UNIVERSITY OF TECHNOLOGY SYDNEY
Faculty of Engineering and Information Technology

Deep Reinforcement Learning for Artificial Intelligence-enabled Autonomous Penetration Testing in Cyber Security

by

Hoang Khuong Tran

Thesis Submitted
in Fulfillment of the
Requirements for the Degree

Doctor of Philosophy

Supervisors: Prof. Chin-Teng Lin
Prof. Yu-Kai Wang

Sydney, Australia

December, 2022

Certificate of Authorship/Originality

I, Hoang Khuong Tran, declare that this thesis is submitted in fulfilment of the requirements for the award of Doctor of Philosophy, in the School of Computer Science, Faculty of Engineering and Information Technology at the University of Technology Sydney.

This thesis is wholly my own work unless otherwise referenced or acknowledged. In addition, I certify that all information sources and literature used are indicated in the thesis.

This document has not been submitted for qualifications at any other academic institution.

This research is supported by the Australian Government Research Training Program.

Signature: Production Note:
Signature removed prior to publication.

Date: December 30th, 2022 Place: Sydney, Australia

ABSTRACT

Deep Reinforcement Learning for Artificial Intelligence-enabled Autonomous Penetration Testing in Cyber Security

by

Hoang Khuong Tran

Penetration Testing (PT) is the set of methods used to enhance the security of a networked system by exploiting potential vulnerabilities. It is the practice of simulating attacks on computer systems, networks, or web applications to test their security and identify vulnerabilities that an attacker could exploit. Penetration testers use various tools and techniques to probe the defenses of a system and uncover weaknesses. These methods require significant time and resources for training and execution. There is a shortage of skilled professionals to deal with the increasingly complex cyber security landscape. Conventional approaches in penetration testing require threat model of the target network to gain context to the exploits and vulnerabilities. These requirements present intractable challenges for penetration testing tools in dealing with rapidly changing network software and attack dimensions.

Artificial intelligence (AI) and reinforcement learning (RL) can potentially be used in penetration testing to automate certain tasks and improve the efficiency of the testing process, such as identifying targets, generating attack strategies, and adapting to changes in the target system. AI-enabled PT has been under research in recent years due to the insurgence of new deep learning advances in which RL is considered an appropriate learning framework to develop such applications thanks to its capability in learning a sequential decision-making process without any labelled dataset through interacting with the environment. For example, an AI-powered tool could analyse the network architecture and system configurations and suggest

potential attack vectors based on this information. An RL system could learn from previous testing experiences and use this knowledge to adapt to new situations and improve its performance over time.

However, there are multitudes of challenges in using RL to develop an automatic penetration testing application. The main challenges are the large and structured configuration of the state space and action space which are unconventional in typical deep RL works. Other challenges include the partial observability, the scarcity of rewards and the stochastic dynamics of the environment.

This research aims at understanding the technical challenges presented by an autonomous penetration testing application and developing novel Deep Reinforcement Learning (DRL) frameworks to deal with two problems of scalable autonomous PT, which involve the complexity of the *action space* and the *state space*. By leveraging the recent advances in Multi-Agent Reinforcement Learning (MARL) paradigm, we re-formulate the conventional approach of using a single-agent DRL into a multi-agent learning framework, enabling the decomposition of the complex and structured *action space* into manageable sub-modules each of which is controlled by a DRL agent. The agents are trained cooperatively to develop PT policies under two different representations of the *action space*: a large and discrete action space and a multinomial parameterised action space. We introduced two new frameworks called *Cascaded reinforcement learning agents for large discrete action spaces* and *Multi-agent reinforcement learning for parameterised action spaces* for each of the aforementioned *action space* representations.

The complexity of the *state space* representation in autonomous PT consisting of the non-visual and binary-valued description of the cyber networks, the highly stochastic state transition probability coupled with the sparsity of the reward signals makes it challenging for DRL algorithms to learn in such application. We adapted Hierarchical Reinforcement Learning (HRL), a multi-layer learning approach wherein the high-level layer is trained to assign different sub-goals to the lower level, which in turn learns a primitive policy to achieve the given subgoals. This integration of HRL into the MARL training is innovative and can be devel-

oped into a more general framework for handling complex problem space in different domains. The subgoal learning is facilitated by using the Successor Representation (SR) as it enables the learning of a state abstraction under environment with sparse or no reward. All the proposed approaches can be integrated end-to-end to develop an AI-enabled autonomous penetration testing application.

Dissertation directed by Distinguished Professor Chin-Teng Lin
Australian Artificial Intelligence Institute (AAII)
School of Computer Science
University of Technology Sydney

To my loved ones

Acknowledgments

The 4-year PhD research is one of the toughest but worthwhile experiences in my life. It is a testimony to my perseverance, intellectual ability, and hard work. The Covid-19 pandemic has put a lot of constraints both mentally and physically to my research as well as my well-being. I experienced the passing of my grandpa and the birth of my first child during my PhD. I could not have done this without the unconditional support of my loved ones. There is no word that I can use to express my gratitude to my parents and my wife who have been standing with me throughout the journey.

I would like to sincerely thank Professor Chin-Teng Lin for giving me the opportunity to join the Brain Computer Interface lab at UTS to conduct my PhD research, and for continually supporting me in both work and personal life. I would also like to express my appreciation to Dr. Yu-Kai Wang for being my co-supervisor and his constant encouragement whenever we meet.

Last but not least, I extend my gratitude towards Dr. Junae Kim, Dr. Toby Richer and Max Standen from the Defence Science and Technology Group, Australia for supporting this work by having regular discussions with us and giving valuable feedbacks on the development of the involved researches.

This work was supported by the Australian Defence Science Technology Group (DSTG) under Agreement No: MyIP 10699.

Hoang Khuong Tran

Sydney, Australia, 2022.

List of Publications

Conference

C-1. **Tran, Khuong** and Akella, Ashlesha and Standen, Maxwell and Kim, Junae and Bowman, David and Richer, Toby and Lin, Chin-Teng, "Deep hierarchical reinforcement agents for automated penetration testing" *Proceedings of the 1st International Workshop on Adaptive Cyber Defense, IJCAI 2021*

Journal

J-1. **Tran, Khuong** and Standen, Maxwell and Kim, Junae and Bowman, David and Richer, Toby and Lin, Chin-Teng, "Cascaded reinforcement learning agents for large discrete action space" *Special Issue Machine Learning for Cybersecurity Threats, Challenges, and Opportunities II*

J-2. **Tran, Khuong** and Standen, Maxwell and Kim, Junae and Bowman, David and Richer, Toby and Lin, Chin-Teng, "A Multi-Agent Reinforcement Learning Approach for Multinomial Parameterised Action Space" *IEEE Transactions on Information Forensics and Security* (Under Review)

J-3. **Tran, Khuong**; Lin, Chin-Teng, "Subgoals Discovery Using Deep Successor Feature Representation in Autonomous Penetration Testing" *IEEE Transactions on Systems, Man, and Cybernetics* (In progress)

Contents

List of Figures

List of Tables

Abbreviation

RL - Reinforcement Learning

MARL - Multi-agent Reinforcement Learning

DRL - Deep Reinforcement Learning

MLP - Multi-layer perceptron

CybORG - Cyber Operation Reseach Gym

RNN - Recurrent Neural Network

GRU - Gated Recurrent Unit

DQN - Deep Q-Network

HRL - Hierarchical Reinforcement Learning

MDP - Markov Decision Process

SMDP - Semi-Markov Decision Process

PAMDP - Parameterised Action Markov Decision Process

POMDP - Partially Observable Markov Decision Process

CTF - Capture The Flag

SR - Successor Representation

SF - Successor Feature

DSR - Deep Successor Representation

ICT - Information Communication Technology

Chapter 1

Introduction

This chapter highlights the motivation of our research and the topic under investigation. It presents the scope of the research and an overview of the organisation of the dissertation.

1.1 Motivation

With the ever-increasing computerisation of society and connectivity of computers, in particular through the Internet of Things, the number of systems vulnerable to cyber-attacks have increased significantly according to Gupta et al. (2017). The development of skilled cyber security professionals requires both highly tailored training courses and practical experience in defending systems. As a result, there is a growing shortage in this field (Oltsik 2018).

Automation has been used by cyber-criminals from the development of the first viruses and internet worms (Orman 2003). Defensive automation tools such as Snort (Zhou et al. 2010) have been used for many years, but these systems have generally relied upon signatures of specific malicious activity rather than behavioural heuristics. Such systems have been easy for skilled attackers to defeat. There is now significant interest in the application of Machine Learning (ML) to detect malicious behavior (Buczak and Guven 2015). The as-yet unsolved problem is how to apply Artificial Intelligence (AI) techniques to the actions taken in response so as to both

counter threats and preserve the functionality of the system.

The development of effective autonomous defenses requires sophisticated attacks to train against. Penetration Testing – the organised attack of a computer system in order to test existing defences – has been used extensively to evaluate the security of Information Communication Technology (ICT) systems. This is a time-consuming process and requires in-depth knowledge for the establishment of a strategy that resembles a real cyber-attack. Available automatic penetration testing tools cannot learn to develop new strategies in response to simple defensive measures. As a result, automatic penetration testing tools that use AI are desirable for simulating real attackers with an attacking strategy instead of performing brute-force attacks when hacking into a system.

Artificial Intelligence-enabled defense systems can provide an automatic response to abrupt real-time attacks. Machine learning based approach is a learning paradigm in which an autonomous agent learns on its own from actual or simulated data to develop an action policy that it can use when exercised in an environment. Leveraging ML algorithms to create an autonomous and responsive defense system is a promising approach in this era, whereby human resources can be freed up to focus on more critical tasks (Geluvaraj et al. 2019). Machine learning approaches have been used extensively in the field of information security. These approaches focus on the detection of attacks including network intrusion, malware, and anomalous behaviour on cyber-physical systems (García-Teodoro et al. 2009; Xin et al. 2018; Ding et al. 2018).

There are three major branches of ML algorithms, namely supervised learning, unsupervised learning and reinforcement learning. Supervised and unsupervised methods require prepared data, either with or without labels, respectively. These algorithms learn the underlying structures and patterns in previously seen data in order to make predictions on unseen data. These methods are susceptible to deception, given the adversarial nature of cyber security (Vo et al. 2022). Reinforcement Learning (RL), in contrast, is a form of learning in which an autonomous decision-making agent explores and exploits the environmental dynamics and discovers a suit-

able policy for acting in such environments. Deep Reinforcement Learning (DRL) uses neural networks as the non-linear parameterised functions to approximate the state and action policy. This learning paradigm is suitable for adversarial and online cyber security scenarios in which attack and defense policies are learned and improved via continuous interactions. Real scenarios often have data or patterns that are not simulated or generated in advance. As a result, it is necessary to develop autonomous AI-based agents using DRL that can learn a responsive defensive strategy from attack patterns that are generated from both simulated and real-time scenarios.

There are several challenges in applying DRL or any Machine Learning algorithms in developing responsive ICT systems. These include large discrete action spaces that are inherent in cyber security networks, partial observability, stochasticity, combativeness and the stability of the DRL algorithms in complex problem space (Nguyen and Reddi 2019).

The use of RL or DRL in cyber security has sparked interests in the research community in recent years, especially against adversarial attacks. There are multiple use cases of applying Deep Reinforcement Learning to cyber-physical systems, from building intrusion detection applications to adversarial attack-defence game settings (Nguyen and Reddi 2021). Tabular RL and deep RL also have been trained to perform automatic attacks on simulated enterprise networks (Walter et al. 2021), privilege escalation (Kujanpää et al. 2021), and penetration testing (Schwartz and Kurniawati 2019a). These works present the feasibility of training RL agents to learn a sequence of actions to perform attacks on simulated cyber scenarios, expanding the research frontiers towards developing AI-based attackers. In particular, work done by Tran et al. (2021) is the first to directly target the scalability of using DRL in large scale cyber scenarios.

Ghanem and Chen (2020a) supports the use of RL in developing efficient penetration testers in terms of maintaining high accuracy and reliability of the system outputs while reducing time consumption for human PT experts. A plethora of recent works in adapting deep RL in automating the learning of the attack graph and

network penetration have shown encouraging results in reducing the dependency on human experts(Tran et al. 2022; Applebaum et al. 2022; Chen et al. 2022; Cody 2022; Niculae et al. 2020).

In summary, the use of RL in PT aim to achieve the following advantages over the conventional approaches to PT (Ghanem and Chen 2020a):

- Reduce the systematic cost, involving both time and money, in deploying regular tests to verify network security.
- Reduce network downtime and security exposure during the tests.
- Reduce human-related errors due to attending to boring and repetitive tasks in PT.
- Develop more flexible and optimised attack strategy that would not otherwise identified and investigated by human experts.

1.2 Scope

This study contributes a major accomplishment toward the learning of attack strategies and effective countermeasures in practical systems: a DRL framework for handling the high-dimensional action space and state space that is present in cyber security scenarios. This framework is demonstrated on an Autonomous Cyber security Operation (ACO) simulator: the Cyber Operations Research Gym (CybORG).

CybORG provides a simulated training platform for an adversarial setting with two teams: the red team (attackers) and the blue team (defenders). A game-based scenario contains a number of subnets, each of which holds a variable number of hosts with various attributes. Some hosts in the network possess valuable and sensitive assets, which referred to as flags in the context of cyber security, to which the attacker aims at gaining access. The numbers of hosts and action properties present high-dimensional state and action spaces in which an autonomous agent from either the red or blue team must learn a suitable action policy. Due to the complexity of the ACO project, it is challenging to develop AI-based agents to integrate with

the CybORG simulator and emulator. Many intricate details on the reaction of the network in response to attack or defensive actions have to be abstracted away from the perspective of RL agents in order to reduce training time; as typical RL algorithms need thousands or millions of training episodes to collect the necessary experiences for learning.

Engineering systems in real world normally have complexity that are usually insurmountable to conventional computation methods and can become intractable due to the immense number of possible states and actions to take into account when making decisions. As a result, an intelligent and versatile sequential decision making framework is needed to handle such systems (Andriotis and Papakonstantinou 2019). The system complexity includes (1) the stochastic nature of the environment dynamics, (2) the large state and action space in which the discrete and continuous nature of the state and action space can each have its own challenge, and (3) the sparsity of the reward signal from the environment. Each of these challenge has its own body of RL research where different techniques are developed to address. Value or policy based reinforcement learning methods by (Watkins and Dayan 1992; Watkins 1989; Sutton et al. 2000) are the foundations to modern deep reinforcement learning algorithms. Regarding the large state and action spaces, Kochenderfer and Hayes (2005); Dulac-Arnold et al. (2015); Andriotis and Papakonstantinou (2019); Arulkumaran et al. (2016) have developed different RL algorithms and training regimes to tackle the big problem space. Hierarchical Reinforcement Learning or HRL is another sub-branch of RL which deals with rewards sparsity (Sutton et al. 1999; Vezhnevets et al. 2017; Dietterich 1998). Additionally, there are ideas to form a feudal structure within the state reasoning capability of the agent to learn latent space abstraction; and as a result, decompose the problem into smaller sets of sub-problem in order to tackle the challenges of (2) and (3). The work in this area is notably known to originate from (Dayan and Hinton 1993; Dietterich 1998; Parr and Russell 1998). However, there is no known frameworks which are specifically designed to handle environments which captured the three aforementioned challenges.

As a result, this research aims to introduce a unified and versatile Reinforcement Learning framework to handle the problem with large state and action spaces, with the presence of reward scarcity. Most of the learning environments in reinforcement learning are associated with a stochastic nature, otherwise it would be straightforward and trivial to learn in a deterministic setting. The general idea that this research focuses on is to use the Multi-Agent Reinforcement Learning paradigm (Buşoniu et al.) to tackle the large problem space. As Tavakoli et al. (2018) suggested, multi-agent system is a potential research direction for tackling problem scalability when facing with complicated domain in practical applications. For example, Tampuu et al. (2015) had extended the single agent Deep Q-Learning algorithm into the multi-agents setting to let them cooperatively solve a common problem. However, this adaptation faces the problem of achieving stable convergences which cannot be resolved without having access to additional information and assumptions in the environment and is confined to the context of two agents.

In human society, there is no significant problem in the our world that can be solved by a single entity. It is always the collective work of a group of individuals with clear milestones and objectives that makes notable impact on the world. In that societal structure, there are individuals with different perspectives, skill sets and capability, with focuses on solving different parts of a problem such that the whole team would achieve the final goal. Feudal reinforcement learning is a research direction that embraced this concept and there have been works trying to realise the idea following the line of researches done by Dayan and Hinton (1993) and Ahilan and Dayan (2019).

This dissertation aims at addressing the following challenges posed by an AI-enabled automatic penetration testing application. Each challenge and its solution form a research objective in this dissertation.

- Investigate the adaptability of the deep reinforcement learning, especially multi-agent reinforcement learning in addressing the large and discrete action space of scalable cyber security networks.

- Investigate the applicability of cooperative multi-agent reinforcement learning in handling structured and parameterised action space in autonomous penetration tester.

- Investigate the feasibility of using model-free and domain-agnostic state representation learning model to train RL agents in the complex state space and sparse reward environment presented by the cyber security domain.

This work does not investigate the use of conventional PT tools in developing attack and defense strategies in a simulated platform. Firstly, most of PT tools are proprietary and only used by information technologist with expertise in penetration testing. Secondly, penetration testing is not a single set of consolidated techniques or programs but rather a combination of multiple interconnected tools and technologies. These tools and technologies are problem specific and therefore used by trained professionals. And finally, there is limited work in applying PT tools to autonomously simulated scenarios, as these platforms are developed with interfaces suitable for running AI-enabled agents, not with conventional PT tools. The structure of this dissertation is described in the next section.

1.3 Thesis overview

Figure 1.1 summarises the main research objectives and the contributions made by this dissertation.

Chapter 1 The first chapter introduces the research topic, the scope and the presentation of the main research objectives.

Chapter 2 The second chapter provides the readers with the technical background to reinforcement learning and related methodologies. It covers an introduction to penetration testing and the capture-the-flag (CTF) scenarios in CybORG, which is the main simulator being used throughout the thesis. Finally, this chapter reviews current literature on the related topics.

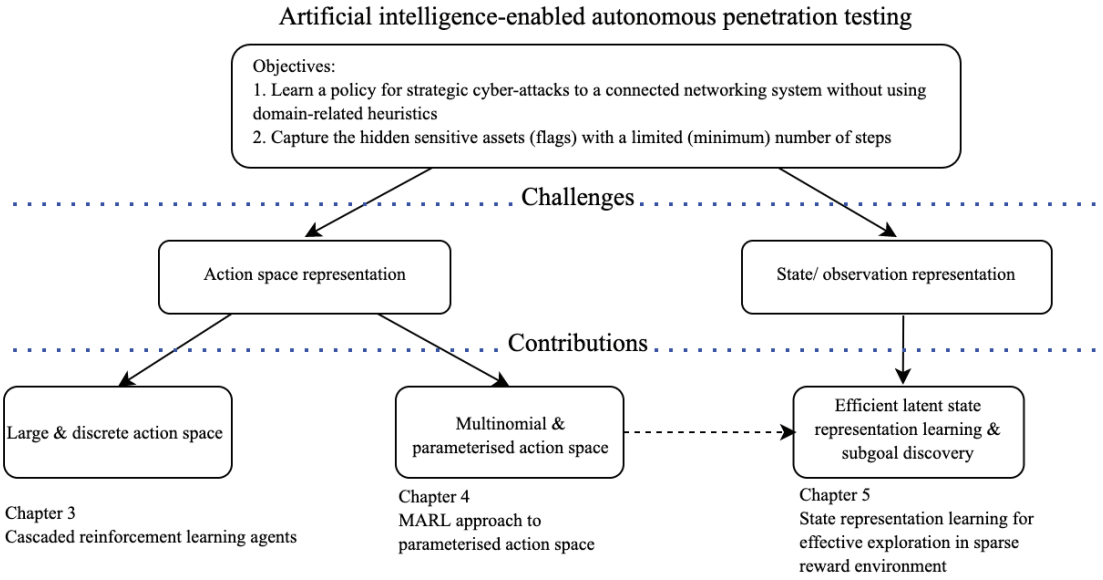


Figure 1.1 : Research objectives

Chapter 3 The third chapter proposes a novel multi-agent reinforcement learning architecture to tackle the action space scalability. The combinatorial nature of the action space in pen-tester easily increases the action space to sizes which are non-typical to RL algorithms. The large and discrete action space presented by an autonomous penetration tester is the first challenge in scaling up the use of AI/ RL for such application. This chapter proposes *Cascaded reinforcement learning agents for large discrete action space* which is a re-formulation of a single agent DRL into a multi-agent setting. The proposed architecture is tested on multiple scenarios and shown to outperform the single-agent dueling deep q-network implementation with similar neural network sizes. This chapter lays the foundation in using multi-agent reinforcement learning in dealing with complex system for the following chapters.

Chapter 4 This chapter tackles a multinomial parameterised action space representation which increases the realism of the cyber simulation. In this chapter, we adapt the *Cascaded reinforcement learning agents for large discrete action space* from previous chapter and design a multi-agent learning for the parameterised action space. The architecture, called *Multi-Agent for Parameterised Action* or MAPA,

relies on a sample efficient value-based algorithm called double dueling q-learning with cooperative training strategy. The proposed methodology shows superior performances in multiple CybORG scenarios when compared to a baseline approach in reinforcement learning literature targeting at parameterised action space. The work developed in this chapter is used for the next chapter.

Chapter 5 In chapter 5, we look at state representation learning in cyber-security simulator and penetration testing. We examine different state representation learning models and propose a novel approach in combining deep auto-encoder and successor feature representation in reducing the state dimensionality and extracting important features for the cyber scenarios. This latent representation is useful in extracting subgoals in the absence of the reward signals and can be used with hierarchical reinforcement learning to tackle sparse reward environments.

Chapter 6 This chapter concludes the dissertation summarising all the main contributions proposed by this work. We highlight certain drawbacks in the current work for practical deployments and propose potential approaches for future research in the domain.

Chapter 2

Background and Literature Review

This chapter gives the readers an introduction to the background of penetration testing and reinforcement learning. It also describes different reinforcement learning formulations and the available algorithms which are used in the following chapters. The chapter concludes with a literature review on different challenges this work tries to resolve in developing AI-enabled autonomous penetration testing.

2.1 Penetration Testing

2.1.1 Introduction

This section provides a brief overview of the penetration testing landscape and how it could be formulated as a RL problem. For a deep dive of the topic, we refer the interested readers to this research(Sarraute 2013).

Penetration testing has been used for a long time in improving the security of cyber systems (Schwartz and Kurniawati 2019b). This is a carefully designed plan of attack which contains multiple steps to compromise valuable assets in a network. These consists of *Network scan*, *Host and Subnet scan*, performing different types of exploit to find vulnerability and to run script execution. The intermediate success allows the red (attacker) agent to gain deeper access to the network. The details on the types and variety of attack actions differ across networks and systems, however they do share similarities in general (Niculae et al. 2020). The knowledge learned

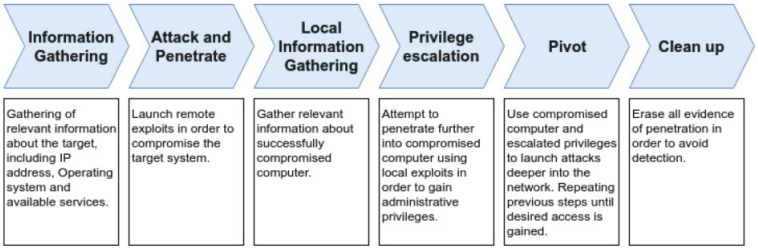


Figure 2.1 : A general penetration testing process (Sarraute 2013).

from these planned attack helps network administrators and designers to find and fix detected vulnerabilities. Descriptions of main attack types are illustrated in Figure 2.1.

In a simulation environment used for pen-testing, there are three main classes of agents: red, blue and green agents. Red agents represent the attacker whose actions targeting at finding vulnerability and compromise the networks. Blue team is the defender whose actions are to detect any potential suspicious behaviors from the attacker and exercise defending actions. Green agents represent users' activities in the network that are neither attack or defense.

There are different attempts to build a complete cyber simulator that bridges the gap between reality and simulation (Schwartz and Kurniawati 2019b), CybORG is the recently developed simulator that shares common interfaces between a simulator wherein certain action effects are abstracted away from the decision making agents, and an emulator where time delay and effects of the actions are taken into account. During the course of this research, CybORG had gone through different versions with changes to scenario representations. CybORG v2020 (Baillie et al. 2020) was used in the first research objective with large discrete action space and CybORG v2021 (Standen et al. 2021) is being used for the second and the third research objectives.

As this research focuses on developing RL algorithms for enhancing automatic penetration testers, interested readers are encouraged to read the work of (Schwartz and Kurniawati 2019b) on developing penetration testing simulator and a compre-

hensive review of applying DRL in cybersecurity done by Nguyen and Reddi (2021) for broader understanding of these topics.

According to Standen et al. (2021), Autonomous Cyber Operations or ACO are the research direction concerning with designing the defence of computer network systems via learning autonomous strategy using machine intelligence. These crucial operations are needed for business and national security systems wherein the complexity far exceeds the coverage of human operators and specialists. It is challenging to develop AI for ACO domain as it usually involves multi-disciplinary research. Similar to designing game AI, there is the adversarial component which makes it even more challenging. The developed AI agent has to be robust and adaptive to the unknown and varying opponents. ACO is also affected by the need of a well-developed simulator, which is critical for testing any proposed algorithm. On this front, we have, in collaboration with Defence Science and Technology Group in Australia, developed and tested the ACO simulation environment called CybORG with full support for AI and reinforcement learning research. This dissertation is the first work to actually employ deep reinforcement learning algorithms to exercise the developed CybORG simulated environment.

2.1.2 Threat Models in Penetration Testing

The purpose of penetration testing is to develop systematic attack plan to find security flaws in a network, which are also known as *vulnerabilities* or *security bugs*. These vulnerabilities are then *exploited* to the attacker's advantage in which case unintended code execution, data breach on computer hardware and software can occur. This will eventually lead to the computer system being in controlled by malicious entities.

There are principled mechanisms where an exploit is implemented. An *attack vector* is a method in which the vulnerability reveals itself. For instance, an attack vector can be a phishing email with malicious links or attachments which trigger harmful code execution upon open by an application or user. Once the vulnerability is revealed, the attacker can run code execution to trigger buffer overflow, string for-

matting or race condition. Payloads, which are malicious codes, can be executed to change system accounts, configurations, or simply gain the highest privilege control of the computer.

As illustrated in Figure 2.1, there are different steps contributing to a successful penetration testing process.

- Information gathering: the actions in this step serve the purpose of gathering relevant network information such as active IP addresses (network discovery), opening ports and and running services (port scanning), and operating systems (OS identification).
- Attack and Penetrate: with the data collected in the previous phase, the attacker then launches a suitable exploits.
- Local Information Gathering: during this step, the attackers compile the information collected from the successful code execution in the second phase. This helps the attackers update their belief on the network connections, and reveal more vulnerabilities as they have more access to the compromised hosts.
- Privilege Escalation: at this phase, the attacker can claim higher privilege, such as administrative access to a compromised host.
- Pivot: Once a host is fully controlled by losing its administrative right, the attacker can have complete views of the connected subnets and is able to launch its attack to any of the other hosts in the connected subnets.
- Clean Up: If the previous attacks go un-noticed, the attacker can remove its footprints and minimise the chance of being detected.

2.1.3 Automated Penetration Testing Tools

There are many industry-level tools which have been developed to cope with the large and complex cyber security landscape. Nmap(Lyon 2008), which is a network scanner, is used to provide network information such as operating system, currently

running services and open ports. The collected information is analysed by vulnerabilities scanners such as OpenVAS(Rahalkar 2019) or Nessus(Anderson 2003). Penetration testing framework such as Metasploit(Kennedy et al. 2011) is then used to search and to launch exploits based on the gathered information. The mentioned automated tools enhance the efficiency of the entire testing process. However, it also requires substantial amount of training time and technical expertise to complete a thorough and successful pentest, and human is still an important factor for identifying unfamiliar exploits.

Another AI related approach for automated penetration testing is via constructing attack graph(Ingols et al. 2006). This is a system modeling technique where each node in the graph represents a specific network state and the edge is the state transition after each exploit. Once the graph is constructed, classical planning techniques are used to search for an attack path to penetrate the network(Boddy et al. 2005). However, in order to leverage this method, it requires the attacker having the complete knowledge of the network topology, which is neither impractical and intractable due to the hidden nature of network penetration and the complexity of real world networks.

The necessity of having an autonomous penetration testing tool built with self-learning intelligence without prior knowledge motivates the research community into developing RL-based method for cyber security applications.

2.1.4 CybORG Overview

CybORG is designed as a gaming interfaces where classes of agents can interact on a shared scenario. Each scenario can be configured to have different number hosts, subnets, ip addresses and other parameters of a real network such as users, passwords etc... Each game lasts a number of time steps and contains hidden valuable assets which are referred to as flag which is similar to capture the flag (CTF) challenges (Figure 2.2). The game is modeled as discrete time decision making process where the agents exercise actions on the environment for each time steps. The observations presented to the blue, red and green teams are different. From an attacker's point

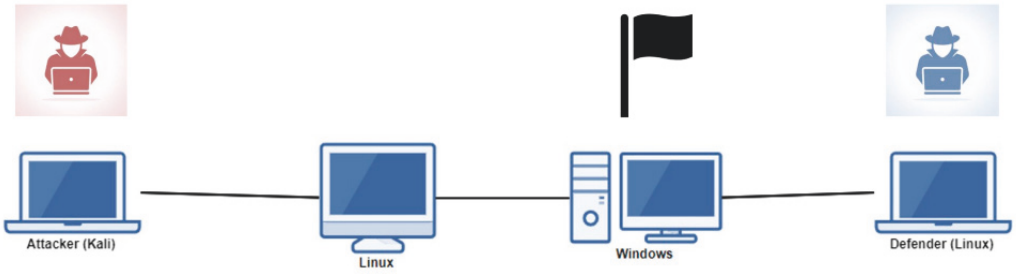


Figure 2.2 : CybORG diagram

.

of view, it can only observe parts of the compromised network. From the defender's viewpoint, it does not know which actions are being executed by the red team and the green team is not aware of any on-going attack or defense actions.

The action space of CybORG can be a large discrete set of action identifiers as in CybORG v2020 or can be structured into hierarchy of main attack type and corresponding parameters space which is available in CybORG v2021.

The CybORG design follows OpenAI gym interface (Brockman et al. 2016) where the state space, action space and environmental interactions follow certain functional requirements. This makes it easier to develop AI agents for CybORG. The agents get a main reward for capturing the required number of hidden assets after which the game was reset to its initial state. After each actions, the environment returns a next state, and some extra information which the agents can take advantage to use as a proxy to the true game state.

The actions' effects are stochastic in nature. An action can be invalid or valid, depending on the availability of the network. For example, the agent cannot attack a host or a subnet that it does not know about, even though that host does exist in the true game state. For valid actions, they also have configurable success probabilities. The state transition dynamics is also probabilistic as the next state follows a probabilistic distribution for any given pair (s_t, a_t). This makes the game highly stochastic hence computationally expensive to apply classical planning algorithms

or model-based reinforcement learning.

We hope CybORG will become a benchmark environment for future AI-assisted automatic penetration testing development given its sophisticated design and an easy-to-use interface.

2.2 Reinforcement Learning

Intelligent agent, as defined by Russell and Norvig (2010), is an entity that perceives the world through its sensors and interact with the surrounding via its actuators. Through the sequence of exercised actions, the intelligent or rational agent acts in such a way to maximise its utility or some performance measures. Machine learning (ML) is the field of research to develop algorithms that improve the learning of any computer agents through experiences. There are three main categories of learning processes in ML, namely, supervised learning, unsupervised learning and reinforcement learning (Mitchell 1997). Supervised learning requires a labelled dataset with many examples of matching input (e.g. X) and output (e.g. y) pairs in order to learn a mapping function $f : X \rightarrow y$. The learnt function is then used to make prediction on unseen data. Unsupervised learning, on the other hand, extracts latent patterns in unlabelled datasets such as groupings or clusters of data.

Reinforcement learning (RL) is different from the former two methods as it does not need a provided static dataset. Instead, the RL agent learns by interacting with an unknown environment and builds up its dataset of experiences. The learning signals or feedbacks are not labels but scalars of rewards of whether its previous sequence of action leads to a positive or negative outcome. Figure 2.3 illustrates the agent-environment interaction model in RL. The goal of the RL agent is to learn a sequential decision making process or a policy, given its observation of the world and the reward signals, to maximise its utility of accumulative rewards.

2.2.1 Formulation

Markov Decision Process

Puterman (1994) developed a mathematical framework called Markov Decision Process or MDP to formulate the problem of sequential decision making. This thesis adapts the notations following the book *Introduction to Reinforcement Learning* by Sutton and Barto (2018a).

The RL agent is exposed to an *Environment* where it gets information about the world via *State* or *Observation*, exerts certain *Action* and receives *Reward* and transitions to the *Next state* as a result of taking that action. In RL, the agent does not have the information on the environment dynamics such as the state transition or reward functions. It needs to learn a *Policy* of what action to take in each state in order to achieve the maximum accumulative rewards in the future. This learning process is formalised as a sequential decision making process, and represented mathematically as a Markov Decision Process.

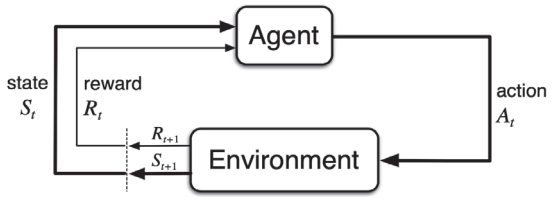


Figure 2.3 : Agent - Environment Interaction

The MDP is specified through a tuple of information as $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma \rangle$. At each time step t the agent receives a state representation of the environment $s_t \in \mathcal{S}$, it then takes an action $a_t \in \mathcal{A}$ following an action policy $\pi(a|s)$ to interact with the environment. In response, the environment returns a reward signal $r_{t+1} \in \mathcal{R}$ and transition to a new state $s_{t+1} \in \mathcal{S}$ (or $s' \in \mathcal{S}$) with a next state probability distribution $P(s'|s,a) = \mathcal{T}(s,a,s')$, with $\mathcal{T} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0,1]$. The state transition dynamics of the environment is defined in Equation 2.1.

$$p(s', r|s, a) \doteq Pr(S_{t+1} = s', R_{t+1} = r|S_t = s, A_t = a) \tag{2.1}$$

The crucial thing about the Markov decision process is that the probability distribution of the next state completely depends on the immediate preceding state and action. The state, therefore, must include all the necessary information for the agent to make the decision. The MDP formulation gives a convenient way to define some utility functions that the agent can assess and optimise its decision. The *value function* of a state s under a *policy* π is defined as in Equation 2.2:

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s], \forall s \in \mathcal{S} \tag{2.2}$$

with G_t is the return and defined as in Equation 2.3:

$$G_t \doteq R_{t+1} + R_{t+2} + R_{t+3} + ... + R_T = \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k \tag{2.3}$$

The discount factor $\gamma \in [0, 1]$ prevents the return from exploding to infinity for scenarios with long or infinite horizon T. It also a parameter controlling how much important future reward is to the agent. A discount factor $\gamma = 0$ leads to a *myopic* agent where it basically just optimises its action for achieving the immediate reward without any regard for future return. As the value of the discount factor increases, the agent considers more about the discounted future return.

The state value function tells the agent how good it is to be in that state. However, it is even better that there is a utility function telling the agent how good it is to be in *that* state and taking *that* specific action in order to achieve the highest possible reward. Equation 2.4 describes the state-action value function.

$$q_\pi(s, a) \doteq \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s, A_t = a], \forall s \in \mathcal{S}, \forall a \in \mathcal{A} \tag{2.4}$$

In RL, the agent is expected to learn an optimal policy π^* to maximise its expected utility or the Q-value function. The optimal state value function $V^{\pi^*}(s)$ and the optimal state-action value function $Q^{\pi^*}(s, a)$ gives the maximised returns the agent can obtain in such state and taking the action (Equation 2.5).

$$
\begin{aligned}
V^{\pi^*}(s) &= \max_{\pi} V^{\pi}(s) \\
Q^{\pi^*}(s,a) &= \max_{\pi} Q^{\pi^*}(s,a)
\end{aligned}
\tag{2.5}
$$

The above optimal value functions can be found by recursively solving a system of equations, often referred to as *Bellman optimality equations* (Equation 2.6).

$$
\begin{aligned}
V^{\pi^*}(s) &= \max_{a \in \mathcal{A}} \sum_{s',r} p(s', r|s, a)[r + \gamma V^{\pi^*}(s)] \\
Q^{\pi^*}(s,a) &= \sum_{s',r} p(s', r|s, a)[r + \gamma \max_{a'} Q^{\pi^*}(s', a')]
\end{aligned}
\tag{2.6}
$$

In practice, the state transition dynamics $\mathcal{T}$ are unknown. As a result, there are different methods to estimate the optimal value functions. A model-based RL agent estimates the transition dynamics $\mathcal{T}$ and the reward function $\mathcal{R}$ before learning $Q^{\pi^*}(s,a)$. A model-free RL agent, on the other hand, learns and estimates the Q-values or the policy directly with trade-off between sample and computation efficiency. Approximating the optimal value functions provides the agent with the knowledge of the state value of the environment; and therefore the ability to select the best action to take in each of those states to achieve highest possible returns.

Partially Observable Markov Decision Process

In complex real world problems, it is not possible for a single agent to observe everything from the world. Even if it is possible do so, there are still latent state information that the agent needs to learn in order to infer the ground truth states. Partially observable environments are common in multi-agent system in which each agent can only observe the local information available to itself. In a partially observable environment, the agents cannot observe directly the states of the environment but only infer the true state via its local observations. As defined by Kaelbling et al. (1998), POMDP is formulated as a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \Omega, \mathcal{O}, \gamma \rangle$ with the additional notations:

- Ω is the set of observations that the agents can observe.

- $\mathcal{O} : \mathcal{S} \times \mathcal{A} \rightarrow \Pi(\omega)$ is the observation function that maps a state and an action to a distribution over observations. Generally in reinforcement learning, this function is unknown to the agent and it has to be learned.

As a result, a POMDP agent as illustrated in Figure 2.4 needs to have a state estimator in order to aid the policy learning.

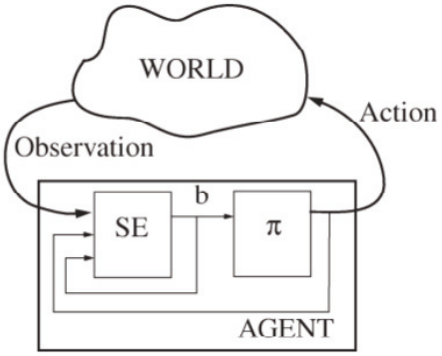


Figure 2.4 : A POMDP agent with a state estimator (SE) to assist itself in interprete the world state from the observation.

In RL, memory units such as recurrent neural network (RNN) and its variants; including gated recurrent unit or GRU, and long short term memory or LSTM (Goodfellow et al. 2016) are common methods for maintaining an internal latent state representing the entire history of agent's experiences. These remedies mitigate the affect of not maintaining of the Markov properties; in which the agent must have enough information at each time-step to make decision and not rely on past data. Recurrent networks have been used with deep q-learning to solve POMDP problems (Hausknecht and Stone 2015c). It is notoriously hard to leverage model-based RL to solve POMDP (Banerjee et al. 2012). As a result, model-free RL is preferred to learn the policy directly.

Semi-Markov Decision Process

MDP and POMDP provide the formulation for two fundamental problems in sequential decision making process. However, in complex problem domains, there is another

formulation which is proved to be useful in learning more complex policies. During the development of RL, it is suggested that actions may not need to be executed for every time step of the environment, or the simulator. Just as human performs his or her actions for certain tasks to make breakfast, he or she would first need to come up with a *plan* to get milk and cereal, and then actually executes a number of steps to move to the fridge or open the cub-board to get the item. According to Sutton et al. (1999), the terms *option*, used in the semi-Markov Decision Process, refers to a generalisation of actions that are expanded over several time steps. There are three components to formulate an option: a policy $\pi : \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$; a termination condition $\beta : \mathcal{S}^+ \rightarrow [0, 1]$; and a set of initial states $\mathcal{I} \subseteq \mathcal{S}$. A valid option $\langle I, \pi, \beta \rangle$ in state s_t if and only if $s_t \in \mathcal{I}$. If the option is selected, it will take actions according to policy π until meeting the termination condition in β. After finishing one option, the agent can move on to take either the next primitive action or another option depending on its policy. The policy over options is normally denoted as $\mu : \mathcal{S} \times \mathcal{O} \rightarrow [0, 1]$. The *option* framework is powerful in defining specific actions that are spanning over several time steps of executing. Learning options or policy over options helps the agents narrow down the state space that it needs to explore and have some intermediate rewards to guide its learning. This is useful when trying to apply RL to complex problem domains where the state and action space are enormous and the reward signals are sparse. Without this policy over options, the agent may not be able to expand its explored regions given no reward signal is achieved after several thousands of exploration episodes or steps. This *option* framework later gives rise to the development of Hierarchical Reinforcement Learning.

Decentralised Markov Decision Process

Bernstein et al. (2013) and Oliehoek (2010) formulated the Decentralised Markov Decision Process or Dec-MDP for Multi-Agent Stochastic game environment, especially for the cooperative decision making under uncertainty. Under partially observable setting, a decentralised POMDP is a multi-agent system version of POMDP. This formulation aids the development of MARL in which each agent does not observe

the same information. The transition probability $P(s'|s, a^1, a^2, ..., a^m)$ and the reward function $R(s, a^1, a^2, .., a^m)$ take into account the action a^i of each agent i in the environment. Ω^i is the set of observations of agent i. $\mathcal{O}$ is the table of all observation probability where a joint probability $\mathcal{O}(o^1, o^2, ..., o^m|a^1, a^2, ..., a^m, s')$ defines the probability that each observation $o^1, o^2, ..., o^m$ are observed by agent $1, 2, ...m$, given that each of them executes the action $a^1, a^2, ...a^m$ and the state transitions to next state s'. Additionally, each agent i has a set $\mathcal{A}_o^i$ which contains the actions that are available for each observation $o^i \in \Omega^i$.

For each $a^1, a^2, ..., a^m, s'$, define $\omega(a^1, ..., a^m)$ to be the set of observations that have non-zero chance of occurring given the execution of actions $(a^1, a^2, ..., a^m)$ and next state is s'. The requirement is that the state must be uniquely determined from the observation $(o^1, o^2, ..., o^m)$ which associated with $(a^1, a^2, ..., a^m)$ and next state is s'.

A local policy δ^i is a mapping from the agent's histories of observations $o_1^i, ..., o_t^i$ to action $a^i \in \mathcal{A}_{ot}^i$. A joint policy $\delta = (\delta^1, ..., \delta^m)$ is the grouped of local policies of each agents in the environment. The goal is to find the optimal joint policy to yield a optimal reward.

According to Bernstein et al. (2013), a decentralised setting with m-agent where $m > 2$, the finite horizon m-agent Dec-POMDP is NEXT-complete or intractable with polynomial time. Therefore, in order for MARL to work, there must be certain assumptions to the problems and environment. In this dissertation, the formulation of MARL in Dec-POMDP is developed with relaxation to the constraints presented to the group of agents. This makes it feasible to extend single-agent deep reinforcement learning approach to multi-agent setting.

2.2.2 Reinforcement Learning Algorithms

Previous sections described different problem formulations which are common in defining a sequential decision making process, there are many reinforcement learning methods that can be used to address these problems. However, it is highly dependent upon the type of problems that some methods are preferred to others. This research

aims to develop a reinforcement learning framework which leverage many of the core reinforcement learning algorithms.

Overall, there are two main umbrellas of algorithms in solving the RL equations as illustrated in Figure 2.5*. There are value-based methods which trying to learn the value or action value functions. The famous Q-Learning or Temporal Difference learning algorithm falls under this branch. On the other hand, there are the policy-based methods which directly learn a policy through the samples of interaction. REINFORCE or Proximal Policy Optimisation algorithms (Schulman et al. 2017) are some famous examples that fall under this branch. However, for each of these methods, it exists certain kind of trade off on the bias and variance during learning, as well as sample efficiency. As a result, a hybrid version of both value based and policy based methods gives rise to the actor critic framework, in which a value function is learned to help the training of the policy gradient and reduces its variance. Some famous examples of actor-critic methods are Advantage Actor Critic, Advantage Asynchronous Actor Critic or Deep Deterministic Policy Gradient (Arulkumaran et al. 2017).

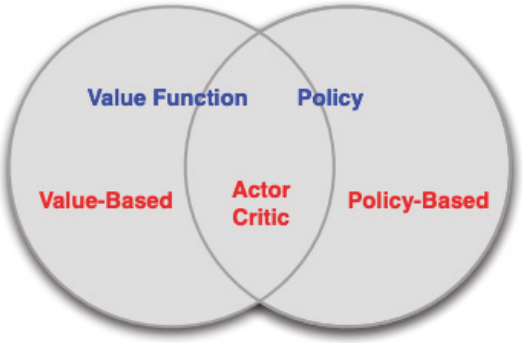


Figure 2.5 : Reinforcement Learning methods (Silver 2015).

On policy based method: Dynamic Programming

This is a class of algorithms that aim to directly solve the above state value and action value functions. The requirement for this method to work is a model of how

*Adapted from the infamous David Silver's lecture notes (Silver 2015)

the environment works which meant the state transition probability is provided.

The value of a state s would then be calculated using the below Equation 2.7:

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi[G_t|S_t = s] \\ &= \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s], \forall s \in \mathbf{S} \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')] \end{aligned} \tag{2.7}$$

where $\pi(a|s)$ is the probability of taking action a in state s under policy π.

The action value function is then solved as:

$$\begin{aligned} q_\pi(s) &\doteq \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s, A_t = a] \\ &= \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')] \end{aligned} \tag{2.8}$$

The gist of DP is to create a lot of episodes of the interaction between the agent and the environment and uses a tabular data structure to store all the possible values of state and action and update it accordingly to the experienced episodes. This is of course can only work in finite state and action domain.

On policy based method: Monte Carlo Methods

Monte Carlo methods are a class of algorithms that belongs to the group of model free reinforcement learning in which the agent does not need to know the underlying dynamics of the environment. It can learn the optimal policy through interacting with the environment. For each episode or trial of interaction, the agent will collect a series of state, its action when it is in that state, the rewards it gets and the next states. The agent then learns to approximate the value function to represent how good it is to be in each state of the environment. Again the concept here best understood when using tabular format to store the value of the individual state. For more complicated scenario, neural networks are used to approximate the state and its value. The drawbacks of MC methods is that the agent has to experience the whole episode until the termination of the episode in order to learn the value of the states it has gone through. This becomes infeasible in environment where there is

non-zero probability of non-termination. For example, stock trading environment, self driving car simulation etc... The advantage of MC methods is its low bias in the estimation of the state value function, since it updates the state value function based on actual environment returns.

Value based method: Temporal Difference Learning

Tabular approach for Q-Learning to learn optimal policy is guaranteed for convergence. However this is not feasible in practical problems due to the large complexity of the state and action space. Function approximation approach has been proposed for example by (Gordon 1995) and recently is implemented with neural networks (Mnih et al. 2013, 2015). Additional engineering practices are implemented to enhance the stability and convergence probability of Deep Q-Network (DQN). These include a replay memory buffer $\mathcal{D}$ to store experience tuple (s, a, r, s') to update DQN parameter θ through the loss function $L_j(\theta_j) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{D}}[(r+\gamma \max_{a'} Q(s', a'; \hat{\theta}_j) - Q(s, a; \theta_j)^2]$. There are two networks used to learn Q-value function namely a target network parameterised by $\hat{\theta}_j$ and an evaluation network parameterised by θ_j. The target network are used to select the action and updated less frequently than the evaluation network.

Temporal difference learning is the central idea in reinforcement learning. It combines the advantage model free learning from Monte Carlo methods and the dynamic update from dynamic programming to update the state and state - action value functions.

The fundamental formula to update the state value function in TD learning is:

$$\begin{aligned} V(S_t) &\leftarrow V(S_t) + \alpha[G_t - V(S_t)] \\ &\leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1} - V(S_t)] \end{aligned} \tag{2.9}$$

The main take away that is also the main difference between TD learning and MC methods is the temporal difference in the update of the state value function. The value of the state is updated immediately after 1 time step of experience in the environment. The immediate reward and the state value of the next state is used to guide the update of the value of the current state. This way the agent does not

need to wait till the end of the episode to update the state value, instead it uses the estimate of the next state to update the current state. Doing this injects bias into the updates of the state value, however with a lot of training episodes, it is proven that TD learning can converge to optimal policy. One of the famous TD learning algorithm is Q-Learning where the action-value function is updated using TD learning:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \tag{2.10}$$

François-Lavet et al. (2018) summarises the DQN algorithm in the sketch figure below 2.6:

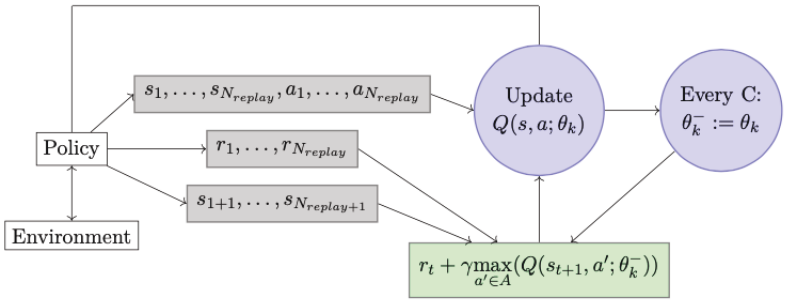


Figure 2.6 : DQN algorithm

Policy based method: Policy Gradient

Another major branch in reinforcement learning deals with policy methods. Dynamic programming, Monte Carlo and temporal difference learning are all value based methods where the value of the state is estimate in order to derive the optimal policy. Policy methods directly use the return to find the policy $\pi(a|s)$ directly. Common algorithms that fall under this category include REINFORCE, Proximal Policy Optimisation and Actor Critic frameworks. The performance measure is defined generally as $J(\theta)$ and to maximise the performance, θ should be updated as:

$$\theta_{t+1} = \theta_t + \alpha \widehat{\nabla J(\theta_t)} \tag{2.11}$$

and $\nabla J(\theta_t)$ is established as:

$$\nabla J(\theta) \propto \sum_s \mu \sum_a q_\pi(s,a) \nabla \pi(a|s,\theta) \tag{2.12}$$

2.2.3 Hierarchical Reinforcement Learning

Hierarchical Reinforcement Learning is an approach aiming at learning abstraction over state space or action space, in order to address challenging tasks of having sparse rewards or long horizons. The idea of building up learning hierarchy in RL dates back to the work conducted by Dayan and Hinton (1993), who first proposed the concept of HRL in which agents at different layers of decision making observe variable spatial resolution to search for the solution (Figure 2.7). The agents at higher layers look at a coarser level of details while the lower-level agents interact with the primitive details of the environment.

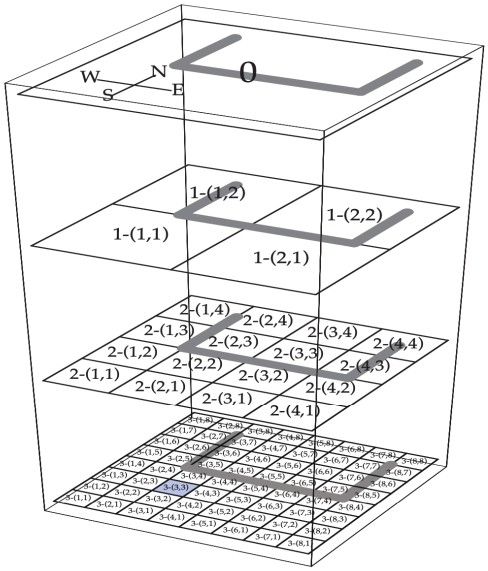


Figure 2.7 : Feudal Architecture with spatial abstraction

The idea has been revisited recently and inspires new line of research into using HRL to solve complex problems, especially challenging ones with hard exploration and sparse reward signal. Figure 2.8 presents one such problem in RL benchmarks. The screenshot is from an Atari game called *Montezuma's Revenge*.

Figure 2.8 : A screen of the Atari game called Montezuma's Revenge. The agent has to climb different ladders, avoid the ghosts and get to the key in order to transition to a new screen. The agent has to learn this entire sequence of actions without any reward signal.

Precup (2000) defines options as temporal extended actions and will be chosen and executed by the agents for a number of time steps until termination either by reaching maximum step size or achieving terminal states. According to (Sutton et al. 1999), a Markovian option is defined as a triple $\langle I, \pi, \beta \rangle$ where I is the set of states that can be used as the inputs for the options, π is the option's policy over the input states and β is the termination condition.

Another framework of HRL was proposed by Parr and Russell (1998) which considered everything in the environment as state machine and the agent learned through navigating to different state abstraction. However this idea has problem of realisation and had not been visited by recent line of research.

Inspired by the idea from the work of Dayan and Hinton (1993), a hierarchical structure in (Vezhnevets et al. 2017) with two network layers namely a manager and a worker was implemented to solve the game Montezuma which had failed many deep reinforcement learning algorithms because of its lack of reward signal. The

manager learns the latent space representation to discover import sub-goals and use it to direct the exploration of the agents. Kulkarni et al. (2016) developed a more complete framework upon the HRL idea where the manager layer uses intrinsic reward to train the worker agent to achieve the sequence of sub-goals.

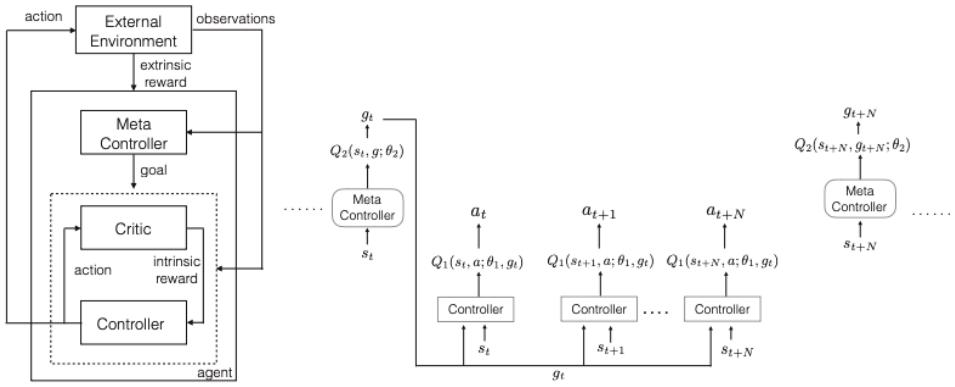


Figure 2.9 : HRL with temporal abstraction sub-goals

As shown in Figure 2.9, the meta-controller takes the external observation and extrinsic reward signal and learns to generate sub-goals for the controller. The controller performs primitive actions on the environment and checks whether it satisfies the sub-goals from the meta-controller. If it is, then the controller gets an intrinsic reward. The goals from the meta-controller can be an option or a policy over options which expand several time steps.

Many approaches and frameworks proposed in (Nachum et al. 2018; Zhang et al. 2019; Ghavamzadeh et al. 2006) are improvements upon the mentioned HRL structures. Nachum et al. (2018) provides an efficient framework for Hierarchical Reinforcement Learning with high level policy giving a series of goal to low level policy, in addition to a low level policy correction to reduce the training samples and increase performance.

2.2.4 Multi-Agent Reinforcement Learning

Multi-agent reinforcement learning or MARL is an advancing research area in current reinforcement learning. The use of multi-agent extends the learning capability

of single-agent algorithms. This dissertation does not attempt to cover the theory of MARL but rather to provide a brief introduction to MARL. There are major challenges in MARL which are listed as follows:

- Non-stationarity: As agents perform training together in the environment, each agent's policy will evolve and cause unexplanable difference and discrepancy to the other agents' policy.
- Coordination: The main aim of having multi-agents working together to solve a complex problem is to leverage the coordination of the agents. This is however, is still a challenging problem.
- Scalability: Most research works claims to have MARL scalable to large number of agents. However, there is always trade-off between scalability and performance of each agent.

Panait and Luke (2005) defines a multi-agent system is the one in which there are many autonomous computational mechanisms perform actions in an environment. There are certain constraints such that each agent is not able to know the state of the world that the other agents are observing (partially observable) or they do not know the internal states of the other agents.

The exploration and learning process of a MAS is inherently more complicated compared to a single agent environment. This is due to the fact that multi-agents are interacting with the environment, each trying a stochastic policy and targeting to maximise its own utility. A naive extension of single agent algorithms to a multi-agent scenario in which each agent has to learn its policy and assumes all of the other agents belong to the environment lead to a violation of Markov property since the stationarity of the environment dynamics no longer hold (Tuyls and Weiss 2012; Laurent et al. 2011). As demonstrated in (Tan 1993b) and (Matignon et al. 2012), this *independent q-learner* approach only works in practice for a limited case of problem setting with very few number of agents.

An extension of the MDP is used to model the multi-agent learning behaviour.

This is called a stochastic Markov game which has the tuple $(\mathcal{S}, \mathcal{N}, \mathcal{A}, \mathcal{T}, \mathcal{R})$. Details description of each item are as follows:

- $\mathcal{S}$ contains all the local observations of each agent with $\mathcal{S} = S_1 \times S_2 \times ... \times S_n$ where S_i is the individual observation from agent i.
- $\mathcal{N}$ is the set of N agents.
- $\mathcal{A}$ is the set of joint action from N agents and expressed as $\mathcal{A} = A_1 \times A_2 \times ... \times A_n$.
- $\mathcal{R}$ is the reward function for N agents and expressed as $\mathcal{R} = R_1 \times R_2 \times ... \times R_n$.
- $\mathcal{T}$ is the transition function of the states and the joint action for N agents and expressed as $\mathcal{T} = S \times A_1 \times ... \times A_n$.

For an agent i, the set of all agents excluding agent i is denoted as $-\mathbf{i} = \mathcal{N} \setminus \{\mathbf{i}\}$. The value function of the agent depends on the joint action and policy as $\mathbf{a} = (a_i, a_{-i})$ and $\boldsymbol{\pi}(s, \mathbf{a}) = \prod_j \pi_j(s, a_j)$.

$$V_i^{\boldsymbol{\pi}}(s) = \sum_{\mathbf{a} \in \mathcal{A}} \boldsymbol{\pi}(\boldsymbol{s}, \mathbf{a}) \sum_{s' \in \mathcal{S}} \mathcal{T}(s, a_i, \mathbf{a_{-i}}, s')[R_i(s, a_i, \mathbf{a_{-i}}, s') + \gamma V_i(s')]. \tag{2.13}$$

As a result of the above value function, the optimal policy of each agent depends on the policies of the other agents.

$$\begin{aligned} \pi_i^*(s, a_i, \boldsymbol{\pi_{-i}}) &= \arg\max_{\pi_i} V_i^{(\pi_i, \boldsymbol{\pi_{-i}})}(s) \\ &= \arg\max_{\pi_i} \sum_{\mathbf{a} \in \mathcal{A}} \pi_i(s, a_i) \boldsymbol{\pi_{-i}}(s, \mathbf{a_{-i}}) \\ &\sum_{s' \in \mathcal{S}} \mathcal{T}(s, a_i, \mathbf{a_{-i}}, s')[R_i(s, a_i, \mathbf{a_{-i}}, s') + \gamma V_i^{(\pi_i, \boldsymbol{\pi_{-i}})}(s')] \end{aligned} \tag{2.14}$$

The joint policy $\boldsymbol{\pi_{-i}}(s, \mathbf{a_{-i}})$ is non-stationary since the agents' policies keep on updating during the training. It has been studied and confirmed by Littman (2001) that in a cooperative environment it is not possible to guarantee the convergence of optimal policies for each agent without extra assumptions on the scenario.

2.2.5 Hierarchical Multi-Agent Reinforcement Learning

This research aims to integrate the hierarchical reinforcement learning methods into the MARL in order to better coordinate the learning and execution of agents. HRL provides different methods to learn and discover sub-goals in the environment. These discovered goals can then be used to assign to each lower agents.

Ghavamzadeh et al. (2006) was one of the first to propose the idea of integrating Hierarchical Reinforcement Learning into a Multi-agent System in order to speed up learning and enhance coordination between agents. They extend the MAXQ framework of single agent HRL (Dietterich 1998) into the MARL structure. The agents in the paper are homogeneous and learn from the same task hierarchy with a decentralised training strategy. The task hierarchy is designed prior to the training.

Ahilan and Dayan (2019) recently suggested that the initial idea of hierarchy in reinforcement learning in fact can be incorporated into MARL in order to improve the coordination between agents. There is a manager agent who controls the a finite list of sub-goals and assigns accordingly to the lower level agents.

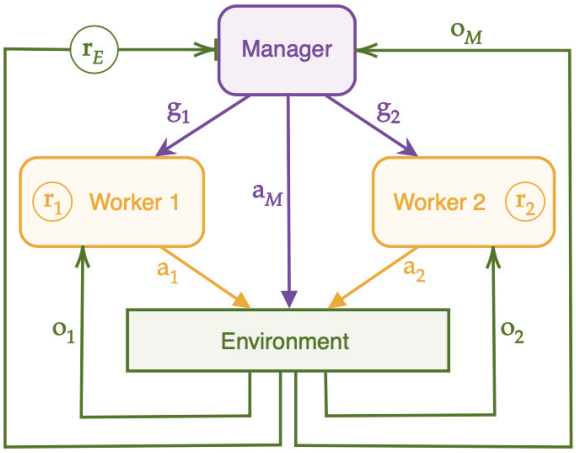


Figure 2.10 : Feudal structure in MARL

However, these are the main drawbacks of current work in integrating HRL to MARL:

- The proposed algorithms are limited to small scale problems where the number

of actions in the problem space are small. Most of the works are tested on simple grid world scenarios where agents have at most 10 primitive actions. As a result, scalability in problem space is not addressed.

- There is no sub-goals discovery mechanism. All the sub-goals or sub-tasks are predefined.

2.3 Deep neural networks

2.3.1 Multi-layer perceptron

Deep neural network is a form of parameterised approximation function which is a composition of multiple affine transformation of the inputs into more abstract representation, with element-wise non-linear function $\sigma(.)$ applied after each layer. Throughout this dissertation, we will use multi-layer perceptron (MLP) or feed forwards neural network as the base neural network model. Fundamentally, MLP is a composition function as shown in Equation 2.15 (Kipf 2020).

$$MLP_\theta = f^L \circ \sigma \circ f^{L-1} \circ \ldots \sigma \circ f^2 \circ \sigma \circ f^1 \tag{2.15}$$

Each MLP layer is a parameterised function as follows (Equation 2.16):

$$f^l(h^{l-1}; w, b) = {h^{l-1}}^T w + b \tag{2.16}$$

with w and b are the weights or the parameters of the layer. h^{l-1} is the output from the previous layer $l-1$ and $h^0 = x$ where x is the input to the network.

2.3.2 Autoencoders

In this thesis, we use a neural network architecture called Autoencoder to reduce the state dimensionality. The autoencoder extracts a latent representation of an input vector and it does so via an encoding-decoding process. The network applies an encoding function $f : \mathbb{R}^d \rightarrow \mathbb{R}^k$ to the input x (e.g. $h = f(x)$) where $k << d$. A decoding function $g : \mathbb{R}^k \rightarrow \mathbb{R}^d$ then tries to reconstruct the original input

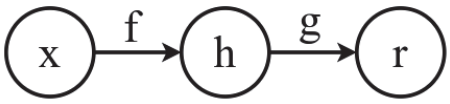


Figure 2.11 : The general architecture of an autoencoder. The input x is mapped into a latent representation h via the encoding function f. The decoding function g reconstructs h back to the original input r.

$r = g(h)$ (Goodfellow et al. 2016). Figure 2.11 illustrates the encoding-decoding process.

Both functions f and g can be implemented using different neural network architectures in deep learning literature. In this work, we use MLP as the base models for both of these functions. The weights of the MLP are trained to minimise the reconstruction error $L(x, g(f(x))$ as it tries to make x and r to be similar.

2.4 Literature Review

2.4.1 Reinforcement learning in penetration testing

Recent interests and success of deep reinforcement learning have motivated the research community to leverage this learning paradigm to developing autonomous penetration testing framework. The self-learning nature of RL makes it suitable for cyber security research to develop better attack strategy where the agent can learn attack strategy without domain knowledge or network modelling as it is the case for automated penetration testing tools.

The first research direction is to develop a useful simulation platform for cyber security researchers to apply AI methods. Many simulation platforms have been created recently and most share common interfaces for algorithms to be developed. These platforms include gym-idsgame(Hammar and Stadler 2020), CyberBattleSim(Team. 2021), NAsim(Schwartz and Kurniawati 2019a) and CybORG. Each simulated environment has different interpretation of network's state and state transition during exploit execution. Some environments such as CybORG and CyberBattleSim also present the adversarial nature of penetration testing wherein there

exist both the red team (the attacker) and the blue team (the defender), making it useful for creating a more dynamic attack and defense strategies.

The second more active research area is to adapt and advance reinforcement learning algorithm to learn penetration policy. For example, Schwartz and Kurniawati (2019a); Tran et al. (2022); Ghanem and Chen (2018) are amongst some of the researchers demonstrate the feasiblity of training an RL agent from scratch to learn penetration testing. Tran et al. (2021) proposed a hierarchical structure of agents to deal with the large action space in complex penetration testing scenarios. The current research literature leverages both value based and policy based RL algorithms which open full potential for applying DRL in advancing the development of autonomous penetration testing tools.

2.4.2 Deep reinforcement learning with large action space

Pen-testing applications share similar state-based transitions that are similar to games. The entire attacks and defenses strategy can be modeled as a Markov Decision Process with state machine diagram, state transition matrix and an action space. In many scenarios, the game can also be modelled as a partially observable MDP, where the attacker can only observes a partial view of the network configurations. This makes PT an ideal playground to train RL agents where they can be trained by interacting with the simulator or emulator (Standen et al. 2021). This approach has another advantage of not having any assumption on the domain knowledge as well as network topology and configurations.

RL and its deep variants from tabular Q-learning to deep Q-network (DQN) have been used to solve PT problems ranging from *Capture The Flag* (CTF) (Zennaro and Erdodi 2020) to research in injection vulnerabilities (Erdődi et al. 2021). Works done by Schwartz and Kurniawati (2019a) and Hu et al. (2020) have shown the feasibility of using function approximators such as deep neural networks to extract state features in large scale scenarios for DRL agents to learn action policy. In these works, PT is formulated as a partially observable markov decision process (POMDP) within which RL or DRL is used to model the environment and to learn

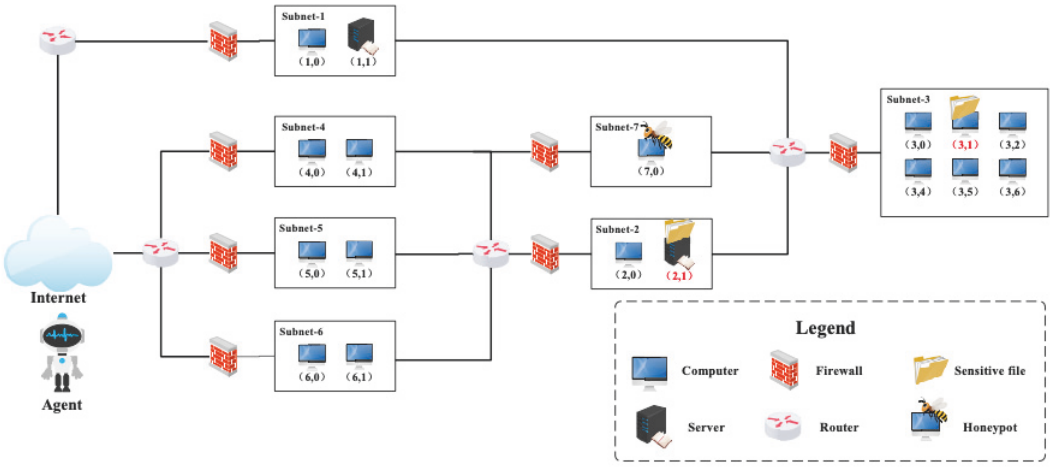


Figure 2.12 : An illustration of a typical cyber security network used in pen-testing simulators (Zhou et al. 2021).

exploit strategy. The attacking agent is presented with some level of exposed public facing hosts and subnets. The internal network connection and activities are completely hidden from the attacker's observation, as well as the defenders' behavior. Figure 2.12 describes the details of a typical cyber security network configuration.

The attacker, usually referred to as the red agent, has the initial observation of known hosts in the public facing subnet via the available routers and firewalls. The agent is also equipped with a set of actions which can be exercised on the network in order to *scan*, *ping* or performing *privilege escalation* to gain higher access to control the hosts. The complexity of the penetration testing simulator varies between platforms and at the time of developing this thesis, there is no known common benchmark for cyber security and AI researchers. For example, NASim (Schwartz and Kurniawati 2019a), which is an open-source platform for simulating network scenarios, constructs the observation vector containing the information of the hidden asset. It, however, does not describe how the hosts and subnets are connected. The state transition $P(s'|s, a)$ is also closed to being deterministic after a successful action execution. These implementation details can be helpful for the RL agents to update its policy during learning. Another open-source platform, CybORG (Standen et al. 2021), implemented a more realistic simulator wherein the *flag* is completely hidden

from the observation vector, and the state transition is stochastic even with successful action execution. Additionally, there are more entities other than hosts and subnets involved within a scenario such as *ip_ address*, *password*, *user*, ... Combinatorically, the available action space easily scales up to thousands for relatively-sized scenarios. As a result, typical DRL algorithms without specific improvements failed to learn successful attack policy in such pragmatic scenarios.

Tabular RL or deep RL (DRL) have been used as a proof-of-concept solutions to small networks or capture the flag challenges with relatively small action spaces. Zhou et al. (2021) recently proposed an enhanced dueling deep-Q network solution using multiple extensions including prioritised experience replay (Schaul et al. 2015b), noisy nets (Fortunato et al. 2017), soft Q-learning (Haarnoja et al. 2017), and intrinsic curiosity (Pathak et al. 2017a), which are all standard implementations in RL literature, targeting at improving performance at scale. Extending DRL to networks with a larger action space is still a standing challenge not only in cybersecurity setting but also in RL context (Schwartz and Kurniawati 2019a).

Inspired from previous works, we attempted to extend DRL agents in pen-tester to deal with large discrete action space. RL literature have seen notable works that focused on improving DRL for large scale action space. For example, Dulac-Arnold et al. (2015) proposed an architecture, named Wolpertinger, that leverages an actor-critic framework for reinforcement learning. This framework attempts to learn a policy within a large action space problem that has sub-linear complexity. It uses prior domain knowledge to embed the action space and clusters similar actions using the nearest-neighbour method to narrow down the most probable action selection. However, in a sparse reward environment, this method can be unstable as the gradient cannot propagate back to enhance learning of the actor network.

In another study, carried out by Zahavy et al. (2018), the use of an elimination signal was proposed to teach the agents to ignore irrelevant actions, which reduces the number of possible actions to choose from. A rule-based system calculates the action elimination signal by evaluating actions and assigning negative points to

irrelevant actions. Then, the signal is sent to the agent as part of an auxiliary reward signal to help the agent learn better. This framework is best suited to domains where a rule-based component can easily be built to embed expert knowledge and facilitate the agent's learning by reducing the size of the action set.

In Tavakoli et al. (2018) different network branches were incorporated to handle different action dimensions, while a growing action space was trained using curriculum learning to avoid overwhelming the agents (Farquhar et al. 2019). The authors of Chandak et al. (2019) focused on learning action representations. Here, the agent was able to infer similar actions using a single representation. In a similar vein, in de Wiele et al. (2020), a solution was proposed targeted at learning action distributions. All these approaches sought to find a way to reduce a large number of actions. However, they cannot be used to solve automating penetration testing problems for several reasons. First, the action space in penetration testing is entirely discrete and without continuous parameters. This is different from other areas of testing where the action space is parameterised and comprised of a few major actions that may contain different continuous parameters. Second, each action in a penetration test can have a very different effect, such as attacking hosts in different subnets or executing different methods of exploits. This is again different in nature to a large, discretised action space, where a group of actions can have spatial or temporal correlations (Tavakoli et al. 2018; de Wiele et al. 2020; Dulac-Arnold et al. 2015).

All these approaches aimed at finding a way to reduce the large number of actions. However they cannot be applied to solve automating PT for the following reasons. Firstly, the action space in PT is entirely discrete without continuous parameters. This is different with other works where the action space is parameterised and comprised of few major actions within which contain different continuous parameters. Secondly, each action in PT can have very different effects such as attacking hosts in different subnets or different method of exploits. This nature is again different with large discretised action space, where group of actions can have spatial or temporal correlation , which was addressed in (Tavakoli et al. 2018; de Wiele et al. 2020;

Dulac-Arnold et al. 2015).

To the best of our knowledge, this investigation is the first to propose a non-conventional deep reinforcement learning architecture to directly target the large action space problem in penetration-testing settings. The architecture involves an explicit action decomposition scheme based on a structure of cascaded reinforcement learning agents. Here, the agents are grouped hierarchically and each agent is trained to learn within its action subset using the same external reward signal. This is a model-free methodology where no domain knowledge is required to decompose the set of actions. Further, the architecture is proven to scale efficiently to more complex problems. The solution also takes advantage of value-based reinforcement learning to reduce the sample complexity, and, with some modifications, it can be extended to other problem domains where the action space can be flattened into a large discrete set.

2.4.3 Deep reinforcement learning with parameterised action space

One of the first attempts to solve a parameterised action space was to convert the parameter space into another flat action space and expand it into a large discrete set together with the main action type space. An example of this method was explained in the previous section, which we showed leads to a gigantic action space even with a relatively simple scenario. This naive method transforms the structured action space back into a conventional flat one(Sherstov and Stone 2005), but it also removes the advantage of reducing a large action set using hierarchical action selection. Additionally, it also destroys the semantics of the actions and their parameters. For these reasons, Hausknecht and Stone(Hausknecht and Stone 2015b) instead convert the selection of discrete action types into a continuous set, merging it into the parameter space. A deep deterministic policy gradient(Lillicrap et al. 2015) is then used to learn the final action selection. Even though this is a feasible way of training such networks, it is a hard strategy to scale to larger applications. Masson et al.(Masson et al. 2016) takes the route of alternately learning a discrete action selection and a parameter selection using an actor critic algorithm. However, this approach does not account for the dependency of the parameters on the action type

nor the number of variable parameters for each action. Wei et al.(Wei et al. 2018a) proposed an implementation very close to our architecture; however, the identifiers of the main action are passed on to the parameter network instead of the state-action values as in our proposal. In our experiments, it is clear that passing on the action identifier does not help the parameter network to learn as it needs to also learn the action representation. In this regard, the authors also highlight the instability of this training approach. Recent works by Xiong et al. (Xiong et al. 2018) and Bester et al. (Bester et al. 2019) use deep q-learning to learn both the discrete action type and a discretised continuous parameter selection. However, the parameter selection is passed onto the main action selection, which means a discrete action is learned based on the estimated parameters. This is only suitable for applications where all discrete action types share the same set of parameters.

Along the lines of applying reinforcement learning to penetration testing applications, different attempts have been made to demonstrate the feasibility of such an approach(Hu et al. 2020; Zennaro and Erdodi 2020; Ghanem and Chen 2020b). Further efforts to scale reinforcement learning algorithms for complex penetration testing scenarios have been proposed by(Tran et al. 2022) and (Schwartz and Kurniawati 2019b). However, all these studies are conditioned on a conventional flat action space to train reinforcement learning algorithms. But, as mentioned in the previous section, keeping a flat action space would be a major challenge when trying to emulate a realistic penetration testing simulator. The combinatorial nature of the complex action structure in an autonomous penetration testing application would make it impossible to use a flattened action space. Hence, an improved deep q-network approach was proposed wherein two network heads are used to learn concurrently the attack action and the host value(Zhou et al. 2021). Further, no coordination is used when training the independent network heads.

To our best knowledge, ours is the first approach to use a modified deep reinforcement learning algorithm as a way of tackling the parameterised structure of the action space in a penetration testing application. Further, our experiments demonstrate the superiority of our approach in terms of convergence and stability

as compared to previous works.

Furthermore, CybORG, the main simulator used in this work, has certain differences in the action space representation that may make it impractical to leverage previous works:

- The action space in CybORG contains a main attack action type followed by a number of parameters such as a host, a subnet, a password and session selection. Each parameter has a completely different meaning and more than one parameter can be used for each attack type.
- The selection of the parameters depends on which attack type is selected, and the dependency not only reflects the validity of the final action selection but also expresses its semantics.
- During training, the cardinality of each parameter set is variable, as new hosts and subnets are discovered; and it also differs depending on the type of parameter. This parameterised action space is shown in Figure2.14.

As a result, we developed a MARL-inspired architecture where independent agents are used to learn each dimension of an action selection. Using a MARL affords flexibility when selecting actions and parameters. A separate reinforcement learning agent is trained to output the state-action values of the action selection. These values are then passed onto the second reinforcement learning agent to condition on and learn different parameter selections. The entire multi-action learning process is differentiable thanks to the adaptation of a coordination strategy known as QMIX.

At the time of writing this thesis, we are aware of another work that also formulates the problem of parameterised action space into a multi-agent reinforcement learning where each agent manages a sub-action space (Li et al. 2021b). This is an interesting encounter as it proves our approach is valid and we would love to compare our proposal with this recent research in the future work.

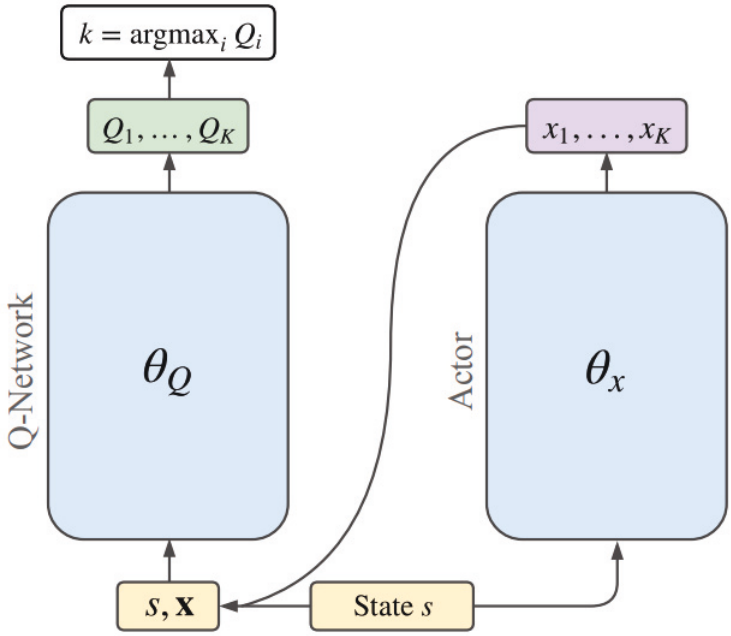


Figure 2.13 : The P-DQN algorithm by Xiong et al. (2018)

.

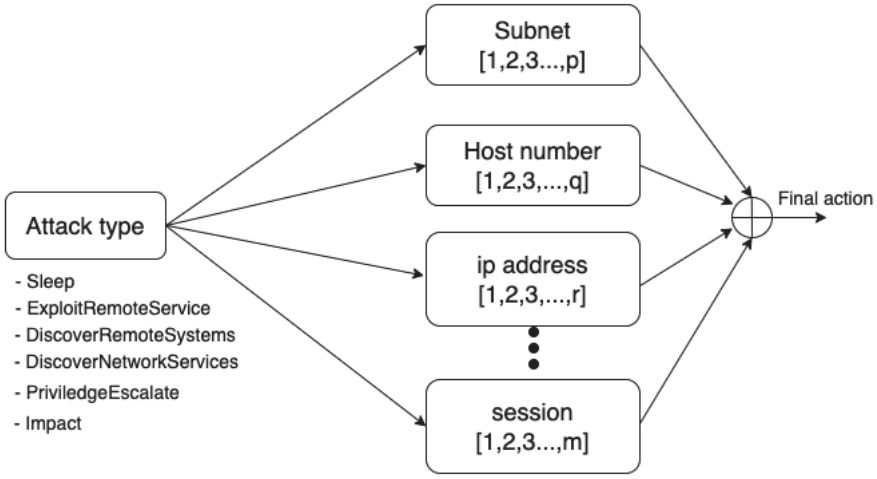


Figure 2.14 : Parameterised multinomial action space

2.4.4 State representation learning in deep reinforcement learning

State representation learning in RL

In this work we look at a reinforcement learning setting wherein an agent needs to learn a policy π which maps a state $s \in \mathcal{S}$ to a primitive action $a \in \mathcal{A}$. The agent aims to maximise an expected discounted future return $G_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'+1}$, where $\gamma \in [0, 1]$ is the discount factor and r_t is the reward returned by the environment after the agent exercised the action a_t. In value-based reinforcement learning, the model-free RL agent is trained to give a good estimate of a state-action value function parameterised by θ in the case of neural networks, e.g $Q(s, a; \theta)$. A temporal difference learning method is used to optimise the estimate of $Q(s, a; \theta)$ by minimising the loss function in Equation 2.17:

$$L(\theta) = \mathbb{E}_{(s,a,r,s' \sim \mathcal{D})}[(r + \gamma max_{a'} Q(s', a'; \theta) - Q(s, a; \theta))^2] \tag{2.17}$$

where $\mathcal{D}$ is the experience replay buffer. When the environment has large state action space and/or with sparse learning reward signal, it is challenging for the agent to update its estimate of $Q(s, a; \theta)$ (Mnih et al. 2015). This problem is coupled with the exploitation-exploration trade-off in RL which makes it hard for RL agents to collect useful transitions for learning. To facilitate the learning of RL agents, a good state representation learning needs to satisfy some of the following purposes:

- The state representation can extract critical feature of the environment dynamics into a small latent vector with dimension $k << d$ where k is the size of the latent vector and d is the original dimension of the state vector.
- A domain-agnostic state representation which can be applied in non-visual domain would be useful. As majority of current RL literature in state representation deal with continuous state information or visual information as in games or in robotics simulation. As a result, these methods do not translate directly to binary-value vector as in cyber applications.
- And it provides meaningful semantics to the state abstraction which gives an advantage for interpretability of the model.

Successor Representation

Successor representation (SR) was first introduced by Dayan (1993), which described the concept of representing the expected discounted future states encounter. In other words, SR summarises the future states into the current state representation. In a tabular setting, the representation of state s_t would be the discounted visitation count of each reachable future states starting from s_t and executing policy π. The value of state s_t can be calculated as in Equation 2.18 (Gershman 2018):

$$V(s_t) = M(s_t, s_{t+1})R(s_t) + M(s_t, s_{t+2})R(s_{t+2}) + M(s_t, s_{t+3})R(s_{t+3}) \tag{2.18}$$

where $M(s_t, .)$ is a vector representing the state s_t with each element is the discounted future visitation count of subsequent states. For generalisation, the value of state s_t is the sum of SR and its future rewards as in Equation 2.19:

$$V(s_t) = \sum_{s'} M(s, s')R(s') \tag{2.19}$$

In deep learning with neural network as the function approximator, the SR is referred to as Successor Feature (SF) and denoted as $\psi^\pi(s, s')$. Concretely $\psi^\pi(s, s')$ is calculated as an expectation as denoted in Equation 2.20:

$$\psi^\pi(s, s') = \mathbb{E}[\sum_{t=0}^{\infty} \gamma^t \mathbb{1}(s_t = s')|s_0 = s] \tag{2.20}$$

where $\mathbb{1}(.)$ is the indicator function. Temporal difference (TD) learning can be used to estimate $\psi(s)$ from sampled data as in Equation 2.21 (Machado et al. 2020):

$$\hat{\psi}(s_t, j) \leftarrow \hat{\psi}(s_t, j) + \eta(\mathbb{1}(s_t = j) + \gamma\hat{\psi}(s_{t+1}, j) - \hat{\psi}(s_t, j)) \tag{2.21}$$

where η is the step size for update and the policy notation π has been dropped to avoid cluttering the equation.

Barreto et al. (2017) extended the SR into SF in the context of non-linear function approximator (Equation 2.22).

$$\psi^{\pi}(s) = \mathbb{E}[\sum_{t=0}^{\infty} \gamma^t \phi(s_t) | s_0 = s] \tag{2.22}$$

The successor feature of state s_t can also be extended into the successor feature of state s_t and action a_t as in Equation 2.23, which is similar to the formulation of state-action value function.

$$\psi^{\pi}(s, a) = \mathbb{E}[\sum_{t=0}^{\infty} \gamma^t \phi(s_t, a_t) | s_0 = s, a_0 = a] \tag{2.23}$$

and their relationship is shown in Equation 2.24:

$$\psi^{\pi}(s) = \mathbb{E}_{a \sim \pi(s_t)}[\psi(s, a)] \tag{2.24}$$

It can be shown that the SF of a state-action pair can be used to recover the state-action value function as shown in Equation 2.25 (Barreto et al. 2017):

$$Q(s, a) = \psi(s, a)^T w \tag{2.25}$$

where w is the weight of a reward prediction function (Equation 2.26):

$$R(s, a) = R(s, a; w) = \phi(s, a)^T w \tag{2.26}$$

As a result, SF is considered to be a hybrid approach connecting model-based and model-free reinforcement learning.

Temporal abstraction with hierarchical reinforcement learning

We take advantage of learning the temporal abstraction approach proposed by Kulkarni et al. (2016), wherein the decision making process of DRL agents are separated into two levels:

- The meta-controller at the top level learns a goal driven policy $\pi^{meta}(g_t|s_t)$ mapping the current given state s_t into a selected sub-goal g_t.

- The controller at the lower level learns the primitive policy $\pi^{controller}(a_t|(s_t, g_t))$ mapping a concatenated representation of state and sub-goals into a primitive action a_t executable on the environment. In value-based RL, the controller basically learns to estimate a Universal Value Function Approximators (UVFA) (Schaul et al. 2015a) (e.g $V((s_t, g_t); \theta)$.

If the controller is able to achieve the assigned sub-goals in a given T time steps, it receives an intrinsic reward from the meta-controller (Figure 2.15).

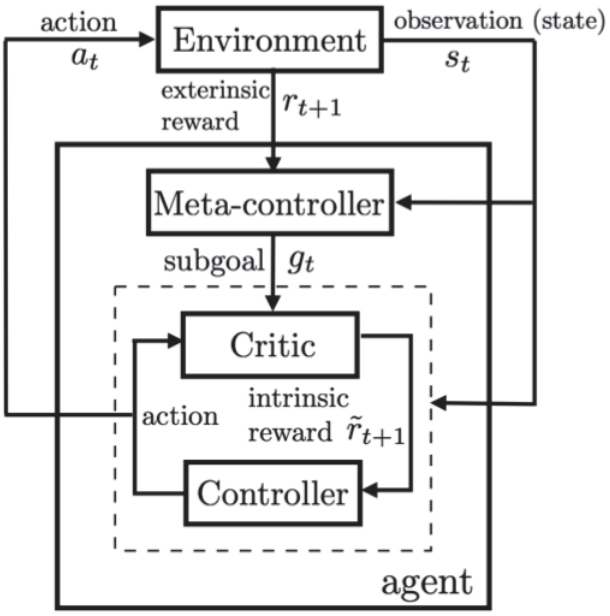


Figure 2.15 : Zoom-in version of the hierarchical meta-controller/controller framework (Rafati and Noelle 2019a)

As a result, the controller is trained to optimise the objective of maximising the cumulative discounted intrinsic reward $\tilde{r}_t$ as shown in Equation 2.27:

$$\tilde{G}_t = \sum_{t'=t}^{t+T} \gamma^{t'-t} \tilde{r}_t(g_t) \tag{2.27}$$

The controller needs to learn a state action value function by minimising the following loss function (Equation 2.28):

$$L(\theta^1) = \mathbb{E}_{(s,g,a,\tilde{r},s')\sim\mathcal{D}_1}[(\tilde{r} + \gamma max_{a'} Q^1(s', g, a'; \theta^1) - Q^1(s, g, a; \theta^1))^2] \tag{2.28}$$

The meta-controller optimises its objective with respect to the external reward r_t as shown in Equation 2.29 and its corresponding loss function (Equation 2.30):

$$G_t = \sum_{t'=t}^{t+T} \gamma^{t'-t} r_t(g_t) \tag{2.29}$$

$$L(\theta^2) = \mathbb{E}_{(s,g,r,s')\sim\mathcal{D}_2}[(r + \gamma max_{a'} Q^2(s', g; \theta^2) - Q^2(s, g; \theta^2))^2] \tag{2.30}$$

In the above equations, $Q^1(.;\theta^1)$, $Q^2(.;\theta^2)$ are the action value functions of the controller and meta-controller respectively, and $\mathcal{D}_1$ and $\mathcal{D}_2$ are the replay buffer of the controller and meta-controller respectively.

The standing challenge of using HRL is that of how to achieve the subgoals from the environment. Subgoals discovery or options learning are challenging without obtaining good feature engineering with the help of domain knowledge. Ramesh et al. (2019) attempted to learn subgoals by clustering the successor representation of the environment. The idea takes advantages of the SR wherein similar states result in similar SR and dissimilar states would have low cosine similarity between the SR vectors. It was demonstrated on a tabular maze scenario and small scale robotics simulation platform. States clustering is another line of work (Rafati and Noelle 2019a) as latent vectors are clustered into distinct regions and using rewards as an indicator of good centroids. These centroids are then translated into subgoals which the meta-controller can give to the controller. Eigenoption discovery achieves the subgoals by establishing the SF matrix and extract the top eigenvectors of this SF representation (Machado et al. 2017b,a). These early works demonstrated the success on tabular maze-base scenario or game specific setting which is not clear on how to extend them to a discrete domain.

Inspired by the previous researches, we combine the HRL approach with meta-controller/ controller framework and the SF representation to extract subgoals from

the SF vectors and use them for learning in HRL.

- Firstly we train the SF using random policy to collect state transitions and trajectory.
- These transitions are then used to train the deep encoder-decoder module and a SF learning module. The learned latent vector at the encoder's output is then the condensed representation of the state vector as well as the environment dynamics.

Chapter 3

Cascaded reinforcement learning agents for action space decomposition

Organised attacks on a computer system to test existing defences, i.e., penetration testing, have been used extensively to evaluate network security. However, penetration testing is a time-consuming process. Additionally, establishing a strategy that resembles a real cyber-attack typically requires in-depth knowledge of the cyber security domain. This paper presents a novel architecture, named deep cascaded reinforcement learning agents, or CRLA, that addresses large discrete action spaces in an autonomous penetration testing simulator, where the number of actions exponentially increases with the complexity of the designed cyber security network. Employing an algebraic action decomposition strategy, CRLA is shown to find the optimal attack policy in scenarios with large action spaces faster and more stably than a conventional deep Q-learning agent, which is commonly used as a method for applying artificial intelligence to autonomous penetration testing.

3.1 Introduction

Developing effective autonomous defences typically requires that models are trained against sophisticated attacks. Penetration testing has been used extensively to evaluate the security of ICT systems. However, this is a time-consuming process. Additionally, establishing a strategy that resembles a real cyber-attack requires in-depth

knowledge of the cyber security domain. Moreover, the currently available automatic penetration testing tools are not capable of learning new strategies to respond to simple defensive measures. So, instead of testing all possible values when hacking into a system, it is often more desirable to use an autonomous penetration-testing tool that includes artificial intelligence so as to simulate a real attacker with a real attack strategy.

Reinforcement learning is a learning paradigm in which an autonomous decision-making agent explores and exploits the dynamics of its environment and, in so doing, discovers a suitable policy for acting in that environment. This makes reinforcement learning a suitable candidate for tackling the problem of automating the penetration-testing process. Deep reinforcement learning with function approximators as neural networks have achieved multiple state-of-the-art results on a variety of platforms and with a range of applications, as demonstrated by Mnih et al. (2015) and Lillicrap et al. (2015). Value-based methods, such as some variants of deep Q-learning (DQN) algorithms (Watkins and Dayan 1992), have become the standard in many approaches thanks to their simplicity and effectiveness in learning from experienced trajectories (Wang et al. 2015; Hessel et al. 2017b; Fan et al. 2020). However, one of the limitations of applying DQN in practical applications is the complexity of the action space in different problem domains. DQN requires an evaluation for all the actions at the output of the policy network and uses a maximisation operation on the entire action space to select the best action to take in every step. This is problematic for large action spaces as multiple actions may share similar estimated values (Farquhar et al. 2019). Penetration testing is one such practical application where the action space scales exponentially with the number of host computers in different connected sub-networks. Applying conventional deep reinforcement learning to automate penetration testing would be difficult and unstable as the action space can explode to thousands of executable actions even in relatively small scenarios (Schwartz and Kurniawati 2019a).

There are studies that address the intractable problem of having large discrete action spaces. For example, Tavakoli et al. (2018) proposed an action branching ar-

chitecture where actions are grouped into sub-spaces and jointly trained by different network heads. Prior domain knowledge can also be used to embed or cluster similar actions in the action space, which effectively reduces the cardinality of distinct action representations (Dulac-Arnold et al. 2015). These investigations inspired us to develop a new action decomposition scheme for devising penetration-testing attack strategies that is efficient, versatile, and works with different network configurations. The proposed method leverages the multi-agent reinforcement learning (MARL) paradigm to decompose the action space into smaller subsets of a space where a typical reinforcement learning agent is capable of learning.

As such, in this chapter, we propose a novel architecture to directly address the large discrete action spaces of autonomous penetration testers. Using an adapted binary segment tree data structure, the action space is algebraically decomposed into sub-branches, each of which is an order of magnitude smaller. A combined learning mechanism is then used to cooperatively train the agents in the cascaded structure. The proposed algorithms are tested on CybORG, which is an autonomous cyber security simulator platform developed by Standen et al. (2021) designed to empirically validate the learning and convergence properties of an architecture. CRLA's performance demonstrates better and more stable learning curves in comparison to a conventional DQN at the same network size.

This chapter is organised as follows: Section 3.2 provides a reinforcement learning formulation to the context of penetration testing. Section 3.3 explains the details of the proposed algorithm, and Section 3.4 describes the experiments undertaken for the paper. Finally, Section 3.5 presents the results and discussion on the current investigation as well as considering future research directions.

3.2 Background

The problem to be considered in this paper is a discrete-time reinforcement learning task modelled by a Markov decision process or MDP formalised by a tuple $\langle \mathcal{S}, \mathcal{U}, \mathcal{P}, \mathcal{R}, \gamma \rangle$. Unlike previous investigations, we do not wish to formulate the penetration testing as a POMDP because we want to clearly demonstrate the ef-

fects of the proposed architecture without compounding factors. No previous studies have used any POMDP-specific solutions, and, further, our method can still be applied as a feature to a POMDP solution. At each time step t, the agent receives a state observation $s_t \in \mathcal{S}$ from the environment. In our penetration testing setting, this represents the observable part of the compromised network. To interact with the environment, the agent executes an action $u_t \in \mathcal{U}$, after which it receives another state s_{t+1} according to the state transition probability function $\mathcal{P} : \mathcal{S} \times \mathcal{U} \times \mathcal{S} \to \mathbb{R}$, which returns a probability distribution of the next state s_{t+1} given the current state s_t and action u_t, and a reward signal r_t given by $\mathcal{R} : \mathcal{S} \times \mathcal{U} \to \mathbb{R}$. The agent's aim is to maximise its utility, defined as the total sum of all rewards gained in an episode $R_t = \sum_{\tau=t}^{\infty} \gamma^{\tau-t} r_\tau$, where $\gamma \in [0, 1]$ is the discount factor used to determine the importance of long-term rewards.

In MARL, the action space $\mathcal{U}$ is constructed from the action space $\mathcal{A}^i$ of each individual agent. Normally u is a vector comprised of all the individual actions of k agents $u = (a^1, a^2, a^3, ..., a^k)$ (Rashid et al. 2018). In this paper, u will be used to refer to the final action which will be exercised on the environment and a^i is the action generated by the agent i, which is dubbed as an action component of u.

One common approach in deep reinforcement learning is to use a value-based method, such as deep q-learning or its variants, to learn a (near) optimal policy $\pi : \mathcal{S} \to \mathcal{A}$ which maps the observed states or observations to actions. The state-action value function, as defined below, tells each agent i the expected utility or discounted future return of taking the action a_t^i in the state s_t:

$$\begin{aligned} Q^{\pi,i}(s, a^i) &= \mathbb{E}[R_t | s_t = s, a_t = a^i] \\ &= \mathbb{E}_{s'}[r + \gamma \, \mathbb{E}_{a^{i'} \sim \pi^i(s')}[Q^{\pi,i}(s', a^{i'})] | s_t = s, a_t = a^i] \end{aligned} \tag{3.1}$$

By recursively solving Equation (3.1) for all state action pairs, and taking the action with the highest Q-value in the next state, the optimal action value function, which is defined as $Q^*(s, a) = \max_\pi Q_\pi(s, a)$, can be derived. This optimal solution is shown in the Equation (3.2):

$$Q^{i*}(s, a^i) = \mathbb{E}_{s'}[r + \gamma \max_{a^{i'}} Q^{i*}(s', a^{i'}) | s, a^i] \tag{3.2}$$

In non-trivial problems where there are large numbers of states and actions, $Q(s, a)$ is often approximated by parameterised functions, such as neural networks (e.g., $Q(s, a; \theta)$), in which similar states are estimated with similar utility.

Generally, DQNs or reinforcement learning algorithms are well-known for becoming unstable as the action space becomes larger, for example, of the order of hundreds or thousands of actions (de Wiele et al. 2020). As a result, the performance of these algorithms degrades as the action space increases (Dulac-Arnold et al. 2015). With reinforcement learning, the action space will have a different format and representation depending on the type of problem being tackled. Typically, these action spaces can be classified into two main categories: continuous and discrete. In a discrete action problem, the action space is most commonly constructed as a flat action space with each primitive action identified as one integer. Alternatively, the action space might be parameterised into a hierarchically-structured action space with a few main types of actions and sub-actions underneath (Masson and Konidaris 2015). It is harder, however, to learn in a parameterised action space. This is because the agent needs to learn two Q-functions or policies—one to choose the main action and another to choose the sub-action or the parameters of each action. On the other hand, converting a hierarchically structured action space into a flat representation could result in a substantially larger sized action space, which also hinders learning with DQN agents. In this paper, we focus on a large and discrete flat action space. Additionally, we propose a multi-agent learning framework where the agents are grouped into a hierarchical structure.

In a flat and discrete setting, the action space is encoded into integer identifiers ranging from LOW_INT to MAX_INT—for instance, from 0 to 999 in an action space of 1000 actions. Each number is translated into a valid primitive action that is executable by the simulator. In our investigation, we do not assume any prior knowledge of the semantics or relationships amongst these individual actions. The main idea of the proposed solution is to decompose the overall action space $\mathcal{U}$ into smaller sets of integers $\mathcal{A}^1, \mathcal{A}^2, \ldots, \mathcal{A}^L$, where $|\mathcal{A}^i| << |\mathcal{U}|$ $(\forall i \in [1, L])$. A primitive action u_t in $\mathcal{U}$ can be formed as a function over the action components ${a_t}^i$ in those

smaller sets: $u_t = f(a_t^1, a_t^2, \ldots, a_t^L)$, with ${a_t}^i \in \mathcal{A}^i$. Intuitively, the integer identifiers of the final actions, which can be large in value, are built up algebraically using smaller integer values. The growing numerical representation of the action space is illustrated in Figure 3.1. In the next section, we explain the intuition behind this architecture in detail, along with how to define the function as well as the smaller sets of action identifiers.

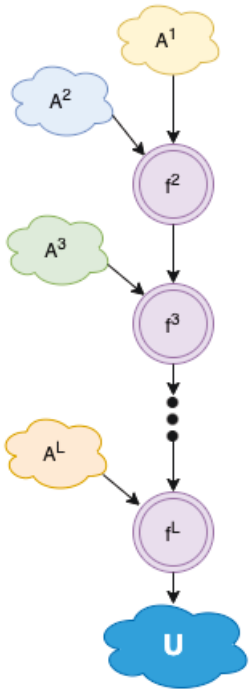


Figure 3.1 : Action space composition. The final action space $\mathcal{U}$ is constructed by chaining multiple action component's subspaces $\mathcal{A}^i$.

3.3 Methodology

3.3.1 Algebraic Action Decomposition Scheme

The main contribution of this paper is to propose an action decomposition strategy to reduce large discrete action spaces into manageable subsets for deep reinforcement learning. Our solution also includes a cooperative training algorithm that stabilises the learning process and supports better convergence properties.

As mentioned in the previous section, a large action space U is decomposed into L action component subspaces. However, rather than using different neural network heads to manage these subsets, as proposed by Tavakoli et al. (2018), separate DQN agents are used instead. The benefits of this approach are highlighted in the next section.

The idea of decomposing a large action space into smaller action component sets is similar to having a tree structure with multiple layers of action selections. Each node in the tree acts like an action component selection node in that it picks a corresponding component from one of its child nodes. Each node has its own range of identifiers. In order to reach a leaf node with a specific action value, the parent node needs to pick the right children to reach a specified leaf. Figure 3.2, which is an adaptation from a structured parameterised action space in (Fan et al. 2019a), illustrates the idea of having different action selection layers. Inspired from a segment tree data structure, each non-leaf node in the tree contains a range of identifiers that are reachable from that node. All nodes in the tree share the same branching factor, which is considered to be a hyper-parameter of the model. Heuristics are used to calculate the total number of nodes and the number of levels, as shown in Equation (3.3), where b is the branching factor. The levels are constructed by filling in the nodes from top to bottom, one level at a time, noting that the final tree should be a complete tree.

$$|T| = \log_b(|\mathcal{U}|) \tag{3.3}$$

Instead of having a different number of agents in each level to represent ranges of integer values, a linear algebraic function shifts the action component identifier values into appropriate ranges.

$$a_{out}^{i+1} = f(a_{out}^{i}, a^{i+1}) = a_{out}^{i} \times \beta^{i+1} + a^{i+1} \tag{3.4}$$

In Equation (3.4), the coefficient β^{i+1} represents the number of action component selection nodes or agents at the level $(i+1)^{th}$. The value for β^{i+1} shifts the integer

values of the action components to particular output ranges. This results in having only one agent per level, as shown in Figure 3.3. This action decomposition scheme shows that, instead of having a single reinforcement learning agent explore and learn an entire action space of $|\mathcal{U}|$, which can be extremely large, a few smaller reinforcement learning agents can be used to explore and learn in a much smaller version of the action component subspace. The outputs of these smaller agents are then chained together to produce the final action. The complexity of having multiple agents on different levels is similar to traversing a tree in computer science. The number of levels L is of the order of $L \approx \log_b |\mathcal{U}|$. As a result, only a handful of agents are required to construct an action space even if there are millions of possible actions.

To understand the intuition behind the proposed action composition approach, let us structure those small action component sets into a hierarchy, as shown in Figure 3.2. Each circle in the illustration represents an agent and the arrows are the possible action components. The three smaller sets of action components, $\mathcal{A}^1$, $\mathcal{A}^2$ and $\mathcal{A}^3$, are chosen and, for simplicity, each has the same action size of 3. The green highlighted path represents the sequence of action components needed to produce the primitive action 21 which is executable by the environment. The branching factor b and the number of levels L are 3 in this example. There are two things worth noting here: First, to reconstruct the original action space $|\mathcal{U}| = 27$, each agent in the hierarchy only needs a small action size of 3. Second, as illustrated in the figure, we need $1 + 3 + 9 = 13$ agents in total to fully specify the 27 actions. However, we do not have to implement all 13 agents. In fact, we only need three agents, one for each level of the action component selection tree.

If the first-level agent picks the action component number 2, and the second-level agent picks the action component number 1, the actual action component identifier created after the second level is $2 * 3 + 1 = 7$, where 3 is the size of the action space of each agent in the second level. Similarly, the final primitive action identifier is $7 * 3 + 0 = 21$, where 0 is the action component taken by the corresponding agent at the third level. To generalise this into a compact formula, let us consider the action

component signals a^i and a^{i+1} output by the agents at levels i and $i+1$, where $0 <= a^i <= |\mathcal{A}^i| - 1$ and $0 <= a^{i+1} <= |\mathcal{A}^{i+1}| - 1$. The action component identifier constructed up to level $i+1$ is $a^i \times |\mathcal{A}^{i+1}| + a^{i+1}$. The executable primitive action is $u = a^{L-1} \times |\mathcal{A}^L| + a^L$, where L is the number of levels in the action hierarchy.

The design of the small individual action component sets $\mathcal{A}^1, \ldots, \mathcal{A}^L$, as well as selecting the value of L, have to be performed manually as no domain knowledge is incorporated into structuring these action component sets. One rule of thumb is to limit the size of each set to be relatively small, say, 10 to 15 actions. The value of L is then selected accordingly to reconstruct the original action set $\mathcal{U}$. Each agent in the hierarchy can be implemented using any conventional reinforcement learning algorithm, for example, Dueling DQN (Wang et al. 2015), which is the core reinforcement learning algorithm we have used in this study. This is our Cascaded Reinforcement Learning Agent architecture, or CRLA for short. An operational diagram of the CRLA architecture is depicted in Figure 3.3.

Agents in the hierarchy can be trained in parallel and independently by optimising their own loss functions (Equation (3.5)). This involves sampling a batch size of p transitions from the replay buffer, where θ^i are the parameters of the agent i's network and θ^{-i} are the parameters of the target network. The latter parameters are used to stabilise the training, as mentioned in the original DQN paper (Mnih et al. 2015).

$$L^i(\theta^i) = \sum_{j=1}^{p} [(y_j^{i,DDQN} - Q^i(s, a^i; \theta^i))^2] \tag{3.5}$$

where

$$y^{i,DDQN} = r_t + \gamma \max_{a'^i} Q(s', a'^i; \theta^{-i}) \tag{3.6}$$

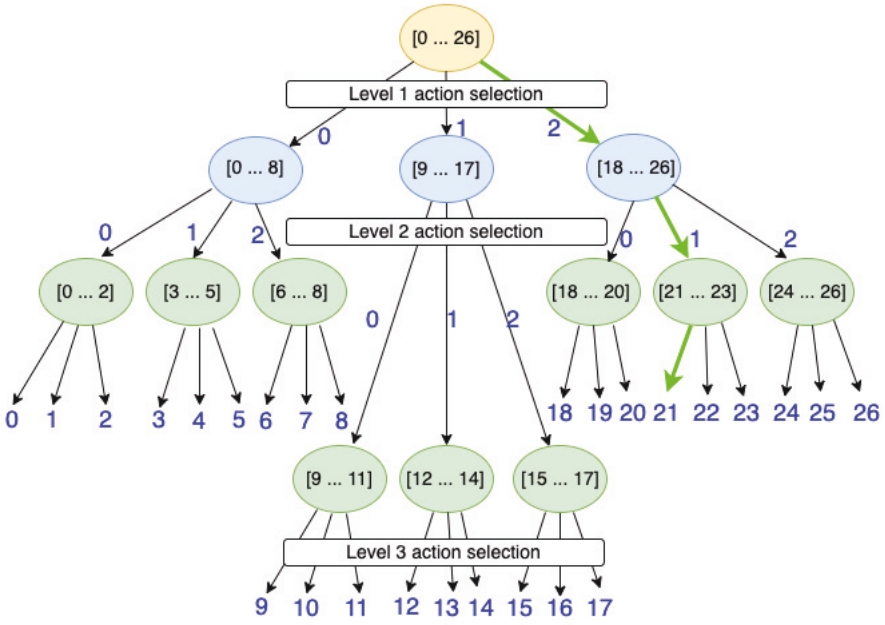


Figure 3.2 : An illustration of a tree-based structure for hierarchical action selection. The primitive action identifiers are located on the leaf nodes. Each internal node contains the action range of its children.

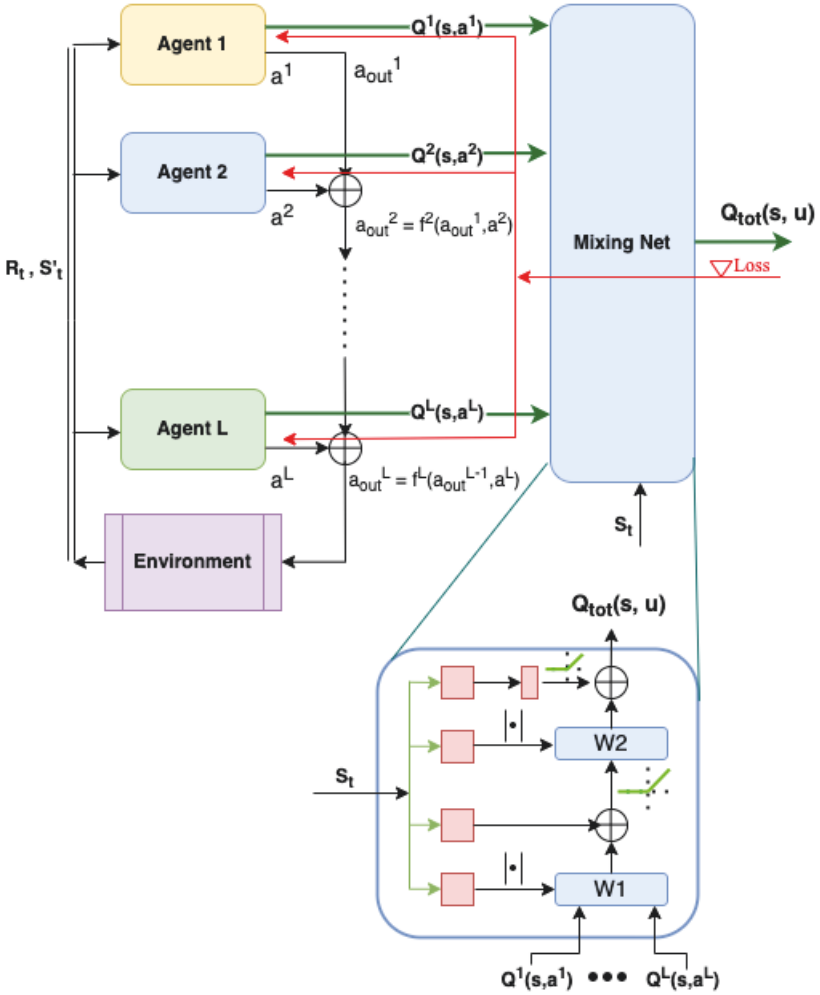


Figure 3.3 : The operational diagram of the proposed CRLA architecture. DRL agents are grouped into a cascaded structure of L levels. The state-action values of all agents are fed into a MixingNet implemented as a hyper-network (at bottom of the figure) to optimise the agents altogether. The paths of the gradient are shown as red arrows.

3.3.2 Cooperative Multi-Agent Training

The multi-agent training approach discussed in the previous section resembles the popular independent Q-learning (IQL) method (Tan 1993a) in which independent agents are trained simultaneously in the same environment. Despite its simplicity, IQL performs surprisingly well in practice and is usually a good benchmark against more advanced MARL algorithms (Leibo et al. 2017). The major drawback of IQL

is that it does not consider non-stationarity in the policies of individual agents, nor do the agents take each other's policies or actions into account when updating their own policy. As a result, changes in observation are not the direct result of each agent's behaviour, and, hence, there is no convergence guarantee even with infinite exploration.

To provide a sound theoretical guarantee of convergence with CRLA, we adapted a well-known multi-agent training network called QMIX (Rashid et al. 2018). QMIX trains an individual agent's network properly while taking the changing policies of the other agents into consideration. QMIX is shown in the right half of Figure 3.3 under the name Mixing Net. Generally, QMIX enables the training of a joint action value function by aggregating all the individual functions and optimising a combined action value function $Q_{tot}(s, u)$ with $u = f(a^1, a^2, \ldots, a^L)$.

We had two reasons for choosing QMIX. First, we needed a coordination mechanism that would allow individual agents to account for changes to its own update as a result of another agent's policy. QMIX provides an end-to-end learning scheme that takes in all the values of individual action components (e.g., $Q^1(s, a^1), \ldots, Q^L(s, a^L)$), plus the state information, in order to learn an optimal $Q^*_{tot}(s, u)$. It does this by minimising the following loss function:

$$L(\theta) = \sum_{j=1}^{p} [(y_j^{tot} - Q_{tot}(s_t, u; \theta)^2] \tag{3.7}$$

and $y^{tot} = r_t + \gamma \max_{u'} Q(s', u'; \theta^-)$

Different functions and network architectures can then be used to construct Q_{tot} (Sunehag et al. 2017b). We tried to construct Q_{tot} through a summation function and a feed-forward neural network but, eventually, we selected a two-layered hyper-network to output the weight of the Mixing Net as this yielded a better result experimentally (Equation (3.8)):

$$Q_{tot}(s_t, u; \theta) = g(Q^1(s, a^1), \ldots, Q^L(s, a^L); \theta_{mixer}) \tag{3.8}$$

This hyper-network was proposed by Ha et al. (2016) and used in Rashid et al. (2018). It has the advantages of using a small number of parameters $W1$ and $W2$

to output the weights for a larger network, in this case the Mixing Net. As a result, it helps to avoid a linear time complexity when scaling up to more agents or inputs. In previous studies on QMIX, the Mixing Net takes in a set of state-action values from homogenous agents as well as the environment's true states. This is different from our setting since each agent in the hierarchy can have a variable number of action components and we do not have access to the environment states. With respect to implementation, we had to concatenate h time steps of observations as the inputs to the Mixing Net. We also leveraged the vectorised computing feature in Pytorch (Paszke et al. 2017) to efficiently adapt to the heterogenous input values. During back-propagation, the learning signal is distributed to each agent, which means they can consider other agents' policies when updating their own parameters to achieve a global optimum. This behaviour would not be possible without the use of this coordinating network.

The second reason for using QMIX is that its parameters ($W1$ and $W2$ in Figure 3.3) are learned using a two-layer hyper-network with a non-negative activation function. This extra function is used to constrain the parameters of the Mixing Net to be positive, allowing QMIX to represent any factorisable and non-linear monotonic function $\frac{\partial Q_{tot}}{\partial Q^i} >= 0$ for all agents i. This results in an equality between taking the *argmax* for Q_{tot} and taking the *argmax* for each individual Q^i (Equation 3.9):

$$\underset{\mathbf{u}}{\operatorname{argmax}}\, Q_{tot}(\tau, \mathbf{u}) = \begin{pmatrix} \operatorname{argmax}_{a^1} Q_1(\tau, a^1) \\ \vdots \\ \operatorname{argmax}_{a^n} Q_n(\tau, a^n) \end{pmatrix} \tag{3.9}$$

In summary, if all agents learn their own optimal policy cooperatively, the entire architecture can derive the optimal global policy.

3.3.3 CRLA Implementation

Even though CRLA is inspired by a MARL framework, there are distinct features that help CRLA learn efficiently during training. First, all agents share the same state vector and reward signal. The only difference between the agents is the action

it selects. Consequently, the multi-agent training and action selection can be configured to run in parallel instead of serially. All agents can also share a replay buffer containing the tuple (s, u, r, s', d), where the action u is a vector comprising all the individual action components a^i.

With an appropriate configuration for the multi-agent DQN training, and the use of a vectorisable programming framework, such as Pytorch (Paszke et al. 2017), all agents can be trained and used for inference in parallel. Moreover, the training is optimised to be faster than for a serial multi-agent approach. In addition, since the tuple of information is mostly the same for all agents (except for the outputted action component of each agent, as shown in Figure 3.4), this is also true for the use of the replay buffer, where a single replay buffer is used for all agents.

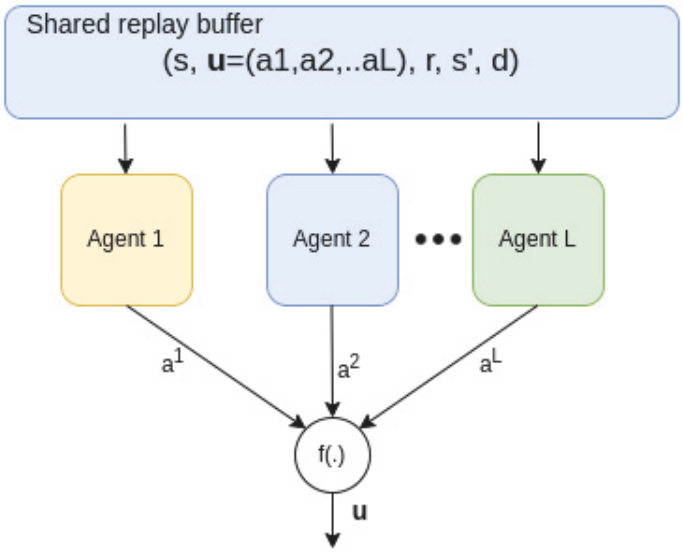


Figure 3.4 : Parallel training and execution. All agents share the replay buffer from which experiences can be sampled in batches and used for training in parallel.

3.3.4 Algorithms Pseudocode

The pseudocode for the main CRLA algorithm is given in Algorithm 1 and the sub-routine for initialising the agents given the environment action space $\mathcal{U}$ and a branching factor b is provided in Algorithm 2.

Algorithm 1 CRLA—Cascaded Reinforcement Learning Agents

Initialise a shared replay buffer $\mathcal{D}$
Initialise a list of agent $\mathbf{A}$ = initialise_agents(env)
Initialise QMIXER(θ) and QMIXER_target(θ)
for i_episode = 0 ... MAX_EPISODES **do**
    Initialise an empty set of compromised hosts $\mathbf{H} = \emptyset$
    Reset the environment S_0 = env.reset()
    $s = S_0$
    for step = 0 ... MAX_STEPS **do**
        for agent in agent_list **do**
            $a^i = \epsilon$_greedy(s, ϵ^i)
        end for
        Construct final primitive action: $\mathbf{u} = f(a^0, a^1, a^2, ..., a^k)$
        Collect (s, u, r, s', d) by executing action $\mathbf{u}$ on the environment
        Perform reward shaping and update $\mathbf{H}$ if needed:
            r = reward_function(state, $\mathbf{H}$, r)
        Store $(s, \mathbf{u}, r, s', d)$ into $\mathcal{D}$
        Sample a batch of $(s, \mathbf{u}, r, s', d)$ from $\mathcal{D}$ for learning
        Compute q-value for each agent $\mathbf{q}^i(s, a^i)$
        Compute target q-value for each agent $\mathbf{q}^{target-i}(s', a'^i)$
        $Q(s, \mathbf{u})$ = concat($\mathbf{q}^i(s, a^i)$) from all agents
        $Q^{target}(s, \mathbf{u})$ = concat($\mathbf{q}^{target-i}(s', ai^i)$) from all agents
        Compute $\mathbf{Q}_{tot}(s, \mathbf{u})$ = QMIXER($Q(s, \mathbf{u})$, s)
        Compute $\mathbf{Q}_{tot}^{target}(s, \mathbf{u})$ = QMIXER($Q^{target}(s, \mathbf{u})$, s)
        Compute TD target $y^{tot} = r + \gamma \mathbf{Q}_{tot}^{target}(s, \mathbf{u})$
        Perform SGD to minimise $L(\theta) = MSE(\mathbf{Q}_{tot}(s, \mathbf{u}), y^{tot})$
    end for
end for

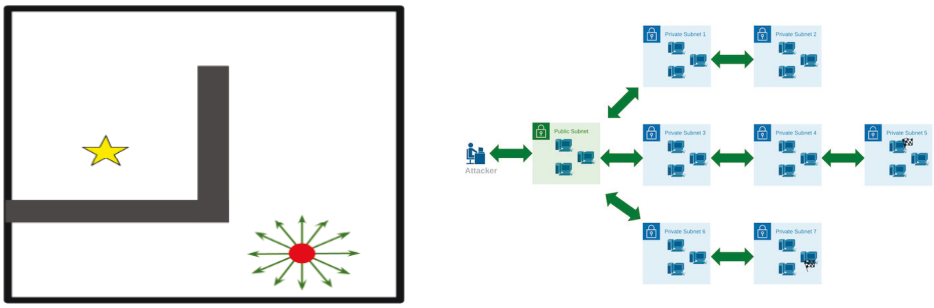
Algorithm 2 Initialise the agents

```
def initialise_agents(env: Environment, b: branching factor)
begin
|U| = env.action_space
|T| = log_b(|U|)
Build a complete tree with branching factor b
Initialise a Dueling DQN agent for each action selection level
Return agent_list
end
```

3.4 Experiments

3.4.1 Toy Maze scenario

As a proof of concept to verify the performance of the proposed algorithm, we set up an experiment with a toy-maze scenario. An agent, marked as the red dot in Figure 3.5a, tries to control some actuators to move itself toward the target—the yellow star. The agent can essentially move continuously in any direction. However, the actuators' output is discretised into a variable number of action outputs to make this a discrete action environment. The size of the action set is $|\mathcal{U}| = 2^n$, where n is the number of actuators. These are represented by green arrows pointing in the direction of movement displacement. The agent receives a small negative reward for every step in the maze and, if it can get to the target within 150 steps, it receives a reward of 100. The environment was adapted from Chandak et al. (2019) to create different testing scenarios with a variable number of actions. The fact that the agent does not get any positive reward unless it reaches the target within the pre-specified number of steps makes this a challenging scenario with sparse rewards.



(a) Maze scenario

(b) An example of 24 hosts scenario

Figure 3.5 : Two simulated scenarios. (**a**) A toy-maze-based scenario. (**b**) A CybORG scenario of 24 hosts.

3.4.2 The CybORG Simulator

To validate the performance of CRLA in a more practical scenario, we used an autonomous penetration testing simulator as our testbed. CybORG provides an experimental environment for conducting AI research in a cyber security context. The testbed is designed to enable an autonomous agent to conduct a penetration test against a network. Apart from being open source and having the appropriate OpenAI gym interface for applying different deep reinforcement learning algorithms, this environment represents a real application where we are faced with different aspects of complexity. The sparsity of reward signals, a large discrete action space, and a discrete state representation mean we cannot rely on any computer vision techniques to facilitate learning. Details of CybORG's complexity and how we executed the experiments follow.

The simulation provides an interface for the agent to interact with the network. A vector based on the observed state of the network is also provided to the agent. Through this interface, the agent performs actions that reveal information about the simulated network and they successfully penetrate the network by capturing flags on the hosts within it.

The simulation can implement various scenarios. These scenarios detail the topology of the target network, the locations of the flags, and specific information

about the hosts, such as the operating system and services the host is running. The target network features a series of subnets that contain hosts. The hosts in a subnet may only act upon hosts in the same subnet or adjacent subnets. An illustrative representation of a possible scenario configuration with 24 hosts and one attacker agent is presented in Figure 3.5b. In this example scenario, there are two hidden flags: one on a host in subnet 5 and one in subnet 7. There are three hosts in each of the subnets. The scenarios can differ in terms of the number of hidden flags and the locations of the hosts that contain the flags, along with the number of hosts. This determines the complexity of the state and action spaces. In the 24-host example, 550 actions are available to an attacking agent. The simulation includes three action types: host-to-host actions, host-to-subnet actions, and on-host actions. Each action may be performed on a host, and each host-to-host action or host-to-subnet action may target another host or subnet, respectively.

In each scenario, p hosts are grouped into q subnets. If a simulation supports m types of host-to-host actions, n types of host-to-subnet actions, and o types of on-host actions, the action space for the host-to-host actions is $m \times p \times (p-1)$, the action space for the host-to-subnet actions is $n \times p \times q$, and the action space for the on-host actions is $o \times p$. This leads to a large potential action space, even in a small network.

The success of the actions depends on both the state of the simulation and the probability of success. The state of the simulation consists of various types of information, such as the operating system on each host and the services that are listening on each host. The agent sees an unknown value for each value it does not know. As the agent takes actions in the simulation, its awareness of the true state of the simulation increases. An action fails if the agent does not have remote access to a target host or subnet. However, an action also has a probability of failing even if the agent has access to the target. This probability reflects the reality that attempts to exploit vulnerabilities are seldom, if ever, 100% reliable.

Our simulation was deliberately configured to pose a highly sparse environment. Agents began the game with no knowledge of the flags' locations, so finding the hosts

with the flags or locating the hidden assets was completely random. A large positive reward was given for capturing each hidden flag, while a small positive reward was given for every host that was successfully attacked. The flags were hidden from the state space; hence, the agent could not determine the location of the flags by observing the state vector. This simulation differs significantly from the typical reinforcement learning environments in which either the objective or the target is visible to the agent.

3.4.3 Neural Network Architecture

A diagram of the neural network's architecture is shown in Appendix A.1 with a comparison between a single-agent DDQN and the proposed CLRA. Given a scenario with four agents, the size of each layer in CLRA is one-fourth of the corresponding layer in the single-agent architecture. The sizes were selected after a fine-tuning process wherein the networks were able to converge stably using the minimum number of neurons. The action space of each DDQN agent in CRLA was varied from 8 to 12. As a result, there were at most four DDQN agents created for scenarios of up to 5000 actions. Due to the running complexity of the CybORG simulator, the algorithm's performance was tested with scenarios of up to 100 hosts and a maximum action space of 4646 actions. All the experiments were conducted on a single RTX6000 GPU.

Table 3.1 displays the values for the hyper-parameters used in the proposed algorithm.

3.5 Results

3.5.1 Maze

Figure 3.6 presents a performance comparison of policy training between a single-agent DDQN and the proposed CRLA architecture. The tested scenario had an action size of 4096 with $n = 12$ actuators. The results were reproduced over multiple random seeds with the shaded regions showing the variances between runs. The left panel shows the cumulative scores of the agent as the training proceeded, while

Table 3.1 : Hyper-parameter settings

Hyper-parameter	Value
Replay buffer size	1000000
Network size	256
Decay rate	0.998
Discount factor	0.99
QMIX hyper-net 1	32
QMIX hyper-net 2	32
Learning rate	0.0001
Learning frequency	20
GRU hidden dim	128

the right panel shows the number of steps required to reach the goal. Ideally, the cumulative scores would increase towards a maximum reward at the end of the training and the number of steps would reduce to a stable value.

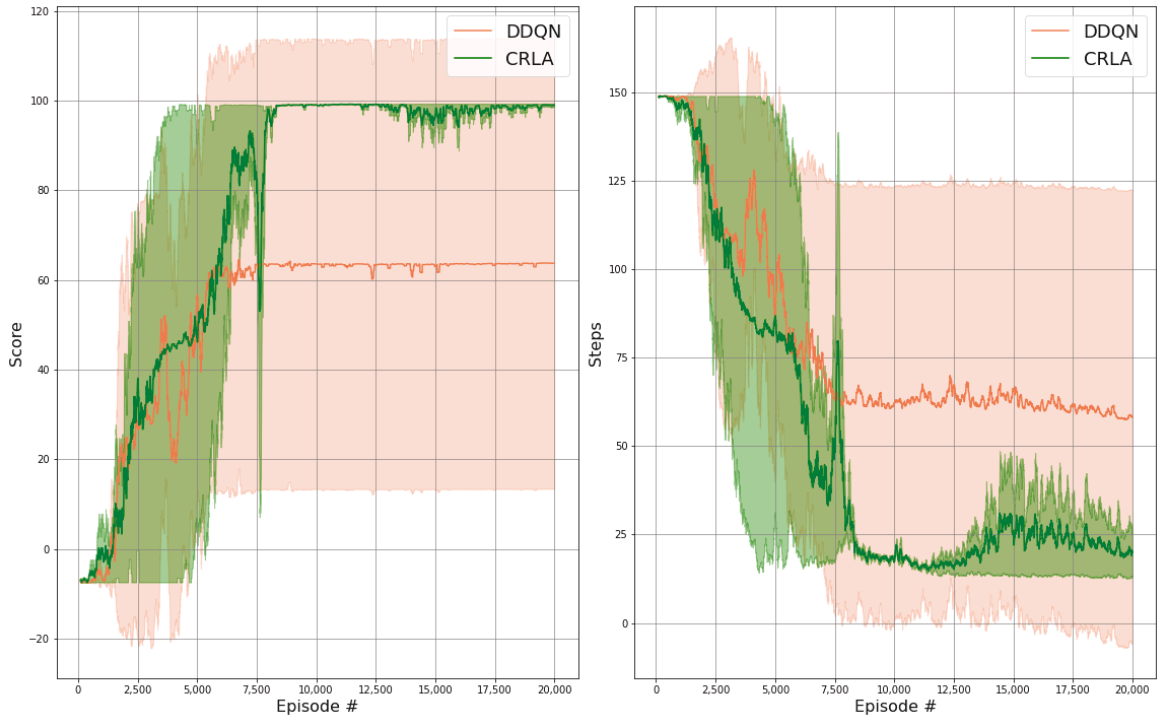


Figure 3.6 : Results of CRLA and single-agent DDQN on a maze scenario with 4096 actions. Left panel: the cumulative scores throughout the training. Right panel: the required number of steps to capture the flag throughout the training.

3.5.2 CybORG

We tested the proposed algorithm in multiple CybORG scenarios with different numbers of hosts to see how well the CRLA learned given action spaces of increasing size. Each performance in Figure 3.7 was repeated five times with different random seeds to ensure reproducibility. We examined two indicators to verify performance: the maximum score attainable by the agents (shown in the left panel) and the number of steps to capture the flag (shown in the right panel). For each scenario, the maximum score the agent could receive was approximately 20 points (minus some small negative rewards for invalid actions taken to reach the assets).

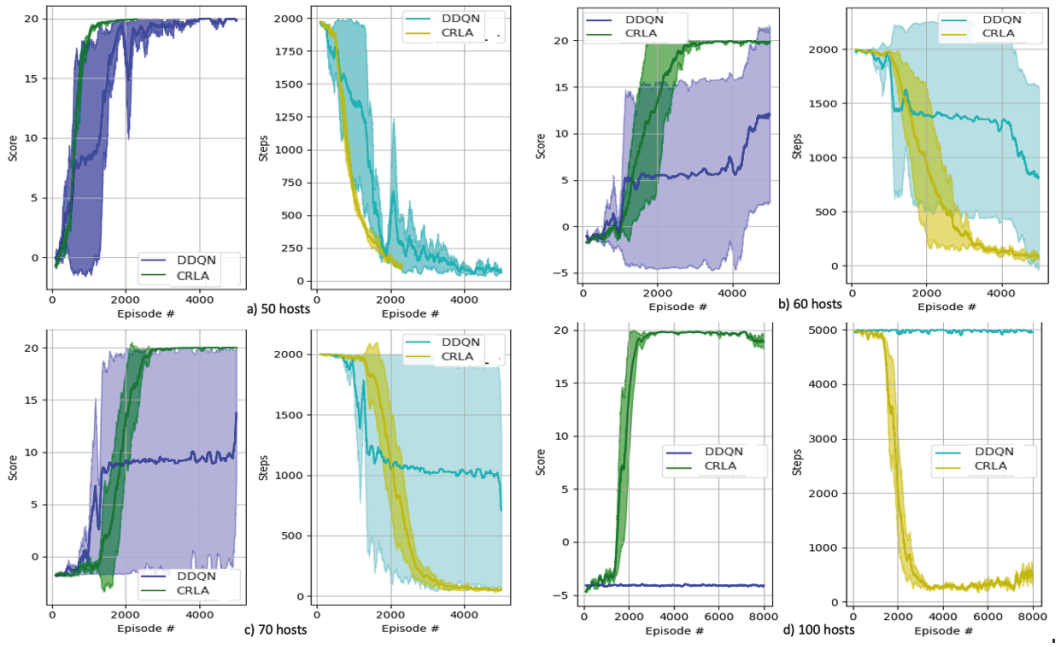


Figure 3.7 : Results of CRLA and single DDQN on different CybORG scenarios. Sub-figures from left to right, top to bottom: (**a**) 50-hosts scenario, (**b**) 60-hosts scenario, (**c**) 70-hosts scenario, (**d**) 100-hosts scenario.

We tested the CRLA algorithm in a variety of scenarios with different configurations of hosts and action spaces, as shown in Table 3.2. In terms of scenario complexity, we varied the number of hosts from 6 to 100 and increased the action space from 49 to 4646 actions. However, we only added two more agents to train. In all the scenarios tested, CRLA's convergence results were either similar to or superior to the DDQN agent, with the dependent factors being the complexity of the scenario and the size of the action space.

Figure 3.7 presents the algorithms' performance on four scenarios where the complexity was significant enough to showcase the superiority of the algorithm while not being too computationally expensive for repeated training. In scenarios where both DDQN and CRLA were able to learn the optimal policy, CRLA showed faster and more stable convergence than DDQN. DDQN's performance on scenarios with 60 and 70 hosts was unstable as it only successfully learned the optimal policy in one out of four runs. In contrast, CRLA held up its performance on the scenario

with 100 hosts where DDQN failed to achieve any progress during training.

For the 100 hosts scenario, DDQN was not able to explore the action space at all, nor could it learn the policy, while CRLA took approximately 4000 episodes to grasp the scenario and begin to converge on the optimal policy. Further, the policy learned by CRLA in each of the tested scenarios was optimal in terms of having a minimum number of actions taken.

Table 3.2 : Configurations of tested scenarios.

Hosts	**State Space**	**Action Space**	**Number of Agents**
6	39	49	2
9	55	120	2
12	71	182	2
18	217	342	2
24	285	550	2
50	573	1326	3
60	685	1830	4
70	797	2414	4
100	1133	4646	4

This simulation serves as the first demonstration of applying deep reinforcement learning to an automatic penetration-testing scenario with such a large action space (Ghanem and Chen 2020b).

3.5.3 Cooperative Learning with QMIX

We verified CRLA's learning, with and without the use of QMIX, by testing the two variants on the 60-host scenario. It can be seen from Figure 3.8 that CRLA with QMIX demonstrated slightly better stability with a smaller variance between runs in comparison to the independent CRLA without QMIX. The cooperative training us-

ing QMIX gives a better theoretical guarantee of convergence. However, implementing this empirically requires extra hyper-parameter tuning for the hyper-network.

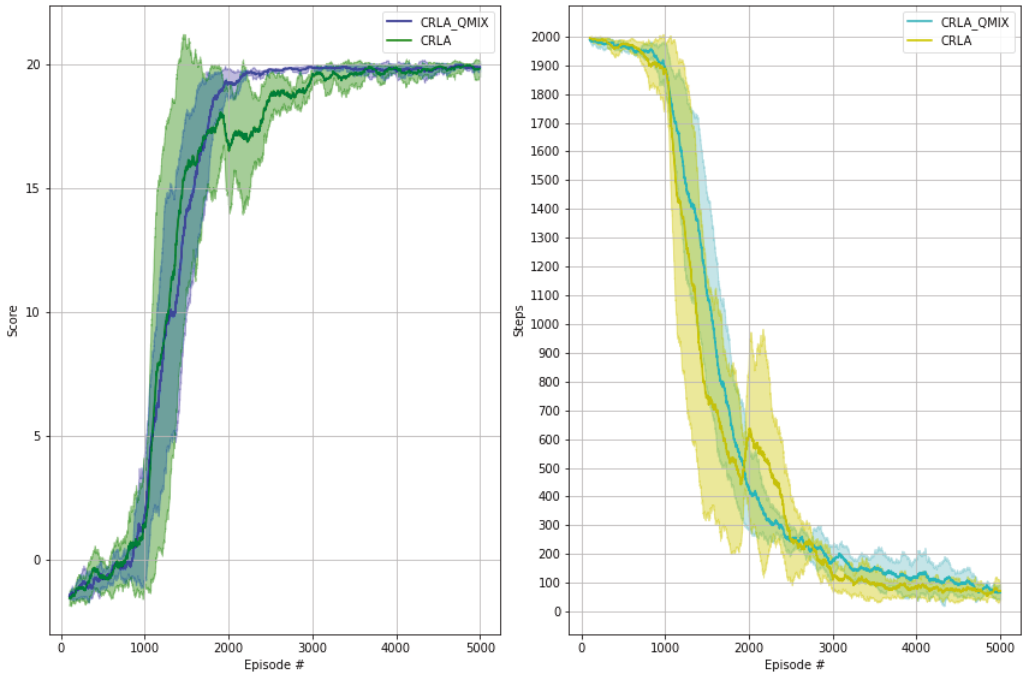


Figure 3.8 : Performances of CRLA with and without QMIX on the 60-hosts scenario.

3.5.4 Discussion

To examine how well each agent in CRLA learned, we looked into the state and action representations that the trained agents learned by visualising the state representations using the t-SNE method (Van der Maaten and Hinton 2008). Figure 3.9 shows the results. As can be seen, the state representations for the 50-host scenario were quite similar across the three agents. Surprisingly, the learned state representations were mapped into nine separate clusters, which matched the nine private subnets in the 50-host configuration. This configuration itself had one public subnet and nine private subnets, each with five hosts, even though this knowledge was not provided or observable by the agents. The coloured markers represent the identifier of the action component each agent would take in certain states. The action space

of 1326 was mapped into smaller subsets of 10 to 15 action components per agent.

Figure 3.9 : State representation from the 3 agents of the 50-hosts scenario. The clusters are coloured based on the action components.

After training, each agent learned that only two to three action components in its own subsets were needed to optimally capture the hidden assets. This visualisation opens up new research directions where further examination of the state and action representations might yield better progress in applying deep reinforcement learning to even more complex penetration-testing scenarios.

3.6 Chapter summary

This chapter introduced a new cascaded agent reinforcement learning architecture called CRLA to tackle penetration-testing scenarios with large discrete action spaces. The proposed algorithms require minimum prior knowledge of the problem domain while providing competitively better performance than conventional DQN agents. We validated the algorithms on simulated scenarios from CybORG. In all tested scenarios, CRLA showed superior performance to a single DDQN agent.

The proposed algorithm is also scalable to larger action spaces with a sub-linear increase in network computational complexity. This is because the individual networks are trained and used for inference separately and their outcomes are used sequentially to compute the final action. Despite CRLA's promising results, various challenges remain to be resolved. Future work will extend CRLA to incorporate

the learning of sub-goals. This should make CRLA perform better in environments where the rewards are much sparser. Additionally, efficient exploration in large action spaces is still an outstanding problem that deserves further research.

Chapter 4

A multi-agent reinforcement learning approach to multinomial parameterised action space in autonomous penetration testing

In this chapter, we propose a novel approach to learning these strategies based on deep reinforcement learning that works on representations of the parameterised action space. To automate the process using artificial intelligence, simulators such as the Cyber Operations Research Gym or CybORG help researchers develop algorithms that learn attack and defence strategies. However, the complexity in the structured action space implemented in these simulators presents challenges to conventional deep reinforcement learning algorithms. In this paper, we propose a novel approach to learning these strategies based on deep reinforcement learning that works on representations of a multinomial parameterised action space in CybORG. The presented architecture uses a multi-agent system to separately learn each action component but follows a cooperative training to achieve convergence and stability. When tested in a range of scenarios of varying complexity with up to 100 hosts, our approach delivers superior performance and better learning efficiency than previous architectures.

4.1 Introduction

Recent interest in developing artificial intelligence-based penetration testers and threat detection tools has led to the development of multiple cyber security simulation platforms. These game-based simulated scenarios describe dynamic activities in an abstract enterprise of connected networks where different types of agents operate and interact with each other and change the underlying network information. The high-level abstraction of complex security concepts and the interconnection interfaces of computer networks in these open-source platforms has seen the implementation of autonomous agents trained through reinforcement learning. Some of these environments formulate security games using Markov decision processes (MDP) or partially observable MDPs (POMDPs). For example, an autonomous attacking agent, usually referred to as the red agent, is presented with an observation that represents an observable network from the attacker's perspective. It then learns to take different actions that will alter the underlying network information. The aim is to penetrate the protected network and compromise the host(s) to acquire sensitive assets. Notably, the transition dynamics and reward functions are generally unknown to the agents, which makes reinforcement learning ideal for such applications.

A growing number of open-source platforms have been developed to experiment with reinforcement learning as it pertains to cyber security. These include but are not limited to *gym-idsgame* (Hammar and Stadler 2020), *CyberBattleSim* (Team. 2021) and *NAsim* (Schwartz and Kurniawati 2019a). Each of these environments abstracts the attacking actions differently; they also have a different influence on the state transition dynamics. Both affect the realism with which these simulated scenarios are represented. Other factors include the ability to enable concurrent adversarial training of both the red team (the attacker) and the blue team (the defender). The ease with which the simulation can be scaled to a number of hosts and networks also needs to be considered. On this front, CybORG, fulfills all of the requirements. The simulator in CybORG is designed for ease of scalability. The number of hosts, subnets, attack types and its required parameters for a specific

attack can all be configured. Figure4.1 illustrates the action space implemented in CybORG. The varying number of connected subnets and hosts dictates the complexity of the scenario.

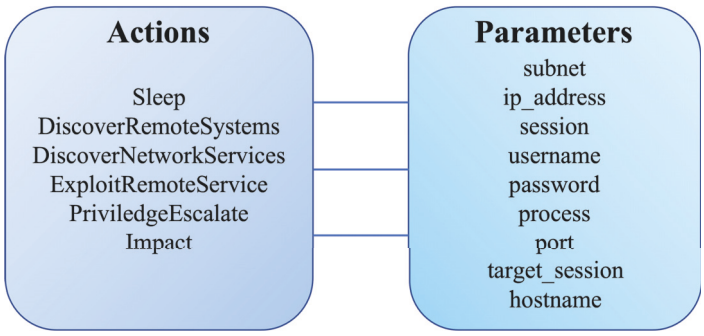


Figure 4.1 : CybORG action components.

Figure 4.2 illustrates an example scenario from CybORG with 15-hosts connected via different subnets. The red attacker is presented initially with the public subnet and has to figure out the penetration path to compromise the host with the hidden asset, represented by the flag, using an attack path consisting of a series of actions. The placement of the flag and the subnet connections are configurable.

Table 4.1 displays a detailed breakdown of the action space for the mentioned 15-host scenario. The *action* component has 6 possible values, shown as 6 types of attack in Figure 4.1. Other components include the different parameters available for each of the attacks. A valid attack action is a combination of an action type and a corresponding value for each parameter. As an example, the *Sleep* action may not use any parameters, whereas the *ExploitRemoteService* needs to identify the host, its opening ports and running processes, and the available subnet.

Action space representation plays a crucial role in defining a reinforcement learning problem. Conventional reinforcement learning transforms a primitive action space into a flattened set of discrete action identifiers. Applying this transformation to the CybORG action space means all the applicable actions would be mapped to single integers in the range of 0 to a max_value equal to the combined multiplication of all the values on the right column of Table4.1. For instance, given

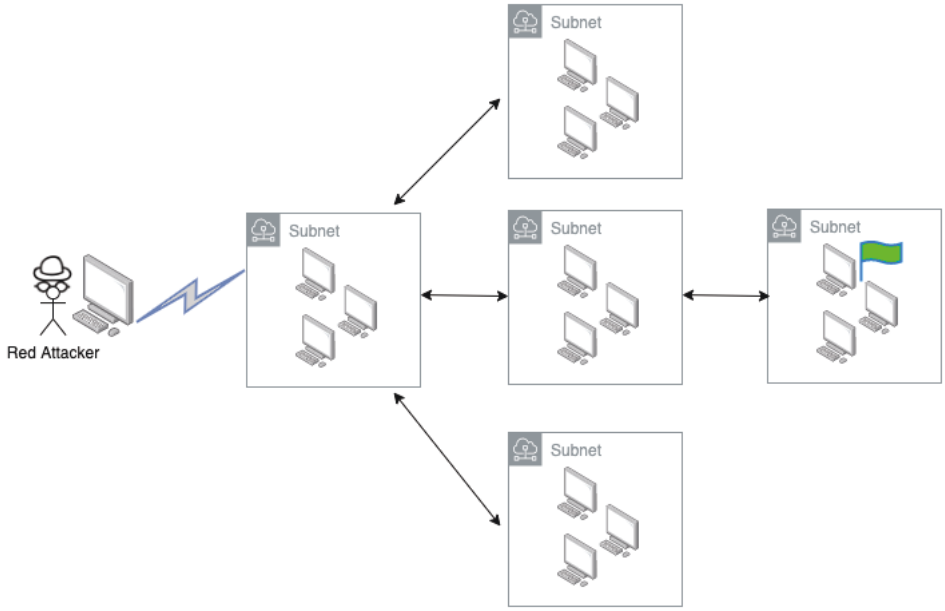


Figure 4.2 : An example of a 15-hosts scenario in CybORG

Table 4.1 : The action components and their possible values in the 15-hosts scenario in CybORG

Action component	Number of possible values
action	6
subnet	6
ip address	16
session	1
username	6
password	1
process	69
port	4
target session	8
host name	16

the 15-hosts scenario, the maximum integer identifiers of the flattened action space would be $6 \times 6 \times 16 \times 6 \times 69 \times 4 \times 8 \times 16 = 122,093,568$. However, notably, this

gigantic action space would cause any typical deep reinforcement learning algorithm to fail(Dulac-Arnold et al. 2015).

The parameterised action space in reinforcement learning is a setting wherein an exercised action consists of a main action type a_t and its parameters x_t formulated as $\langle a_t, x_t \rangle$(Masson et al. 2016). The main action is discrete, but the parameters can be either discrete or continuous. This action space representation is common in games such as StarCraft II(Vinyals et al. 2019) and robotics(Kober et al. 2013). In the setting of continuous parameters, the values are discretised and considered as another discrete set of *parameter* actions(Hausknecht and Stone 2015b). Learning is done by assuming that all parameters have the same semantic meaning and can be represented by a single neural network that is jointly trained with another separate network for selecting the main action type(Bester et al. 2019; Fan et al. 2019b). However, these approaches do not consider settings with multi-parameter selection and different parameter sub-spaces given the main selected action.

In this study, we propose a novel method of training reinforcement learning algorithms. First, the main action type is chosen using a separated neural network. The action parameters are then selected conditioned on the main action. These selected action components are then trained cooperatively using a coordination algorithm inspired from multi-agent reinforcement learning, which ensures a stable learning process. The proposed method not only performs well on tested CybORG scenarios in comparison to existing algorithms, it also maintains the semantic dependencies between the parameters and the actions. As such, the contribution of our work is threefold: 1) Separate networks are used to learn the actions and the parameters that make combined action-parameter selection scalable; 2) We make it possible to have varying set cardinalities or even different parameter representations as each parameter is handled by a separate neural network; thus, the approach is highly customisable. 3) We offer an adapted cooperative training algorithm based on multi-agent reinforcement learning (MARL) named QMIX to co-train the action and parameter networks. The algorithm includes an auxiliary training task that learns a good state representation for stable learning. Our experiments also demon-

strate the learning scalability of the proposed approach when trained on scenarios of up to 100 hosts.

4.2 Background

The problem to be considered in this chapter is a discrete time reinforcement learning task modelled by a parameterised action Markov decision process or PAMDP, formalised by a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$. We followed the parameterised action space formulation in (Masson et al. 2016), adapting the notations for this work. At each time step t, the agent receives a state observation $s_t \in \mathcal{S}$ from the environment. To interact with the environment, the agent executes an action $a_t \in \mathcal{A}$, after which it receives another state s_{t+1} according to the environment transition function $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$, and a reward signal r_t given by $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$. The agent aims to maximise a utility, defined as the total sum of all reward gained in an episode $R_t = \sum_{\tau=t}^{\infty} \gamma^{\tau-t} r_\tau$, where $\gamma \in [0, 1]$ is the discount factor used to determine the importance of long-term rewards. The goal is to learn a policy function $\pi : \mathcal{S} \to \mathcal{A}$ which is a probability distribution over the action space given the current state $\pi(s_t)$, given that the current state is all the agent needs to make decision.

In our PT setting, the red agent does not have access to the true state of the environment. It is only presented with the observable part of the compromised network. Most works on applying RL to penetration testing model the environment as a partially observable Markov decision process or POMDP(Hausknecht and Stone 2015c) where the agent does not have access to the true environment states but receives an observation $o_t \in \Omega$ from the observable space Ω. The observation is generated from an underlying probability distribution function $o_t \sim \mathcal{O}(s_t)$ which is generally unknown. Under POMDP, one needs to estimate a belief probability distribution function $b_{t+1}(s_{t+1}) = Pr(s_{t+1}|o_t, a_t, b_t)$ with $b(s)$ is the probability distribution over environment states to model the world states for decision making. However, this function is generally intractable as we do not have access to the true state of the world nor the probability function of the states. In the RL literature, recurrent neural networks are often used to relax the Markov assumption by giv-

ing the policy function information from previous time steps, e.g. $\pi(s_t, h_t)$ where $h_t = f(s_{t-1}, s_{t-2}, \ldots, s_0)$(Hausknecht and Stone 2015c).

In this study, gated recurrent units(Dey and Salem 2017) or GRUs are used to help the agent better estimate the underlying state distribution function by maintaining a latent state vector h_t that summarises past transitions. As a result, we will not distinguish the difference between o_t and s_t in this work.

In this study, the penetration tester models the interaction of the red attacker with the CybORG simulator through game-based episodes. Each episode starts with an initial observation o_0. The RL agent then starts to exercise different actions and receives new observations. The episode ends when either the RL agent captures the flag, in which case a done flag d_t is set to *True*, or the number of time step exceeds a predefined limit T. The collected transitions (o_t, a_t, r_t, d_t) within an episode, where $t = [0, \ldots, T]$, are called a trajectory τ. A replay buffer $\mathcal{D}$ is used to store the experienced trajectories, from which experiences will be randomly selected for learning(Mnih et al. 2015).

The action space comprises a set of discrete actions, $\mathcal{A}_d = [D] = \{a_1, a_2, \ldots, a_D\}$ and a set of P parameters, represented by a domain $\mathcal{X}_p$ that can be continuous or discrete or a hybrid of both types. In this work, we have limited the parameter representation to be discrete: $\mathcal{X}_p \subset \mathbf{Z}^+$. In the general setting, it is not necessary for each action a_d to be associated with a single parameter x_k. This is a common approach in many works dealing with a parameterised action space. In CybORG, some attack types do not need any parameters (e.g, $x = \emptyset$), while others may require multiple supporting parameters. Hence, we extended the parameterised action Markov decision process (PAMDP) to support a variable number of parameters for each action. Thus, the action space can be re-written as in Equation 4.1.

$$\mathcal{A} = \bigcup_{d \in [D]} \{a = \langle a_d, (x_1, \ldots, x_p) \rangle | x_p \in \mathcal{X}_p\} \tag{4.1}$$

In PAMDP, the state transition probability and the reward function are $\mathcal{P}(s'|s, \langle a_d, (x_1, \ldots, x_p)\rangle$ and $\mathcal{R}(s, \langle a_d, (x_1, \ldots, x_p)\rangle, s')$ respectively. In this paper we have used the value-

Table 4.2 : Notation table

Symbol	Meaning
s	State
o	Observation
a	Action
r	Reward
t	Time step t
d	Done flag
α	Learning rate
γ	Discount factor
θ	Neural network parameters
$\pi(\theta)$	Policy parameterised by θ
r_t	Reward received at time step t
$\mathcal{D}$	Replay buffer
τ_i	A trajectory i

based method deep Q-network which learns a parameterised policy $\pi(\theta) : \mathcal{S} \rightarrow \mathcal{A}$ to maximise the expected discounted return of taking an action a_t in the state s_t and following the policy π thereafter (e.g., the state-action value function $Q(s, a; \theta) = E[R_t|s_t = s, a_t = a, \pi]$). The state-action value function is updated recursively using temporal difference learning as illustrated in Equation 4.2(Sutton and Barto 2018b). Table 4.2 summarises all the typical notations used in the equations of this study.

$$Q(s_t, a_t; \theta) \leftarrow Q(s_t, a_t; \theta) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a; \theta) - Q(s_t, s_t; \theta)] \tag{4.2}$$

In non-trivial environments, deep neural networks are used to estimate the state action value function. To optimise the network parameters, the following loss func-

tion needs to be minimised (Equation4.3).

$$L_i(\theta_i) = \sum_{j=1}^{b} [(y_j - Q(s, a; \theta_i))^2] \tag{4.3}$$

where

$$y = r_t + \gamma \max_{a'} Q(s', a'; \theta_i^-) \tag{4.4}$$

The subscript i indicates the estimated value of the function and the parameters at the i^{th}iteration. The loss function is updated using random batches of size b from the replay buffer. To stabilise the training, another set of target parameters θ^- with Polyak updates is used to estimate the temporal difference target y(Mnih et al. 2015).

The action space in penetration testing takes on large values as the number of hosts increases. As a result, deep dueling Q-learning is used to help the agents learn a large action space. This learning scheme has been shown to help DQNs learn a more stable state-action value by using an action advantage estimate(Wang et al. 2015) is used. The action advantage function is calculated as per Equation4.5:

$$A^\pi(s, a) = Q^\pi(s, a) - V^\pi(s) \tag{4.5}$$

where $V^\pi(s) = E[Q^\pi(s, a)]$.

Notably, other variants of the DQN algorithm which have been shown to be better than the deep dueling DQN algorithm(Hessel et al. 2017a). However, it is not within our scope to develop another state-of-the-art DQN algorithm. Rather, our aim is to propose a new architecture for multinomial parameterised action spaces. And using the duelling DQN algorithm lets us demonstrate how our mechanism works and its efficiency in isolation from other improvement techniques.

4.3 Methodology

4.3.1 Multi-agent learning for parameterised action space

Figure 4.3 illustrates the hierarchical and multiagent learning nature of our work in the mentioned PAMDP. We refer to this architecture as *Multi-Agent for Parameterised Actions* or *MAPA* for short. The core algorithm is DQN, and so the agents

are referred to as DQN agents. In MAPA, there are two layers of agents. The agents in the first layer learn the main type of attack a_d, which can be any of the previously listed actions. (e.g., *exploit host*, *scan port*, *scan subnet*, or *sleep*). The first DQN agent (DQN#1) learns the state-action value $Q_d(s, a_d)$ and passes it to a second-layer DQN agent via a state-action value embedding layer, which is a fully connected neural network layer. This intermediate neural network of a single-layer transforms the state-action value $Q_d(s, a_d)$ into a smaller representation for the next DQN agent. The DQN agent in the second layer (DQN#2) consists of sub-networks dedicated to different supporting parameters $x_1, x_2, ..., x_p$ (e.g.,*ip address*, *subnet*, *session*, or *host number*) (Figure 4.4). It takes in the state and reward from the environment as well as the previously generated state-action value's embedding from the first-layer agent and estimates the state-action values for all the supporting parameters $Q_1(s, (a_d, x_1), \ldots, Q_p(s, (a_d, x_p)$. This learning is done through P separate sub-networks of different sizes (e.g., 10 hosts, 4 sessions, 6 subnets). Finally, the *Cooridinator* block takes in all the state-action values from the two DQN agents and estimates a total state-action value function $Q_{tot}(s, (a_d, \mathbf{x})$ where $\mathbf{x}$ represents a vector of supporting parameters. A more detailed view of the second DQN agents is shown in Figure 4.4. All the actions are selected by following a greedy policy where $a = \text{argmax}_a\, Q(s, a)$.

Each DQN agent is trained using the temporal difference learning within a deep q-network. For instance, the loss function for DQN#1 follows, noting that where the subscript i has been removed for simplicity:

$$L_d(\theta) = \sum_{j=1}^{b} [(y_j - Q_d(s, a_d; \theta))^2] \tag{4.6}$$

where

$$y = r_t + \gamma \max_{a'} Q_d(s', a'_d; \theta^-) \tag{4.7}$$

The loss function for each of the parameter network $p \in [1, \ldots, p])$ is:

$$L_p(\theta) = \sum_{j=1}^{b} [(y_j - Q(\psi(s, a_d), x_p); \theta))^2] \tag{4.8}$$

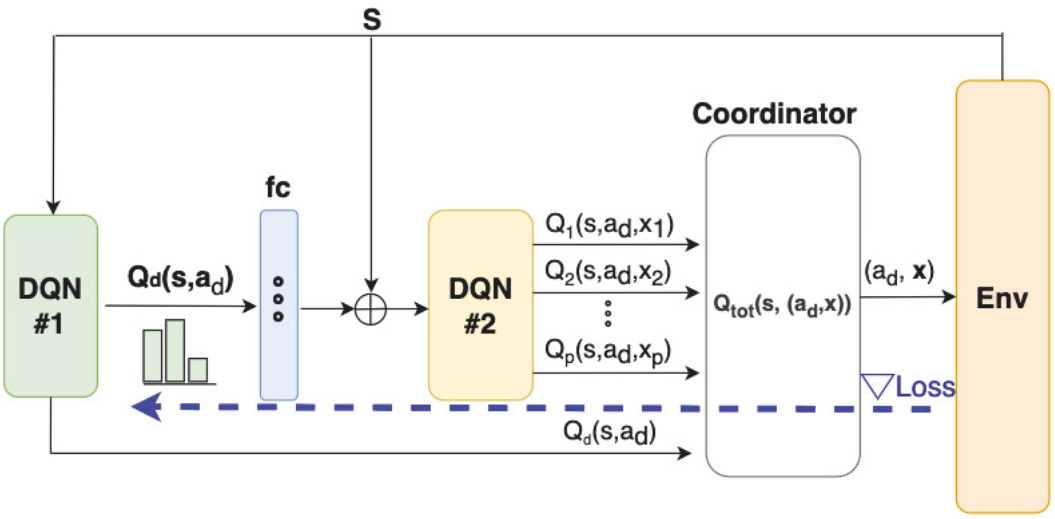


Figure 4.3 : Overview of the MAPA architecture. DQN1 is used to select the attack type. The state-action value of the chosen attack action is passed to DQN2 for parameters' selection.

where

$$y = r_t + \gamma \max_{x'_p} Q(\psi(s, a_d), x'_p; \theta^-) \tag{4.9}$$

and the state-action value's embedding is:

$$\psi(s, a_d) = f(Q_d(s, a_d)) \tag{4.10}$$

Each network head of the second-layer DQN learns to pick the right value for each parameter, given the embedding of the state-action value from the previous DQN agent. This sequential learning mechanism is similar in spirit to (Wei et al. 2018b) in that it retains the semantic dependency between the action and the corresponding parameters. That said, Wei et al. (Wei et al. 2018b) demonstrated this approach with a continuous parameter space and found the learning process would not stabilise. To our understanding, this instability was the result of having multiple separate networks concurrently learn the final composed state-action value function $Q(s, \langle a_d, (x_1, \ldots, x_p)\rangle)$. The strategy resembles a multi-agent learning approach called independent Q-learning (IQL) (Tan 1993c) where each agent, or network head in this case, learns its own policy by optimising for its own loss

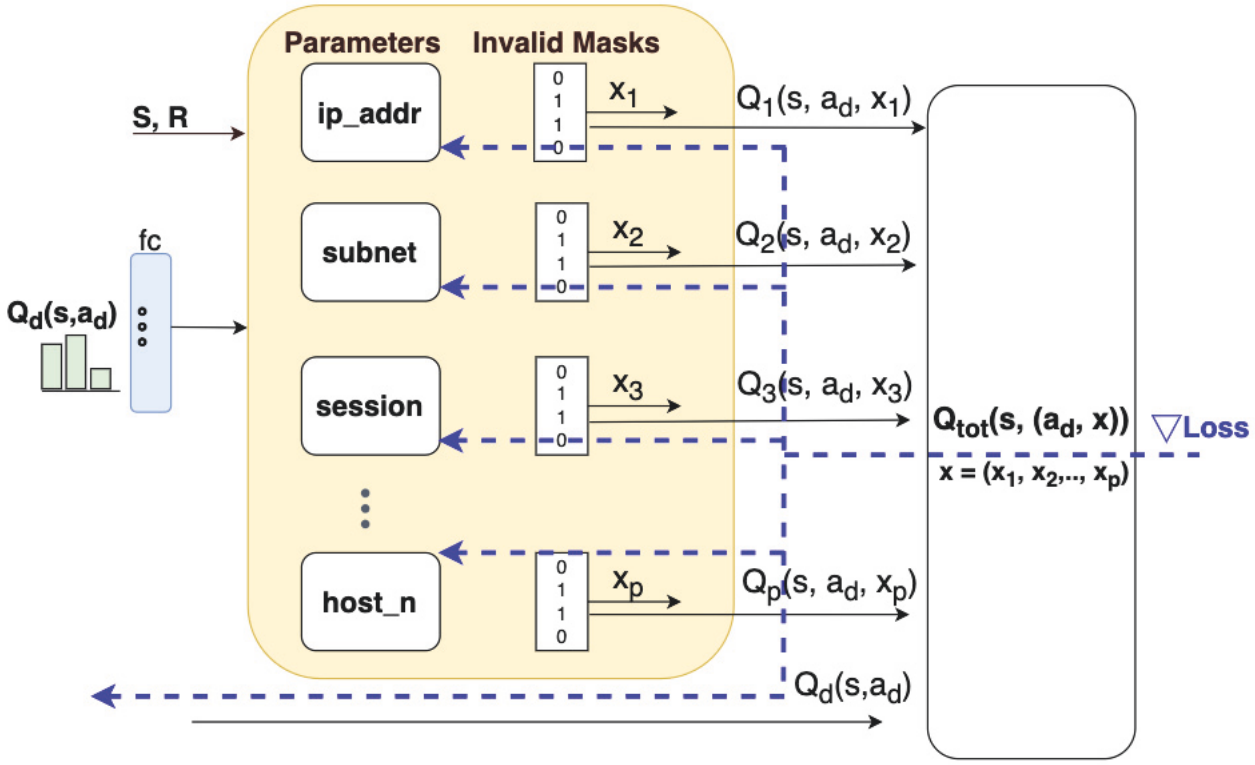


Figure 4.4 : A detailed view of the second DQN agent. Sub-networks are used for different parameters. Invalid masks are applied to the outputs of the sub-networks to mask out invalid parameters.

function (e.g., Equations 4.6 and 4.8). The IQL strategy violates the non-stationary assumption in general MDP, while agent i remains unaware of the changes in the underlying environmental dynamics due to the actions generated by other agents $j \neq i$. As a result, the individual loss function $L_i(\theta_i)$ is unable to propagate the necessary information to update each agent's state-action value, which leads to sub-optimal or even divergent behaviour. In our approach, it is important for each parameter's network to be aware of the output of other networks, as well as the selected attack type from DQN#1. This cooperative learning paradigm is common in multi-agent reinforcement learning. We adapted a well-known network architecture called QMIX (Rashid et al. 2018) to serve as our centralised training update mechanism. The mixing network inside QMIX acts as an aggregating function, combining all

state-action value functions and optimising a total loss function. The dotted blue arrows in both figures display the gradient path flowing backwards in each step of the training. Details of the loss function will be discussed in the next section. The proposed architecture using QMIX implements a fully centralised training and end-to-end differentiable function. During inference, the agents and sub-networks can generate actions directly without going through the QMIX net. Therefore, nothing extra is incurred in terms of a computing overhead.

4.3.2 Cooperative learning

In this paper, we adapted QMIX's cooperative training style to optimise the state-action value functions of both the DQN agents in MAPA. To do this, an aggregation function has to be used to combine all the state-action value functions. However, there are different ways the aggregation function g in Equation 4.11 could be formulated. For example, it could simply be formulated as a summation over the individual state-action value functions (Sunehag et al. 2017a), or as a mean function (as in Equation 4.12) (Tavakoli et al. 2018; Zhou et al. 2021).

$$\begin{aligned} Q_{tot}(s,(a_d,\mathbf{x});\theta) = g(&Q_d(s,a_d;\theta_d),\\ &Q_1(\psi(s,a_d),a_1;\theta_1),\ldots,Q_p(\psi(s,a_d),a_p;\theta_p)) \end{aligned} \tag{4.11}$$

$$\begin{aligned} &Q_{tot}(s,(a_d,\mathbf{x});\theta) = \\ &\qquad \frac{1}{P+1}\sum_{i=1}^{p}(Q_i(\psi(s,a_d),a_i;\theta_i)+Q_d(s,a_d;\theta_d))) \end{aligned} \tag{4.12}$$

In this paper, we implemented g as a neural network with the weights being learned through the training and optimisation process. To ensure the maximisation of Q^{tot} is monotonic with respect to each individual action value (Equation 4.13), a hyper-network (Ha et al. 2016) is used to indirectly learn the weight for the network of g. The monotonicity property is important to ensure that the optimal total state-action value function can be translated to optimality of each state-action value function. This approach shows much better performance experimentally compared to the implementation of g using a summation or mean function.

$$\underset{(a_d,\mathbf{x})}{\operatorname{argmax}}\, Q_{tot}(s,(a_d,\mathbf{x})) = \begin{pmatrix} \operatorname{argmax}_{a_d} Q_d(s,a_d) \\ \operatorname{argmax}_{a_1} Q_1(s,a_1) \\ \vdots \\ \operatorname{argmax}_{a_p} Q_p(s,a_p) \end{pmatrix} \tag{4.13}$$

The total loss function is defined in Equation 4.14. Following this centralised training, the gradient of the loss function can be distributed to all the parameter networks and the first DQN agent, as shown by the dotted blue arrows in Figures 4.3 and 4.4. As a result, each agent updates its parameters taking the loss incurred by other agents into account. This leads to a better learning experience and achieve faster convergence.

$$L_{tot}(\theta) = \sum_{j=1}^{b} [(y_j - Q_{tot}(s,(a_d,\mathbf{x};\theta))^2] \tag{4.14}$$

where $y_j = r_t + \gamma \max_{(a'_d,\mathbf{x}')} Q_{tot}(s',(a'_d,\mathbf{x}');\theta)$.

4.3.3 Auxiliary State Representation Training

The size of the observation vector can become large as the complexity of the simulated scenario increases. To reduce the number of trainable network parameters and to enhance training efficiency, an encoder-decoder architecture(Sutskever et al. 2014) is used to assist the learning of the state representation and reduce the dimensionality of the observation vector. Figure 4.5 shows the neural network architecture implemented for each agent in MAPA. The encoder block takes in the observation vector o_t and performs a non-linear transformation via two fully connected layers (fc) and the ReLu activation functions (relu). The extracted feature $\phi(o_t)$ is then used as the input to the upper branch, which is where the reinforcement learning algorithm is trained, and the decoder block at the lower branch where the state representation learning is trained. The decoder is optimised to reconstruct the next observation o_{t+1} so that the predictive features of the observation can be extracted. In our experiments, the training of this auxiliary state representation helps MAPA achieve better and more stable convergence.

4.3.4 Invalid Action Masking

Figure 4.4 shows another feature of the MAPA implementation, which are the invalid masks applied to the outputs of the parameter networks. The use of invalid masks was first officially introduced and examined by Huang and Ontañón (Huang and Ontañón 2020). Invalid masks are Boolean vectors where invalid or unavailable actions are masked as a Boolean false value. Invalid masks not only reduce the number of action parameter values to learn at each step but also enhance the estimation of the available action parameters. Figure 4.6 shows the application of the mask on the logits of the state-action values. For example, if a host number 10 is not discovered at time step t_1 then randomly exercising an attack action on that host will not be beneficial to the network's learning. Assigning a penalty or negative reward signal to the attempted invalid action is the conventional way of dealing with invalid actions. However, this approach does not prevent an invalid action from being selected during random exploration. It also hinders the learning of other valid actions in the same state/observation. However, using action masking, the invalid action logits are assigned with negative infinity (e.g $-\infty$), which ensures that, under no circumstances, can the invalid action be selected. This would not be the case if a negative reward signal had been used.

4.4 Experiments

4.4.1 CybORG

This section provides the details of the simulated cyber security application called CybORG(Standen et al. 2021) and the experiments. We used CybORG as a testbed for the MAPA algorithm. The penetration tester was modelled as a red agent applying exploits to a network of hosts and subnets. CybORG was developed as an open source environment for developing artificial intelligence-enabled solutions to cyber security challenges, such as autonomous penetration testing.

The white hat attacker, which is often referred to as the red agent in these scenarios, is initially presented with a connection to the public subnet initially (Figure 4.2). However, it has no knowledge of the inner topology of how all the private

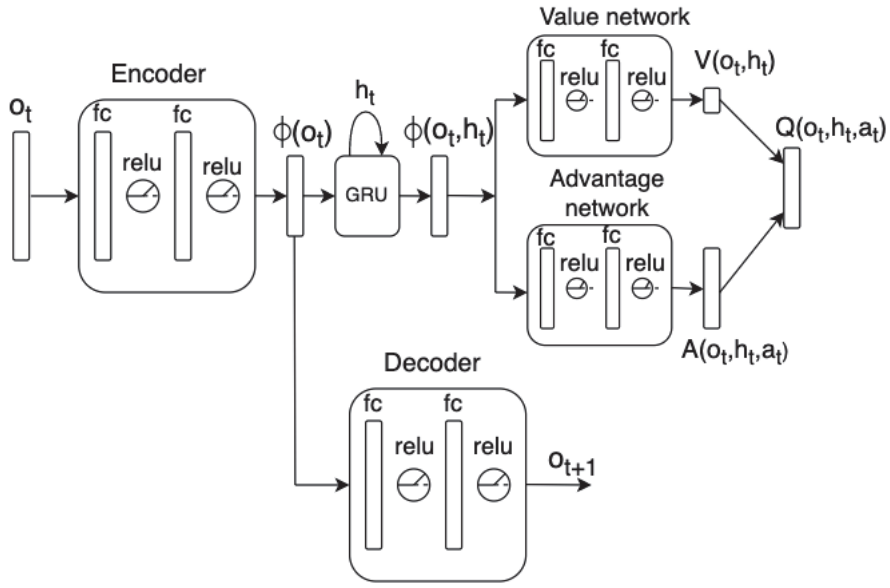


Figure 4.5 : Dueling deep q-network architecture.

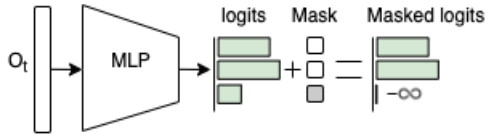


Figure 4.6 : Invalid action mask.

subnets and hosts are connected. The observed state of the network is provided as a 1-dimensional array of floating point values (Figure 4.7). Each value in the array describes bits of information regarding the network connections and states, such as the discovered host ID, the subnet ID, or which of the port IDs is available for exploit. The size of the observation vector depends on the number of hosts in the simulated network.

From this given observation vector, the agent needs to learn a sequence of actions to penetrate the network and reveal more information that eventually lets it find and compromise the host containing the flag. All the tested scenarios have different configurations of the number of hosts, the topology of subnet connections, and the location of the flag. The experiments were performed on a range of configurations

```
tensor([ 6.6667e-01,  0.0000e+00,  1.0000e+00,  1.0000e+00,  7.8571e-01,
        -1.0000e+00,  6.6667e-01, -1.0000e+00, -1.0000e+00, -1.0000e+00,
         4.1901e-01, -1.0000e+00, -1.0000e+00,  0.0000e+00, -1.0000e+00,
        -1.0000e+00, -1.0000e+00, -1.0000e+00, -1.0000e+00, -1.0000e+00,
        -1.0000e+00, -1.0000e+00, -1.0000e+00, -1.0000e+00, -1.0000e+00,
        -1.0000e+00, -1.0000e+00, -1.0000e+00, -1.0000e+00, -1.0000e+00,
        -1.0000e+00, -1.0000e+00, -1.0000e+00, -1.0000e+00, -1.0000e+00,
        -1.0000e+00, -1.0000e+00, -1.0000e+00, -1.0000e+00, -1.0000e+00,
        -1.0000e+00, -1.0000e+00, -1.0000e+00, -1.0000e+00, -1.0000e+00,
        -1.0000e+00, -1.0000e+00, -1.0000e+00, -1.0000e+00, -1.0000e+00,
        -1.0000e+00, -1.0000e+00, -1.0000e+00, -1.0000e+00, -1.0000e+00,
        -1.0000e+00, -1.0000e+00, -1.0000e+00, -1.0000e+00, -1.0000e+00,
        -1.0000e+00, -1.0000e+00, -1.0000e+00, -1.0000e+00, -1.0000e+00,
        -1.0000e+00, -1.0000e+00, -1.0000e+00, -1.0000e+00, -1.0000e+00,
        -1.0000e+00, -1.0000e+00, -1.0000e+00, -1.0000e+00, -1.0000e+00,
        -1.0000e+00, -1.0000e+00, -1.0000e+00, -1.0000e+00, -1.0000e+00,
        -1.0000e+00, -1.0000e+00, -1.0000e+00, -1.0000e+00, -1.0000e+00,
        -1.0000e+00, -1.0000e+00, -1.0000e+00, -1.0000e+00, -1.0000e+00,
```

Figure 4.7 : A 1-dimensional tensor describing the observed state of the network.

with an increasing number of hosts, which represents an increase in complexity.

The environment presents multiple aspects of complexity to RL algorithms. First of all, the red agent receives a single scalar reward if it can compromise the host with the sensitive asset, or the so-called flag. The number of steps required to capture the flag in a random exploration policy varies from a few hundred to thousands of steps. Given this sparsity in the reward signal, it is very challenging for RL algorithms to update the state-action value function. The reward function is constructed to help MAPA learn better without foregoing the realism of the training (Equation 4.15). The $R_{flag\_captured} = 10$ rewards the agent for capturing the flag. There is an extra small positive reward of 0.01 for discovering a new host via a successful attack action (e.g., $R_{discovered\_host} = 0.01$). In a scenario of n hosts, there would be a maximum extra reward of $0.01 \times n$. Finally, a negative cost of $R_{cost} = -0.001$ is applied for each executed action, representing the cost of exercising the attack. This cost also encourages the RL agents to learn the optimal and shortest path to capture the flag.

$$R_t = R_{flag\_captured} + R_{discovered\_host} + R_{cost} \tag{4.15}$$

Secondly, CybORG has a highly stochastic action effect when exercised on the simulated scenarios. Each attack action has a probability of success, and it also depends on the state of the simulation. The simulator's state machine consists of

various types of information, such as the operating system on each host and the services that are listening on each host. The agent sees an unknown value for each value it does not know. As the agent takes actions in the simulation, its awareness of the true state of the simulation increases. An action fails if the agent does not have remote access to a target host or subnet. However, an action also has a probability of failing even if the agent has access to the target. This probability reflects the reality that attempts to exploit vulnerabilities are seldom, if ever, 100% reliable.

We tested the MAPA architecture on different scenarios with an increase in complexity where the number of hosts varied from $12, 15, 18$ to $50, 60, 70, 80, 90$ and 100 hosts. The number of hosts per subnet was 3 for simple scenarios of $12, 15$ and 18 hosts, and this increased to 5 hosts per subnet for other scenarios. As the number of hosts increases, it is more challenging for the red agent to learn the attack policy with different combination of the attack type and the supporting parameters. Figure 4.2 illustrates an example of the 15-hosts scenario used in the experiments.

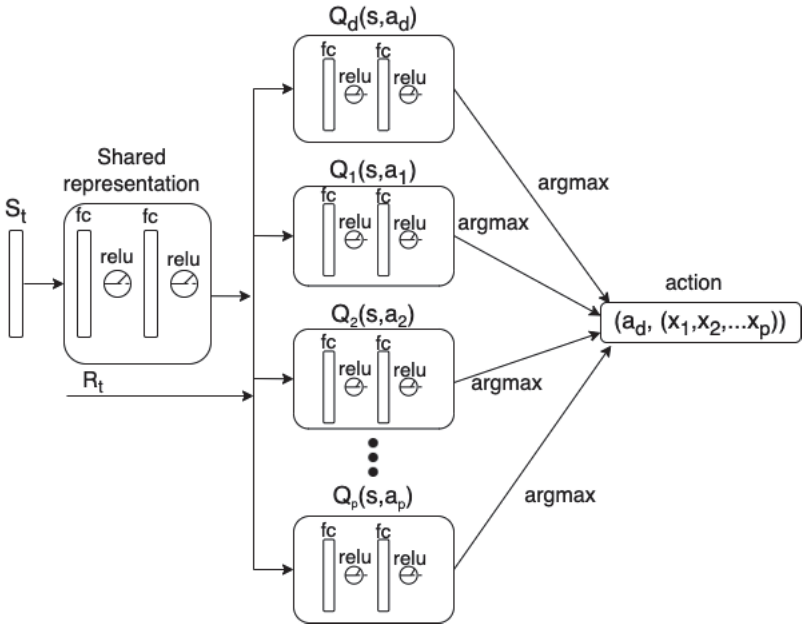


Figure 4.8 : An adaptation of the action branching architecture used as the baseline method.

4.4.2 Neural network architecture

The neural network architecture is illustrated in Figure 4.5. Each fully connected layer has a size of 256 neurons. The outputs of the state-action value and the advantage value function has different sizes depending on the scenario configurations. It is worth noting that even though MAPA uses the dueling DQN algorithm, our architecture can be applied to any value-based method.

We compared MAPA's performance to a baseline method with an independent branching sub-network and a non-coordinated learning scheme for selecting the attack type and action parameters (Figure 4.8). This resembles the learning strategy used in (Tavakoli et al. 2018; Zhou et al. 2021), which can be thought of as the conventional approach for multinomial parameterised action spaces. It is important to point out that, in these works, together with the proposed training method for parameterised or multinomial action space, the authors also used other improvement features such as prioritised experience replay(Schaul et al. 2015b), and random network distillation (Burda et al. 2018). These are well-known improvements that benefit the agents' learning, and can easily overcompensate for a sub-optimally performing action architecture.

Additionally, to better support reproducibility with both MAPA and the baseline method, we ran each experiment with 5 random seeds. All training was conducted using a single RTX6000 GPU.

Table 4.3 displays the values for the hyper-parameters used in the proposed algorithm.

4.5 Results

Our first step was to validate MAPA against the baseline methods (Tavakoli et al. 2018; Zhou et al. 2021), which train the sub-components separately and optimise the average of the individual state-action value functions. Note that we tested MAPA solely using dueling deep q-learning and without any other features as a way of verifying our approach alone. Performance was assessed in terms of the

Table 4.3 : Hyper-parameter settings

Hyper-parameter	Value
Replay buffer size	1000000
Network size	256
Decay rate	0.998
Discount factor	0.99
QMIX hyper-net 1	32
QMIX hyper-net 2	32
Learning rate	0.0001
Learning frequency	20
GRU hidden dim	128

accumulated reward scores and the number of steps required to capture the flag. An ideal performance would be a reward score converging on 10 coupled with a small number of stable steps. This would show that the RL agent had successfully learned an optimal attack policy.

Figure 4.9 compares the performance of three approaches - the baseline, and MAPA with and without the encoder-decoder to learn the state representations - on the 12-hosts scenario. The left panel shows the cumulative scores the agent obtained through each episode. The right panel displays the number of steps required to capture the hidden flag. The shaded regions are the variances between runs.

As shown, the baseline approach failed to consistently learn a flag-capturing strategy even in the 12-hosts scenario. This is reflected by both a high variance in learning, and the sub-optimal scores and number of required steps towards the end of the training. The MAPA variants, on the other hand, both showed a solid performance, with much lower variances and good scores. The scores reach a peak towards the end of the training with the minimum number of steps to capture the flag. This improvement in learning is because of the cooperative training in QMIX. QMIX allows different sub-networks to consider others' values when updating their

own parameters. As a result, within the experienced trajectories, the agents in MAPA were able to estimate accurately their state-action values. This is not the case if the aggregation is simply a summation or an average function as in the baseline method. MAPA with the auxiliary state representation learning shows signs of early convergence and was highly stable towards the end of the training. This indicates that the encoder-decoder architecture was not only able to learn a predictive state representation, it was also able to use that representation to improve the training of the reinforcement learning policies.

Figure 4.10 shows MAPA's performances in the $12, 15$ and 18-hosts scenarios. Here, it was able to learn the optimal attack policy after approximately 2000 training episodes. We further tested MAPA on more complex scenarios with host number up to 100 hosts (Figure 4.11 and Figure 4.12). In large scenarios of 80 to 100 hosts, we extended the number of training episodes to $10,000$ so the agents had more time to explore. Due to the realism of CybORG, the training on these scenarios took significantly more wall-clock time, ranging from 6 to 48 hours for a successful run. This imposes a practical limit on the training resources consumed, and so we could not test MAPA at larger scales. However, across the experiments, MAPA delivered solid convergence in terms of both accumulated scores and the number of steps to reach the target. TThus, we conclude that using a multi-agent approach to handle multinomial parameterised action spaces is feasible.

4.6 Chapter summary

In this work, we extend the capability of deep reinforcement learning to tackle the multinomial parameterised action spaces in cyber security research. We proposed a novel architecture that transforms single agent reinforcement learning into a special parameterised action space within a multi-agent learning framework, in which each component of the action space is managed by an RL agent. Leveraging *QMIX*, which is a centralised training paradigm, we successfully trained a MAPA architecture to perform penetration testing in CybORG scenarios. Excellent performance was demonstrated on a range of scenarios. The results show that the DRL agents

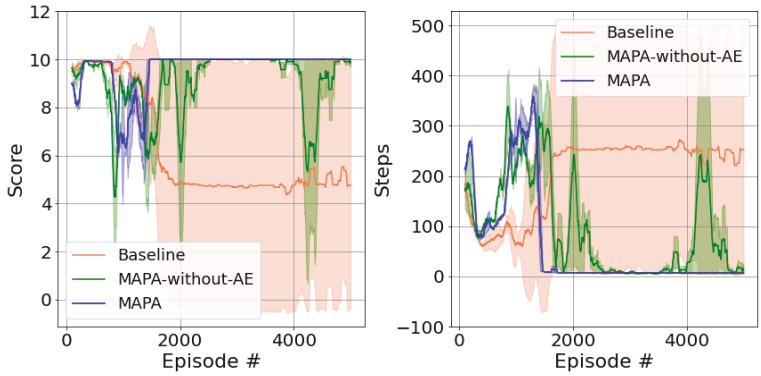


Figure 4.9 : Performance comparison between MAPA and the baseline with action branching. Left panel: the accumulative score during training. Right panel: the number of steps to capture the flag during training.

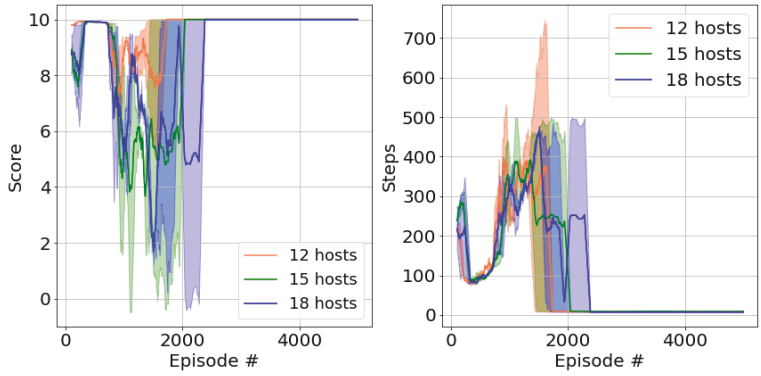


Figure 4.10 : Performance with the 12, 15 and 18-hosts scenarios. Left panel: the accumulative score during training. Right panel: the number of steps to capture the flag during training.

successfully learned the optimal attack policy in a minimum number of steps. In additional to having solid performance, MAPA maintained the dependencies between the parameter selection and the main attack action. This is critical for exploring the interpretability of the decision-making process of DRL agents.

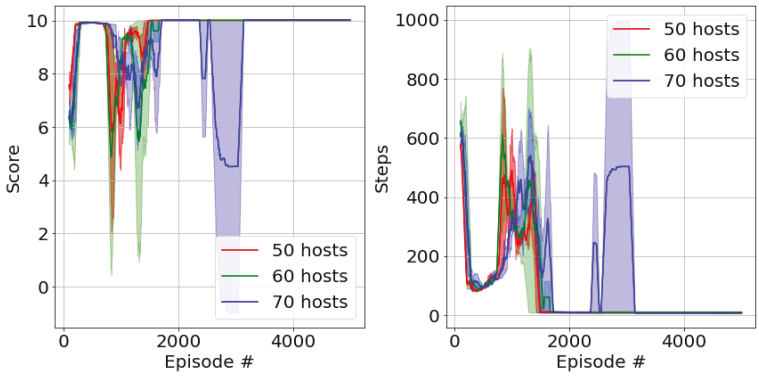


Figure 4.11 : Performance with the 50, 60 and 70-hosts scenarios. Left panel: the accumulative score during training. Right panel: the number of steps to capture the flag during training.

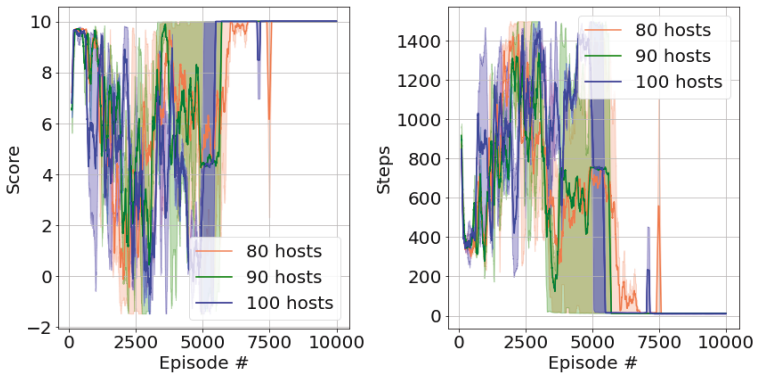


Figure 4.12 : Performance with the 80, 90 and 100-hosts scenarios. Left panel: the accumulative score during training. Right panel: the number of steps to capture the flag during training.

Chapter 5

State representation for effective learning in sparse reward environment

This chapter attempts to address one notable challenge in complex domains involving large state spaces with delayed or sparse rewards. The huge and complex state space presents a significant challenge in terms of exploration for RL agents. The absence of dense rewards prevents the agent from learning a good policy which can lead to higher reward regions in the environment. We propose a learning scheme involving state representation learning (SRL) and hierarchical reinforcement learning (HRL) in order to discover useful subgoals in the environment; and use those to enhance the learning of the RL agent. The model-free learning method reduces the reliance of the RL agents on the reward signals by learning a state representation using temporal difference learning, known in the RL literature as Successor Feature (SF) representation. We demonstrate the feasibility and efficiency of the proposed approach on *hard* settings of CybORG scenarios which have complex structured state space with sparse reward signals.

5.1 Introduction

Large scale applications present a multitude of challenges to RL agents. Previous chapters proposed novel solutions to tackle different action space representations, such as large action spaces and structured parameterised action spaces. Another

typical challenge of using RL in solving real-world problems is that of having large and complex state spaces. It is common in practice that the agents do not have access to the ground truth state of the environment but are only exposed to partially observable information. As a result, the agents have to do learning in a partially observable MDP (POMDP) which is much more challenging. There is no known theoretically supported solution for doing reinforcement learning in POMDP. Current works use different recurrent neural network architectures to mitigate the lack of the underlying environmental states (Hausknecht and Stone 2015a; Li et al. 2015). The learning becomes even more difficult when the environment has delayed or sparse reward feedbacks. These two main obstacles are common in many engineering domains (Ni et al. 2021). Therefore, learning a good state or observation* representation is an active research area in RL community. A good and useful state representation (SR) to the DRL agents must describe a large number of possible state transitions in a reduced form of latent vectors. The efficient and effective SR captures the most information from previous transitions and possibly future prediction. There are different approaches to learning a good SR in RL, for example, learning from demonstration or expert data (Hester et al. 2017; Brys et al. 2015), or from random exploration data (Li et al. 2021a). SRL can also be assisted with the present of the reward signal which is the main learning signal in reinforcement learning (Rafati and Noelle 2019b), or even without any reward signal (Merckling et al. 2021). The SR learned by the latter can focus on extracting the environmental dynamics via its observable state transitions.

State representation learning can be done by following the environment primitive dynamics (Merckling et al. 2021), or by learning an abstraction of the similar states (Rafati and Noelle 2019a). The learned abstraction can be temporal (Kulkarni et al. 2016) or spatial (Dayan and Hinton 1993). Spatial abstraction represents topographically related states with similar latent representations. This is espe-

*In RL literature, it is common to use *state representation* even though the precise terminology should be *observation representation* in POMDP. We will use *state representation* for most parts unless there is a need to differentiate the two terms.

cially useful wherein states can be distinguished by positional information such as in games or maze-based environments. However, it is possible for states to be in a close proximity and still being temporally distant. This situation is illustrated in the Figure 5.1.

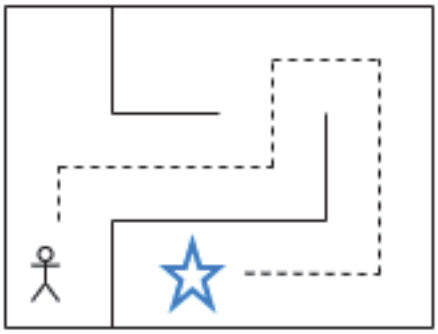


Figure 5.1 : The agent, represented by the humanoid figure, and the star are at spatially closed states but temporally distant.

State abstraction over temporal dimension can assist the agents in learning higher temporal policies over the primitive time steps of the environment. This learning paradigm, which is usually dubbed as hierarchical reinforcement learning (HRL), follows the Semi-MDP formulation wherein the agents perform learning over different timescales with the high-level *meta-controller* making decisions every $\mathbf{T}$ time steps and the low-level *controller* exercises primitive actions on every $\mathbf{t}$ time steps with the environments. Commonly, $\mathbf{t} = 1$ and $\mathbf{T} > 1$. This requires the agents to learn intermediate or *sub-goaling* states which can be achieved in $\mathbf{T}$ time steps.

HRL frameworks were proposed in the early 90s by a variety of works done by Dietterich (2000); Barto and Mahadevan (2003); Hengst (2004). In these original works, the main policy was decomposed into sub-tasks or sub-routines which can be achieved in a smaller number of time steps. A high-level controller learns to pick the sub-routines at certain states as well as the sequence of sub-routines to follow, whereas a lower-level controller would execute the primitive actions within the selected sub-routines until termination. These early works demonstrated that given a proper task decomposition, HRL can indeed learn policies at different tem-

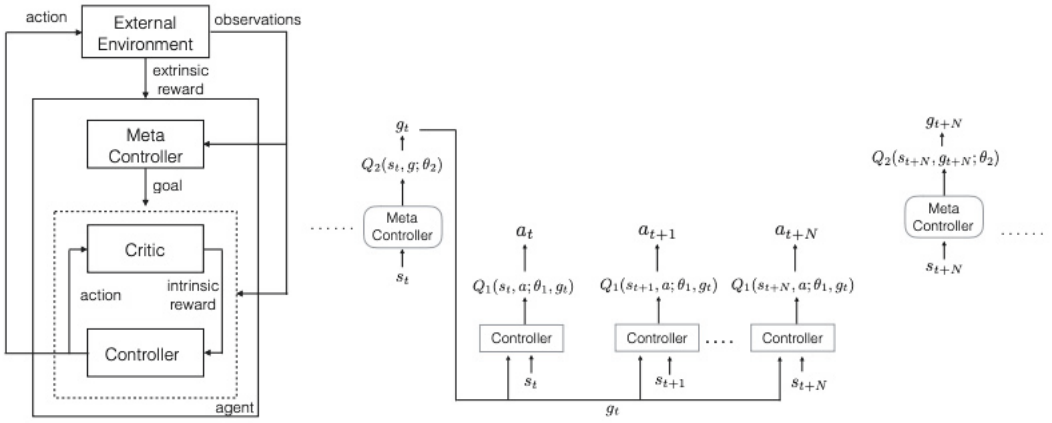


Figure 5.2 : Hierarchical reinforcement learning framework. The meta-controller generates a goal g_t every N^{th} time steps (e.g., $\mathbf{T} = N$ and $N > 1$), during which the controller executes a sequence of primitive actions $a_t \ldots a_{t+N}$ to achieve the assigned subgoal. The learning of the controller is guided by the intrinsic reward while the meta-controller is trained by the extrinsic reward(Kulkarni et al. 2016).

poral abstraction. This learning mechanism resembles the intelligence of biological organisms (Badre et al. 2010).

There are two main approaches to develop HRL algorithms. The first approach focuses on breaking down a policy into options or sub-policies. There are researches targeting at learning HRL under the option framework (Precup 2000; Sutton et al. 1999; Bacon et al. 2017). These learned options can also be used for transfer learning between similar MDPs (Konidaris and Barto 2007). The second approach to HRL is to identify intermediate sub-goaling states or landmarks in the environment, which are often referred to as *subgoals* (McGovern and Barto 2001a; Şimşek et al. 2005). In this work, we decided to follow the second approach of learning the representative sub-goaling states to guide the policy acquisition of the agents. The general working mechanism of HRL is shown in Figure 5.2.

Subgoal discovery is a standing challenge in HRL literature as this has to be accomplished online with learning the policy to act in the unknown environment. Different approaches have been proposed to address the subgoal discovery and learning such as using state clustering techniques to identify bottle neck regions in the

state space, or leverage domain heuristics (McGovern and Barto 2001b; Machado and Bowling 2016). Once the subgoals are discovered, HRL employs a multi-level decision-making process where a meta-controller learns a sequence of subgoals to assign to the controller. The controller learns to achieve those assigned subgoals via intrinsic rewards, or also known as *intrinsic motivation* (Kulkarni et al. 2016; Vigorito and Barto 2010). The design of the intrinsic reward is another open research question and there are many mechanisms of constructing a *good* intrinsic reward function (Aubret et al. 2019). Generally, a learned state representation or feature extraction function (e.g., $\phi(s_t)$) is used to measure the novelty of state transitions (Pathak et al. 2017b), which is translated to a *curiosity*-driven reward to encourage the agent to explore novel and unvisited areas of the state space. Under the HRL framework with meta-controller, the intrinsic reward is calculated as the distance between the current state feature s_t and the assigned subgoal s_g (Equation 5.1). As a result, the agent is guided towards visiting states that are similar to the assigned subgoals.

$$r_t^{intrinsic} = ||\phi(s_g) - \phi(s_t)||_2^2 \tag{5.1}$$

5.2 Problem formulation

In this work, we consider CybORG scenarios with hard exploration and sparse reward settings. In CybORG, the difficulty level of a scenario depends on multiple factors such as the number of actions and supporting parameters, the number of hosts in the network and how many subnets they are grouped into, and the placement of the sensitive asset represented by the flags.

Training on scenarios with a large number of hosts (e.g., 50 hosts) would require heavy computational resources to run the training and simulation (Figure 5.3). As a result, for the convenience of developing the algorithms, we create extra scenarios for the 18-hosts setting. We refer to the scenario presented in Figure 5.4 as a *medium*-level scenario and the one in Figure 5.5 as a *hard*-level scenario. The main difference lies in the position of the flag with a flag placed deeper in the network would present

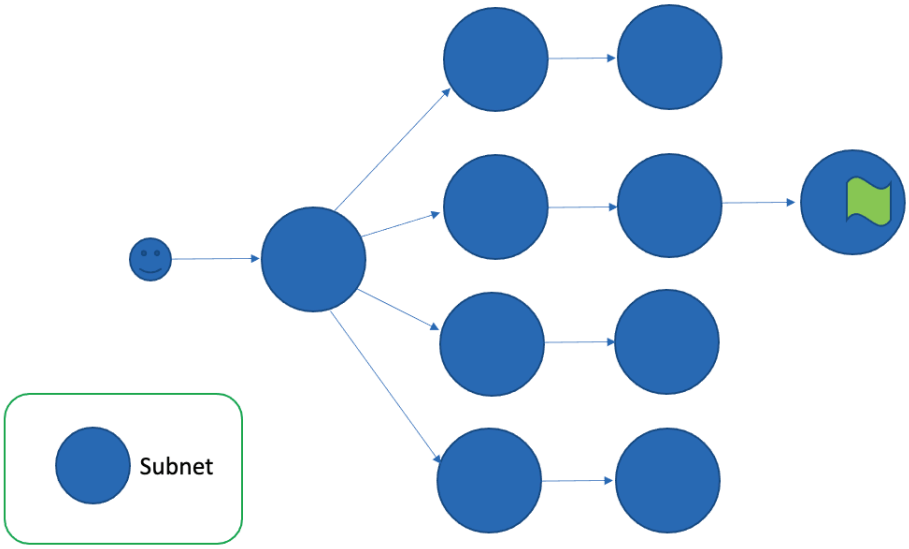


Figure 5.3 : A CybORG scenario configuration with 50 hosts organised into 10 subnets.

a *harder* setting, as it requires a more focused attack strategy to penetrate a deep network.

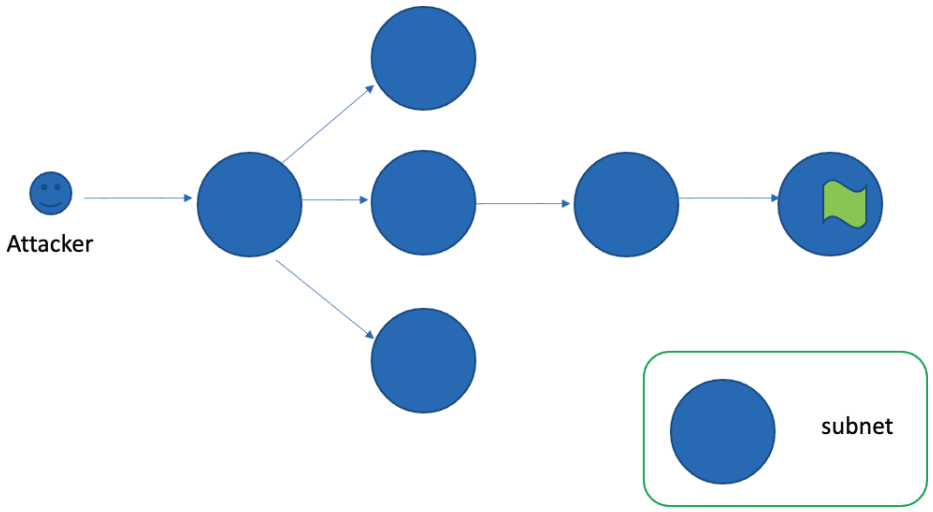


Figure 5.4 : A *medium* CybORG scenario configuration with 18 hosts organised into 6 subnets.

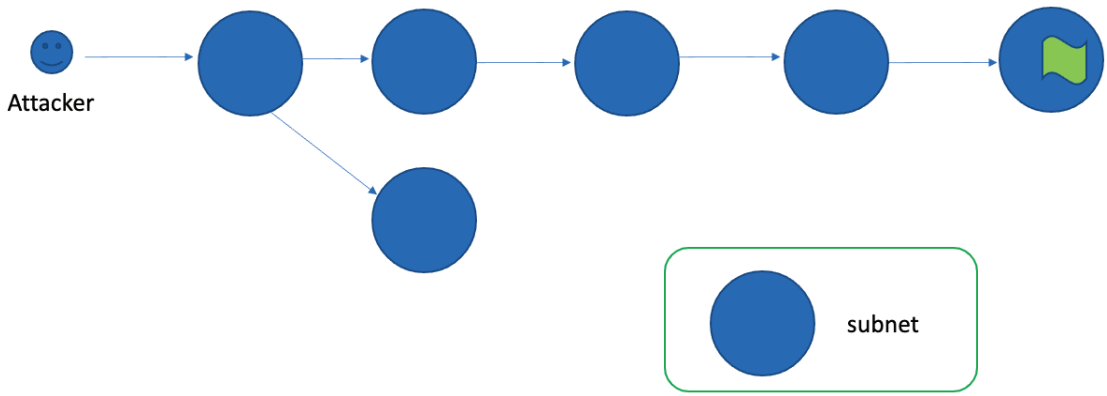


Figure 5.5 : A *hard* CybORG scenario configuration with 18 hosts organised into 6 subnets.

5.3 Methodology

We investigate the use of deep neural networks to assist the agents in learning a good state presentation. As mentioned in previous sections, a good state representation would be able to help the agents learn better in complex environment with sparse rewards. Common neural network architectures for learning state representation in RL use either an auto-encoder (Figure 5.6) or a variational auto-encoder (Figure 5.7). The latter can be useful for next state prediction by implicitly learning the environment dynamics while the former is useful for state dimensionality reduction. Moreover, the variational auto-encoder is more popular in continuous domains as it can be trained to learn the distribution of the states (Doersch 2016) by fitting the mean (e.g.,μ) and the variance (e.g.,σ) of the state distribution.

5.3.1 State representation learning using the auto-encoder architecture

To extract a reduced latent representation of the observation vector from the CybORG environment, a deep auto-encoder with neural network is used. The latent vector z_t is then fed as an input to the *Multi-Agent for Parameterised Action* architecture from the previous Chapter 4 (Figure 5.8).

The auto-encoder is useful for extracting a compressed latent representation

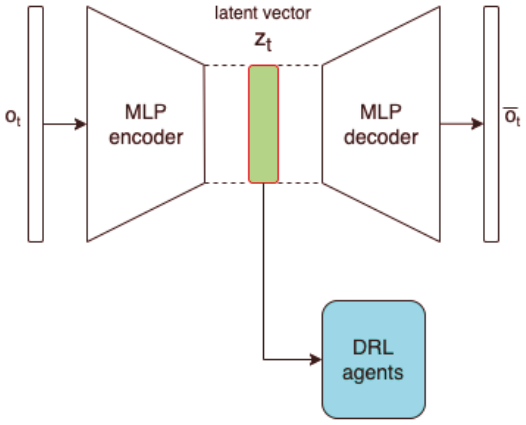


Figure 5.6 : An Auto-encoder for reduced state representation.

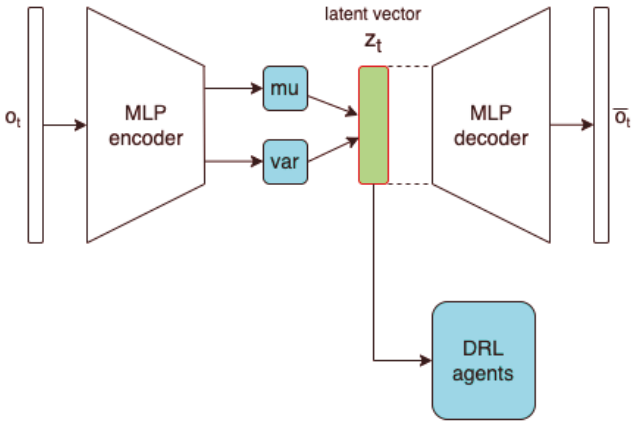


Figure 5.7 : A variational auto-encoder for learning state distribution.

which is possible to reconstruct the original states with minimum loss. This reduced vector helps the agent focus on only meaningful components of an observation during learning, hence improve training efficiency. The training process is separated into two phases wherein during the first phase, the DRL agents only follow the uniform random policy to explore and collect observations from the environment. These observations are then used to train the encoder-decoder block with the main loss function being the L_2 mean square error loss $L_2^{mse}(\tilde{s}_t, s_t)$ which measures the difference between the input observation and the reconstructed observation $\tilde{s}_t = D(E(s_t))$.

The performance of the auto-encoder is shown in Figure 5.9 with the top left

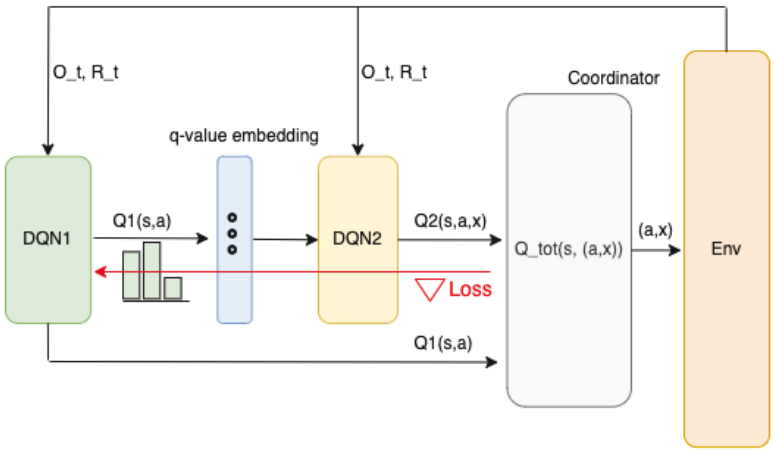


Figure 5.8 : Multi-Agent for Parameterised Action or MAPA from Chapter 4.

subplot showing the cumulative episodic return while the steps to capture the flag is shown on the bottom left subplot. The two curves on each subplots represent different runs with randomly chosen seeds. Other subplots describe loss statistics and action value estimates as the training proceeds. The auto-encoder was trained on roughly $300-500$ episodes of random exploration. Once trained, the fixed encoder is used to convert observation to feature vector and feed it as the input to *MAMA* for reinforcement learning. With the usage of an auto-encoder for latent vector extraction, the learning is sped up to convergence at around episode 800, whereas in previous *MAPA* training, the convergence signal starts at around episode 1200s. Therefore, using an auto-encoder is a good way to make learning more efficient by leveraging the reduced state representation for RL.

Figure 5.10 demonstrates the difference between with and without using auto-encoder to process the state vector. The green curve is the *MAPA* performance on 12 hosts scenario with the use of auto-encoder while the gray curve is the architecture without auto-encoder. The performance shown by the green curve is much better and more stable than the gray curve.

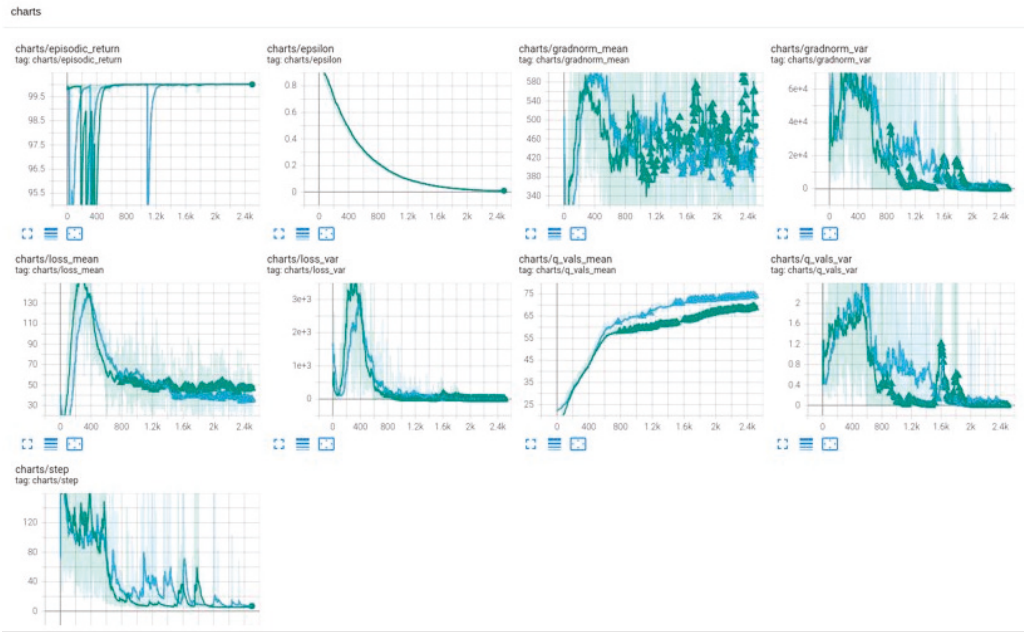


Figure 5.9 : The auto-encoder training conducted on 12-hosts scenario. From left to right, top to bottom: cumulative episodic returns, epsilon rate for exploration, mean of the gradient norm, variance of the gradient norm, mean of the loss function, variance of the loss function, mean of the estimated state-action values, variance of the estimated state-action values, the number of steps to capture the flag.

5.3.2 State representation learning with successor representation

In the previous section, the deep auto-encoder has shown to improve learning efficiency. From the state representation learning perspective, however, the encoder merely acts as a table lookup for state embedding. This is a barrier to scaling up the application of RL to more complex environment where it may be intractable to learn all possible states during exploration phase. Additionally, modern reinforcement learning literature has introduced different approaches for learning a better state representation using an architecture known as variational auto-encoder or VAE, which is helpful in learning the distribution of the latent representation, instead of just learning the reduced latent embedding. We also have experimented with VAE in this work but found no encouraging results. In CybORG, the observation vector contains discrete and binary values instead of continuous real numbers,

Figure 5.10 : Performance comparison between with and without auto-encoder. The green curves are the MAPA performance with auto-encoder while the gray curves show the MAPA performance without auto-encoder

.

whereas VAE has been known to be widely adopted in continuous domains such as game screen, images etc...

Introduction to successor feature

Recently, another state representation learning technique called successor representation (for tabular domain) (Dayan 1993) or successor feature (when used with function approximation) (Barreto et al. 2017) is revived and adapted to various RL problems.

The successor representation (SR) is defined as the expected discounted future state occupancy. It is an implicit method to take into account the future states

visitation (Equation 5.2).

$$\begin{aligned} M(s, s', a) &= \mathbb{E}[\sum_{t=0}^{\infty} \gamma^t \mathbb{1}[s_t = s'] | s_0 = s, a_0 = a] \\ &= \mathbb{1}[s_t = s'] + \gamma\, \mathbb{E}[M(s_{t+1}, s', a_{t+1})] \end{aligned} \tag{5.2}$$

The SR decouples the state-action value function from the reward prediction component. The Equation 5.3 illustrates this decoupling approach in estimating the state-action value function.

$$Q^{\pi}(s, a) = \sum_{s' \in \mathcal{S}} M(s, s', a) R(s') \tag{5.3}$$

When used with neural network as the function approximator, SR is referred to as the successor feature or SF, and is described as $\psi(s, a; \theta)$ with θ being the network parameters.

$$\psi(s, s') = \mathbb{E}_{s' \sim P, a \sim \pi}[\sum_{t=0}^{\infty} \gamma^t \mathbb{1}(s_t = s') | s_0 = s] \tag{5.4}$$

SF can also be trained following temporal difference learning similar to the deep q-learning procedure:

$$\hat{\psi}(s_t, :) = \hat{\psi}(s_t, :) + \alpha[\mathbb{1}(s = s_t)] + \gamma \hat{\psi}(s_{t+1}, :) - \hat{\psi}(s_t, :) \tag{5.5}$$

$\psi(s_t)$ is parameterised by θ and can be estimated using sampled episodes:

$$\psi(s_t; \theta) = \mathbb{E}[\phi(s_t) + \gamma \psi(s_{t+1}; \theta)] \tag{5.6}$$

The benefits of learning an SF representation are two folds. First of all, SF stands in between the model-based and model-free approaches in reinforcement learning. The SF maintains an implicit state dynamics of the environment. That means a good SF can be used to cluster state space in such a way that states with similar future state visitation can be grouped into a cluster. Secondly, the reward component is separated from the learning of the action value function, makes it less dependent on the reward signal from the environment.

Successor feature training process

Following Barreto et al. (2017), the training process of successor feature involves three components: a state feature extractor, a reward predictor, and a SF representation. The state feature extractor is the training of the auto encoder-decoder as in Equation 5.7.

$$L^a = (D(E(s_t) - s_t)^2 \tag{5.7}$$

where $D(.)$ and $E(.)$ are the decoder and encoder modules. This is the usual loss function for training the auto-encoder. The reward predictor is trained by minimising a regression loss as shown in Equation 5.8.

$$L^r = (R(s_t) - \phi(s_t)w)^2 \tag{5.8}$$

where $\phi(s_t) = E(s_t)$ and w is the weight of the predictor network.

The SF representation is trained using temporal difference learning. The loss function is shown in Equation 5.9.

$$L^m = \mathbb{E}[(\phi(s_t) + \gamma\psi^{target}(\phi(s_{t+1}), a') - \psi(\phi(s_t), a))^2] \tag{5.9}$$

where $\psi^{target}(.)$ is the lagged target network. This is similar to the target q-network used in training deep q-learning, and $a' = \text{argmax}_a\, \psi(s_{t+1}, a)w$.

For this work, we have modified the loss equation L^m to incorporate the training of SF into QMIX. By taking advantage of the relationship between action value function and SF and multiplying both sides of Equation 5.9 with the weight of the predictor network, the DQN loss function is rewritten as:

$$L^m = \mathbb{E}[(\phi(s_t) * w + \gamma\psi^{target}(\phi(s_{t+1}), a') * w - \psi(\phi(s_t), a) * w)^2] \tag{5.10}$$

where $\phi(s_t) * w = R(s_t)$ and $\psi(\phi(s_t), a) * w = Q(s, a)$.

The entire architecture is represented in Figure 5.11, wherein we have introduced two extra branches containing the decoder block and the successor feature block.

There are different training procedures in the literature, and it is more or less a trial and error process for each application. However, it is important to optimise

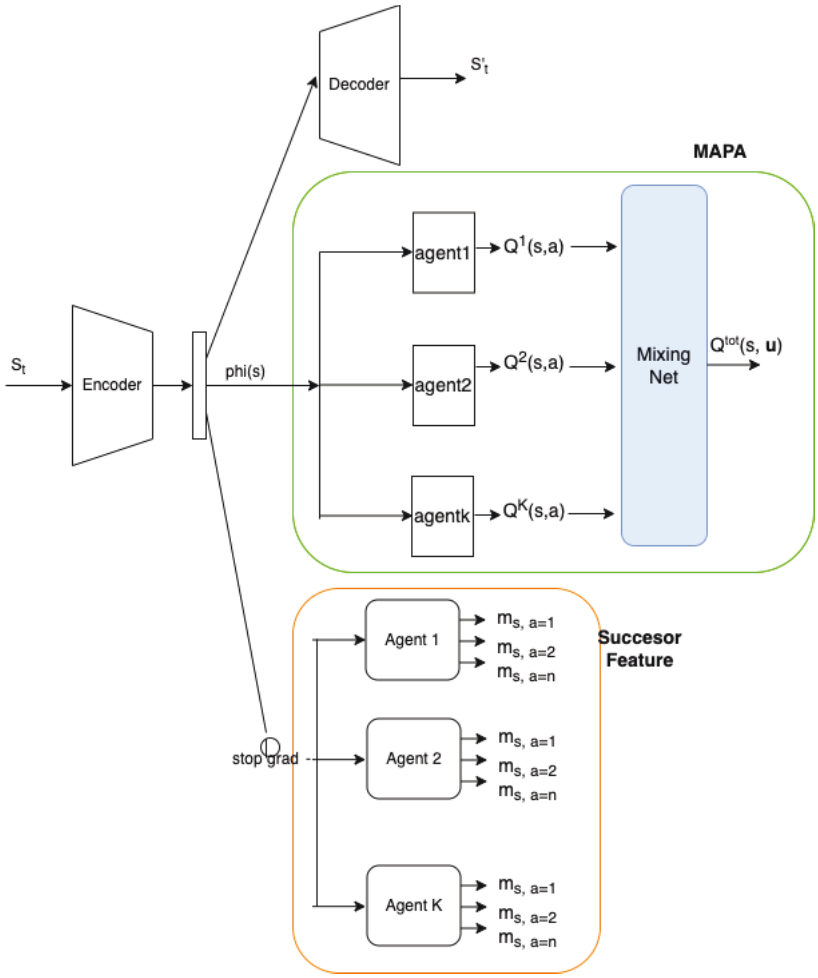


Figure 5.11 : Successor feature with MAPA architecture

the L^a and L^r separately from L^m. We trained alternatively the combined $L = L^a + L^r$ and L^m, and used separate episode roll-outs to train the encoder and reward predictor until convergence before training the SF network. To our experiences, separating the training process gives a better performances. We consider the process of training the encoder-decoder and the SF network as an *offline* process, as the training data can be collected from previous policies. In our work, we just use random policy to train the encoder-decoder network and the SF network. Once we have a well-trained encoder and SF network, the MAPA can then be trained following an *online* manner using the learned latent vector to update its policy.

5.3.3 Successor feature with hierarchical reinforcement learning

The learned SF from exploratory trajectories are clustered into k sub-groups, which we consider as sub-goals in the HRL framework. These clusters generally contain states with similar SF representation, which mean they have similar temporal abstraction. The learning process is described in Figure 5.12. Initially, the state feature extraction function (e.g., $\phi(s)$) and the SF function (e.g., $\psi(s)$) are trained from exploratory trajectories collected from uniform random policy. The use of random policy serves the purpose of learning the environment dynamics via the visited states. It is also useful in CybORG as we do not need to take into account the various action complexities during this initial phase. The trained SF function is used to transform the input states into latent representation $\psi(s)$ which are then clustered into groups using a clustering algorithm called spectral clustering (Damle et al. 2016). The centroids of these clusters are treated as sub-goaling states. These clusters represent states with similar successor representation or temporal abstraction, since SF represents the future visitation of each states. This initial training process are done offline, in a sense that they rely only on off-policy roll-outs. The online training process integrates the meta-controller and controller to update the agents' policies on the environment.

5.3.4 Algorithm pseudocodes

This section specifies the algorithmic pseudocodes for the 2 sub-routines of training the encoder-decoder block in Algorithm 3, and the SF clustering in Algorithm 4.

Algorithm 3 Encoder training

Input: Agents with uniform random policy

Output: Encoder module $\Phi(.;\theta_e)$

1: Do random walks in the environment.

2: Collect and store the experiences $e = (s, a, r, s')$ into replay buffer $\mathcal{B}$.

3: Train the Encoder-Decoder module $\mathcal{E}$ using samples from $\mathcal{B}$ until convergence.

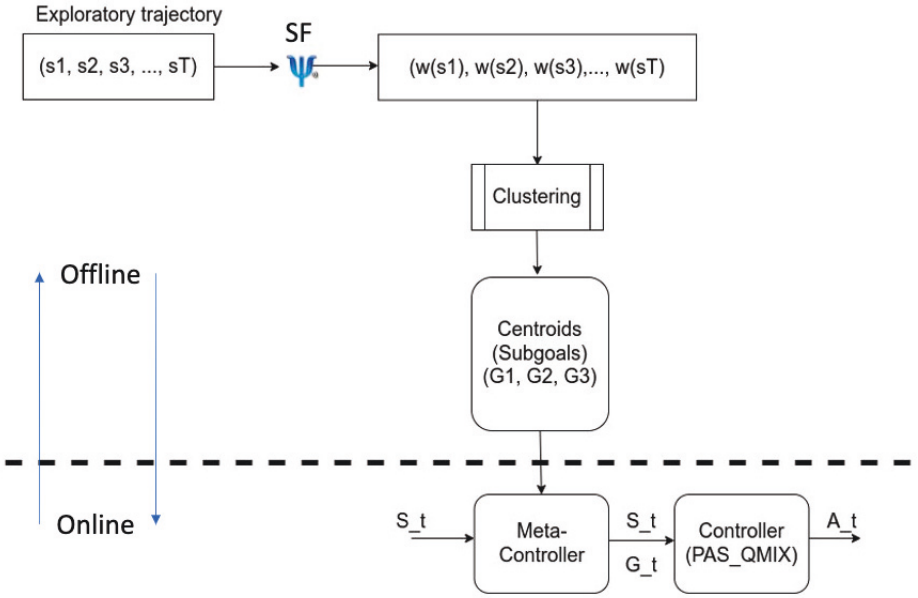


Figure 5.12 : Hierarchical reinforcement learning with successor feature. Samples of exploratory trajectories are used to train a good SF representation. They are then clustered into k groups. The centroids of these clustered are treated as sub-goals in HRL. The meta-controller learns to assign a sequence of subgoals to the controller for execution.

5.4 Experiments

5.4.1 CybORG 2021

The following experiments are performed on the same environment as described in the previous Chapter 4. The differences are in the configurations of the tested scenarios. A simplified diagram of CybORG is shown in Figure 5.13. A scenario is configured by the YAML files which allow the creation of different number of hosts and subnets. Different number of hosts can be connected within each subnet. However for simplicity, we keep the same number of hosts per subnet across the tested scenarios. The red agent is exposed to a minimum view of the network at the beginning of each game, wherein it can only access to the public subnet. The observation vector contains an array of binary values, each of which reflects the

Algorithm 4 Subgoal discovery

Input: Replay Buffer $\mathcal{B}$, SF function $\Psi(.;\phi)$

Output: Subgoal list $\mathcal{G}$

1: **for** experiences $e = (s, a, r, s') \in \mathcal{B}$ **do**
2: Collect SF of s: $\psi(s) = \Psi(s;\phi)$
3: **end for**
4: Perform Spectral Clustering on the collected SF
5: Store the cluster centroids to the subgoal list $\mathcal{G}$

discovered host id, subnet id, available port id etc... The action space consists of a number of main attack types that can be performed by the red agents. Some of this were shown in Figure 2.14. It also has different sets of parameters which will be used by each of the attack type.

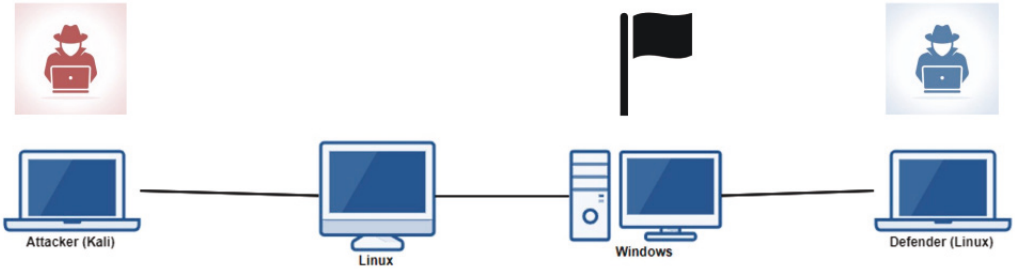


Figure 5.13 : CybORG scenario (Standen et al. 2021)

5.4.2 Neural network architecture

Each agent in our experiments uses a 3-layer Dueling DQN network architecture (Figure 4.5). However, the proposed architecture MAPA can be used together with any variants of DQN algorithms such as prioritised experience replay, double DQN etc.. The performance of MAPA is compared against a typical single DRL agent with non-coordinated and independent branching sub-network for attack action and parameters. All agents share similar neural network architecture with the first two

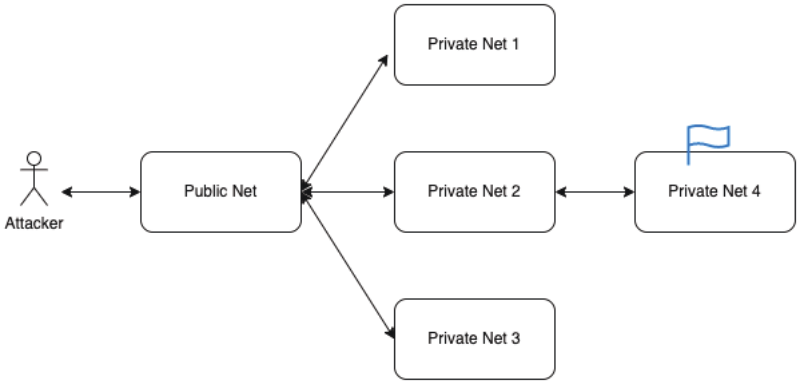


Figure 5.14 : An example of 15-hosts scenario

Table 5.1 : Hyper-parameter settings

Hyper-parameter	Value
Replay buffer size	1000000
Network size	256
Decay rate	0.998
Discount factor	0.99
QMIX hyper-net 1	32
QMIX hyper-net 2	32
Learning rate	0.0001
Learning frequency	20
GRU hidden dim	128

layers have 512 and 256 neurons to evaluate the advantage functions while the last layer has 64 neurons to compute the action values. All the training are conducted using a single RTX6000 GPU.

Table 5.1 displays the values for the hyper-parameters used in the proposed algorithm.

5.5 Results

We tested the initial implementations on 15 and 18-hosts scenarios to validate the feasibility of the approach. The following performances are collected from the experiments on 15-hosts scenario that showcase the clear advantage of using HRL with subgoals extracted from the successor features.

Figure 5.15 and 5.16 present the loss reduction during the training of the *offline* phase using trajectories of random exploratory policy. The losses were collected for each of the 4 agents in the MAPA architecture. The top row of each figure shows the average of the loss values while the second row displays the variances of the losses. For both metrics we should observe a trending down representing reduced loss values.

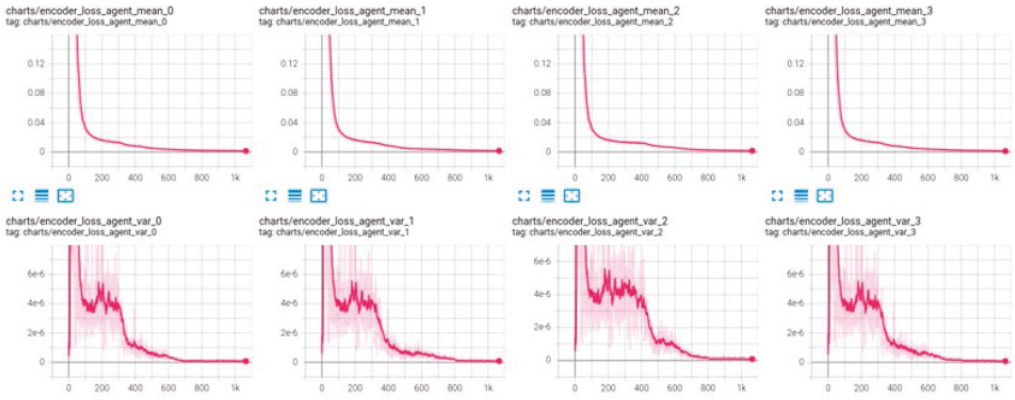


Figure 5.15 : Deep auto-encoder training losses on 15-hosts scenario. Each column contains the loss metrics for each of the 4 agents. Top row shows the average or mean of the loss while the variances are shown in the second row.

In order to examine the learned SF, we rolled out random trajectories for 150 time steps on the 15-hosts scenario. For each trajectory we collected the observations as well as its SF representation. For each pair of SF vectors, we calculated the cosine distance wherein a value of 0 means they are identical and larger values reflect the dissimilarity between the pairs. The heat map is shown in Figure 5.17 illustrating the correlation between experienced states in the trajectory.

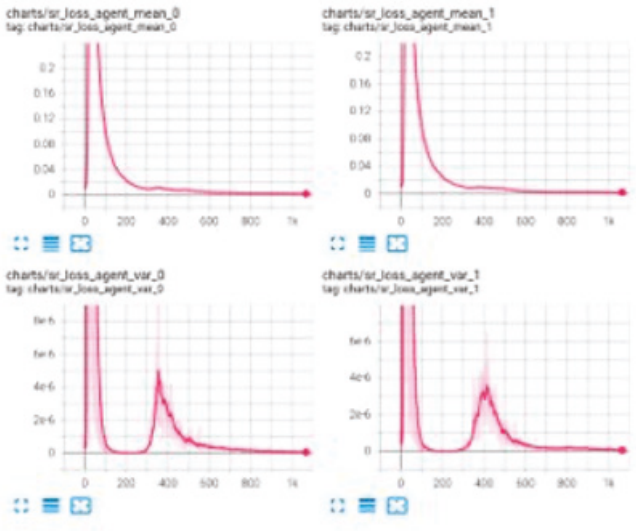


Figure 5.16 : Successor feature training losses on 15-hosts scenario. Each column contains the loss metrics for each of the representative 2 agents. Top row shows the average or mean of the loss while the loss variances are shown in second row.

It should be noted that due to the stochasticity nature of CybORG, it is common for consecutive observations to be similar or even identical. The action taken failed to transition the current state to the next state, or the underlying state transition is not reflected in the observation vector. This explains some of the long blobs of black color in the above heat map. It is important to point out the existence of black blobs for distant observations, which are multiple time steps away from each other. This is because for any pair of observations o_t and o_{t+k}, it is possible that in another trajectories they would share similar future states, or they would be consecutive to each other in another episode.

We use *networkx*[†] package to visualise the graph of unique observations in a random trajectory and assign to each observation a label extract from performing spectral clustering on the collected SF. The self-connection represents the fact that the scenario does not transition to next state for certain actions. The number of cluster is a hyper-parameter which controls how many dissimilar clusters we would

[†]https://networkx.org/

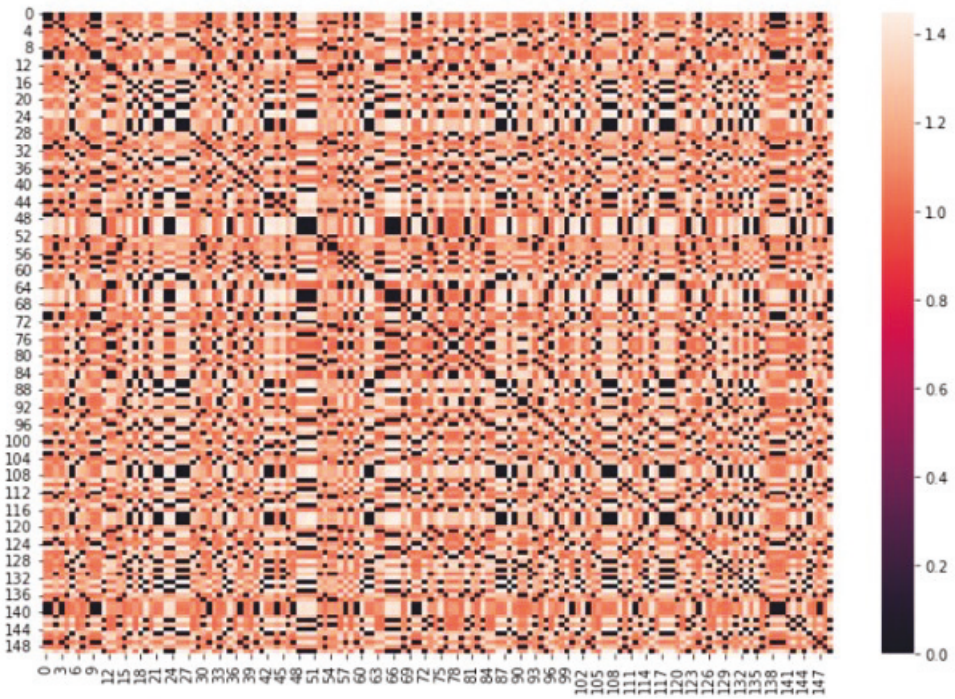


Figure 5.17 : The heat map of the SF matrix for a sampled random trajectory. Small values represent similar SF representations while larger values represent variant degree of dissimilarity between any SF pair of states.

like to observe. The observation graph is shown in Figure 5.18. The graph confirms our understanding of SF as the colors do not concentrate on consecutive observations but slightly scatter across the trajectory.

Finally we performed the comparison between the use of HRL and the original *MAPA* on a special configuration of 15-hosts wherein all the 5 subnets are sequentially connected and the flag was hidden in the innermost subnet (Figure 5.19 and Figure 5.20). This is particularly challenging because the DRL agents need to search for a very special instance of the attack strategy which allows it to penetrate the network subnet by subnet. Random actions are not very useful in deep subnet connections since it is easy to get stuck in local optima and failed to expand the compromised network. The gray curves are the performance metrics of the original *MAPA* architecture.

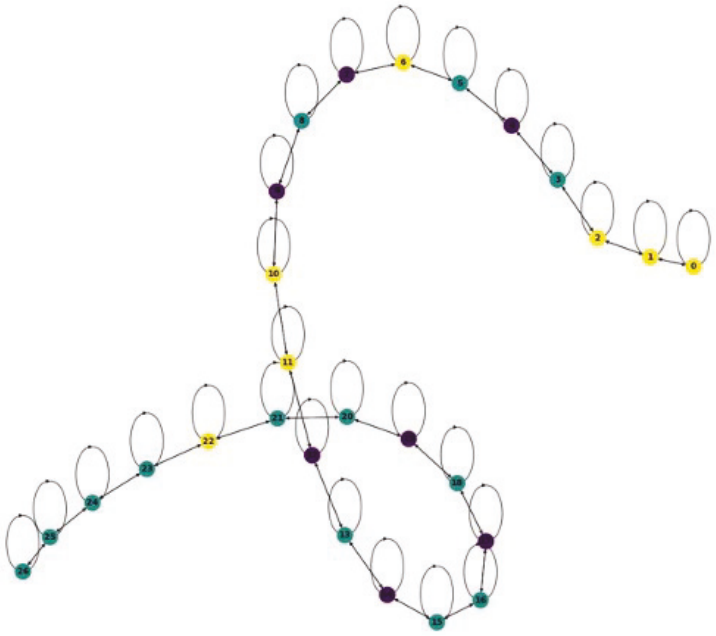


Figure 5.18 : CybORG observation transition graph

It is interesting to see the *MAPA* was able to achieve pretty good performance early on in the training as random exploration still takes place most of the time. The performance of the updated policy started collapsing at later phase in the training because it failed to search for the particular strategy that allowing itself to penetrate through multiple subnets. The agent's policy must be able to take into account long term consequences in these scenarios, in other words, the state representation must span multiple time steps for the agents to avoid getting stuck in local optima.

The HRL approach struggles to learn at the beginning because it needs to train the two layers of decision making, namely the meta-controller and the controller. However, after approximately 2500 episodes, it started picking up the attack policy and was able to maintain the policy until the end of the training. This reflected on both the cumulative rewards, as well as the number of steps to capture the flag.

The following figures demonstrate the performance of the MAPA with HRL approach on the *hard* setting of the 18 hosts scenario. Figure 5.21 shows the ac-

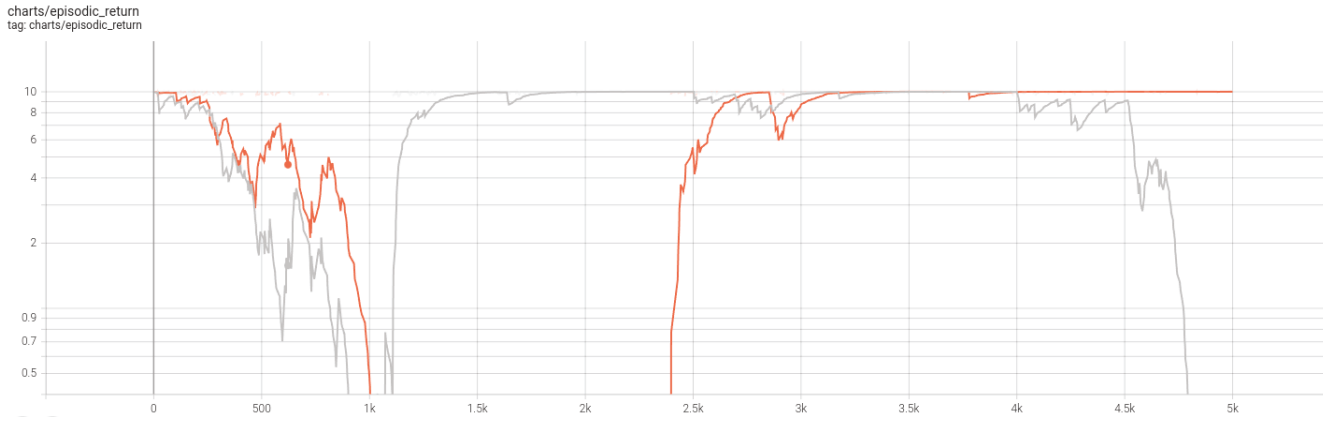


Figure 5.19 : Cumulative rewards on 15 hosts scenario. The orange curve displays the performance of the HRL approach and while the gray curve is the performance of the original MAPA without HRL.

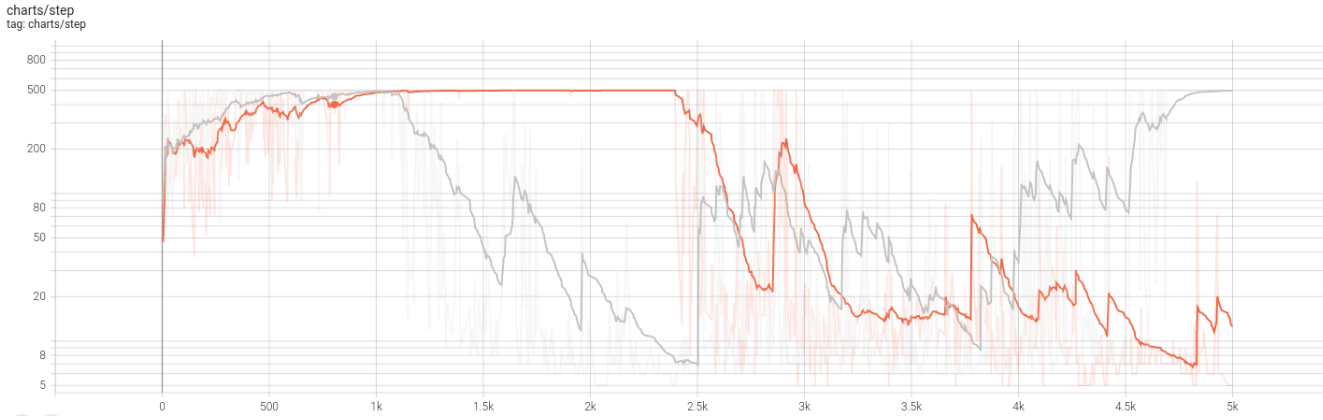


Figure 5.20 : Steps to finish on 15 hosts scenario. The orange curve displays the performance of the HRL approach and while the gray curve is the performance of the original MAPA without HRL.

cumulative scores (the left panel) and the number of required steps to capture the flag (the right panel) on the *hard*-level setting of 18 hosts. This is a very challenging scenario, but the MAPA with HRL approach manages to converge to optimal returns and steps towards the end of the training. Figure 5.22 presents the training loss of the SF representation from the 4 agents during training.

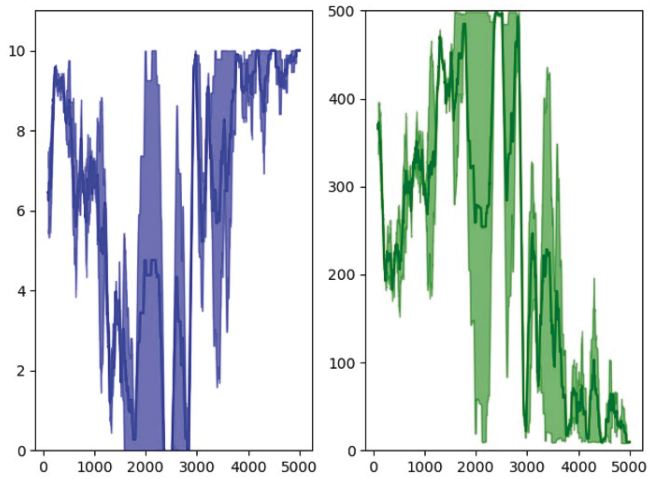


Figure 5.21 : Cumulative returns (left) and number of steps to capture the flag (right) on the *hard*-level 18 hosts scenario using the MAPA with HRL approach.

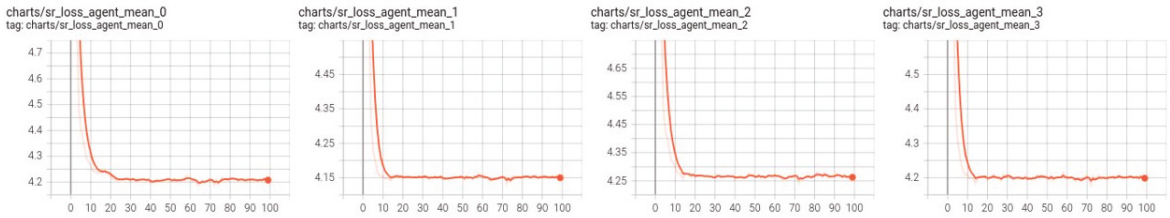


Figure 5.22 : The SF training losses of the 4 agents using the MAPA with HRL approach.

Figure 5.23 summarises the cumulative rewards achieved during the training of the method. The HRL approach shows solid performances across 3 scenario configurations.

5.6 Chapter summary

In this work, we addressed the challenge of state representation learning in CybORG, a simulated cyber environment. We incorporated both dimensionality reduction with deep auto-encoder as well as using successor feature representation as an auxiliary task which allow the latent vector to encoder partly the environment dynamics. HRL

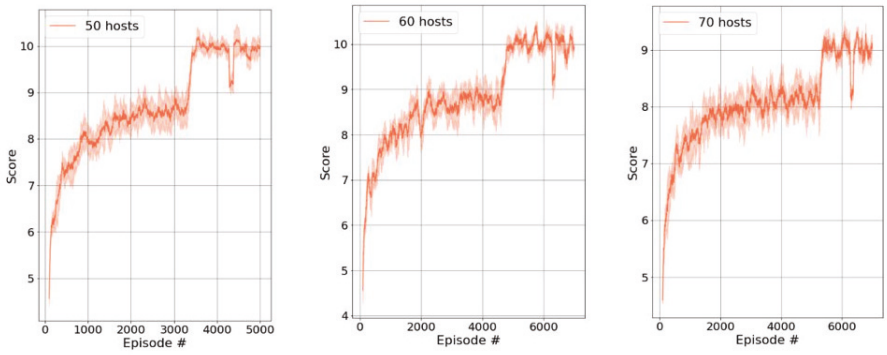


Figure 5.23 : Cumulative rewards during training for scenarios with 50, 60 and 70 hosts.

with meta-controller/ controller framework is used wherein subgoals are extracted from the learned SF representation. Current results supported the feasibility of the approach and we are working on extending this approach to more complex scenarios. Eventually, we hope this will not only achieve better convergence but provide an interpretable method on understanding the attack strategy selection of the DRL agents.

Chapter 6

Conclusion

The final chapter revisits the main objectives of this study and summarises the main contributions. We give brief discussions on the proposed approaches and suggest future work to improve the performances in developing AI for cyber security applications.

6.1 Overview

This doctoral dissertation looked at the applicability of deep reinforcement learning and deep neural network architectures in cyber security related applications. In particular, we examined DRL algorithms in developing and enhancing the performances of autonomous penetration testing. Current pen-testers in the industry are heuristics-based which require well-trained professionals to execute attack plans. It is desirable for both businesses and security providing company to leverage advances in AI to develop an artificial intelligence-based automatic pen-tester. Our research is the first to develop non-conventional DRL algorithms to directly address the challenges posed by a well-developed cyber simulator called *CybORG*. The central idea of this research is whether we can develop an AI-enabled pen-tester with the minimum amount of cyber-related domain knowledge, and it is capable of dealing with the dynamic cyber security networks. This objective motivates the use of deep reinforcement learning which is the learning paradigm that requires neither supervised learning dataset nor heuristic-based rules. The learning of DRL agents basically is

an non-linear optimisation process. Given a proper problem formulation, we expect the DRL agent capable of dealing with different challenges posed by autonomous PT such as having large discrete action space, multinomial parameterised action space and complex state representation learning.

6.2 Summary of contributions

6.2.1 Cascaded Reinforcement Learning Agents for large discrete action space

In chapter 3, we proposed a novel framework for dealing with the particular challenge of large and discrete action space by reformulating the single RL agent problem with a large action space $\mathcal{A}$ into a multi-agent learning with action space decomposition. This solution is not only solving the large discrete action space but also presenting a scalable approach to the particular challenge. As the size of the action space increases, the number of agent (or the computational complexity) only increases following a logarithmic scale (e.g $log_b|A|$), which is sublinear. The proposed architecture proved to be stable on various settings of scenario's complexity and it was able to learn the optimal attack policy to capture the hidden assets. It is built on top of the dueling deep q-network which is a value-based method in RL, providing the advantage of being sample efficient.

We developed an elegant solution to algebraically decompose the large action space representation into smaller sub-spaces using an adaptation of the segment tree data structure. Each level of the action selection is represented by a DRL agent which eventually lead to a sublinear complexity as the number of required agents would not be more than the height of the tree. The agents are cooperatively trained using an adapted mechanism from MARL called QMIX, which is a *MixingNet* with hyper-networks. This centralised and cooperative training allow each individual agent to consider other agents' action during learning, which is crucial in this application as the final exercised action is the combination of all the individual actions.

The proposed method, named CLRA, demonstrated its superiority, in terms of convergence and stability, to conventional DRL algorithms such as dueling deep q-learning and its variants, which are normally used as a proof-of-concept solution in current cyber security application research.

6.2.2 Multi-Agent for Parameterised Action in CybORG

The focus of Chapter 4 is on another action space representation called *parameterised action space*. Expressing the action space as a large and discrete set of action identifiers is not scalable to highly complex scenarios in terms of practical numerical computation since most programming languages have an upper bound on the representable integers in computer. As a result, *CybORG* v2021 has utilised another form of action representation where a final attack action comprised of multiple action components, for example attack type, host id and subnet id. The main difference in this parameterised action space representation is that each attack type can have variable number of supporting parameters, which is not the case in many RL literature in this setting according to the author at the time of this work*.

We leverage the learned experiences from the previous chapter and utilise the multi-agent framework to select sub-action components. This is the first work to successfully keep the semantic dependency between the main attack type and its support parameters and still demonstrate stable convergences during training on a variety of scenario configurations. Using *Mixing network*, invalid action masking and code optimisation, the proposed solution has shown much better performances compared to current baseline approaches with independent network heads. The empirical results of this chapter are important and substantially promising to be incorporated into the development of cyber related applications since this action space representation is common in the domain, and we hope this would lay the foundation for future works in this area.

*RL is a fast-moving research area, new papers are published on a weekly or even daily basis and on many platforms.

6.2.3 State representation for effective learning in complex environment with sparse reward

The penultimate chapter 5 investigates and presents our initial attempts in learning an effective state representation learning module for CybORG. While the majority of current RL literature used variational auto-encoder (VAE) to learn the latent state distribution, it is not transferable to pen-tester applications due to the complex, discrete and partially observable nature of the problem. Noisy perturbation and sampling process from VAE would change the underlying state representation in CybORG; which is not a typical problem for other continuous domains such as images, or robotics simulation.

We incorporated deep auto-encoder for state dimensionality reduction and the successor feature representation for encoding environment dynamics. The learned latent representation achieved better and more stable performance thanks to the reduced and condensed latent vector. The extracted latent information does not only detect the critical bits from the original observation vector but also encode the future state visitation. This acts as a non-reward estimation of the future trajectory.

Furthermore, we experimented with using the learned latent representation to perform sub-goal discovery with HRL wherein a meta-controller/ controller framework is used to concurrently learn a goal driven policy and a low-level primitive policy. Even though there is limited experiment being conducted with this approach, early results are encouraging and we are working on extending the sub-goal discovery with successor feature to other benchmark environments such as the *atari* environments (Mnih et al. 2013).

6.3 Limitations and Future works

The current research is the first stepping stone towards developing AI-enabled automatic penetration testers. Even though initial successes proved the feasibility of our works, there are many challenges that need to be resolved for successful implementation in practice. The operational deployment of the proposed solutions is outside the scope of the thesis. We highlight some potential issues and propose methods

that future research should focus on.

6.3.1 Variable state space and action space

The state space and the parameterised action space in actual scenarios may have temporal dependency where the range of the attack type or supporting parameters appear at different time steps. A compromised host can be reset back to its initial state or the system introduces new hosts to the network. These changes affect the information received by the agents and impact its action space, violating the MDP assumption. As a result, more sophisticated deep learning algorithms are needed to deal with variable and changing input vectors or maintain input and output permutation invariant. As part of an on-going collaborative research, we are looking at the applicability of the *Set Transformer* (Lee et al. 2018) and the newly introduced *Perceiver* from Deepmind (Jaegle et al. 2021) to improve both the state and action space representation learning.

6.3.2 Adversarial cyber operations

In order to improve the defence strategy, we would need to incorporate the attack and defense policies in a single setting. Adversarial training in reinforcement learning is a promising research direction and interesting works are being conducted on this front (Pattanaik et al. 2017). It is well-known for deep learning in general and reinforcement learning in particular to be brittle against adversarial attack (Pinto et al. 2017). Especially in reinforcement learning where any unobserved change to the environment dynamics would make DRL agents converge to sub-optimal policy. DSTG Australia have organised an on-going challenge called *cage challenge* (cag 2022) to encourage world-wide practitioners and researchers to address the adversarial setting in cyber operations.

Our current works in both the action space and state space representation learning are based mainly on empirical works. We need further efforts on theoretical modelling, particularly to cyber security domains where the nature of the inputs and outputs are significantly different from typical the DRL settings.

6.3.3 Human-machine collaboration

Cyber security is a critical domain that can potentially undermine national interests or security. As a result, there is no single approach which relies solely on algorithms or AI in general (Chen et al. 2019). Kantchelian et al. (2013) stated that any useful and practical security systems using AI-enabled algorithms must engage with the human operators or experts in its operation. In our work, we indirectly incorporate human knowledge by structuring the reward function and leveraging valid action masking for the DRL agents to exploit useful information. In critical circumstances, human authorities have to make judgements or intervene based on the output of the AI system. Future deployment of AI-enabled autonomous penetration testers have to go through rigorous examination from domain experts and specialists, and should be used only as an assistive method to any defence systems to mitigate the risks of having catastrophic accidents.

Appendix A

Appendix

A.1 Appendix: CRLA Network Comparison

Figure A.1 shows a comparison between the network architectures of a single-agent Dueling DQN and the CRLA network.

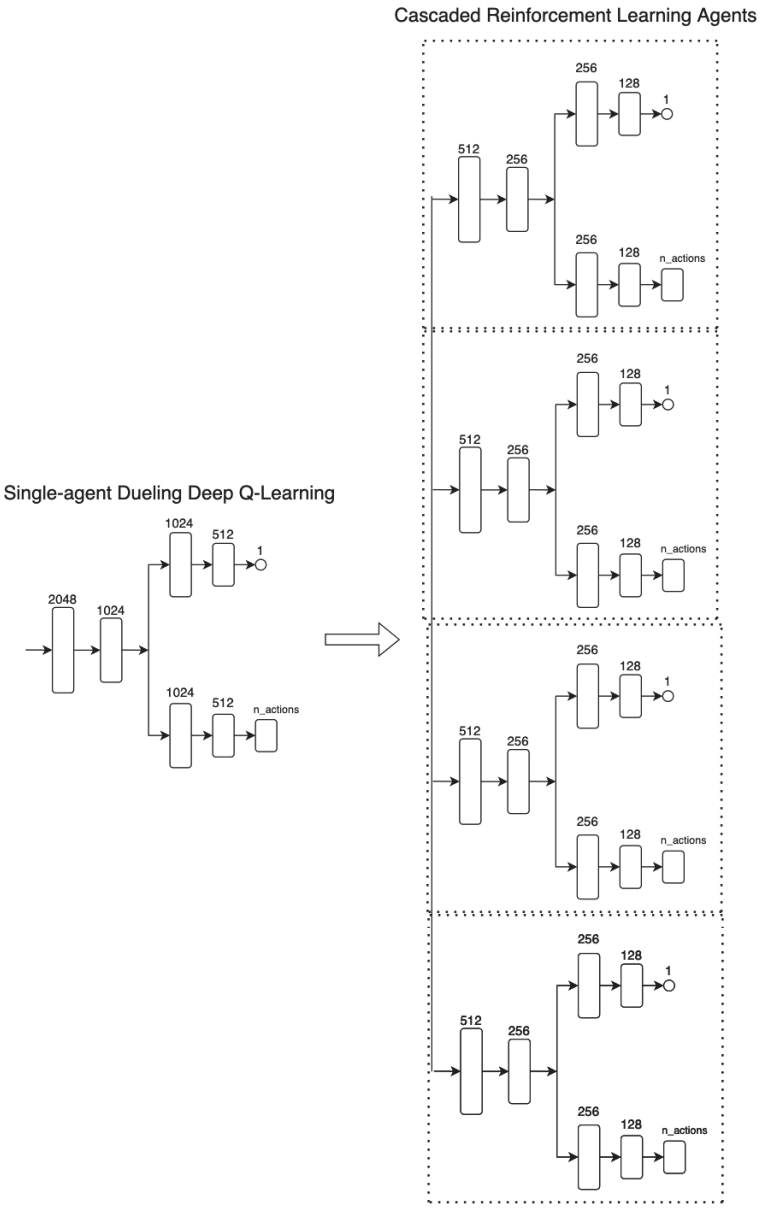


Figure A.1 : Neural network architectures comparison between the single-agent duelling deep Q-learning and the proposed cascaded reinforcement learning agents approach.

Bibliography

2022, 'Cyber autonomy gym for experimentation challenge 2', https://github.com/cage-challenge/cage-challenge-2, created by Maxwell Standen, David Bowman, Son Hoang, Toby Richer, Martin Lucas, Richard Van Tassel, Phillip Vu, Mitchell Kiely.

Ahilan, S. & Dayan, P., 2019, 'Feudal multi-agent hierarchies for cooperative reinforcement learning', *CoRR*, vol. abs/1901.08492.

Anderson, H., 2003, 'Introduction to nessus', *SecurityFocus Printable INFOCUS*.

Andriotis, C. & Papakonstantinou, K., 2019, 'Managing engineering systems with large state and action spaces through deep reinforcement learning', *Reliability Engineering & System Safety*, vol. 191, p. 106483.

Applebaum, A., Dennler, C., Dwyer, P., Moskowitz, M., Nguyen, H., Nichols, N., Park, N., Rachwalski, P., Rau, F., Webster, A. & Wolk, M., 2022, 'Bridging automated to autonomous cyber defense: Foundational analysis of tabular q-learning', *Proceedings of the 15th ACM Workshop on Artificial Intelligence and Security*, AISec'22, Association for Computing Machinery, New York, NY, USA, p. 149–159, <https://doi-org.ezproxy.lib.uts.edu.au/10.1145/3560830.3563732>.

Arulkumaran, K., Deisenroth, M. P., Brundage, M. & Bharath, A. A., 2017, 'Deep reinforcement learning: A brief survey', *IEEE Signal Processing Magazine*, vol. 34, no. 6, p. 26–38.

Arulkumaran, K., Dilokthanakul, N., Shanahan, M. & Bharath, A. A., 2016, 'Classifying options for deep reinforcement learning', *CoRR*, vol. abs/1604.08153.

Aubret, A., Matignon, L. & Hassas, S., 2019, 'A survey on intrinsic motivation in reinforcement learning', *CoRR*, vol. abs/1908.06976.

Bacon, P.-L., Harb, J. & Precup, D., 2017, 'The option-critic architecture', *Proceedings of the AAAI Conference on Artificial Intelligence*, , vol. 31.

Badre, D., Kayser, A. S. & D'Esposito, M., 2010, 'Frontal cortex and the discovery of abstract action rules', *Neuron*, vol. 66, no. 2, pp. 315–326.

Baillie, C., Standen, M., Schwartz, J., Docking, M., Bowman, D. & Kim, J., 2020, 'Cyborg: An autonomous cyber operations research gym', *arXiv preprint arXiv:2002.10667*.

Banerjee, B., Lyle, J., Kraemer, L. & Yellamraju, R., 2012, 'Sample bounded distributed reinforcement learning for decentralized pomdps', *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence*, AAAI'12, AAAI Press, pp. 1256–1262, <http://dl.acm.org/citation.cfm?id=2900728.2900906>.

Barreto, A., Dabney, W., Munos, R., Hunt, J. J., Schaul, T., van Hasselt, H. P. & Silver, D., 2017, 'Successor features for transfer in reinforcement learning', *Advances in neural information processing systems*, vol. 30.

Barto, A. G. & Mahadevan, S., 2003, 'Recent advances in hierarchical reinforcement learning', *Discrete event dynamic systems*, vol. 13, no. 1, pp. 41–77.

Bernstein, D. S., Zilberstein, S. & Immerman, N., 2013, 'The complexity of decentralized control of markov decision processes', .

Bester, C. J., James, S. D. & Konidaris, G. D., 2019, 'Multi-pass q-networks for deep reinforcement learning with parameterised action spaces', <https://arxiv.org/abs/1905.04388>.

Boddy, M. S., Gohde, J., Haigh, T. & Harp, S. A., 2005, 'Course of action generation for cyber security using classical planning.', *ICAPS*, pp. 12–21.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J. & Zaremba, W., 2016, 'Openai gym', .

Brys, T., Harutyunyan, A., Suay, H. B., Chernova, S., Taylor, M. E. & Nowé, A., 2015, 'Reinforcement learning from demonstration through shaping', *Twenty-fourth international joint conference on artificial intelligence*, .

Buczak, A. & Guven, E., 2015, 'A survey of data mining and machine learning methods for cyber security intrusion detection', *IEEE Communications Surveys & Tutorials*, vol. 18, pp. 1–1.

Burda, Y., Edwards, H., Storkey, A. J. & Klimov, O., 2018, 'Exploration by random network distillation', *CoRR*, vol. abs/1810.12894.

Buşoniu, L., Babuška, R. & Schutter, B., n.d., 'Multi-agent reinforcement learning: An overview', .

Chandak, Y., Theocharous, G., Kostas, J., Jordan, S. M. & Thomas, P. S., 2019, 'Learning action representations for reinforcement learning', *CoRR*, vol. abs/1902.00183.

Chen, J., Hu, S., Zheng, H., Xing, C. & Zhang, G., 2022, 'Gail-pt: An intelligent penetration testing framework with generative adversarial imitation learning', *Computers & Security*, p. 103055.

Chen, T., Liu, J., Xiang, Y., Niu, W., Tong, E. & Han, Z., 2019, 'Adversarial attack and defense in reinforcement learning-from ai security view', *Cybersecurity*, vol. 2, no. 1, pp. 1–22.

Cody, T., 2022, 'A layered reference model for penetration testing with reinforcement learning and attack graphs', *2022 IEEE 29th Annual Software Technology Conference (STC)*, IEEE, pp. 41–50.

Damle, A., Minden, V. & Ying, L., 2016, 'Robust and efficient multi-way spectral clustering', <https://arxiv.org/abs/1609.08251>.

Dayan, P., 1993, 'Improving generalization for temporal difference learning: The successor representation', *Neural Computation*, vol. 5, no. 4, pp. 613–624.

Dayan, P. & Hinton, G. E., 1993, 'Feudal reinforcement learning', Hanson, S. J., Cowan, J. D. & Giles, C. L. (eds.) *Advances in Neural Information Processing Systems 5*, Morgan-Kaufmann, pp. 271–278, <http://papers.nips.cc/paper/714-feudal-reinforcement-learning.pdf>.

de Wiele, T. V., Warde-Farley, D., Mnih, A. & Mnih, V., 2020, 'Q-learning in enormous action spaces via amortized approximate maximization', .

Dey, R. & Salem, F. M., 2017, 'Gate-variants of gated recurrent unit (gru) neural networks', *2017 IEEE 60th international midwest symposium on circuits and systems (MWSCAS)*, IEEE, pp. 1597–1600.

Dietterich, T. G., 1998, 'The maxq method for hierarchical reinforcement learning.', *ICML*, , vol. 98Citeseer, pp. 118–126.

Dietterich, T. G., 2000, 'Hierarchical reinforcement learning with the maxq value function decomposition', *Journal of artificial intelligence research*, vol. 13, pp. 227–303.

Ding, D., Han, Q.-L., Xiang, Y., Ge, X. & Zhang, X.-M., 2018, 'A survey on security control and attack detection for industrial cyber-physical systems', *Neurocomputing*, vol. 275, pp. 1674 – 1683.

Doersch, C., 2016, 'Tutorial on variational autoencoders', <https://arxiv.org/abs/1606.05908>.

Dulac-Arnold, G., Evans, R., van Hasselt, H., Sunehag, P., Lillicrap, T., Hunt, J., Mann, T., Weber, T., Degris, T. & Coppin, B., 2015, 'Deep reinforcement learning in large discrete action spaces', *arXiv preprint arXiv:1512.07679*.

Erdődi, L., Sommervoll, Å. Å. & Zennaro, F. M., 2021, 'Simulating sql injection vulnerability exploitation using q-learning reinforcement learning agents', *Journal of Information Security and Applications*, vol. 61, p. 102903.

Fan, J., Wang, Z., Xie, Y. & Yang, Z., 2020, 'A theoretical analysis of deep q-learning', Bayen, A. M., Jadbabaie, A., Pappas, G., Parrilo, P. A., Recht, B., Tomlin, C. & Zeilinger, M. (eds.) *Proceedings of the 2nd Conference on Learning for Dynamics and Control*, , vol. 120 of *Proceedings of Machine Learning Research*PMLR, pp. 486–489, <https://proceedings.mlr.press/v120/yang20a.html>.

Fan, Z., Su, R., Zhang, W. & Yu, Y., 2019a, 'Hybrid actor-critic reinforcement learning in parameterized action space', *CoRR*, vol. abs/1903.01344.

Fan, Z., Su, R., Zhang, W. & Yu, Y., 2019b, 'Hybrid actor-critic reinforcement learning in parameterized action space', *arXiv preprint arXiv:1903.01344*.

Farquhar, G., Gustafson, L., Lin, Z., Whiteson, S., Usunier, N. & Synnaeve, G., 2019, 'Growing action spaces', *CoRR*, vol. abs/1906.12266.

Fortunato, M., Azar, M. G., Piot, B., Menick, J., Osband, I., Graves, A., Mnih, V., Munos, R., Hassabis, D., Pietquin, O., Blundell, C. & Legg, S., 2017, 'Noisy networks for exploration', <https://arxiv.org/abs/1706.10295>.

François-Lavet, V., Henderson, P., Islam, R., Bellemare, M. G. & Pineau, J., 2018, 'An introduction to deep reinforcement learning', *Foundations and Trends® in Machine Learning*, vol. 11, no. 3-4, p. 219–354.

García-Teodoro, P., Díaz-Verdejo, J., Maciá-Fernández, G. & Vázquez, E., 2009, 'Anomaly-based network intrusion detection: Techniques, systems and challenges', *Computers & Security*, vol. 28, no. 1, pp. 18 – 28.

Geluvaraj, B., Satwik, P. M. & Ashok Kumar, T. A., 2019, 'The future of cybersecurity: Major role of artificial intelligence, machine learning, and deep learning in cyberspace', Smys, S., Bestak, R., Chen, J. I.-Z. & Kotuliak, I. (eds.) *International Conference on Computer Networks and Communication Technologies*, Springer Singapore, Singapore, pp. 739–747.

Gershman, S. J., 2018, 'The successor representation: Its computational logic and neural substrates', *Journal of Neuroscience*, vol. 38, no. 33, pp. 7193–7200.

Ghanem, M. C. & Chen, T. M., 2018, 'Reinforcement learning for intelligent penetration testing', *2018 Second World Conference on Smart Trends in Systems, Security and Sustainability (WorldS4)*, IEEE, pp. 185–192.

Ghanem, M. C. & Chen, T. M., 2020a, 'Reinforcement learning for efficient network penetration testing', *Information*, vol. 11, no. 1.

Ghanem, M. C. & Chen, T. M., 2020b, 'Reinforcement learning for efficient network penetration testing', <https://www.mdpi.com/2078-2489/11/1/6>.

Ghavamzadeh, M., Mahadevan, S. & Makar, R., 2006, 'Hierarchical multi-agent reinforcement learning', *Autonomous Agents and Multi-Agent Systems*, vol. 13, no. 2, pp. 197–229.

Goodfellow, I., Bengio, Y. & Courville, A., 2016, *Deep learning*, MIT press.

Gordon, G. J., 1995, 'Stable function approximation in dynamic programming', Tech. rep., Carnegie Mellon University, Pittsburgh, PA, USA.

Gupta, A., Tsai, T., Rueb, D., Yamaji, M. & Middleton, P., 2017, 'Forecast: Internet of things — endpoints and associated services, worldwide, 2017', <https://www.gartner.com/en/documents/3840665>.

Ha, D., Dai, A. M. & Le, Q. V., 2016, 'Hypernetworks', *CoRR*, vol. abs/1609.09106.

Haarnoja, T., Tang, H., Abbeel, P. & Levine, S., 2017, 'Reinforcement learning with deep energy-based policies', <https://arxiv.org/abs/1702.08165>.

Hammar, K. & Stadler, R., 2020, 'Finding effective security strategies through reinforcement learning and self-play', *2020 16th International Conference on Network and Service Management (CNSM)*, IEEE, pp. 1–9.

Hausknecht, M. & Stone, P., 2015a, 'Deep recurrent q-learning for partially observable mdps', <https://arxiv.org/abs/1507.06527>.

Hausknecht, M. & Stone, P., 2015b, 'Deep reinforcement learning in parameterized action space', *arXiv preprint arXiv:1511.04143*.

Hausknecht, M. J. & Stone, P., 2015c, 'Deep recurrent q-learning for partially observable mdps', *CoRR*, vol. abs/1507.06527.

Hengst, B., 2004, 'Model approximation for hexq hierarchical reinforcement learning', *European Conference on Machine Learning*, Springer, pp. 144–155.

Hessel, M., Modayil, J., van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M. & Silver, D., 2017a, 'Rainbow: Combining improvements in deep reinforcement learning', <https://arxiv.org/abs/1710.02298>.

Hessel, M., Modayil, J., van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M. G. & Silver, D., 2017b, 'Rainbow: Combining improvements in deep reinforcement learning', *CoRR*, vol. abs/1710.02298.

Hester, T., Vecerik, M., Pietquin, O., Lanctot, M., Schaul, T., Piot, B., Sendonaris, A., Dulac-Arnold, G., Osband, I., Agapiou, J. et al., 2017, 'Learning from demonstrations for real world reinforcement learning', .

Hu, Z., Beuran, R. & Tan, Y., 2020, 'Automated penetration testing using deep reinforcement learning', *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, IEEE, pp. 2–10.

Huang, S. & Ontañón, S., 2020, 'A closer look at invalid action masking in policy gradient algorithms', <https://arxiv.org/abs/2006.14171>.

Ingols, K., Lippmann, R. & Piwowarski, K., 2006, 'Practical attack graph generation for network defense', *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, IEEE, pp. 121–130.

Jaegle, A., Gimeno, F., Brock, A., Zisserman, A., Vinyals, O. & Carreira, J., 2021, 'Perceiver: General perception with iterative attention', <https://arxiv.org/abs/2103.03206>.

Kaelbling, L. P., Littman, M. L. & Cassandra, A. R., 1998, 'Planning and acting in partially observable stochastic domains', *Artificial Intelligence*, vol. 101, no. 1, pp. 99 – 134.

Kantchelian, A., Afroz, S., Huang, L., Islam, A. C., Miller, B., Tschantz, M. C., Greenstadt, R., Joseph, A. D. & Tygar, J., 2013, 'Approaches to adversarial drift', *Proceedings of the 2013 ACM workshop on Artificial intelligence and security*, pp. 99–110.

Kennedy, D., O'gorman, J., Kearns, D. & Aharoni, M., 2011, *Metasploit: the penetration tester's guide*, No Starch Press.

Kipf, T. N., 2020, 'Deep learning with graph-structured representations', .

Kober, J., Bagnell, J. A. & Peters, J., 2013, 'Reinforcement learning in robotics: A survey', *The International Journal of Robotics Research*, vol. 32, no. 11, pp. 1238–1274.

Kochenderfer, M. J. & Hayes, G., 2005, 'Modeling and planning in large state and action spaces', *Proceedings of the Workshop on Planning and Learning in A Priori Unknown and Dynamic Domains*, pp. 16–22.

Konidaris, G. D. & Barto, A. G., 2007, 'Building portable options: Skill transfer in reinforcement learning.', *IJCAI*, , vol. 7pp. 895–900.

Kujanpää, K., Victor, W. & Ilin, A., 2021, 'Automating privilege escalation with deep reinforcement learning', *Proceedings of the 14th ACM Workshop on Artificial Intelligence and Security*, AISec '21, Association for Computing Machinery, New York, NY, USA, p. 157–168, <https://doi.org/10.1145/3474369.3486877>.

Kulkarni, T. D., Narasimhan, K., Saeedi, A. & Tenenbaum, J. B., 2016, 'Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation', *CoRR*, vol. abs/1604.06057.

Laurent, G. J., Matignon, L. & Fort-Piat, N. L., 2011, 'The world of independent learners is not markovian', *Int. J. Know.-Based Intell. Eng. Syst.*, vol. 15, no. 1, pp. 55–64.

Lee, J., Lee, Y., Kim, J., Kosiorek, A. R., Choi, S. & Teh, Y. W., 2018, 'Set transformer: A framework for attention-based permutation-invariant neural networks', <https://arxiv.org/abs/1810.00825>.

Leibo, J. Z., Zambaldi, V. F., Lanctot, M., Marecki, J. & Graepel, T., 2017, 'Multi-agent reinforcement learning in sequential social dilemmas', *CoRR*, vol. abs/1702.03037.

Li, S., Zhang, J., Wang, J., Yu, Y. & Zhang, C., 2021a, 'Active hierarchical exploration with stable subgoal representation learning', <https://arxiv.org/abs/2105.14750>.

Li, W., Wang, X., Jin, B., Luo, D. & Zha, H., 2021b, 'Structured cooperative reinforcement learning with time-varying composite action space', *IEEE Transactions on Pattern Analysis and Machine Intelligence*.

Li, X., Li, L., Gao, J., He, X., Chen, J., Deng, L. & He, J., 2015, 'Recurrent reinforcement learning: A hybrid approach', <https://arxiv.org/abs/1509.03044>.

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D. & Wierstra, D., 2015, 'Continuous control with deep reinforcement learning', *arXiv preprint arXiv:1509.02971*.

Littman, M. L., 2001, 'Value-function reinforcement learning in markov games', *Cognitive Systems Research*, vol. 2, no. 1, pp. 55 – 66.

Lyon, G. F., 2008, *Nmap network scanning: The official Nmap project guide to network discovery and security scanning*, Insecure. Com LLC (US).

Machado, M. C., Bellemare, M. G. & Bowling, M., 2017a, 'A laplacian framework for option discovery in reinforcement learning', *International Conference on Machine Learning*, PMLR, pp. 2295–2304.

Machado, M. C., Bellemare, M. G. & Bowling, M., 2020, 'Count-based exploration

with the successor representation', *Proceedings of the AAAI Conference on Artificial Intelligence*, , vol. 34pp. 5125–5133.

Machado, M. C. & Bowling, M., 2016, 'Learning purposeful behaviour in the absence of rewards', *arXiv preprint arXiv:1605.07700*.

Machado, M. C., Rosenbaum, C., Guo, X., Liu, M., Tesauro, G. & Campbell, M., 2017b, 'Eigenoption discovery through the deep successor representation', *arXiv preprint arXiv:1710.11089*.

Masson, W. & Konidaris, G. D., 2015, 'Reinforcement learning with parameterized actions', *CoRR*, vol. abs/1509.01644.

Masson, W., Ranchod, P. & Konidaris, G., 2016, 'Reinforcement learning with parameterized actions', *Thirtieth AAAI Conference on Artificial Intelligence*, .

Matignon, L., Laurent, G. j. & Le fort piat, N., 2012, 'Review: Independent reinforcement learners in cooperative markov games: A survey regarding coordination problems', *Knowl. Eng. Rev.*, vol. 27, no. 1, pp. 1–31.

McGovern, A. & Barto, A. G., 2001a, 'Automatic discovery of subgoals in reinforcement learning using diverse density', .

McGovern, A. & Barto, A. G., 2001b, 'Automatic discovery of subgoals in reinforcement learning using diverse density', *Proceedings of the Eighteenth International Conference on Machine Learning*, ICML '01, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, p. 361–368.

Merckling, A., Perrin-Gilbert, N., Coninx, A. & Doncieux, S., 2021, 'Exploratory state representation learning', .

Mitchell, T. M., 1997, *Machine learning, International Edition*, McGraw-Hill Series in Computer Science, McGraw-Hill, <https://www.worldcat.org/oclc/61321007>.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. & Riedmiller, M., 2013, 'Playing atari with deep reinforcement learning', *arXiv preprint arXiv:1312.5602*.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G. et al., 2015, 'Human-level control through deep reinforcement learning', *Nature*, vol. 518, no. 7540, p. 529.

Nachum, O., Gu, S., Lee, H. & Levine, S., 2018, 'Data-efficient hierarchical reinforcement learning', .

Nguyen, T. T. & Reddi, V. J., 2019, 'Deep reinforcement learning for cyber security', *CoRR*, vol. abs/1906.05799.

Nguyen, T. T. & Reddi, V. J., 2021, 'Deep reinforcement learning for cyber security', *IEEE Transactions on Neural Networks and Learning Systems*, pp. 1–17.

Ni, T., Eysenbach, B. & Salakhutdinov, R., 2021, 'Recurrent model-free rl can be a strong baseline for many pomdps', <https://arxiv.org/abs/2110.05038>.

Niculae, S., Dichiu, D., Yang, K. & Bäck, T., 2020, 'Automating penetration testing using reinforcement learning', .

Oliehoek, F., 2010, *Value-Based Planning for Teams of Agents in Stochastic Partially Observable Environments*, UvA proefschriften, Amsterdam University Press, Amsterdam, <https://cds.cern.ch/record/1315051>.

Oltsik, J., 2018, 'Research suggests cybersecurity skills shortage is getting worse', <https://www.csoonline.com/article/3247708/research-suggests-cybersecurity-skills-shortage-is-getting-worse.html>.

Orman, H., 2003, 'The morris worm: A fifteen-year perspective', *IEEE Security & Privacy*, vol. 1, no. 05, pp. 35–43.

Panait, L. & Luke, S., 2005, 'Cooperative multi-agent learning: The state of the art', *Autonomous Agents and Multi-Agent Systems*, vol. 11, no. 3, pp. 387–434.

Parr, R. & Russell, S., 1998, 'Reinforcement learning with hierarchies of machines', *Proceedings of the 1997 Conference on Advances in Neural Information Processing Systems 10*, NIPS '97, MIT Press, Cambridge, MA, USA, pp. 1043–1049, <http://dl.acm.org/citation.cfm?id=302528.302894>.

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L. & Lerer, A., 2017, 'Automatic differentiation in pytorch', .

Pathak, D., Agrawal, P., Efros, A. A. & Darrell, T., 2017a, 'Curiosity-driven exploration by self-supervised prediction', <https://arxiv.org/abs/1705.05363>.

Pathak, D., Agrawal, P., Efros, A. A. & Darrell, T., 2017b, 'Curiosity-driven exploration by self-supervised prediction', *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, JMLR.org, p. 2778–2787.

Pattanaik, A., Tang, Z., Liu, S., Bommannan, G. & Chowdhary, G., 2017, 'Robust deep reinforcement learning with adversarial attacks', <https://arxiv.org/abs/1712.03632>.

Pinto, L., Davidson, J., Sukthankar, R. & Gupta, A., 2017, 'Robust adversarial reinforcement learning', *International Conference on Machine Learning*, PMLR, pp. 2817–2826.

Precup, D., 2000, 'Temporal abstraction in reinforcement learning', .

Puterman, M. L., 1994, *Markov Decision Processes: Discrete Stochastic Dynamic Programming*, 1st edn., John Wiley & Sons, Inc., New York, NY, USA.

Rafati, J. & Noelle, D. C., 2019a, 'Efficient exploration through intrinsic motivation learning for unsupervised subgoal discovery in model-free hierarchical reinforcement learning', <https://arxiv.org/abs/1911.10164>.

Rafati, J. & Noelle, D. C., 2019b, 'Learning representations in model-free hierarchical reinforcement learning', *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence and Thirty-First Innovative Applications of Artificial Intelligence Conference and Ninth AAAI Symposium on Educational Advances in Artificial Intelligence*, AAAI'19/IAAI'19/EAAI'19, AAAI Press, <https://doi.org/10.1609/aaai.v33i01.330110009>.

Rahalkar, S., 2019, 'Openvas', *Quick Start Guide to Penetration Testing*, Springer, pp. 47–71.

Ramesh, R., Tomar, M. & Ravindran, B., 2019, 'Successor options: An option discovery framework for reinforcement learning', *arXiv preprint arXiv:1905.05731*.

Rashid, T., Samvelyan, M., de Witt, C. S., Farquhar, G., Foerster, J. N. & Whiteson, S., 2018, 'QMIX: monotonic value function factorisation for deep multi-agent reinforcement learning', *CoRR*, vol. abs/1803.11485.

Russell, S. & Norvig, P., 2010, *Artificial Intelligence: A Modern Approach*, 3rd edn., Prentice Hall.

Sarraute, C., 2013, 'Automated attack planning', *CoRR*, vol. abs/1307.7808.

Schaul, T., Horgan, D., Gregor, K. & Silver, D., 2015a, 'Universal value function approximators', Bach, F. & Blei, D. (eds.) *Proceedings of the 32nd International Conference on Machine Learning*, , vol. 37 of *Proceedings of Machine Learning Research*PMLR, Lille, France, pp. 1312–1320, <https://proceedings.mlr.press/v37/schaul15.html>.

Schaul, T., Quan, J., Antonoglou, I. & Silver, D., 2015b, 'Prioritized experience replay', <https://arxiv.org/abs/1511.05952>.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A. & Klimov, O., 2017, 'Proximal policy optimization algorithms', <https://arxiv.org/abs/1707.06347>.

Schwartz, J. & Kurniawati, H., 2019a, 'Autonomous penetration testing using reinforcement learning', .

Schwartz, J. & Kurniawati, H., 2019b, 'Autonomous penetration testing using reinforcement learning', <https://arxiv.org/abs/1905.05965>.

Sherstov, A. A. & Stone, P., 2005, 'Function approximation via tile coding: Automating parameter choice', *International Symposium on Abstraction, Reformulation, and Approximation*, Springer, pp. 194–205.

Silver, D., 2015, 'Lectures on reinforcement learning', URL: https://www.davidsilver.uk/teaching/.

Şimşek, Ö., Wolfe, A. P. & Barto, A. G., 2005, 'Identifying useful subgoals in reinforcement learning by local graph partitioning', *Proceedings of the 22nd international conference on Machine learning*, pp. 816–823.

Standen, M., Lucas, M., Bowman, D., Richer, T. J., Kim, J. & Marriott, D., 2021, 'Cyborg: A gym for the development of autonomous cyber agents', *CoRR*, vol. abs/2108.09118.

Sunehag, P., Lever, G., Gruslys, A., Czarnecki, W. M., Zambaldi, V., Jaderberg, M., Lanctot, M., Sonnerat, N., Leibo, J. Z., Tuyls, K. & Graepel, T., 2017a, 'Value-decomposition networks for cooperative multi-agent learning', <https://arxiv.org/abs/1706.05296>.

Sunehag, P., Lever, G., Gruslys, A., Czarnecki, W. M., Zambaldi, V. F., Jaderberg, M., Lanctot, M., Sonnerat, N., Leibo, J. Z., Tuyls, K. & Graepel, T., 2017b, 'Value-decomposition networks for cooperative multi-agent learning', *CoRR*, vol. abs/1706.05296.

Sutskever, I., Vinyals, O. & Le, Q. V., 2014, 'Sequence to sequence learning with neural networks', Ghahramani, Z., Welling, M., Cortes, C., Lawrence, N. & Weinberger, K. (eds.) *Advances in Neural Information Processing Systems*, , vol. 27Curran Associates, Inc., <https://proceedings.neurips.cc/paper/2014/file/a14ac55a4f27472c5d894ec1c3c743d2-Paper.pdf>.

Sutton, R. S. & Barto, A. G., 2018a, *Reinforcement Learning: An Introduction*, 2nd edn., The MIT Press, <http://incompleteideas.net/book/the-book-2nd.html>.

Sutton, R. S. & Barto, A. G., 2018b, *Reinforcement learning: An introduction*, MIT press.

Sutton, R. S., McAllester, D. A., Singh, S. P. & Mansour, Y., 2000, 'Policy gradient methods for reinforcement learning with function approximation', *Advances in neural information processing systems*, pp. 1057–1063.

Sutton, R. S., Precup, D. & Singh, S., 1999, 'Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning', *Artificial Intelligence*, vol. 112, no. 1, pp. 181 – 211.

Tampuu, A., Matiisen, T., Kodelja, D., Kuzovkin, I., Korjus, K., Aru, J., Aru, J. & Vicente, R., 2015, 'Multiagent cooperation and competition with deep reinforcement learning', .

Tan, M., 1993a, 'Multi-agent reinforcement learning: Independent versus cooperative agents', *Proceedings of the Tenth International Conference on Machine Learning (ICML 1993)*, Morgan Kauffman, San Francisco, CA, USA, pp. 330–337, <http://web.media.mit.edu/~cynthiab/Readings/tan-MAS-reinfLearn.pdf>.

Tan, M., 1993b, 'Multi-agent reinforcement learning: Independent vs. cooperative agents', *In Proceedings of the Tenth International Conference on Machine Learning*, Morgan Kaufmann, pp. 330–337.

Tan, M., 1993c, 'Multi-agent reinforcement learning: Independent vs. cooperative agents', *Proceedings of the tenth international conference on machine learning*, pp. 330–337.

Tavakoli, A., Pardo, F. & Kormushev, P., 2018, 'Action branching architectures for deep reinforcement learning', *Thirty-Second AAAI Conference on Artificial Intelligence*, .

Team., M. D. R., 2021, 'Cyberbattlesim', https://github.com/microsoft/cyberbattlesim, created by Christian Seifert, Michael Betser, William Blum, James Bono, Kate Farris, Emily Goren, Justin Grana, Kristian Holsheimer, Brandon Marken, Joshua Neil, Nicole Nichols, Jugal Parikh, Haoran Wei.

Tran, K., Akella, A., Standen, M., Kim, J., Bowman, D., Richer, T. & Lin, C.-T., 2021, 'Deep hierarchical reinforcement agents for automated penetration testing', <https://arxiv.org/abs/2109.06449>.

Tran, K., Standen, M., Kim, J., Bowman, D., Richer, T., Akella, A. & Lin, C.-T., 2022, 'Cascaded reinforcement learning agents for large action spaces in autonomous penetration testing', *Applied Sciences*, vol. 12, no. 21, p. 11265.

Tuyls, K. & Weiss, G., 2012, 'Multiagent learning: Basics, challenges, and prospects', *AI Magazine*, vol. 33, no. 3, p. 41.

Van der Maaten, L. & Hinton, G., 2008, 'Visualizing data using t-sne.', *Journal of machine learning research*, vol. 9, no. 11.

Vezhnevets, A. S., Osindero, S., Schaul, T., Heess, N., Jaderberg, M., Silver, D. & Kavukcuoglu, K., 2017, 'Feudal networks for hierarchical reinforcement learning', *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, JMLR. org, pp. 3540–3549.

Vigorito, C. M. & Barto, A. G., 2010, 'Intrinsically motivated hierarchical skill learning in structured environments', *IEEE Transactions on Autonomous Mental Development*, vol. 2, no. 2, pp. 132–143.

Vinyals, O., Babuschkin, I., Czarnecki, W. M., Mathieu, M., Dudzik, A., Chung, J., Choi, D. H., Powell, R., Ewalds, T., Georgiev, P. et al., 2019, 'Grandmaster level in starcraft ii using multi-agent reinforcement learning', *Nature*, vol. 575, no. 7782, pp. 350–354.

Vo, V. Q., Abbasnejad, E. & Ranasinghe, D. C., 2022, 'Query efficient decision based sparse attacks against black-box deep learning models', <https://arxiv.org/abs/2202.00091>.

Walter, E., Ferguson-Walter, K. & Ridley, A., 2021, 'Incorporating deception into cyberbattlesim for autonomous defense', *arXiv preprint arXiv:2108.13980*.

Wang, Z., Schaul, T., Hessel, M., Van Hasselt, H., Lanctot, M. & De Freitas, N., 2015, 'Dueling network architectures for deep reinforcement learning', *arXiv preprint arXiv:1511.06581*.

Watkins, C. J. & Dayan, P., 1992, 'Q-learning', *Machine learning*, vol. 8, no. 3-4, pp. 279–292.

Watkins, C. J. C. H., 1989, 'Learning from delayed rewards', .

Wei, E., Wicke, D. & Luke, S., 2018a, 'Hierarchical approaches for reinforcement learning in parameterized action space', <https://arxiv.org/abs/1810.09656>.

Wei, E., Wicke, D. & Luke, S., 2018b, 'Hierarchical approaches for reinforcement learning in parameterized action space', *arXiv preprint arXiv:1810.09656*.

Xin, Y., Kong, L., Liu, Z., Chen, Y., Li, Y., Zhu, H., Gao, M., Hou, H. & Wang, C., 2018, 'Machine learning and deep learning methods for cybersecurity', *IEEE Access*, vol. 6, pp. 35365–35381.

Xiong, J., Wang, Q., Yang, Z., Sun, P., Han, L., Zheng, Y., Fu, H., Zhang, T., Liu, J. & Liu, H., 2018, 'Parametrized deep q-networks learning: Reinforcement learning with discrete-continuous hybrid action space', <https://arxiv.org/abs/1810.06394>.

Zahavy, T., Haroush, M., Merlis, N., Mankowitz, D. J. & Mannor, S., 2018, 'Learn what not to learn: Action elimination with deep reinforcement learning', *CoRR*, vol. abs/1809.02121.

Zennaro, F. M. & Erdodi, L., 2020, 'Modeling penetration testing with reinforcement learning using capture-the-flag challenges and tabular q-learning', *arXiv preprint arXiv:2005.12632*.

Zhang, Z., Li, H., Zhang, L., Zheng, T., Zhang, T., Hao, X., Chen, X., Chen, M., Xiao, F. & Zhou, W., 2019, 'Hierarchical reinforcement learning for multi-agent MOBA game', *CoRR*, vol. abs/1901.08004.

Zhou, S., Liu, J., Hou, D., Zhong, X. & Zhang, Y., 2021, 'Autonomous penetration testing based on improved deep q-network', *Applied Sciences*, vol. 11, no. 19.

Zhou, Z., Chen, Z., Zhou, T. & Guan, X., 2010, 'The study on network intrusion detection system of snort', *2010 International Conference on Networking and Digital Society*, , vol. 2pp. 194–196.