Artificial intelligence-based cloud network control

Abderrahmane Redha Alliche

UNIVERSITÉ CÔTE D'AZUR

ÉCOLE DOCTORALE
SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

THÈSE DE DOCTORAT

Contrôle du réseau cloud basé Intelligence Artificielle

Abderrahmane Redha ALLICHE

Laboratoire d'Informatique, de Signaux et Systèmes de Sophia Antipolis (I3S)
UMR7271 UniCA CNRS

Présentée en vue de l'obtention du grade de docteur en Informatique **d'**Université Côte d'Azur

Dirigée par : Lucile SASSATELLI, Professeure des universités, Université Côte d'Azur
Co-dirigée par : Ramon APARICIO PARDO, Maître de conférences, Université Côte d'Azur
Soutenue le : 07 Juin 2024

Devant le jury, composé de :
Guillaume URVOY-KELLER, Professeur des universités, Université Côte d'Azur
Lucile SASSATELLI, Professeure des universités, Université Côte d'Azur
Ramon APARICIO PARDO, Maître de conférences, Université Côte d'Azur
Stefano SECCI, Professeur des universités, Conservatoire National des Arts et Métiers
Fen ZHOU, Maître de conférences-HDR, Université d'Avignon
Massimo TORNATORE, Professor, Politecnico di Milano, Italy

CONTRÔLE DU RÉSEAU CLOUD BASÉ INTELLIGENCE ARTIFICIELLE

Artificial Intelligence-based cloud network control

Abderrahmane Redha ALLICHE

Jury :

Président du jury
Guillaume URVOY-KELLER, Professeur des universités, Université Côte d'Azur

Rapporteurs
Stefano SECCI, Professeur des universités, Conservatoire National des Arts et Métiers
Fen ZHOU, Maître de conférences-HDR, Université d'Avignon

Examinateurs
Massimo TORNATORE, Professor, Politecnico di Milano, Italy

Directeur de thèse
Lucile SASSATELLI, Professeure des universités, Université Côte d'Azur

Co-directeur de thèse
Ramon APARICIO PARDO, Maître de conférences, Université Côte d'Azur

---

Université Côte d'Azur

Abderrahmane Redha ALLICHE

Contrôle du réseau cloud basé Intelligence Artificielle

xiii+121 p.

Ce document a été préparé avec LaTeX2e et la classe `these-ISSS` version v. 2.10.

Impression : output.tex – 24/7/2024 – 8:39

Révision pour la classe : these-ISSS.cls,v 2.10 2020/06/24 14:16:37 mpelleau

"The happiness of your life depends upon the quality of your thoughts."
***Marcus Aurelius**, Meditations*

Contrôle du réseau cloud basé Intelligence Artificielle

Résumé

L'explosion du nombre d'utilisateurs d'Internet et du volume de trafic constitue un défi majeur pour la gestion efficace des réseaux de diffusion de contenu (CDN). Bien que ces réseaux aient amélioré le temps de réponse en exploitant la mise en cache dans des serveurs cloud proches des utilisateurs, les services non mis en cache continuent de poser des problèmes de gestion de trafic. Pour répondre à cette problématique, les réseaux overlay cloud ont émergé, mais ils introduisent des complexités telles que les violations d'inégalités triangulaires (TIV).

Dans ce contexte, l'application du paradigme des réseaux à définition logicielle (SDN) combinée aux techniques d'apprentissage par renforcement profond (DRL) offre une opportunité prometteuse pour s'adapter en temps réel aux fluctuations de l'environnement. Face à l'augmentation constante du nombre de serveurs edge, les solutions distribuées de DRL, notamment les modèles d'apprentissage par renforcement profond multi-agent (MA-DRL), deviennent cruciales. Cependant, ces modèles rencontrent des défis non résolus tels que l'absence de simulateurs réseau réalistes, le surcoût de communication entre agents et la convergence et stabilité.

Cette thèse se concentre donc sur l'exploration des méthodes MA-DRL pour le routage de paquets dans les réseaux overlay cloud. Elle propose des solutions pour relever ces défis, notamment le développement de simulateurs de réseau réalistes, l'étude du surcoût de communication et la conception d'une solution MA-DRL adaptée aux réseaux overlay cloud. L'accent est mis sur le compromis entre la performance et la quantité d'information partagée entre les agents, ainsi que sur la convergence et la stabilité de l'entraînement.

Mots-clés : Apprentissage par Renforcement Profond Multi-Agent ; Routage de Paquets Distribué ; Réseaux Cloud Overlay ; Contrôle de Réseaux Autonomes ; Intelligence Artificielle; Optimisation

Artificial Intelligence-based cloud network control

Abstract

The exponential growth of Internet traffic in recent decades has prompted the emergence of Content Delivery Networks (CDNs) as a solution for managing high traffic volumes through data caching in cloud servers located near end-users. However, challenges persist, particularly for non-cacheable services, necessitating the use of cloud overlay networks. Due to a lack of knowledge about the underlay network, cloud overlay networks introduce complexities such as Triangle inequality violations (TIV) and dynamic traffic routing challenges.

Leveraging the Software Defined Networks (SDN) paradigm, Deep Reinforcement Learning (DRL) techniques offer the possibility to exploit collected data to better adapt to network changes. Furthermore, the increase of cloud edge servers presents scalability challenges, motivating the exploration of Multi-Agent DRL (MA-DRL) solutions. Despite its suitability for the distributed packet routing problem in cloud overlay networks, MA-DRL faces non-addressed challenges such as the need for realistic network simulators, handling communication overhead, and addressing the multi-objective nature of the routing problem.

This Ph.D. thesis delves into the realm of distributed Multi-Agent Deep Reinforcement Learning (MA-DRL) methods, specifically targeting the Distributed Packet Routing problem in cloud overlay networks. Throughout the thesis, we address these challenges by developing realistic network simulators, studying communication overhead in the non-overlay general setting, and proposing a distributed MA-DRL framework tailored to cloud overlay networks, focusing on communication overhead, convergence, and model stability.

Keywords: Multi-Agent Deep Reinforcement Learning; Distributed Packet Routing; Cloud Overlay Networks; Autonomous Network Control; Artificial Intelligence; Optimization

Acknowledgement

This thesis manuscript marks the end of a journey that lasted almost four years of work. It was a valuable and enriching experience, both personally and professionally, culminating in a significant milestone in my career.

First and foremost, I extend my sincere gratitude to my advisors, Dr. Ramon Aparicio-Pardo and Prof. Lucile Sassatelli, for their support, guidance, and endless encouragement throughout this journey. Your expertise, openness to ideas, and mentorship have shaped me into the researcher I am today.

I thank the reviewers for their availability and willingness to evaluate this work. I also would like to express my gratitude to the members of the individual monitoring committee, Prof. Luis Velasco and Prof. Guillaume Allibert, for their invaluable feedback and unwavering support during the annual review process.

I extend my appreciation to all the individuals in the I3S lab for their kindness and support. Additionally, I am grateful to the COATI team members at INRIA, with whom I spent my first two years of the thesis. Your passion, enthusiasm, and shared dedication to research made every challenge surmountable.

I am profoundly grateful to my wife, Sara, whose unwavering love, patience, and understanding sustained me during the challenging moments of this endeavor. Your support and belief in me kept me motivated and focused. This thesis is dedicated to you.

To my parents, Samir and Anissa, I owe an immense debt of gratitude for instilling in me perseverance and determination. Your unwavering support, sacrifices, and encouragement have been the driving force behind my success.

I extend my heartfelt gratitude to all those who supported me throughout this journey, including my brother Yacine, my friends, family, and colleagues. I am especially grateful to Dr. Hassan Adda and his family, whose assistance was invaluable during my first year in France.

Additionally, I want to express my deep appreciation to my uncle Djamel and his wife Assia, whose unwavering support has been a constant source of strength.

Finally, I also wish to honor the memory of my grandfather Mustafa and my late uncle Noureddine. Though they left us too soon, I know they would have been proud to see my success. May they rest in peace.

This thesis represents not only the culmination of years of hard work and dedication but also a testament to the collective support and encouragement of those around me. I am deeply grateful for the opportunity to embark on this journey, and I look forward to the next chapter with immense gratitude and humility.

Table of contents

Chapter 1

Introduction

1.1 Motivation

The traffic over the Internet has scaled massively during the last decades: the number of Internet users worldwide has reached 4.7 billion in 2020 (Ritchie, Mathieu, Roser, & Ortiz-Ospina, 2023). As of 2023, approximately 67% of the world's population, equivalent to 5.4 billion people, is connected to the Internet (*Facts and Figures 2023 - Internet Use*, s. d.) According to the Cisco Annual Internet Report (*Cisco Annual Internet Report - Cisco Annual Internet Report (2018–2023) White Paper*, s. d.), the landscape of Internet traffic is shaped by several key factors. Firstly, a substantial portion of this traffic stems from video streaming, particularly following the advent of Ultra-High-Definition (UHD) or 4K compatible devices. Additionally, platforms dedicated to file-sharing, cloud computing services, and online gaming significantly contribute to the overall volume of Internet data. Moreover, the increasing popularity of remote work and online learning has increased the usage of video conferencing applications, further intensifying Internet traffic. Lastly, the growth of the Internet of Things (IoT), characterized by Machine-to-Machine (M2M) connections facilitated by interconnected homes, cities, and vehicles, has notably augmented the global Internet traffic load.

To handle this high demand, Content Delivery Networks (CDNs) have emerged as a practical solution by providing data caching on cloud servers near the end user. Video streaming accounts for a significant portion of internet traffic, which is why CDNs are essential for data-heavy services such as YouTube and Netflix. Doan et al. (Doan, Bajpai, & Crawford, 2020) showed that caching reduces throughput by up to three times. However, some services like live video streaming are non-cacheable or cacheable for only short periods. In such cases, the requested content is downloaded from the origin server via a cloud overlay network operated by the CDN provider.

Overlay networks are virtual or logical networks built on top of a physical network (underlay networks). They provide flexible traffic routing between nodes that are not directly connected by a physical link, such as CDN servers positioned at the Internet's edge. The overlay nodes are connected via tunnels traversing different Autonomous Systems managed by different Internet Service Providers. These tunnels are virtual or logical links corresponding to paths in the underlying network.

The complexity of routing traffic within cloud overlay networks stems from four factors. First, the lack of knowledge about the underlay topology, which continues to grow in complexity since the Internet's infrastructure has expanded, leading to more intricate network topology: more and more Autonomous Systems (AS) (*CIDR REPORT for 15 Sep 23*, s. d.). As a consequence, each overlay link could pass through multiple network operators (i.e., multiple AS). Each AS is running a routing policy which is also unknown by the overlay nodes. This leads to the presence of Triangle Inequality Violations, in which the direct tunnel connecting two remote servers may not

be the optimal one. Second, the presence of unknown dynamic underlay traffic (generated by the underlay nodes) makes predicting the overlay links capacities more complex and hard to handle by classical methods like Border Gateway Protocol (BGP) (Rekhter, Li, & Hares, 2006) or Cisco Overlay Routing Protocol (OMP). Those methods rely on shared administrative weights that may not properly represent the rapid variation of the inaccessible underlay network metrics like the delay and the loss rate. Third, similar to the variability observed in underlay traffic, overlay traffic exhibits dynamic and complex characteristics (S. Yang & Kuipers, 2014; Sun et al., 2021). Routing such dynamic traffic typically involves formulating it as a constrained shortest path problem and solving multiple instances of general multi-commodity flow problems, each with a modified traffic matrix. However, finding the optimal solution in this setting is NP-hard (Trimponias, Xiao, Wu, Xu, & Geng, 2019). Also, this approach assumes the model to be fixed with varying traffic for the general multi-commodity flow problem (Quang et al., 2022). Consequently, these methods struggle to effectively manage sudden fluctuations in traffic demand, as they require frequent recomputation of solutions, leading to inefficiencies in network resource management (S. K. Singh, Das, & Jukan, 2015; Hasan, Al-Rizzo, & Al-Turjman, 2017; Kim, Kim, & Lim, 2022). Furthermore, this approach fails to accurately model the network's behavior given the unknown underlay topology and routing policies and the presence of dynamic underlay traffic. These limitations motivate the adoption of Machine Learning (ML) techniques, especially Deep Reinforcement Learning (DRL) (Sutton & Barto, 2018; Bengio, 2009). The latter method has proven its effectiveness by achieving breakthrough results in various complex tasks (Mnih et al., 2015). In the context of network management, DRL can provide the capacity to adapt to sudden changes in traffic, handle the lack of knowledge about the topology, and optimize the performance of the network in real-time (Mukhutdinov, Filchenkov, Shalyto, & Vyatkin, 2019; You et al., 2022; Manfredi, Wolfe, Wang, & Zhang, 2021; L. Chen, Hu, Guan, Zhao, & Shen, 2021). The adoption of Machine Learning method has also been available thanks to Software-Defined Networking (SDN) paradigm. SDN provides flexible management of the network by decoupling the data plane (packet transmission) from the control plane (control operations) (Benzekki, El Fergougui, & Elbelrhiti Elalaoui, 2016; Belzarena, Sena, Amigo, & Vaton, 2016). The control plane can embed the training of the ML model and deploy the resulting policies into the data plane. The last factor of complexity of routing in cloud overlay networks is the increasing number of edge servers. This surge can be attributed to the substantial data volumes generated by data-driven businesses and IoT applications, as well as the diversification of services such as cloud computing and the emergence of the cloud continuum (Yousefpour et al., 2019; Moreschini et al., 2022). This increase in the number of overlay nodes poses a scalability concern, which centralized approaches facilitated by the SDN paradigm struggle to address efficiently. Consequently, researchers are turning to multi-agent approaches, considering each overlay node (i.e., cloud server) as an autonomous agent making decisions on incoming packets based on locally collected information. This problem formulation is termed as Distributed Packet Routing (DPR). Multi-Agent Deep Reinforcement Learning, which it is an extension of Deep Reinforcement Learning, has become popular in solving communication network problems like the DPR problem (Mukhutdinov et al., 2019; You et al., 2022; Manfredi et al., 2021; L. Chen et al., 2021). In this setting, DRL agents collaborate to optimize a global objective. In addition to the scalability and resilience of the network, having a decentralized training approach allows fast response to the changes in the network while circumventing issues linked to maintaining a centralized database.

Although the distributed Multi-Agent Deep Reinforcement Learning (MA-DRL) scheme seems intuitively suited for routing and can extend the classical distributed routing algorithms like

Routing Information Protocol (RIP) (Malkin, 1998) or Back-Pressure (BP) routing (Tassiulas & Ephremides, 1992), the network research community has not thoroughly investigated it for several reasons. First, there is a lack of a common realistic network simulator to test and compare the solutions. Existing simulators often align better with the centralized SDN paradigm, leaving the fully decentralized scheme under-explored. Second, as pointed out by the previous studies (Gronauer & Diepold, 2022; Matignon, Laurent, & Le Fort-Piat, 2012), the agents need to communicate to overcome the non-stationarity of the environment, especially in the cloud overlay setting. This results in a high communication overhead between the agents during the training. Lastly, applying Reinforcement Learning (RL) for the DPR problem is a hard problem due to its multi-objective nature. Indeed, the agents may need to optimize more than one network metric (i.e., delay and loss). This poses challenges when designing the reward, which ensures stability in the learning process.

1.2 Goal of the thesis

Given the motivation detailed above, this Ph.D. thesis is oriented towards studying distributed Multi-Agent Deep Reinforcement Learning methods for the Distributed Packet Routing problem for cloud overlay networks.

This thesis presents an in-depth study of this approach, focusing on its multiple challenges, such as realistic experimentation, communication between the agents, and learning stability. The objective is to address those challenges and design a fully distributed Multi-Agent Deep Reinforcement Learning framework for the Distributed Packet Routing problem in the cloud overlay setting.

To achieve this main objective, we define two goals:

G.1- Realistic network simulator

This first goal aims to provide a realistic platform to evaluate and compare the state-of-the-art models. This simulator should also allow the emulation of control packets to evaluate the communication overhead between the agents. This goal consists of two steps :

— **G1.1- General simulator**
Developing a general realistic network simulator adapted to MA-DRL for the general non-overlay case.

— **G1.2- Simulator for the cloud overlay case**
Extending the general simulator (G1.1) to the cloud overlay case by implementing the overlay topology on top of a physical one. Also, add the underlay background traffic.

G.2- Distributed Multi-Agent Deep Reinforcement Learning framework for Distributed Packet Routing problem

The goal is to propose an efficient distributed Multi-Agent Deep Reinforcement Learning that tackles the Distributed Packet Routing problem. The framework should present competitive results regarding performance, communication overhead, and stability. This goal consists of two steps:

— **G2.1- General framework**
Implement the framework in the general non-overlay case. This sub-goal focuses on this less complex case to analyze and minimize the communication overhead and study its effect on the model's performance.

— **G2.2- Framework for the cloud overlay case**
Extend the general framework to the cloud overlay case. In this case, the proposed framework should handle the lack of knowledge about the underlay topology, the underlay routing policies, and the underlay traffic. The focus is also put on the stability of the agents' learning.

A summary of the goals of the thesis is presented in Table 1.1.

Table 1.1 – Thesis goals

G.1- Realistic network simulator problem in cloud overlay networks	**G1.1- General simulator**
	G1.2- Simulator for overlay case
G.2- Distributed MA-DRL framework for DPR problem	**G2.1- General framework**
	G2.2- Framework for the cloud overlay case

1.3 Methodology

This thesis treats the packet routing problem in a distributed manner using Multi-Agent Deep Reinforcement Learning. We assume the architecture presented in Figure 1.1 for the general Distributed Packet Routing problem. Each network node embeds an Reinforcement Learning agent interacting with its environment. This environment includes the local information gathered by the node and the information coming from its neighbors through a communication channel. After taking action, the agent receives a reward that it uses to improve its policy and adapt to changes in the environment.

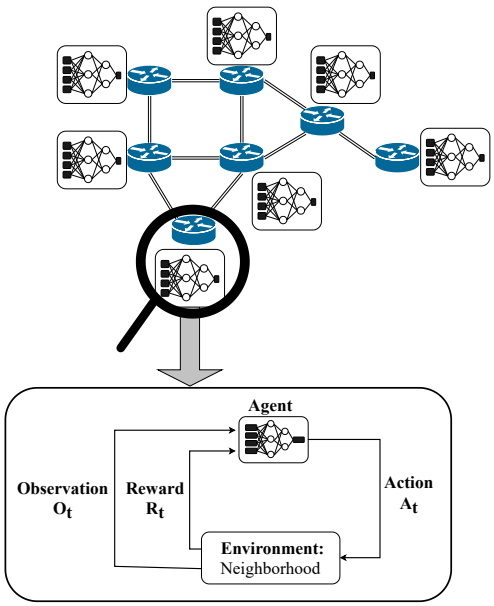


Figure 1.1 – Illustration of the Distributed Packet Routing problem considered in the thesis

Figure 1.2 describes the architecture of a cloud overlay node in this setting. The cloud overlay control plane embeds four elements: the AI Forwarder Agent, the AI Trainer Agent, the AI communicator Agent, and the node database. The packet header information, like the packet destination, is extracted and sent to the AI Forwarded Agent along with locally monitored data for each incoming data packet. This information is used as an input of the Deep Neural Network by the AI Forwarder Agent. The Deep Neural Network then outputs an action, which is the outgoing port. The packet's header and the locally monitored data are also sent to the AI Trainer Agent. The latter treats the information and stores it in the node database. The AI Trainer Agent periodically samples data from the node database to train and update the Deep Neural Network (DNN) weights that are sent to the AI Forwarder Agent. The AI Communicator Agent handles the communication between the cloud overlay node and its neighbors through an in-band control packet exchange. The data collected by the AI Communicator Agent updates the missing information needed to train the agent.

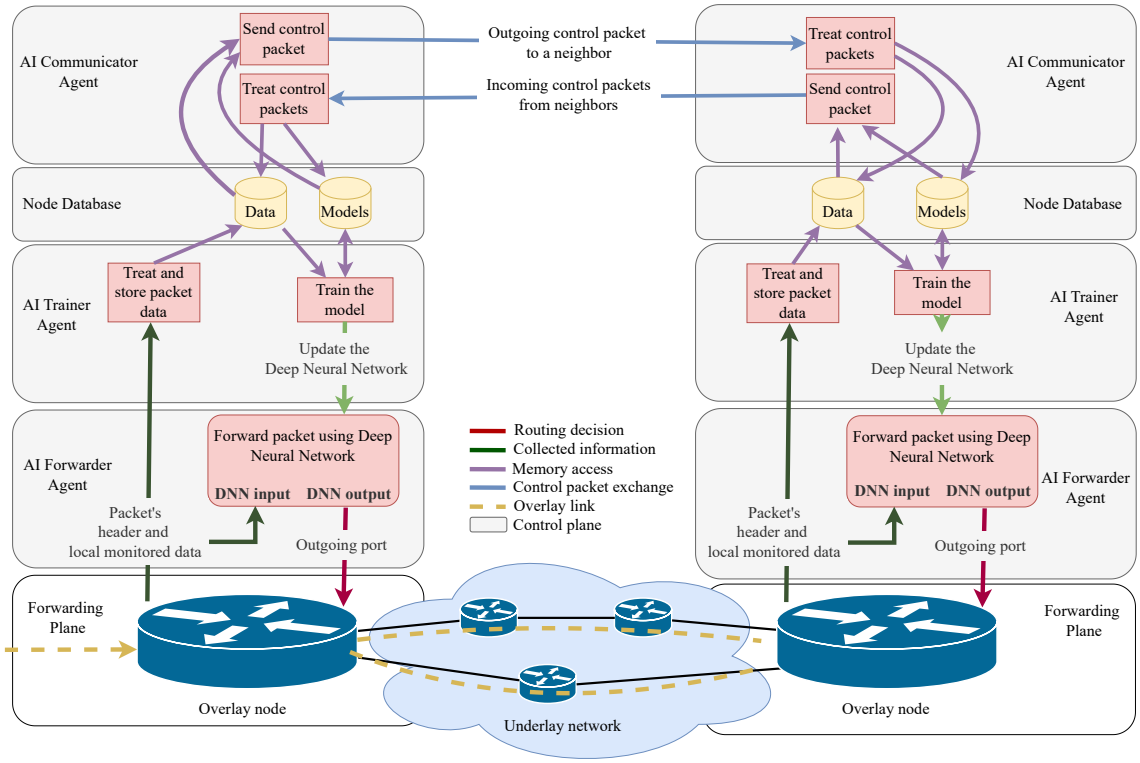


Figure 1.2 – Architecture of the network agent in the distributed MA-DRL setting

Considering the system architecture presented above, the methodology illustrated in Figure 1.3 was followed to achieve the thesis goal.

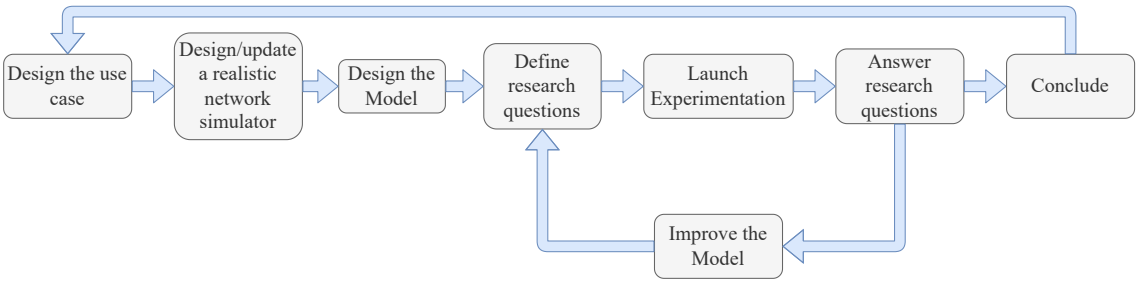


Figure 1.3 – Thesis methodology

First, we define the use case scenario. We start with the general non-overlay case and then tackle the cloud overlay networks. After that, we design or update our network simulator for each of the two scenarios. To ensure realistic simulation, `ns-3` (nsnam, s. d.) is chosen. Then, inspired by the scientific literature, a Multi-Agent Deep Reinforcement Learning framework is designed to address the problem. Research questions are then defined based on performance, communication overhead, and model stability. Based on those questions, an experimentation scenario is designed, and experiments are launched. We analyze the results obtained and answer the previously defined questions. Based on the answers, we can either improve the model and relaunch the experiments or conclude by disseminating the results in international conferences or journals.

1.4 Contributions

As pointed out in the motivation section, applying distributed MA-DRL to address the DPR in cloud overlay networks presents multiple challenges: realistic simulation environment, communication between agents overhead, stability given the lack of knowledge in the overlay setting. Given those challenges, we sort our contributions in the following table (Table 1.2).

Table 1.2 – Contributions

Use case scenario	Build a realistic simulation environment	Analyze and reduce communication overhead	Address the stability	Apply DRL in other cases different from the MA-DRL for DPR problem
DPR problem in general non-overlay networks	**PRISMA**	**IMPACT**	-	-
DPR problem in cloud overlay networks	**PRISMA-2**	**O-DQR**	**O-DQR**	-
Quantum virtual link generation	-	-	-	**QUANTUM**
Cloud network slicing management	-	-	-	**SLICE**

The acronyms used in the previous table correspond to publications detailed in the following and sorted in chronological order.

1. **PRISMA** (G1.1): corresponds to the paper entitled "`Prisma`: a packet routing simulator for multi-agent reinforcement learning" (Alliche, Barros, Aparicio-Pardo, & Sassatelli, 2022). It was presented at the IFIP networking conference 2022 workshop.
2. **IMPACT** (G2.1): the paper entitled "Impact evaluation of control signalling onto distributed learning-based packet routing" (Alliche, da Silva Barros, Aparicio-Pardo, & Sassatelli,

2022). It was presented at the 34th International Teletraffic Congress (ITC) 2022. It also includes an overhead reduction method illustrated in Chapter 4.

3. **SLICE**: corresponds to the paper entitled "Reconfiguring network slices at the best time with deep reinforcement learning" (Gausseran et al., 2022). It was done in collaboration with the COATI team in INRIA and presented at the 11th International Conference on Cloud Networking (CloudNet) 2022.
4. **PRISMA-2** (G1.2): corresponds to the paper entitled "`prisma-v2`: Extension to cloud overlay networks" (Alliche, Barros, Aparicio-Pardo, & Sassatelli, 2023). It was presented at the 23rd International Conference on Transparent Optical Networks (ICTON) 2023.
5. **QUANTUM**: corresponds to the paper entitled "Quantum virtual link generation via reinforcement learning" (Aparicio-Pardo, Cousson, & Alliche, 2023). It was presented at the 23rd International Conference on Transparent Optical Networks (ICTON) 2023.
6. **O-DQR** (G2.2): corresponds to the paper entitled "O-DQR: a Multi-agent Deep Reinforcement Learning for Distributed Packet Routing in Overlay Networks". It is submitted to IEEE Transactions on Network and Service Management.

1.5 Thesis outline

The remainder of this thesis manuscript is organized as follows.

Chapter 2 provides the needed background on Reinforcement Learning, Multi-Agent Deep Reinforcement Learning systems and the application of those methods to the Distributed Packet Routing problem. It also includes a review of the available simulation tools and finishes with a presentation of the constrained models used to address the stability of the model.

Chapter 3 reviews state-of-the-art related to the goals of this thesis, focusing on MA-DRL models applied to Distributed Packet Routing problem, communication between the agents, and stability of the learning using constrained models.

Chapter 4 focuses on goal G.1. It contains two parts corresponding to the two contributions **PRISMA** and **PRISMA-2**. The first concerns the implementation of `PRISMA` (Alliche, Barros, et al., 2022): a realistic network simulator for distributed routing using independent Multi-Agent Deep Reinforcement Learning methods. The second part presents the extension of our simulator to overlay networks, which is called `prisma-v2` (Alliche et al., 2023).

Chapter 5 presents the contribution **IMPACT** (G2.1) related to the general non-overlay network scenario. It has two sections. The first one presents the study done on the communication overhead of Multi-Agent Deep Reinforcement Learning method applied to the Distributed Packet Routing problem (Alliche, da Silva Barros, et al., 2022). This study was motivated by the fact that communication overhead between agents was not deeply covered by the literature in general non-overlay networks. The second section presents a new proposed communication strategy to significantly reduce this overhead. The strategy was presented in the submitted paper O-DQR.

Chapter 6 presents the contribution **Q-DQR** (G2.2), which is a novel distributed Multi-Agent Deep Reinforcement Learning framework for cloud overlay networks. It is based on the submitted journal publication.

Chapter 7 gathers two contributions during the thesis involving Reinforcement Learning applied to network control (Gausseran et al., 2022 ; Aparicio-Pardo et al., 2023), which are **QUANTUM** and **SLICE**.

Finally, chapter 8 concludes this Ph.D. thesis.

CHAPTER 2

Background

In this chapter, we go through the core aspects necessary to understand the Ph.D. thesis. We first introduce the single-agent Reinforcement Learning (RL) framework. Then, we extend the concept to Multi-Agent Reinforcement Learning (MA-RL). We present the Distributed Packet Routing (DPR) problem and the proposed RL methods to solve it, of which the Q-routing paradigm. We also present the available network simulation tools and the RL libraries. Finally, we present the concept of constrained reinforcement learning at the end of this chapter

2.1 Introduction

Reinforcement Learning (RL) has been widely used to tackle networking problems. One classical problem is the Distributed Packet Routing (DPR). It consists of routing packets in a distributed manner where each node (router) acts independently from the others. This problem can be challenging when the traffic becomes highly dynamic and hard to predict using classical routing methods. This motivates researchers to use Deep Reinforcement Learning (DRL) since it can adapt to the changes of the environment, like the changes in the traffic and the topology. Multi-Agent Deep Reinforcement Learning (MA-DRL) is the extension of DRL to multiple agents, which fits naturally with the distributed aspect of packet routing. However, MA-DRL can experience stability issues, especially when the reward includes multiple goals (i.e., optimizing the packet loss and the packet end-to-end delay). In this case, concepts from the theory of constrained models can be used to construct the reward signal in a way that ensures stability. Additionally, a realistic network simulator is required to evaluate the performance of the developed solutions.

The goal of this chapter is to provide the necessary materials to understand the thesis and its contributions. It is structured as follows. First, In Section 2.2, we cover reinforcement learning principles, moving from a single-agent approach to a multi-agent approach. Then, we describe the Distributed Packet Routing in Section 2.3. In Section 2.4, we present available simulation tools. Constrained Reinforcement Learning (CRL) is covered in Section 2.5. Finally, we conclude this chapter in Section 2.6.

2.2 From Single-Agent to Multi-Agent Reinforcement Learning

This section is inspired by Sutton and Barto's book: "Reinforcement Learning: An Introduction" (Sutton & Barto, 2018) and "Multi-agent deep reinforcement learning: a survey" (Gronauer & Diepold, 2022). The goal is to briefly introduce the Reinforcement Learning method.

2.2.1 Single-Agent Reinforcement Learning

The idea of Reinforcement Learning is to learn by interacting with the environment. As illustrated by Figure 2.1, the agent observes the state of the environment at each time t, denoted as s. Based on this state, the agent takes an action a. In response to his action, the agent receives at time $t+1$ a reward r and observes a new state s'. The agent's objective is to maximize the expected performance in a long-term perspective, given the unknown dynamics of the environment.

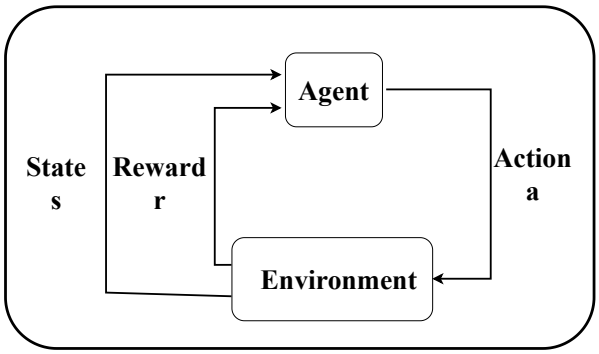


Figure 2.1 – Single-Agent system model

The standard formulation for such a system is the Markov Decision Process (Bellman, 1957), which is defined below.

2.2.1.1 Markov Decision Process formulation

A Markov Decision Process (MDP) is a discrete-time stochastic control process where the Markov property is met: the future behavior of a stochastic process depends only on the present state of the system and is independent of the past states. An MDP is formalized by the tuple $(\mathcal{S}, \mathcal{A}, p, R, \gamma)$ where $\mathcal{S}$ and $\mathcal{A}$ are the state and action space, respectively. Let $p : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to p(X)$ be the transition function describing the probability of a state transition, $R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ be the reward function, and $\gamma \in [0, 1]$ be the discount factor. The agent implements a mapping from the state $s \in \mathcal{S}$ to the probability of selecting each possible action $a \in \mathcal{A}$. This mapping is called a policy and is denoted $\pi(a|s)$.

During an episode and in a fully observable environment, the agent follows its policy and takes action a_t at each time step t after observing a state s_t.

The episode return G_t is defined as:

$$G_t = R(s_t, a_t) + \gamma R(s_{t+1}, a_{t+1}) + \cdots = \sum_{k=0}^{T-t} \gamma^k R(s_{t+k}, a_{t+k})] \tag{2.1}$$

Equation 2.1 includes the possibility that $T = \infty$ and $\gamma < 1$ or $T < \infty$ and $\gamma \in [0, 1]$ for infinite-horizon and finite-horizon problems, respectively. For simplicity, we consider the infinite-horizon problem formulation for the rest of the chapter. The formulations for finite-horizon problems can be found in (Bertsekas, 2012). We note that by changing the discount factor γ, we define the importance of the future interactions ($k > 0$) in the return.

2.2.1.2 State-value function and Q-function

To evaluate the expected performance given a policy π and stating from a state s, we define the *state-value function* $v_\pi : S \to \mathbb{R}$.

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_t | S_t = s] \\ &= \mathbb{E}_\pi[\sum_{t=0}^{\infty} \gamma^k R(s_{t+k}, a_{t+k}) | s_t = s] \end{aligned} \tag{2.2}$$

A fundamental property of the value function (Equation (2.2)) is that it satisfies the recursive *Bellman equation* described by Equation (2.3).

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_t | S_t = s] \\ &= \mathbb{E}_\pi[\sum_{t=0}^{\infty} \gamma^k R(s_{t+k}, a_{t+k}) | s_t = s] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')], \end{aligned} \tag{2.3}$$

where $p(s', r|s, a)$ is the probability of getting the state s' and the reward r after taking the action a and being in the state s, the actions $a \in A$, the state $s \in S$ and the next state $s' \in S$.

Similarly, we define the expected performance given a starting state s, a policy π, and after taking an action a by the action-value function $Q_\pi : S \times A \to \mathbb{R}$, which is also called the *Q-function*.

$$Q_\pi(s,a) = \mathbb{E}[G_t|s_t = s, a_t = a] = \mathbb{E}[\sum_{t=0}^{\infty} \gamma^k R(s_{t+k}, a_{t+k})|s_t = s, a_t = a] \tag{2.4}$$

The *Bellman equation* is also satisfied for the *q-function*, which is given by Equation (2.5).

$$\begin{aligned} Q_\pi(s,a) &= \mathbb{E}[G_t|s_t = s, a_t = a] \\ &= \mathbb{E}[\sum_{t=0}^{\infty} \gamma^k R(s_{t+k}, a_{t+k})|s_t = s, a_t = a] \\ &= \sum_{s',r} p(s', r|s, a)[r + \gamma Q_\pi(s', a')], \end{aligned} \tag{2.5}$$

2.2.1.3 Finding the optimal policy

The goal of the agent is to find an optimal policy that maximizes the expected return. In a single-agent setting, at least one policy is always better than or equal to all the other policies. The optimal policies π^* share the same optimal *state value function*, denoted $v^*(s)$ (Equation (2.6)), and the same optimal *Q-function*, denoted $Q^*(s,a)$ (Equation (2.7))

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

$$\begin{aligned} Q^*(s,a) &= \max_\pi Q_\pi(s,a) \\ &= \sum_{s',r} p(s', r|s, a)[r + \gamma \max_{a'} Q_\pi(s', a')], \end{aligned} \tag{2.7}$$

To improve any policy π to get to the optimal policy π^*, a popular method is to use *temporal-difference* (TD) learning. The advantage of this method is that it allows the agent to update its policy after each transition without waiting for a final state (final return G_t) and without having a model of the environment (Model-Free). It is done by bootstrapping using the estimated value of subsequent states to figure out the value of the current state.

For example, *Q-learning* (C. J. C. H. Watkins, 1989) is an off-policy TD learning algorithm, which is one of the most popular algorithms in Reinforcement Learning. In its simplest form, the *one-step Q-learning* is defined by:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s,a)], \tag{2.8}$$

where α is the learning rate.

Finally, the key idea of Deep Reinforcement Learning is to represent the value function V_π, the policy function Q_π, or both by a Deep Neural Network (DNN).

Additionally, The environment can be partially observable, where the agent cannot observe the entire state s but only an observation o of it. In this setting, we formulate the problem as Partially Observable Markov Decision Process (POMDP), and the agent must take the actions under the uncertainty of the actual environment state.

2.2.2 Multi-Agent Reinforcement Learning

The above setting can be extended to multiple agents. In this case, we use Markov Games (Littman, 1994) as a framework.

2.2.2.1 Multi-Agent Reinforcement Learning under the Markov Games framework

A Markov Game is formalized by the tuple ($\mathcal{N}$, $\mathcal{S}$, $\mathcal{A}^i$, p, R^i, γ), where $\mathcal{N} = \{1, \ldots, N\}$ denotes the set of $N > 1$ agents, $\mathcal{S}$ is the set of the observed states by all the agents, $\mathcal{A}^i$ is the joint action space, which is the collection of the action spaces of all the agents $\mathcal{A}^i = \mathcal{A}^1 \times \cdots \times \mathcal{A}^N$, p is the transition probability for the environment, R^i is the reward function associated to each agent $i \in \mathcal{N}$, and γ is the discount factor.

At each time step t, each agent $i \in \mathcal{N}$ observes the state s_t and takes an action a_t^i based on each agent's policy π^i. The environment then evolves from the state s_t to the state s_{t+1}, and each agent gets an immediate reward r^i.

Similar to the single-agent problem, each agent needs to change its policy to optimize the long-term return. However, unlike the first case, the *state-value function* and the *Q-function* do not depend only on the policy of the agent i denoted π^i but also on the policies of the other agents denoted π^{-i}.

When all the agents learn simultaneously, the optimal policy, which may not be unique, is described as the *Nash equilibrium* (Leyton-Brown & Shoham, 2022). This is a solution where each agent's policy $\pi^{*(i)}$ is the best response to the other agents' policy $\pi^{*(-i)}$ such that $v^i_{\pi^{*(i)},\pi^{*(-i)}}(s) \geq v^i_{\pi^i,\pi^{*(-i)}}(s)$ holds for all state $s \in \mathcal{S}$ and all policies $\pi^i \in \Pi^i \forall i$.

2.2.2.2 Types of Markov Games

Depending on the type of the task, Multi-Agent Reinforcement Learning can be categorized into the following:

1. **Fully cooperative setting**. All the agents are encouraged to collaborate and maximize the team's performance. In this case, the reward can be *equally shared* among the agents.
2. **Fully competitive setting.** The agents aim to maximize their reward and minimize the reward of the others. It can be described as either (i) a *zero-sum* Markov Game $R = \sum_{i=1}^{N} R^i(s, a) = 0$ or (ii) a *competitive game* where the rewards do not sum to zero.
3. **Mixed setting.** It is known as a *general-sum game* where the game does not incorporate restrictions on the agents' goal.

2.2.2.3 Challenges of Multi-Agent Reinforcement Learning

The multi-agent setting brings a lot of challenges and complexities to consider when designing a model. We detail below the most important ones.

1. **Non-stationarity.** As mentioned before, the agents in the Multi-Agent Reinforcement Learning setting update their policies simultaneously. Consequently, the environment appears non-stationary from an agent's perspective, and thus, the Markov assumption of an MDP no longer holds. In this situation, the agents face a moving target problem (E. Yang & Gu, 2004 ; Busoniu, Babuska, & De Schutter, 2008)

2. **Partially observable environment.** Also known as *Partially observable Markov Games* (POMG). Like Partially Observable Markov Decision Process, each agent can only observe a part of the state, denoted o^i. In this setting, the history of the interaction with the environment is meaningful; thus, it is relevant to incorporate history-dependent policies.
3. **Converging to sub-optimal solutions.** It was shown that agents could possibly converge to sub-optimal solutions (Wiegand, 2004). This can be explained by the *shadowed equilibrium* (Fulda & Ventura, 2007 ; Matignon et al., 2012). In essence, it suggests that despite agents' efforts, they may converge to near-optimal solutions but not necessarily globally optimal. This occurs when agents receive minimal rewards by unilaterally deviating from this equilibrium. If the potential gain from deviating is lower than the minimal gain achievable by deviating from another equilibrium, agents tend to remain in this sub-optimal state.
4. **Credit assignment problem.** In a cooperative setting with a joint reward signal, the contribution of each agent's action to the reward can be unknown. Therefore, the agents face a credit assignment problem (Chang, Ho, & Kaelbling, 2003 ; Wolpert & Tumer, 1999).
5. **Alter-exploration problem.** It is a dilemma that is also present in the single-agent case and is amplified by the partial observability of the environment. It represents the tradeoff between taking non-optimal action to explore the environment or taking optimal decisions according to the current knowledge of the environment.

2.2.2.4 Training and execution schemes

The training of the agents is a challenging problem since the complexity of the state and action space increases exponentially with the number of agents. We distinguish two training paradigms: centralized and distributed (Weiß, 1995). The first describes the situation where the agents rely on global state information to update their policies. The second refers to the situation where each agent updates its policy independently without relying on a centralized point.

Regarding execution, we also distinguish two strategies: centralized and decentralized. The first refers to relying on a global controller to make decisions, and the second refers to taking action independently.

In practice, we can have three types of architecture: (i) Centralized Training Centralized Execution (CTCE), (ii) Centralized Training Decentralized Execution (CTDE), and (iii) Distributed Training Decentralized Execution (DTDE). The definition of those architectures may differ in the literature. Still, in this Ph.D. thesis, we consider the following:

— Centralized Training Centralized Execution (CTCE): the agents rely on a centralized point during training and inference.

— Centralized Training Decentralized Execution (CTDE): the agents rely on a centralized point during the training but are independent in inference.

— Distributed Training Decentralized Execution (DTDE): the agents are independent to a centralized point during training and inference. However, they may share information and interact locally at both phases, training and inference.

2.2.2.5 Communication between agents

One of the previously mentioned challenges is the environment's non-stationarity, especially when considering a distributed training architecture. To address this problem, the simplest approach is to neglect the adaptative behavior of the agents by ignoring them (Matignon et al., 2012)

or assuming the others' behavior to be static or optimal (Lauer & Riedmiller, 2000). This solution, which is called *independent learners*, can perform well in simple deterministic environments (Claus & Boutilier, 1998), but often results in poor performance (Matignon et al., 2012; Lowe et al., 2017) and can over-fit and fail to generalize (Lanctot et al., 2017). To handle this, the literature proposed more efficient solutions. Foerster et al. (Foerster et al., 2017) proposed to use an experience replay buffer to store past interactions and decay the outdated transition samples to stabilize the training. Baker et al. (Baker et al., 2019) and Leibo et al. (Leibo, Zambaldi, Lanctot, Marecki, & Graepel, 2017) used short-term experience replay buffers. Bono et al. (Bono, Dibangoye, Matignon, Pereyron, & Simonin, 2019) adopted a centralized training decentralized execution approach. Another approach, called meta-learning, was proposed by Finn et al. (Finn & Levine, 2017) and used by Rabinowitz et al. (Rabinowitz et al., 2018) to construct agent models. The idea of this approach is to allow the agent to learn to predict other agents' future actions. Finally, another research track proposes to address the problem of non-stationarity and partial observability by allowing the agents to communicate. The communication can be either broadcast to all the agents (Foerster, Assael, De Freitas, & Whiteson, 2016), targeted to a specific agent (Das et al., 2019; Jiang & Lu, 2018; A. Singh, Jain, & Sukhbaatar, 2018), or sent only to the neighbors (networked communication) (K. Zhang, Yang, Liu, Zhang, & Basar, 2018; Chu, Wang, Codecà, & Li, 2019).

2.3 Distributed Packet Routing problem

In this section, we review the application of Reinforcement Learning to tackle the Distributed Packet Routing problem. We first formulate the Distributed Packet Routing problem. Then, we detail the *Q-Routing* framework, which is an RL solution to the problem. Finally, we conclude the section by presenting *DQN-Routing*, which is the DRL extension of *Q-Routing*.

2.3.1 Problem formulation

The Distributed Packet Routing problem pertains to the optimization and efficient management of data packet routing in decentralized communication networks. In this context, "distributed" refers to the absence of a centralized routing authority, requiring each node within the network to make autonomous decisions regarding the forwarding of data packets. This problem naturally fits the fully decentralized multi-agent scheme (Distributed Training Decentralized Execution).

The Distributed Packet Routing problem can be formalized as a Multi-Agent POMDP (Littman, 1994): a Markov Decision Process where each agent can only locally observe its environment without knowing the actual state of the whole process.

Let $\mathcal{N}$ be the set of routing nodes (*agents*), where each *agent* n has its own local observation space $\mathcal{O}_n$ and its own action space $\mathcal{A}_n$. When a new packet arrives at time t at the node n, the node n selects as an action a_n the *next hop node* n' to forward the packet to. This decision is taken depending on the local observation of the router o_n (e.g., *the current packet destination*, *the node buffers' occupancy*, ...). As a consequence, the *agent* n receives a reward r_n: the *next-hop packet delay* (i.e., the packet delay to travel from n to n'), and the whole network makes a transition to a new state. This procedure is repeated each time a packet arrives at a node.

2.3.2 Q-Routing framework

The first RL method proposed to tackle the DPR was the *Q-routing* (Boyan & Littman, 1993), based on the classical *Q-learning* algorithm (C. J. C. H. Watkins, 1989). We will use this method as a framework to describe the POMDP and the RL approach used to solve it. Note that other RL methods different from *Q-learning* could also be adapted to the DPR problem.

The framework selects an action a_n that minimizes an estimate of the expected *end-to-end packet delay* from node n to its final destination using the estimate of the neighbors. This idea is extended to Deep Reinforcement Learning by Mukhutdinov et al. (Mukhutdinov et al., 2019) denoted as $Q_n(o_n, a_n; \theta_n)$ (the *Q-function*). This is the output of a DNN with θ_n weights. The DNN is trained to fit the target value Y_n^Q below:

$$Y_n^Q = r_n + \gamma \cdot \tau \cdot (1 - f) \tag{2.9}$$

where f indicates if the next hop is the packet destination, $\gamma \in [0, 1]$ is a discount factor, r is the *next-hop packet delay* and τ is *the remaining end-to-end delay* from the next hop n' to the destination. The latter is computed as follows:

$$\tau = \min_{a_{n'} \in \mathcal{A}_{n'}} Q_{n'}(o_{n'}, a_{n'}; \theta_{n'}) \tag{2.10}$$

where $Q_{n'}(\cdot; \theta_{n'})$ is the output of the DNN of the next hop agent.

The θ_n weights are updated by stochastic gradient descent when minimizing the *Temporal Difference (TD) error*:

$$L_{DQN} = \left(Q_n(o_n, a_n; \theta_n) - Y_n^Q\right)^2 \tag{2.11}$$

We note that compared to classical *Q-learning* presented by Equation (2.8), here, to build the target value Y_n^Q, we use the DNN $Q_{n'}(., \theta_{n'})$ of the neighbor n' instead of using the local target network $Q_n(o', a', \theta_n)$ for the next local observation o' at the node n.

2.4 Simulation tools

After presenting the MA-DRL approach to tackle the DPR problem, we discuss in this section the available simulation tools to evaluate those methods. One of the most popular network simulators is called `ns-3`, which was adapted to handle the distributed MA-DRL case for general non-overlay networks (Alliche, Barros, et al., 2022) and the cloud overlay networks (Alliche et al., 2023). In this section, we present the `ns-3` network simulator and the available tools that transform a network simulation in a DRL environment.

2.4.1 The `ns-3` Network Simulator

The `ns-3` simulator (nsnam, s. d.) is a stable discrete-event network simulator, largely adopted by the research and educational community and accepted as a standard. It is free and open-source under *GPLv2* and developed in `C++` using object-oriented programming. The success of this tool is based on two main features: (i) realistic modeling of the network systems: the protocol stack is properly abstracted by the simulator classes; (ii) a wide range of networking protocols (e.g., IP, TCP, UDP), communication technologies (e.g., Ethernet, Wi-Fi, LTE) and statistical models for channels, mobility, and traffic generation are supported by built-in classes.

2.4.2 Reinforcement Learning environments for network simulation

RL agents are usually implemented using `Python` packages (e.g., `TensorFlow` (*TensorFlow: An end-to-end open source machine learning platform*, s. d.)) and interfaced with an environment where agents take decisions. The OpenAI™`Gym` (*Gym: Gym toolkit for creating reinforcement learning environments*, s. d.) is one of the most popular toolkits to define RL environments in `Python`. `Gym` is used to test and compare different RL algorithms for a variety of problems, like Atari games or robotics (Zamora, Lopez, Vilches, & Cordero, 2016). `Gym` also provides a simple interface to pass the interactions between the agent and the environment without making assumptions about the agent structure. Moreover, the `Gym` toolkit allows the creation of customized environments, which is very convenient since no network environment is natively available in `Gym`.

The first solution to this lack of networking RL environments was `ns3-gym` (Gawłowicz & Zubow, 2019), which interfaces OpenAI ™`Gym` with the `ns-3` network simulator. In other words, `ns3-gym` "transforms" a `ns-3` network simulation into an RL environment by serving as a gateway connecting `ns-3` with `Gym`. Then, it provides an easy-to-use platform for developing and testing RL algorithms for networking problems. The `Python` process (the agent and the `Gym` environment) communicates with the `C++` process (the `ns-3` network simulation) via ZMQ sockets (*Zero MQ: An open-source universal messaging library*, s. d.). `ns3-gym` is also open source under a `GPL` license and can be easily extended. Other tools were released after `ns3-gym`, such as `ns3-ai` (Yin et al., 2020).

2.5 Constrained Reinforcement Learning (CRL)

The optimization for the DPR problem can be multi-objective and can include constraints (like the links capacity constraint). For this reason, we introduce in this section the Constrained Reinforcement Learning (CRL) approach. It is an approach within the field of Reinforcement Learning that focuses on optimizing decision-making processes under specific constraints. In CRL, the objective is to maximize the expected goal while satisfying a set of predefined constraints. This methodology is particularly relevant in scenarios where there are essential limitations or requirements that the learning algorithm must adhere to during the training and inference phases. To understand CRL, we need to present two concepts: (i) the Constrained Markov Decision Processes (CMDPs) and (ii) the Lagrangian Relaxation.

2.5.1 Constrained Markov Decision Processes (CMDPs)

Constrained Markov Decision Processes (CMDPs) are a variant of the classical Markov Decision Processes (MDPs) that incorporate additional safety constraints. It is formulated with the tuple ($\mathcal{S}$, $\mathcal{A}$, p, R, γ, c, d). Similarly to MDPs, $\mathcal{S}$ and $\mathcal{A}$ represent state and action spaces, respectively. p is the transition probability function, R is the reward function and γ is the discount factor. $c(s, a) \in \mathbb{R}$ is the cost function and d is the safety threshold. The cost function evaluates whether the constraints are satisfied under the current state s and after taking the action a. In this framework, the Q-function is the same as in MDPs (see Equation (2.4)).

We add the estimation of the expected discounted cost for a given policy, denoted *C-function*:

$$C_\pi(s, a) = \mathbb{E}[\sum_{t=0}^{\infty} \gamma^t c(s_t, a_t) | s_0 = s, a_0 = a] \tag{2.12}$$

The objective function of a CMDP is formulated as follows:

$$\max_{\pi} \mathbb{E}_{(s,a)\sim\rho_\pi}[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t)], \text{ s.t. } \mathbb{E}_{(s,a)\sim\rho_\pi}[\sum_{t=0}^{\infty} \gamma^t c(s_t, a_t)] \leq d. \tag{2.13}$$

This objective function seeks to maximize the expected cumulative rewards under a policy π while ensuring that the expected cumulative cost over time does not exceed the predefined safety threshold d. Here, ρ_π represents the distribution induced by the policy π over state-action pairs.

As a recap, CMDPs provide a formal framework for decision-making under constraints, where the goal is to balance the optimization of rewards with the satisfaction of safety requirements.

2.5.2 Lagrangian Relaxation

The problem defined in Equation (2.13) can be solved using the Lagrangian relaxation method. In the Lagrangian relaxation, the original constrained optimization problem is transformed into an unconstrained form by introducing Lagrange multipliers, also known as dual variables. These multipliers act as penalty terms associated with each constraint, allowing the incorporation of the constraints into the objective function.

We apply the Lagrangian relaxation to transform the Equation (2.13) and define the unconstrained dual problem as follows:

$$\min_{\lambda\geq 0} \max_{\theta} L(\lambda, \theta) = \min_{\lambda\geq 0} \max_{\theta} [J_R^{\pi_\theta} - \lambda.(J_C^{\pi_\theta} - d)] \tag{2.14}$$

Where

$$J_R^{\pi_\theta} = \max_{\pi} \mathbb{E}_{(s,a)\sim\rho_\pi}[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t)], \tag{2.15}$$

$$J_C^{\pi_\theta} = \max_{\pi} \mathbb{E}_{(s,a)\sim\rho_\pi}[\sum_{t=0}^{\infty} \gamma^t c(s_t, a_t)]. \tag{2.16}$$

2.6 Conclusion

The objective of this chapter is to give the essential information needed to understand the context of the Ph.D. thesis. We first introduced the single-agent DRL and extended them to MA-DRL. Then, we formulated the core problem tackled in this thesis, the DPR problem. We also present the available simulation tools. Finally, we introduced the concept of constrained reinforcement learning, which could be considered to stabilize the agents' learning and integrate the constraints of the environment into the model.

In the following chapter, we will review the literature related to the goals of the thesis.

CHAPTER 3

State-of-the-Art

This state-of-the-art chapter provides a comprehensive survey of the application of Deep Reinforcement Learning (DRL) techniques to the Distributed Packet Routing (DPR) problem. It starts by examining the limitations of existing network simulation tools in supporting DRL algorithms and introduces extensions like `ns3-gym` *and* `ns3-ai` *designed to bridge this gap. Subsequently, it reviews seminal works applying DRL methods and advancements in Multi-Agent Deep Reinforcement Learning (MA-DRL) for routing problems in general non-overlay and cloud overlay networks. Key considerations regarding communication between agents and the stability and safety of MA-DRL solutions are thoroughly discussed.*

3.1 Introduction

The chapter provides a literature review related to the goals of the thesis, which is the application of Multi-Agent Deep Reinforcement Learning (MA-DRL) techniques to networking, specifically addressing Distributed Packet Routing (DPR) challenges in the cloud overlay case.

The first section (Section 3.2) discusses existing network simulation tools, such as `ns-3`, and extensions like `ns3-gym`. Then, Section 3.3 delves into the works exploring Deep Reinforcement Learning (DRL) applications in general non-overlay network cases. It reviews existing works using single-agent and MA-DRL. In Section 3.4, the discussion is extended to packet routing in cloud overlay networks, which introduces additional challenges, such as the lack of knowledge about the underlay topology and traffic. Existing routing protocols along with DRL techniques are reviewed. Section 3.5 investigates a critical aspect of deploying distributed MA-DRL solutions: the impact of network control signaling on training processes. Different approaches to handling communication between agents, including *model sharing* and *value sharing*, are discussed. Furthermore, the section identifies a lack of investigation into the trade-off between performance and communication overhead in the general non-overlay case. Moreover, the final section (Section 3.6) presents the challenges related to the stability and safety of MA-DRL solutions for the DPR problem. It highlights the importance of considering the stability of the MA-DRL in the cloud overlay case.

3.2 DRL tools for networking

One of the most popular network simulating tools is `ns-3` (nsnam, s. d.), a discrete-event simulator implemented in `C++`. It allows researchers to model and test different aspects of network configuration and protocols. However, `ns-3` does not support deep learning algorithms since most of these algorithms rely on open-source `Python` frameworks like `TensorFlow` and `PyTorch`. To enable the interaction between `ns-3` and the latter frameworks, many extensions have been proposed, such as `ns3-gym` (Gawłowicz & Zubow, 2019) and `ns3-ai` (Yin et al., 2020). Another direction for extending `ns-3` is to focus on specific domains or applications of network systems, such as wireless and Internet of Things (IoT) networks. These domains pose new challenges and opportunities for network research. To address these challenges, some extensions have been developed, such as `GrGym` (Zubow, Rösler, Gawłowicz, & Dressler, 2021), which is designed for radio communication, `MR-iNet Gym` (Farquhar, Kafle, Hamedani, Jagannath, & Jagannath, 2023) for wireless networks and `mobile-env` (Schneider, Werner, Khalili, Hecker, & Karl, 2022) which enables the use of DRL in wireless mobile networks.

Only two tools have been developed to allow a `Python` agent to interact with a network simulation: first, `ns3-gym` (Gawłowicz & Zubow, 2019); and `ns3-ai` (Yin et al., 2020). Both are based on the well-known `ns-3` network simulator, but they follow different approaches to implementing a Reinforcement Learning (RL) environment. The main difference is that `ns3-ai` makes use of a shared memory pool as a mechanism to connect the `ns-3` simulator with the `Python` framework (agent and environment). In contrast, the `ns3-gym` adopts a sockets-based approach.

However, `ns3-gym` and `ns3-ai` are suited for the single-agent DRL model and do not provide native support for the MA-DRL approach nor an isolation between the agents. Furthermore, these frameworks are designed for general non-overlay networks and do not support cloud overlay cases.

This lack of features limits the study of MA-DRL methods for the DPR problem in both general non-overlay and cloud overlay cases, which motivates the first goal of the thesis (G.1).

3.3 DRL for packet routing in general non-overlay case

In recent years, works addressing the DPR problem in the general non-overlay case by making use of DRL have been published in (Boyan & Littman, 1993; Mukhutdinov et al., 2019; You et al., 2022; L. Chen et al., 2021; Manfredi et al., 2021). The seminal paper (Boyan & Littman, 1993) was the first to apply a RL method, the Q-learning (C. J. C. H. Watkins, 1989), to DPR, giving rise to the *Q-routing* paradigm. In this work, routing decisions are made based on the packet destination. The study in (Mukhutdinov et al., 2019) applies the Deep Q-Network (DQN) framework to this *Q-routing* by using a Deep Neural Network (DNN) to approximate the Q-value function, yielding its deep learning version: the *DQN routing*.

After *DQN routing*, other proposals of MA-DRL applied to DPR were published in (You et al., 2022; L. Chen et al., 2021; Manfredi et al., 2021). These works concentrate on how to improve the quality of the learned routing from a Machine Learning (ML) perspective (i.e., changing the model, the input features, or the RL algorithm), neglecting networking aspects as the signalling. *DQRC (DQR with Communication)* (You et al., 2022) adds the action history, the future destinations, and the most loaded neighboring node to the packet destination as input. The neural network architecture of *DQRC* also considers an LSTM layer to exploit the new time series features (action history and future destinations). Authors in (Manfredi et al., 2021) enlarge the input algorithm with *relational features*, such as the node buffers' occupancy, the distance from the routing device to the packet destination, or the packet TTL field. The values of these features can vary for individual packets and devices. Still, the features list (and its size) is the same for all the routing nodes, which helps to improve the generalization among the nodes (the same neural network model is used for all the nodes). Finally, the work in (L. Chen et al., 2021) adapts the Proximal Policy Optimization (PPO) (Schulman, Wolski, Dhariwal, Radford, & Klimov, 2017) RL algorithm to the multi-agent scheme of the DPR problem, yielding the Multi-Agent Proximal Policy Optimization (MAPPO) algorithm. In MAPPO, the task to learn is considered as *to find a routing for a given traffic matrix*, using as input the packet destination and the buffers' occupancy. Finally, this work also aims to generalize the routing policies to different traffic matrices (i.e., tasks) by extending MAPPO to the meta-learning framework.

In all the above-mentioned works, the impact of network control signalling on the training process is ignored or superficially considered. In the more recent works (L. Chen et al., 2021; Manfredi et al., 2021), the training is centralized in a node that gathers all the past experiences (forwarding decisions) of all the forwarded packets at all the routers. The model for each routing node is learned at this central node using this large training set and is eventually pushed to the routing agents that will forward the packets. The significant overhead associated with these data transfers is not analyzed, questioning the actual gains of the proposed methods. On the contrary, the first works (Mukhutdinov et al., 2019; You et al., 2022) assumed a distributed training operation where training data are exchanged only locally between neighboring agents at the cost of the *non-stationarity* of multi-agent environments (Littman, 1994), which makes the agents' policies convergence harder. To tackle this problem, Foerster et al. (Foerster et al., 2017) propose that the neighboring agents share their policies (models) to stabilize the learning when larger datasets of past experiences are used (i.e., replay memories). However, current *DQN routing* such

as (Mukhutdinov et al., 2019) operates by sharing the values of model estimates (*value sharing*) instead of sharing the model at every neighboring node (*model sharing*). Value sharing reduces the signalling overhead but yields poor results when replay memories are used (known to stabilize the training of DQN (Mnih et al., 2015)). On the contrary, *DQRC* (You et al., 2022) shares the neighbors' routing models (*model sharing*), but at each training step, which introduces a significant communication overhead not properly evaluated.

The lack of investigations about the trade-off between performance and communication between the agents overhead in the general non-overlay case motivates goal G2.1 of the thesis.

3.4 DRL for packet routing in the cloud overlay case

Overlay networks can be implemented in different scenarios (Peterson & Davie, 2012), such as End System Multi-cast, Peer-to-Peer networks, resilient networks, or the Content Distribution Network (CDN) scenario. The routing protocol chosen depends on these cases. In general, Multiprotocol Border Gateway Protocol (MP-BGP) is the most commonly used protocol to manage a company's overlay network, whereas protocols like Open Shortest Path First (OSPF) or Intermediate System to Intermediate System (IS-IS) can be used to route in the underlay networks. MP-BGP is an extension of Border Gateway Protocol (BGP) that supports Virtual Private Networks (VPN) encapsulation, such as Virtual Extensible LAN (VXLAN), and can share reachability information across the overlay nodes. Even though this method is distributed, it suffers from scalability issues when the cloud overlay nodes are connected in a full mesh. This motivates confederations or route reflectors (RRs) (Bates, Chandra, & Chen, 2000).

Similar to BGP, MP-BGP shares a basic weight to distinguish a preferred path, which does not consider the rapid changes in traffic, topology, or the Quality of Service (QoS) requirements. Other methods have been proposed to tackle some of these limitations. Andersen et al. (Andersen, Balakrishnan, Kaashoek, & Morris, 2001) have optimized the network's resiliency by detecting path outages and periods of degraded performance and recovering from them fast. The authors of (Jones, Paschos, Shrader, & Modiano, 2014) propose an adaptation of backpressure routing for cloud overlay networks to handle traffic changes and maximize throughput.

In the context of Software-Defined Networking (SDN) (Masoudi & Ghaffari, 2016), cloud overlay networks can be managed by a centralized controller, which can exploit information collected by the cloud overlay nodes and design custom control plans. Industrial actors, like Cisco, propose their custom control plan, such as Cisco Overlay Management Protocol (OMP). In the latter protocol, the routes are computed in a centralized controller (vSmart) and are then propagated to the cloud overlay nodes (vEdges), which, by default, are connected in a full mesh. In addition to administrative weights, the controller can use Service Level Agreements (SLA) metrics, like the latency, jitter, and packet loss for each link (via Bidirectional Forwarding Detection (BFD) packets) to choose the overlay link in the case where there is more than one virtual link between two nodes (i.e., load balancing over virtual links). Other studies, like (Tootaghaj, Ahmed, Sharma, & Yannakakis, 2020) and (Quang et al., 2022), proposed cloud overlay network management in the context of Software Defined Wide Area Network (SD-WAN) by formulating the routing problem given the collected information about the nodes using linear programming.

The use of DRL in the general non-overlay case described in the previous section can be extended to the cloud overlay networks. Kamri et al. (Kamri, Quang, Huin, & Leguay, 2021) employed

a single agent DRL to determine the flow splitting ratios for balancing traffic over multiple virtual links to minimize the latency of tunnels while satisfying capacity constraints. The study held by Houidi et al. (Houidi et al., 2022) extends the previous one, using MA-DRL for scheduling flows in an SD-WAN scenario based on their traffic category. Botta et al. (Botta, Canonico, Navarro, Stanco, & Ventre, 2023) studied the scalability of such approaches in SD-WAN scenarios. All the above studies treated cloud overlay routing as a flow routing or a load balancing problem over known paths. Furthermore, the previous works worked under Centralized Traning Decentralized Execution (CTDE) scheme, where the agents take the actions in a decentralized way, but the training relies on a centralized controller or requires some centralization mechanisms (i.e., centralized reward, gathering global network metrics or paths...).

One of the contributions of this thesis (G2.2) is to consider the scenario as DPR problem in a Distributed Training Decentralized Execution (DTDE), which allows : (i) scalability in terms of the number of agents (nodes); (ii) high flexibility of the agents, since they can respond fast to the changes of network conditions (traffic and links capacity); (iii) low overhead compared to a CTDE approach since the agents are allowed only to communicate with their neighbors.

3.5 Communication between the agents

Most studies applying MA-DRL to networking problems have been devoted to designing effective architectures to reach the best performances, ignoring the impact of the communication overhead. However, sharing the DNN model weights like in *DQRC* (You et al., 2022) can have a significant impact on the congestion of the network during training and thus must be taken into consideration when designing MA-DRL models for tackling the DPR problem.

In terms of communication between the agents, we can distinguish two types:

1. The communication needed to train the models, which consists of the agent interacting with the other agents as part of its environment (retrieving the reward and other information necessary to compute the loss)
2. The communication established between the agents to enable an exchange of information between them and prevent the environment from becoming non-stationary (Matignon et al., 2012), and thus stabilize the training.

The first type of communication is often ignored, considering that the reward and observation are directly accessible by the agent without any cost. However, this assumption is not valid anymore in the DPR scheme. In this scenario, each agent retrieves information about its environment from its neighbors, which requires receiving communication packets.

The second type of communication gained more interest from the ML research community in recent years. Foerster et al. (Foerster et al., 2016) propose to design a communication channel between the agents and add the information gathered from the neighbors to the observation. Other studies like the one held by Zhang et al. (K. Zhang et al., 2018) developed more by restricting the communication only between the neighbors. Jiang et al. (Jiang, Dun, Huang, & Lu, 2018) extend this idea by introducing Graph Convolutional Network (GCN), which enables the agents to follow the changes in the network topology. The problem of non-stationarity is handled, in our case, by adopting the *Q-routing* scheme (Boyan & Littman, 1993 ; Mukhutdinov et al., 2019), where each agent has access to the DNN of its neighbors. However, this introduced a significant communication overhead, much more significant than the first type of communication.

One objective of goal G2.2 of this thesis is to address the problem raised by the two types of communication. We propose a novel communication between agents strategy with minimal overhead while preserving performance and stability.

3.6 Stability and safety of MA-DRL solutions

Agents often deal with conflicting or heterogeneous goals. In the context of the DPR problem, each agent seeks to optimize two goals: end-to-end packet delay and packet loss. One approach to address this issue is to design an appropriate reward function that describes all the goals. The authors of *Q-routing* (Boyan & Littman, 1993) and *Deep Q-routing* (Mukhutdinov et al., 2019) consider only the delay and ignore the loss. Other studies, such as (You et al., 2022), incorporate the loss as a pre-defined penalty in the reward. However, the stability of the training depends on the value of this loss penalty, and setting its value in practice can be challenging. If the loss is poorly handled, it can also raise a safety problem during the training, resulting in a high packet loss, which deteriorates the QoS. To address the above issue and properly set the weight for each goal, Constrained Markov Decision Process (CMDP) (Altman, 2021) can be considered. Those weights can be set dynamically using a Primal-Dual algorithm (Bhatnagar & Lakshmanan, 2012; Tessler, Mankowitz, & Mannor, 2018). The latest work, Reward Constrained Policy Optimization (RCPO) (Tessler et al., 2018), was adapted by (Kamri et al., 2021) to find load-balancing weights that minimize the end-to-end delay while respecting capacity constraints. However, their work used a centralized single-agent solution in a general non-overlay setting, where the agent can directly measure link capacities and traffic. This raises a challenge to adopt the RCPO to the cloud overlay scenario, which differs from the work of Khamri et al. (Kamri et al., 2021) in two significant ways : (i) multihop packet routing instead of flow routing over known paths; and (ii) routing in an overlay network, where information about the capacity and occupation of the physical links is hidden.

One objective of goal G2.2 is to stabilize the learning of the distributed MA-DRL solution for the DPR in the cloud overlay case. We propose to adapt the RCPO algorithm to this case and thus provide stability and safety for the model.

3.7 Conclusion

This chapter provides a literature overview of MA-DRL tools and techniques applied to networking, particularly focusing on packet routing challenges. Overall, the chapter highlights the potential of MA-DRL in solving complex networking challenges while acknowledging the need to overcome communication and stability issues in both general non-overlay and cloud overlay scenarios. It sets the stage for the rest of the thesis, where those challenges will be addressed.

In the following chapter, we will tackle the first challenge faced when adopting distributed MA-DRL methods to address the DPR problem, which is the lack of a realistic simulation framework.

CHAPTER 4

Packet Routing Simulator for Multi-Agent Reinforcement Learning

This chapter presents the software contributions of this thesis. It is divided into two parts. In the first one, we present `PRISMA` *(Alliche, Barros, et al., 2022): Packet Routing Simulator for Multi-Agent Reinforcement Learning. To our knowledge, this is the first tool specifically conceived to develop and test Multi-Agent Deep Reinforcement Learning (MA-DRL) algorithms for the Distributed Packet Routing (DPR) problem.*
Indeed, no MA-DRL tools have been developed to tackle the DPR problem, forcing the researchers to implement their own simplified Reinforcement Learning (RL) simulation environments, complicating reproducibility and reducing realism. We developed `PRISMA` *to overcome these issues and offer the community a standardized framework where: (i) the communication process is realistically modeled (thanks to* `ns3`*); (ii) distributed nature is explicitly considered (nodes are implemented as separated threads); (iii) and, RL proposals can be easily developed (thanks to a modular code design and real-time training visualization interfaces) and fairly compared each other.*
In the second part, we present ***prisma-v2*** *(Alliche et al., 2023), a new release of* `PRISMA` *for cloud overlay networks.* ***prisma-v2*** *brings a new set of features. First, it allows the simulation of cloud overlay network topologies by integrating virtual links. Second, this release offers the possibility to simulate control packets, which allows for a better evaluation of the overhead of the network protocol. Last, we integrate the modules along with the core (*`ns-3`*) to a Docker (Docker, 2020) container so that it can run on any machine or platform.* ***prisma-v2*** *is the first realistic overlay network simulation playground that offers the community the possibility to test and evaluate new network protocols based on the MA-DRL scheme.*

4.1 Introduction

In the last years, Reinforcement Learning (RL) (Sutton & Barto, 2018) using Deep Neural Networks (DNNs) (Bengio, 2009), also called Deep Reinforcement Learning (DRL), has obtained ground-breaking results in solving highly complex tasks, such as human-like performance results at Atari video games (Mnih et al., 2015) or beating AlphaGo (Silver et al., 2017) world champion. In communication networks, DRL has also been widely used in many networking technologies and problems. One of them is the DPR (Mukhutdinov et al., 2019; You et al., 2022; L. Chen et al., 2021; Manfredi et al., 2021). In this problem, no complete and centralized view of network topology and traffic demands is available, which poses a challenge. This is the case in multi-hop wireless networks (Tassiulas & Ephremides, 1992) or multi-domain optical networks (X. Chen et al., 2019). More precisely, in the DPR problem, each packet can be potentially routed differently regardless of the flow they belong to. Besides, the per-packet decisions are made locally by distributed agents placed at the routing nodes. These agents exploit local information, such as packet headers, the neighboring nodes, and link states.

Moreover, the DPR can be extended to cloud overlay networks. Overlay networks are virtual or logical networks built over a physical network (called underlay networks). Overlay networks provide flexible and dynamic traffic routing between nodes that are not directly connected by physical links but rather by virtual or logical links that correspond to paths in the underlying network. Those virtual links can be established using technologies like Generic Routing Encapsulation (GRE), Virtual Private Networks (VPN), or network virtualization. The underlay topology is managed by a third party, typically one or more network operators. One particular example of overlay networks is Software Defined Wide Area Network (SD-WAN) (Z. Yang, Cui, Li, Liu, & Xu, 2019), which fully utilizes the bandwidth of all available transport networks serving one location, like Multiple Protocol Label Switching (MPLS) fabric, Internet, and 5G, considering each one of them as an overlay link.

In the context of overlay networks, the problem of routing the traffic between the overlay links, especially in muti-hop scenarios, becomes challenging since the underlay routing policies are unknown and can involve different protocols, like Open Shortest Path First (OSPF), Border Gateway Protocol (BGP) and others. The absence of information about the underlay network topology and routing policies yields the existence of Triangle Inequality Violations (TIV) (Lumezanu, Baden, Spring, & Bhattacharjee, 2009): it is highly possible to find another path relayed by cloud servers which has a much lower delay than following the shortest path in the overlay topology. DRL can also be used to overcome the above challenges in this scenario. The DRL agent can exploit the information gathered from the network to overcome the lack of knowledge about the underlay network. An example is the work of Houdi et al. (Houidi et al., 2023). They followed a Centralized Traning Decentralized Execution (CTDE) approach (Oliehoek, Spaan, & Vlassis, 2008) to optimize load balancing weights for selected overlay paths in the context of SD-WAN.

However, in all the works mentioned above, whether it is in general non-overlay networks or overlay networks, the proposed DRL approaches were evaluated on ad-hoc discrete-time packet-level simulation environments (typically implemented in `Python`). Those simulation environments were tailored to the assumptions and simplifications made by each study. Namely, these works do not generally implement the MA-DRL agents as separated threads or processes (although they claim to be multi-agent studies), and they usually assume that at most one packet per router can be transmitted at each time step, which introduces an artificial synchronization into the routers' dynamics. This has two main consequences: (i) a reduced realism of the simulated communication

process and (ii) an obstacle to fairly comparing this DRL proposals to state-of-the-art approaches (e.g., Bellman-Ford *shortest path* routing algorithm (Bellman, 1958)) or even against each other. This lack of standardized Machine Learning (ML) tools in the networking community (Mestres et al., 2017; Gawłowicz & Zubow, 2019) represents a major issue in guaranteeing reproducibility. We point out that ML reproducibility issues are becoming a serious concern (Pineau et al., 2021).

Consequently, in the last years, some tools (Gawłowicz & Zubow, 2019; Yin et al., 2020) devoted to the application of RL to networks have been developed. These tools allow `Python` RL agents to interact with the popular `ns-3` network simulator (nsnam, s. d.). Nevertheless, these proposals are not natively compatible with the *multi-agent* setting of the DPR problem, where agents collaborate to take decisions in a distributed manner.

We first propose `PRISMA` (*PRISMA tool: An open MARL framework for packet routing.*, s. d.), an open-source RL simulation environment designed to overcome the aforementioned drawbacks in the application of MA-DRL to the DPR problem for the general non-overlay scenario. This simulator serves as a playground where the community can easily validate their own MA-DRL approaches and compare them in a realistic network simulation.

Then, given the challenging aspect of routing in cloud overlay networks, we propose a new release of `PRISMA`, namely **prisma-v2** (*PRISMA-v2: A Packet Routing Simulator for Multi-Agent Reinforcement Learning - Extension to CLoud Overlay Networks*, s. d.), which offers the possibility to experiment MA-DRL approaches in the context of cloud overlay networks.

Before diving into the technical details of the two proposed solutions, we present briefly their features.

4.1.1 Features of `PRISMA`

— A RL framework designed specifically for considering the distinctive characteristics of the DPR problem, serving as a playground where the community can easily validate their MA-DRL approaches and compare them.

— More realistic modeling of the communication process based on: (i) the `ns-3` (nsnam, s. d.) network simulator; and (ii) a multi-threaded implementation for each agent (no artificial synchronization between the nodes).

— A modular code design, which allows researchers to test their own RL algorithm without needing to work on implementing the environment.

— Visual tools based on `TensorBoard` (*TensorBoard: TensorFlow's visualization toolkit*, s. d.) allowing to track network and ML metrics during both the training and testing phases.

4.1.2 Features of prisma-v2

— Cloud overlay topology simulation and control management.

— Ability to add dynamic underlay traffic along with the overlay one.

— High reproducibility of results by supplying containerizing capability using `Docker` (Docker, 2020).

— Refactoring the code for better readability.

— Improve `Tensorboard` (*TensorBoard: TensorFlow's visualization toolkit*, s. d.) logging by incorporating both training and testing phases.

— Implement control signalling packets to simulate the communication between the agents and evaluate the overhead realistically.

The PRISMA source code is publicly available at (*PRISMA tool: An open MARL framework for packet routing.*, s. d.). The **prisma-v2** (*PRISMA-v2: A Packet Routing Simulator for Multi-Agent Reinforcement Learning - Extension to CLoud Overlay Networks*, s. d.) code source is available as the **v0.2** release of PRISMA.

The rest of this chapter is split into two main sections. First, in Section 4.2, we present PRISMA, our proposed realistic network simulation for experimenting MA-DRL on DRL problem for the general non-overlay setting. Second , in Section 4.3, we describe **prisma-v2**, the extension of PRISMA for overlay networks. Finally, we conclude in Section 4.4.

4.2 PRISMA: A Packet Routing Simulator for Multi-Agent Reinforcement Learning

4.2.1 PRISMA description

In the following, we present the technical details of PRISMA. First, a description of the framework's design is given. Then, each feature is explained, ranging from the network definition to simulation visualization using TensorBoard (*TensorBoard: TensorFlow's visualization toolkit*, s. d.).

4.2.1.1 Framework design principles

We consider the following principles:

1. *Training-Forwarding separation*: the agent training does not disturb the packet forwarding.
2. *Modularity*: code easy to reuse and modify.
3. *Realistic simulation*: implement the agent as close as possible to the real world.
4. *Fast prototyping*: easy and rapid modifications of the decision model.
5. *Online simulation tracking*: visualize in real-time the simulation evolution.

For point 1, we choose a multi-threading approach: each node will run on two separate threads (agents). One is dedicated to the training process, and the other to the decision process. Hence, the node agents can be trained at a fixed timestamp without disturbing the action computation at each packet arrival.

For point 2, we built node agents in an object-oriented manner, with separated functions for each task, so that a user can easily modify the agent's behavior without affecting the rest of the code. We document each part of the code and provide scripts for running the simulation.

Point 3 is a very important aspect of the PRISMA design since we need reliable results as close as possible to the deployment of the agents in the real world. For that, we keep the link propagation delay and use ns-3 to simulate the network. We have also integrated the action handling into the environment and added a Poisson random traffic generator.

For point 4 (which is close to point 2), PRISMA is separated into independent modules so that a user may easily modify the code. For example, changing the neural network or the model's inputs will not affect the other parts of the framework.

Finally, for point 5, we give the user the ability to monitor the simulation progress in real-time. For example, the user can track some network or ML metrics (e.g., average delay, buffers occupancy, error progression) to identify training issues in real-time. To do so, `TensorBoard` has been integrated into the framework offering network monitoring. In addition, it offers the possibility to generate custom plots.

4.2.1.2 Feature description

Figure 4.1 depicts the `PRISMA` code structure, showing in *green* our contribution over the existing `ns-3` and `ns3-gym` modules. We describe next these *green* modules.

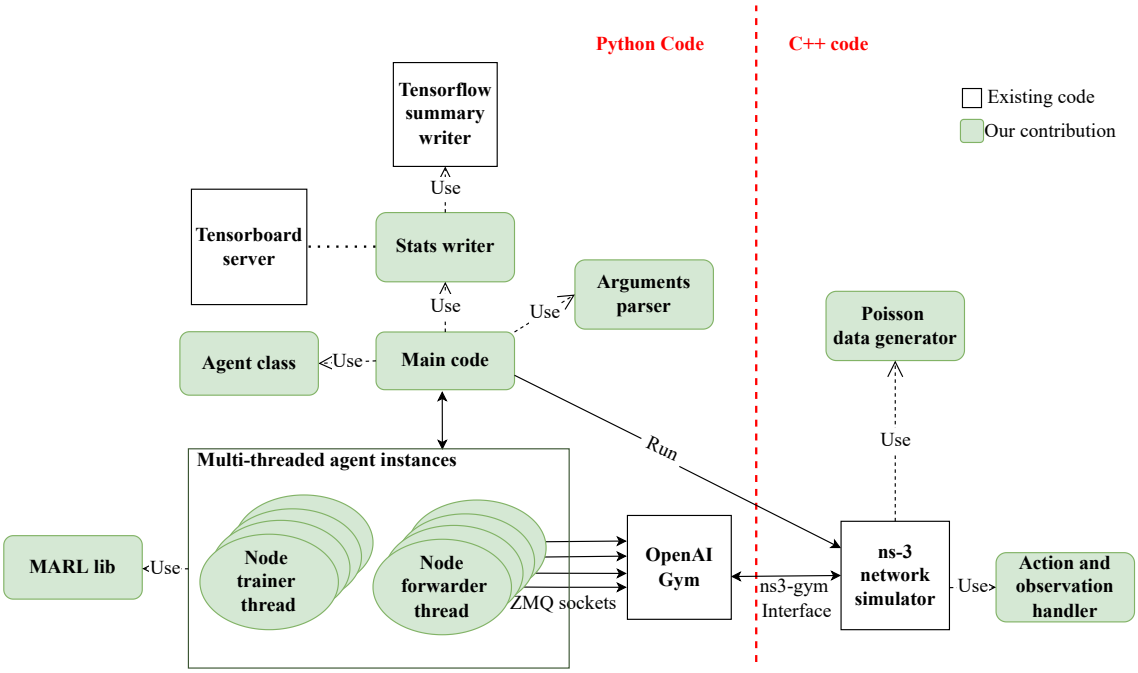


Figure 4.1 – `PRISMA` code structure, separated into two parts: `Python` code and `C++` code, highlighting our contribution in green. The figure shows the multi-threaded approach, where each node agent is composed of two threads: one forwarding the packets and another training the model.

`Main code`. The core of the tool enables to:

- Gather the user arguments from the `Arguments parser` and manage the simulation parameters.
- Instantiate the agents from the `Agent class` and create the threads for each node.
- Run `ns-3` simulator and `TensorBoard` server in separate processes.
- Log the simulation statistics using the `Stats writer`.
- Clean the environment when the simulation is done.

`Agent class`. This class represents a node agent containing two main methods: `run_forwarder` and `run_trainer`. The first one interacts with the `ns3-gym` environment as follows: (i) retrieve the observations, (ii) take action, (iii) compute the reward, (iv) store

the states' transitions in the experience replay buffers, and, (v) update the information about the environment. The second method trains the model, performing a gradient descent step. The `run_forwarder` also tracks the packets using their IDs to build the target value Y_n^Q at the node n, since the reward r, the next hop agent observation $o_{n'}$ and the next hop agent *Q-value* $Q_{n'}(\cdot;\theta_{n'})$ are associated with the packet ID.

`Multi-threaded agent instances`. These are `Agent class` instances. Each one corresponds to a network node. They share different information among themselves and with the `Main code` using static variables. As aforementioned, we decide to allocate two threads for each instance: one calling `run_forwarder` method and the other one calling `run_trainer` method to guarantee that training and forwarding can take place separately.

`Stats writer`. This module is called by the `Main code` and `Agent class` instances to log information about the environment or the training progress. The following variables are tracked to be visualized using `TensorBoard`: average packet delay, loss ratio, Temporal Difference error at each agent, replay buffers' occupancy at each node, exploration ratio for each node, the number of new arriving packets, the number of delivered packets, and, the number of buffered packets. We also add the ability to define custom plots and to compare between execution's hyperparameters.

`Argument parser`. This module is used to retrieve and parse the arguments given by the user in the script call.

`Poisson traffic generator`. This is a `ns-3` application class that generates random packets following a *Poisson* process at each node. The average rate is retrieved from a given traffic matrix, and the packet length is fixed to a given value. The application is not installed in the node itself but in a virtual one directly connected to the real node. To be able to transfer the packet from the generator to the network, we have created a corresponding virtual node that is directly connected to each node. The virtual node generates the packets and sends them to the real node.

`Action and observation handler`. The action a_n and the observation o_n are defined as the output network interface index and the packet destination, respectively. For the local observation o_n, other data, such as the buffer occupancy of the outgoing interfaces, could be considered. This module, implemented as a `ns-3` handler, retrieves o_n from the network simulation when a new packet arrives at the node n and forwards the packet to the output interface a_n. If the buffer interface is full, then the packet is discarded. This first version of `PRISMA` provides an observation containing the destination of the packet and the occupancy of each node's network interface buffers (in number of packets or bytes). We also provide the packet ID in the `info` field so that we can track each packet.

`MARL lib`. With this module, we extend the OpenAI `Baselines` library (*OpenAI Baselines: high-quality implementations of reinforcement learning algorithms*, s. d.) to the MA-DRL approach. The `MARL lib` provides the tools for defining, training, and testing an agent. It contains the following modules:

- `models`: a module containing the Deep Neural Network (DNN) models.
- `Replay buffer`: a class handling the experience replay buffer. It contains methods like `add`, `sample` or `save`.
- `agent`: a class defining an Deep Q-Network (DQN) agent in the context of DPR. It contains methods like `step`, `train`, and `sync neighbor target neural network`
- `utils`: a set of other useful functions like `save_model` and `load_model`.

4.2.2 Usage

We now describe the parameters of PRISMA, and we present the guidelines for using the framework.

4.2.2.1 Framework parameters

PRISMA parameters are separated into the following groups:

— *Global simulation arguments*: it contains the simulation time, the base port, the seed, and whether to run training.

— *Network parameters*: it contains network attributes like the path to the adjacency matrix, the maximum output buffer size, or the load factor.

— *DRL agent arguments*: it concerns the agent's training and contains parameters like the batch size, the learning rate, or the training time step.

— *Session logging arguments*: it contains parameters about the session, like the name of the session and the path to store the result of the simulation.

— *Other parameters*: some misc arguments like whether to run a TensorBoard server and its port number.

4.2.2.2 Usage guide

Before using the framework, the user must install the ns3-gym and the Python dependencies. To do that, we provide installation files.

The main prisma folder is composed of four folders: source, ns-3, examples, and scripts. The source folder contains the MARL lib, the agent class, and utils modules. The ns3 folder contains the ns-3 files: (i) the Poisson data generator, and; (ii) the ns-3 simulation itself (sim.cc). The latter file creates and runs the simulation scenario defined by the examples folder files, which define the network topologies and traffic matrices. Scripts for launching the simulations of the example (Section 4.2.3) are provided in the scripts folder.

The DNN model can be changed in models.py. The reward, observation, and action definitions can be modified in the methods _get_reward (in agent_class located in source.py), Get_Observation (in packet-routing-gym.cc) and Get_action (in packet-routing-gym.cc), respectively. Finally, a user may test the framework by calling main.py with the corresponding arguments. Calling the latter file with the "–help" argument will show all the possible parameters.

4.2.3 Illustrative example

We end this section by presenting an illustrative example. This example aims to show how PRISMA can be used to assist the training and testing of a DRL algorithm to solve the Distributed Packet Routing problem. Namely, we make a Deep Q-Network routing model (Mukhutdinov et al., 2019) learn the *Shortest Path (SP) routing* in backbone network scenarios. The tool can be applied to more challenging ad-hoc wireless networks by modifying the sim.cc file and properly adapting the DQN routing model.

4.2.3.1 Simulation settings

We ran PRISMA in two different machines: (i) a Dell Precision 7920 workstation equipped with an Intel Xeon Gold 6230R Dual CPU (26 Cores, 2.1-4.0GHz Turbo, 128 GB RAM) with 2 NVIDIA RTX A5000 GPUs; and, (ii) an Intel Inspiron 14 laptop equipped with an Intel Core i7-8565U CPU (Quad Core, 1.80 GHz, 8 GB RAM) with one NVIDIA GeForce MX130 GPU.

The DRL agent is implemented in TensorFlow (*TensorFlow: An end-to-end open source machine learning platform*, s. d.) as a three-layer neural network in models.py with a first dense layer of 32 nodes, then two layers of 128 nodes each having a Leaky ReLU activation function. This model considers only the packet destination as input and outputs the estimated end-to-end packet delay based on the selected network interface. The action is retrieved by applying an argmin function to the output layer. In the examples folder, we define the two network topologies considered in this experiment: *Abilene* (11 nodes) and *Geant* (23 nodes), as well as the traffic matrices (randomly generated using a uniform distribution). We fix the packet size to $512KB$, the link propagation delay to $2ms$, and the maximum output buffer length to 30 packets. We chose the reward r_n (the *next hop packet delay*) to be one, which means that the agent policy will aim to minimize the total number of hops from source to destination. These parameters, along with the model hyperparameters, are set as arguments when calling main.py. We expect that *DQN-routing* will have a similar performance to *Shortest Path routing* since both algorithms minimize the number of hops taken by a packet to reach its destination.

4.2.3.2 Model training

We show how the training process is performed successfully. The model is trained at each $7ms$ (network simulation time) with a learning rate of 0.001, a batch size of 512, a load factor of 50%, and a γ of 1. The duration of the simulation time for training was set to 30 seconds. Moreover, we use an ϵ-greedy approach (ϵ decays from 1 to 0.01) to move from exploration to exploitation.

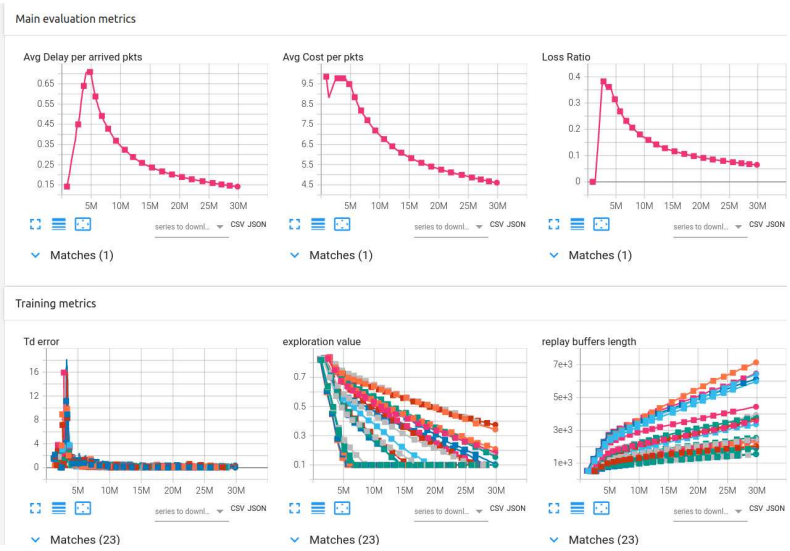
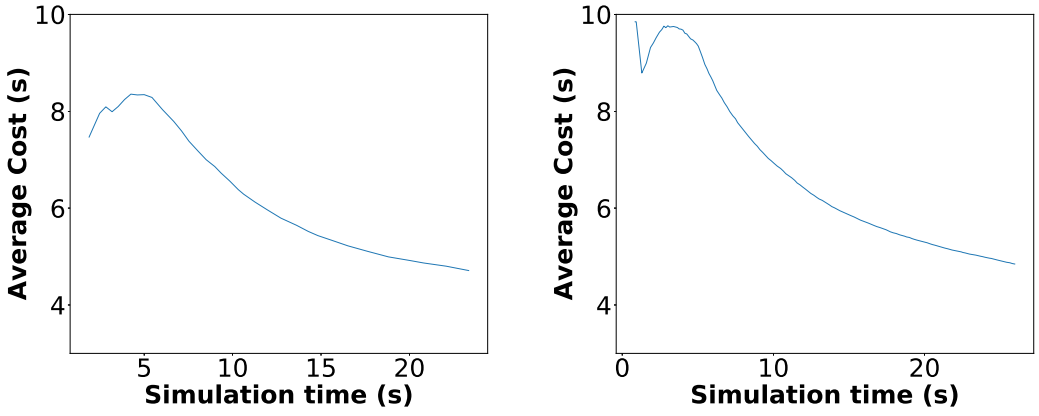


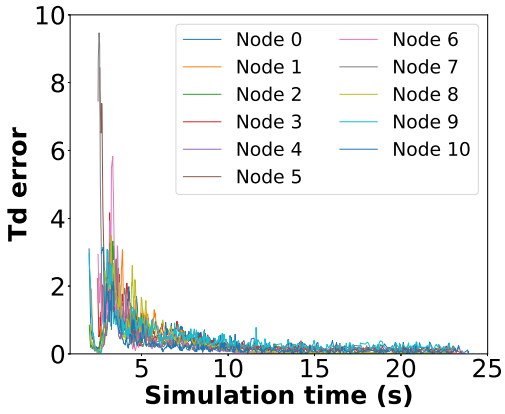
Figure 4.2 – Screenshot of TensorBoard interface. The $x-axis$ represents the time in ns, and the $y-axis$ represents the variables tracked during the simulation. For the training metrics (bottom subfigures), each curve corresponds to an agent.

We can watch the training progress thanks to `TensorBoard` visualization tools (see Figure 4.2). Figure 4.3 shows in detail the most relevant metrics to watch the learning progress for *Abilene* and *Geant*. Figures 4.3a and 4.3b show the average cost over the simulation time. The cost is computed by dividing the sum of the rewards over time by the total number of packets. Since the reward is one at each hop, the cost represents the average hop count per packet. We can observe an early cost increase due to the initial exploration: the network is flooded with packets needing many hops to arrive at their destinations since the forwarding decisions are mainly random. Afterward, we move progressively to the exploitation phase, where the decision comes from the DNN model. As we can see, the packets reduce the hop count to reach their destination since model decisions improve. Figures 4.3c and 4.3d show the Temporal Difference (TD) error for each node over the simulation time. We can observe that this error is decreasing, which means that all the agents are converging (learning) to a feasible routing policy (packets are not forwarded indefinitely). Therefore, the model could learn a SP routing.

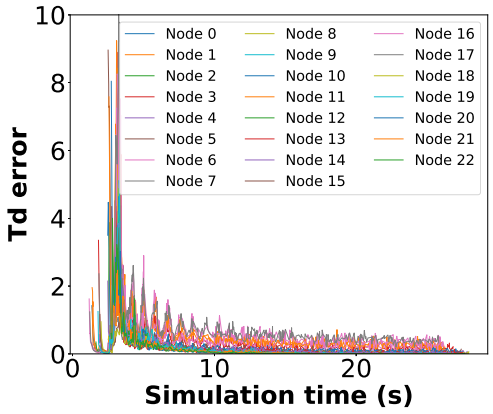


(a) *Abilene's* cost

(b) *Geant's* cost



(c) *Abilene's* TD error



(d) *Geant's* TD error

Figure 4.3 – The average cost per packet and TD errors at each node during training for Abilene (rightmost subfigures) and Geant (leftmost subfigures) topology.

4.2.3.3 Model testing

In this part, we evaluate the performances of the trained *DQN-routing* agents at different traffic intensities (low, medium, high, and very high traffic, in which the load factor was 50%, 100%, 150%, and 200%, respectively). In Figure 4.8, we compare the results with SP routing in terms of the *average end-to-end delay* and the *packet loss ratio*. For low and medium traffic rates, *DQN-routing* presents the same performance as *SP* since both methods use the shortest path decision policy, which is sufficient to handle all the packets at this rate. For high traffic rate, *DQN-routing* outperforms *SP routing* in terms of the *end-to-end delay* for both topologies; and, in terms of *loss ratio*, for Abilene. On the contrary, *SP routing* is slightly better in *loss ratio* in Geant. Anyway, these differences are not significant since we intend to learn a routing close to the SP, which is the case.

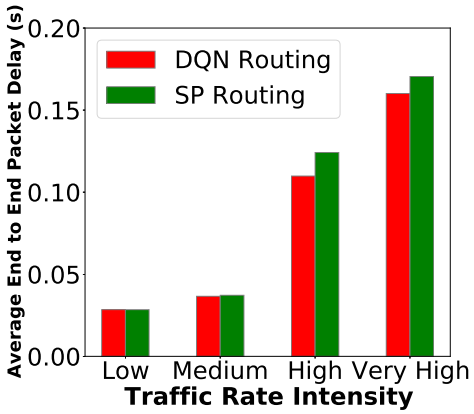


Figure 4.4 – *Abilene's* end-to-end delay

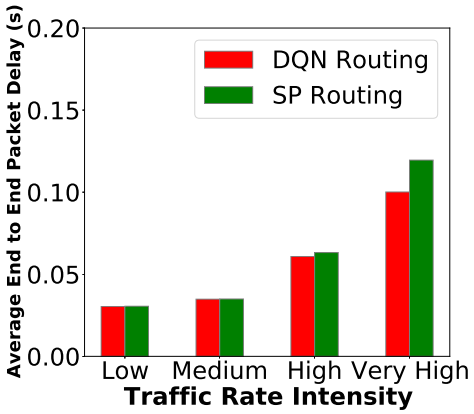


Figure 4.5 – *Geant's* end-to-end delay

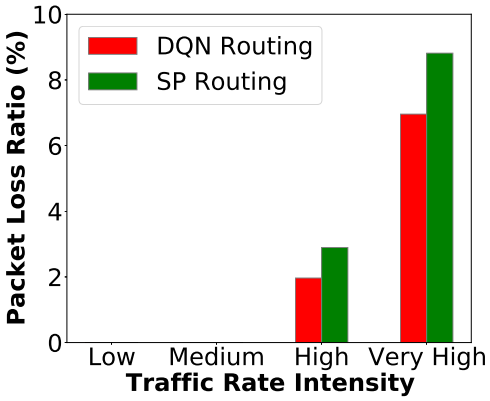


Figure 4.6 – *Abilene's* packet loss ratio

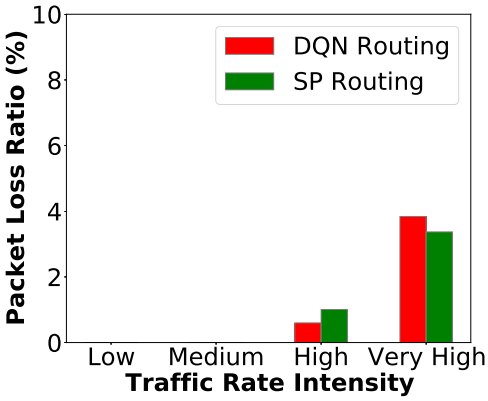


Figure 4.7 – *Geant's* packet loss ratio

Figure 4.8 – The average end-to-end delay per arrived packet and packet loss ratio for different traffic rate intensities for Abilene (right row) and Geant (left row) topology

4.3 prisma-v2: Extension to Cloud Overlay Networks

4.3.1 prisma-v2 Structure

prisma-v2 allows the usage of packet routing in overlay networks. The simulation framework creates two topologies (based on two adjacency matrices): a physical topology containing the physical links between the nodes and an overlay topology where some nodes are connected through virtual tunnels. This overlay topology is the environment used to define the routing policy. The main features of this release are illustrated in Figure 4.9.

In the following, we will explain the different components of **prisma-v2**. Based on Figure 4.9, we can distinguish two parts: `ns-3` part and `Python` part. We first present the `ns-3` part, then the `Python` part, and finally, we will present the changes in the `Tensorboard` logger.

4.3.1.1 `ns-3` part

The **prisma-v2** allows controlling the packet routing policy in overlay networks and also simulating control signalling packets. The simulation framework creates two topologies: an underlay topology containing the physical links and an overlay topology where some nodes are connected through virtual tunnels. This overlay topology is the environment used to define the routing policy.

We implemented three types of control signalling packets: the *target update packets* generated at a fixed time interval and used to simulate the transfer of neural network weights between neighbors; the *replay memory update packets* generated as a response to data packets, to transport the new observation and the reward; and the *tunnel delay estimate packets* created according to a parameterized number of data packet transmissions and used to collect the overlay link's delay.

For managing the data and control signalling packets, we developed the class `PacketManager`, inherited by five specific packet manager classes.

The `SmallSignallingPacketManager` manages the arrival of *replay memory update packets*. The class extracts the overlay link's delay measured during the sending of the data packet.

The `BigSignallingPacketManager` manages the*target update packets*, extracting the information about the neural network weights shared by a node. The `PingForwardPacketManager` receives a *tunnel delay estimate packet* sent and calculates the overlay link's delay in the overlay scenario. The `PingBackPacketManager` receives the response of the *tunnel delay estimate packet* containing the tunnel delay and stores it.

The `DataPacketManager` manages the data packets. It collects the observation information to transmit to the agent. Furthermore, it sends the packet for the next hop based on the agent's action. This class communicates with the `ComputeStats` class, which is responsible for the statistics of the network simulation, such as the number of injected packets by the node, the number of lost packets by the node, and the end-to-end delay of the arrived packets.

4.3.1.2 `Python` part

The left side of figure 4.9 presents the `Python` code structure of **prisma-v2**. Like the first version, the program is launched by calling `main.py` with the simulation arguments.

Packet Forwarder agent objects are instantiated from the `Forwarder` class stored in `forwarder.py`. In contrast, trainer daemon agent objects are instantiated from the `Trainer` class stored in `Trainer.py`. Both classes are children of the `Agent` class stored in `agent.py`. The agent class supplies the blueprint for the two child classes. It implements the static method

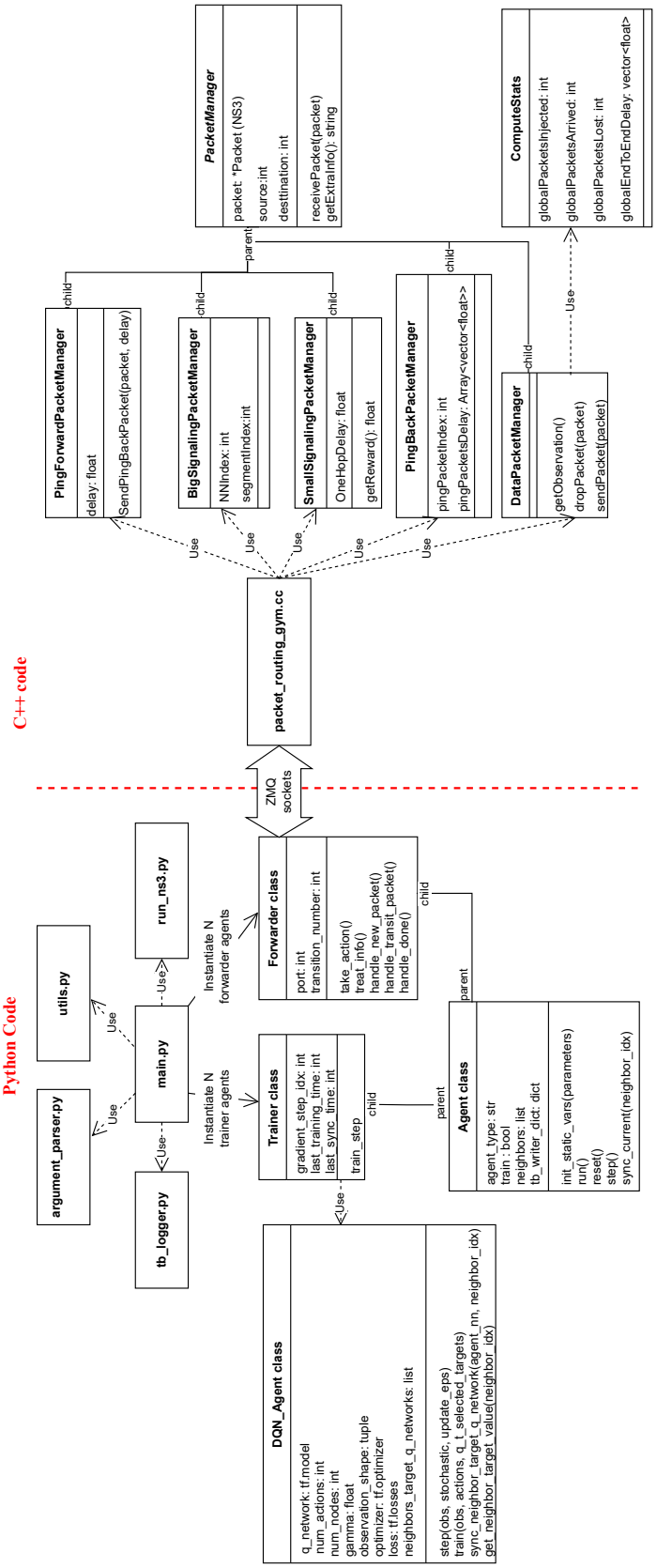


Figure 4.9 – **prisma-v2** code structure

`init_static_vars`. This method sets the static variables used to share information between the different agents and the main program.

The `Agent` class implements the methods `run, reset, and step`, which will be overridden by the children's classes. For the `Trainer` class, the run method will launch the daemon main loop that checks if it is the time to train and calls the `step` method to run a training step. For the `Forwarder` class, the `run` method will start the agent episode and connect it to the network node in `ns-3`. It calls the `step` method to run a transition (waiting for a packet to arrive at the node, taking an action using the `take-action` method, and handling the information returned as a response to the action). We use the field `info` to send relevant information collected from `ns-3` to the `python` agent, like the packet's size or the environment statistics. This field is treated by `treat-info` method of the `Forwarder` class. The `reset` method of the `Forwarder` class will reset the agent environment before starting a new episode.

Along with the agent's classes, we provide the `DQN_Agent` class in `agent.py`, which provides the necessary methods to use an adaptation of DQN model for the Distributed Packet Routing problem like in (You et al., 2022) and (Alliche, da Silva Barros, et al., 2022). This class relies on the files `replay_buffer.py` and `models.py` for the experience replay buffer and the neural network architecture, respectively.

4.3.1.3 `TensorBoard` logger part

In the first release, we used to compute many network metrics (like the delay and loss) in the Python part, which we found is not optimal regarding the realistic claims of `PRISMA`. In **prisma-v2** release, the network metrics (end-to-end delay, packet loss, control overhead) are computed by the `ns-3` part and transmitted to the `python` for being displayed in the `TensorBoard` logger. Moreover, the `TensorBoard` displays in real-time all the metrics to the user during the simulation.

We also improved the logger to automatically store the train and test stats in the same instance. The logs are saved in the session folder, given by the argument `session_name`, respectively.

4.3.2 Usage

In this subsection, we will go through different parts of the code, and explain the basic use cases that a user may encounter to launch a simulation.

4.3.2.1 Installation

After cloning the repository, the user can run **prisma-v2** by three different ways:

— Run locally by installing the required packages and dependencies by calling `install.sh`.

— Create a local docker image by running the docker build command on the root folder. This is possible since we provide a docker definition file. This will copy all the folders in the image so that the user may run **prisma-v2** by calling the docker run command and binding the results' folder to be able to retrieve the results in the host machine.

— Pull a docker image provided in Docker Hub. This image only contains the Linux environment with the requirements installed, so the user needs to bind the **prisma-v2** folder to the image.

We have provided an illustrative example in the `readme.md` file, guiding the user from the installation to training a Multi-Agent Deep Q-Network (MA-DQN) model to solve the DPR problem in an overlay network.

4.3.2.2 Use cases

Modifying or Adding parameters.
Like the previous version, the parameters are organized by groups and can be visualized by calling "python3 main.py –help". They are accessible in the file `argument_parser.py`, where the user can add a new parameter or modify an existing one. In the find `run_ns3.py`, we parse the arguments to `ns-3`, so the user can modify this function to pass arguments to the NS-3 part.
Changing the DRL algorithm.
The state can be modified for in `GetObservation` method of the `DataPacketManager`. The user may modify the DRL model input shape to match the new observation shape. Pre-defined neural network models are already implemented in `models.py`, and a DRL algorithm is implemented in the `DQN_Agent` class. The user may change the latter class to change the learning algorithm.
Sharing the information between the agents.
The agent forwarder and trainer objects share information with the main process using the Agent class static variables. In those variables, we can find shared attributes between the agents like the `DQN_Agent` objects and `pkt_tracking_dict` which tracks transiting packets in the network.
Changing the topology settings.
To change the overlay topology configuration, the user may change some parameters: `physical_adjacency_matrix_path`,
`overlay_adjacency_matrix_path` and `map_overlay_path` for the paths to the underlay topology's adjacency matrix file, the overlay topology's adjacency matrix file, and the map between the indices of overlay and underlay nodes file, respectively; The traffic in underlay topology is handled, by default, using the OSPF protocol, which is used for computing the routing tables. Changing the routing policy is possible by the native `ns-3` method `RecomputeRoutingTables`.
Customize control packets.
For customizing the control packets, the user may create a new class that inherits the class `PacketManager`. For example, The `receivePacket` method recovers the packet information at its arrival, and the method `GetInfo` encapsulates the useful collected information to send it to the agent. For generating control signalling packets, the user may be inspired by the methods `sendSmallSignalingPacket` and `sendPingForwardPacket`, providing the packet size and the information which the control signalling packet should encapsulate. For the *target update packets*, the parameters `syncStep` and `bigSignalingSize` contain the period of time for sharing the model weights and the weights' total size, respectively. For creating new control signalling mechanisms, the user may be inspired by the current mechanisms developed in **prisma-v2**.

4.4 Conclusion

In this chapter, we first presented the PRISMA tool, which is, to the best of our knowledge, the first DRL framework specifically devoted to the DPR problem. PRISMA aims to provide a playground for researchers interested in applying this machine learning paradigm to this challenging problem, allowing fast prototyping and benchmarking. We illustrated its main functionalities by applying the tool to learn in a distributed manner a SP routing policy in two backbone networks.

Then, we presented **prisma-v2**, a new release of PRISMA, extending this tool to cloud overlay networks. This version adds a new set of features, like offering the possibility to add a control signalling packet and measure the impact of having communication between the agent, and so, evaluate the real cost of implementing MA-DRL solutions for DPR in overlay networks.

By providing these two contributions, we can realize realistic simulation and evaluate MA-DRL methods for the DPR in both overlay and general non-overlay network scenarios.

In the next chapter, we will tackle the second scenario, which is the general non-overlay networks. In this case, we evaluate the existing solutions in terms of performance and the overhead due to the communication between the agents. We propose a distributed MA-DRL framework to improve those solutions.

CHAPTER 5

Distributed Learning-based Packet Routing in general non-overlay networks

In recent years, several works have studied Multi-Agent Deep Reinforcement Learning (MA-DRL) for the Distributed Packet Routing (DPR) problem. Unfortunately, these previous works focus on an ideal scenario where the impact of control signalling is neglected, and network simulation is tailored to simplistic assumptions. This chapter presents the first experimental investigation of control signalling mechanisms for distributed learning-based packet routing. We rely on `PRISMA`*, our open-source simulation* `ns-3`*-based module.*

First, we compare two signalling mechanisms between agents (value sharing and model sharing) used in the literature. We investigate the net gains considering off-band signalling and show that routing policies close to those provided by an oracle with full knowledge of traffic and network topology can be discovered with a control overhead of $150\ \%$ *with respect to injected data packets if neighboring agents share their Deep Neural Network (DNN) models. We discuss the generality of our results to underline the importance of assessing net gains of MA-DRL-based routing.*

Then, we propose a novel signalling mechanism called logit sharing. This mechanism significantly reduces the communication overhead while maintaining performance similar to model sharing.

Lastly, we reduce even more the communication overhead by implementing a dynamic control packet sending between the agents. In that manner, the agents will only communicate when necessary, avoiding unnecessary control packet exchange.

5.1 Introduction

After the breakthrough results obtained by Deep Reinforcement Learning (DRL) (Bengio, 2009) in solving highly complex tasks (Mnih et al., 2015), DRL has started to be applied to communication network problems. One of them is the Distributed Packet Routing (DPR) (Mukhutdinov et al., 2019 ; You et al., 2022 ; Manfredi et al., 2021 ; L. Chen et al., 2021). This is a challenging problem because there is no complete and centralized view of network topology and traffic demands (e.g., multi-hop wireless networks (Tassiulas & Ephremides, 1992) or multi-domain optical networks (X. Chen et al., 2019)). In the DPR problem, distributed agents make per-packet forwarding decisions locally, exploiting local information, such as packet headers and neighboring node states. Nevertheless, these works (Mukhutdinov et al., 2019 ; You et al., 2022 ; Manfredi et al., 2021 ; L. Chen et al., 2021) focused on improving the learning performance by modifying the training mechanisms and the model design, neglecting the impact of control signalling on the routing performance. Indeed, the *distributed multi-agent* setting of the DPR problem imposes a non-negligible level of communication between the neighboring agents during the training phase, which is meant to run continuously or frequently to adapt the routing policy to any network change (traffic, topology). This information exchange constitutes the control signalling, and it introduces a certain level of *overhead*. Thus, a trade-off appears between this *overhead* and the *quality* of the learned routing: a minimal amount of signalling (*overhead*) is required to make the agents learn a routing policy at the cost of increased bandwidth requirements. On the other hand, an excessive control *overhead* could slow down data packets and impede the learning process.

This chapter focuses on the study and the reduction of communication overhead between the agents in the general non-overlay scenario. It hence has two parts. The first one focuses on the impact evaluation of this control signalling. To perform the above-mentioned impact evaluation, a realistic modeling of the communication is necessary, in contrast to the simplified simulations performed with ad hoc network simulators in the previous works. For this reason, we use our `PRISMA` tool, detailed in the previous chapter.

In the MA-DRL framework, neighboring agents collaborate by exchanging information, particularly the estimated packet end-to-end delay. In the literature, we can distinguish two ways of sharing this information: one is to share the current value of the estimate (*value sharing*), and the second one is to share the estimating model itself (*model sharing*).

The second part of this chapter proposes two novel methods to reduce the communication overhead between the agents while maintaining the performance. The first method is called *logit sharing*, which is a novel way to share the estimated packet end-to-end delay. It takes the best from both *value sharing* and *model sharing*. The second method consists of dynamically sending control packets by allowing each agent to compute the relevancy of a control packet and decide whether to send it or not. Both methods are evaluated and compared to *value sharing* and *model sharing* using **prisma-v2**.

To summarize, the contributions presented in this chapter are listed as follows:

— To the best of our knowledge, we present the first experimental investigation of control signalling mechanisms for distributed learning-based packet routing. We rely on `PRISMA`, which enables testing MA-DRL-based routing in a reproducible and realistic manner (Alliche, Barros, et al., 2022).

— We compare two signalling mechanisms under the *DQN routing* (Mukhutdinov et al., 2019) framework, where routing nodes periodically communicate copies (*target*) of their Machine

Learning (ML) models to their neighbors (named *model sharing*) or only communicate delay estimates (named *value sharing*).

— We investigate net gains considering off-band signalling and show that (i) *model sharing* between agents is necessary to find routing policies close to the optimal routing of an Oracle with perfect knowledge of topology and traffic, but at the cost of possibly significant control overhead during training (150 % of extra traffic in the considered network settings with respect to data traffic); and (ii) *value sharing* incurs in much more reduced overhead (10 %), but barely improves a Shortest Path (SP) routing policy. We discuss the generality of our results to underline the importance of assessing net gains of MA-DRL-based routing.

— We propose a novel signalling mechanism (named *logit sharing*), which takes the best of the two signalling mechanisms: minimizing the communication overhead to a level comparable to *value sharing* while keeping the high performance as *model sharing*.

— We implement a dynamic control packet mechanism, where the agents can decide whether to send the control packet or not based on the relevancy of that packet.

— We evaluate both methods, and we show that *logit sharing*, along with the dynamic control packet mechanism, keeps similar performance to *model sharing* while reducing the communication overhead by up to 86% compared to *model sharing*.

Section 5.2 evaluates the impact of the control signalling by comparing two signalling mechanisms between agents (*value sharing* and *model sharing*) used in the literature. First, we present a formulation of the problem (Section 5.2.1) and propose an oracle routing policy based on a Minimum Cost Multi-Commodity Flow (MCF) formulation (Section 5.2.2). Then, the MA-DRL framework and the control signalling mechanisms are described in Section 5.2.3 and Section 5.2.5, respectively. Simulation results are analyzed in Section 5.2.6 and discussed in Section 5.2.7.

In Section 5.3, we propose a novel signalling technique, *logit sharing* that aims to overcome the limitations of the techniques evaluated in Section 5.2. Additionally, we propose a dynamic control packet-sending mechanism, which is explained in Section 5.3.2. Then, the two proposed methods are evaluated in Section 5.3.3 and discussed in Section 5.3.4.

Finally, Section 5.4 concludes the chapter.

5.2 Impact Evaluation of Control Signalling onto Distributed Learning-based Packet Routing

5.2.1 Network Model

Let $\mathcal{G}(\mathcal{N}, \mathcal{E})$ be a directed network graph, where $\mathcal{N}$ is the nodes set and $\mathcal{E}$ is the unidirectional links set. A link $(a, b) \in \mathcal{E}$ initiates at node a, terminates at node b and has a capacity of C traffic units (i.e., $Kbps$ or $packets/sec$). The incoming and outgoing neighbor nodes of the node $n \in \mathcal{N}$ are denoted as $\mathcal{N}_n^{in}$ and $\mathcal{N}_n^{out}$, respectively. The traffic matrix $\mathcal{H} = \{h_{sd}, (s, d) \in \mathcal{N} \times \mathcal{N}\}$ counts up the average volumes h_{sd} in traffic units (i.e., $Kbps$ or $packets/sec$) of packet flows between the node pairs $(s, d) \in \mathcal{N} \times \mathcal{N}$. Each node $n \in \mathcal{N}$ is a router equipped with $|\mathcal{N}_n^{out}|$ outgoing network interfaces. Each interface $i \in |\mathcal{N}_n^{out}|$ has its buffer queue of size B. The queue follows a $FIFO$ (*First-In First-Out*) policy. Hence, if a packet arrives at a full buffer, it is rejected (i.e., *lost*). Otherwise, the packet is admitted, eventually getting to the buffer head, and then forwarded to a next-hop node $n' \in \mathcal{N}_n^{out}$. This procedure is repeated at each hop until the packet finally reaches its final destination d (or is lost elsewhere en route). Table 5.1 gathers the notation.

Name	Description
$x_{ab}^{sd} \in [0,1]$	Fraction of the packet flow between the node pair (s,d) to be forwarded via the link (a,b).
$y_{sd} \in [0,1]$	Fraction of the packet flow between the node pair (s,d) that is rejected.
$h_{sd} \in \mathbb{R}_{\geq 0}$	Average flow volume between the node pair (s,d) in traffic units (i.e., $Kbps$).
$C \in \mathbb{R}_{\geq 0}$	Link capacity in traffic units (i.e., $Kbps$).
$B \in \mathbb{R}_{\geq 0}$	Buffer size in data units (i.e., $Bytes$).
$M \in \mathbb{R}_{\geq 0}$	Large number to penalize a loss more than a large delay.

Table 5.1 – General non-overlay model notation

5.2.2 An Oracle Routing Policy

In this subsection, we propose an *oracle* policy to be used as a routing optimality benchmark. We devise a centralized version of the DPR problem, where an *Oracle* observer (different from the routing nodes) has a full knowledge of the network topology and the traffic matrix. This centralized version is formulated as the Minimum Cost MCF (Min Cost MCF) problem solved by the Linear Programming (LP) model (5.1). The optimal solution of this model provides a lowest-cost routing policy, which we call *oracle routing* in the remainder of this chapter.

$$\min_{\{\S,\dagger\}} \sum_{\substack{s\in\mathcal{N}\\ d\in\mathcal{N}}} h_{sd} \left(\sum_{(a,b)\in\mathcal{E}} x_{ab}^{sd} + M \cdot y_{sd} \right) \tag{5.1a}$$

$$\text{s.t.} \sum_{a\in\mathcal{N}_n^{in}} x_{an}^{sd} - \sum_{b\in\mathcal{N}_n^{out}} x_{nb}^{sd} = \begin{cases} -1 + y_{sd}, & \text{if } n = s \\ 1 - y_{sd}, & \text{if } n = d \\ 0, & \text{otherwise} \end{cases}$$

$$s \in \mathcal{N}, d \in \mathcal{N}, n \in \mathcal{N} \tag{5.1b}$$

$$\sum_{\substack{s\in\mathcal{N}\\ d\in\mathcal{N}}} h_{sd} \cdot x_{ab}^{sd} \leq C, \qquad (a,b) \in \mathcal{E} \tag{5.1c}$$

$$x_{ab}^{sd} \in [0,1], \qquad s \in \mathcal{N}, d \in \mathcal{N}, (a,b) \in \mathcal{E} \tag{5.1d}$$

$$y_{sd} \in [0,1], \qquad s \in \mathcal{N}, d \in \mathcal{N} \tag{5.1e}$$

The objective function (5.1a) minimizes a twofold objective: (i) primarily, the total amount of rejected traffic $\sum h_{sd} \cdot y_{sd}$; and, (ii) secondly, the average hop count $\sum h_{sd} \sum x_{ab}^{sd}$. Optimizing (ii) minimizes the average end-to-end delay in this model. The flow conservation constraints (5.1b) ensure that all the traffic admitted at the source is routed until the destination. The constraints (5.1c) limit the link capacity. Finally, the constraints (5.1d) and (5.1e) define the lower and upper bounds of the variables.

The Oracle policy is not necessarily optimal for the DPR problem since it was computed to solve a centralized and flow-based version of the routing problem. The LP model (5.1) works with averaged packets' flows and is unaware of the traffic dynamics at the packets' scale (e.g., a burst of packet arrivals whose data rate exceeds nominal link capacities C). Hence, a per-packet routing algorithm could find policies exploiting outgoing buffers, which are unknown for the LP model (5.1), able to beat this *Oracle routing* in some circumstances (as we will see in Section 5.2.6). Anyway, we use it since it constitutes a high-quality benchmark that is difficult to attain without a global view of the exact network topology and the precise traffic matrix.

5.2.3 The DPR Problem and our *DQN routing* Algorithm

As described in the Background section 2.3, the DPR problem consists of deciding, at each node, the output port (next hop) for each upcoming packet. The objective is to find a global routing policy that minimizes the end-to-end delay for all the packets over the network.

Let $\mathcal{N}$ be the set of routing nodes (*agents*), where each *agent* n has its own local observation space $\mathcal{O}_n$ and its own action space $\mathcal{A}_n$. When a new packet arrives at time t at the node n, the node n selects as action a_n the *next hop node* n' to forward the packet to. This decision is taken depending on the local observation of the router o_n.

As a consequence of the decision, the *agent* n receives a reward $r_{n'}$ from the next hop node n': the *next-hop packet delay* (i.e., the packet delay to travel from n to n'). Hence, a transition is made to a new state. When the packet arrives at a node, this procedure is repeated. The action a_n is selected to minimize an estimate of the expected *end-to-end packet delay* from the node n to its final destination.

In the next paragraphs, we detail (i) the *node observation* representation, (ii) the *node action*, and (iii) the *reward* definition.

Node observation.

The *node observation* $O_n \in \mathcal{O}_n$ is represented as $o_n = \left(d, \{b_i, i \in 1 \dots |\mathcal{N}_n^{out}|\}\right)$. This is the concatenation of *two* components: (i) *the current packet destination* d and (ii) *the buffers' occupancy* b_i, in $Bytes$, of each node interface i at the node n.

Node action.

The *node action* $a_n \in \mathcal{A}_n$ is the choice of the out-neighbor $n' \in \mathcal{N}_n^{out}$ of the node n (i.e., the outgoing interface to this next hop node n') to forward the packet to the buffer head.

Reward.

The *reward* $r_{n'}$ is the *next-hop packet delay*, defined as the time required by the packet to travel from the buffer tail of n up to the buffer tail of the next hop node n'. Then, it is computed as $r_{n'} = l + q$, where: (i) l denotes *the link transmission delay*, the transmission latency in the link connecting n and n'; and, (ii) q refers to *the queuing delay*, the time spent by the packet in the outgoing buffer of node n taking to n'. If in the next hop node n', the outgoing buffer where the packet should be forwarded is full, the packet is lost, and $r_{n'}$ is considered infinity since the packet did not arrive at n'. In practice, we use the *worst-case* end-to-end delay: $\frac{|\mathcal{N}| \times (B+1)}{C}$, that is, the delay of traversing over the $|\mathcal{N}|$ nodes when all the output buffers are full. Thus, the reward can be measured in time units (typically, in $seconds$). This reward definition allows the *end-to-end delay* estimate $Q_n(\cdot; \theta_n)$ to account for both buffer delays and packet losses, but giving more weight to the packet losses to favor its minimization in priority, similarly to the LP model of the *oracle routing* (see Section 5.2.2).

5.2.4 Neural Network Architecture

In this subsection, we detail the architecture of the neural network $Q_n(s_n, a_n; \theta_n)$, which is depicted in Figure 5.1. This architecture is inspired by (You et al., 2022).

The input layer is split into two parts according to the description of the node state s_n in Section 5.2.3: (i) *the packet destination*, as a $|\mathcal{N}|$-element vector in *one-hot encoding* (i.e. the position corresponding to the destination is set to one, the rest to zero); and, (ii) *the buffer occupancy of outgoing buffers of node n*, as an $|\mathcal{N}_n^{out}|$-element vector, whose values are layer normalized.

The first hidden layer is also split into two parts, each being a 32-neuron fully connected layer fed by their respective input. Afterward, their outputs are concatenated before feeding two 64-neuron fully connected layers.

Finally, the output layer is fully connected with as many neurons as the node action space (i.e., out-degree $|\mathcal{N}_n^{out}|$ of n). The value of each output neuron is simply the estimate of the *Q-value* $Q_n(s_n, a_n; \theta_n)$. All the activations are ELUs (Exponential Linear Units).

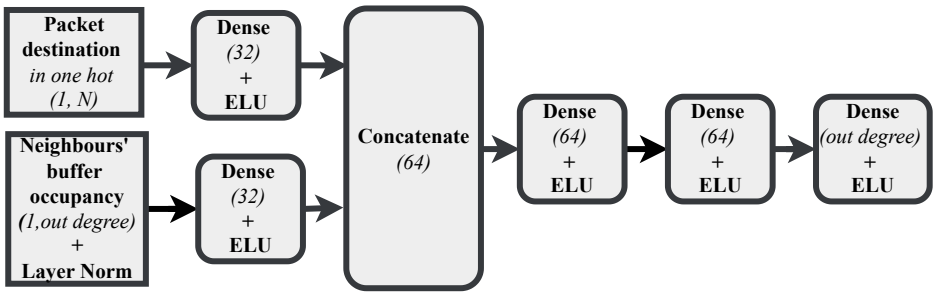


Figure 5.1 – NN layers architecture

5.2.5 MA-DRL Signalling: *value sharing* or *model sharing*

DPR consists of forwarding decisions taken in a distributed manner without cooperation between the routing nodes. However, with MA-DRL, control signalling exchanges are necessary to enable agent training since agents need to share their past observations (i.e., experiences), network metric estimates, and ML models before the training takes place.

We consider two agent information-sharing techniques proposed by the literature: (i) *value sharing* and (ii) *model sharing*. The main difference between the two techniques is how the loss function (defined in Equation (2.9)) is computed during the training phase. We present in the following each of these techniques along with the control packets associated with it.

5.2.5.1 *Value sharing*

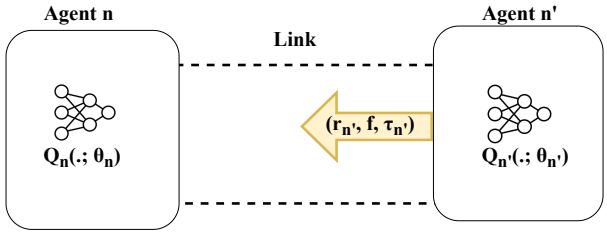
Value sharing was used by Mukketdidinov (Deep Q Routing) (Mukhutdinov et al., 2019). In this technique, the target value $\tau_{n'}$ is collected directly from the neighbor node n' control packet that we refer to as *replay memory update packet*. These packets are used to complete the entries in the local *replay memory* of each agent with the neighbors' information since a state-action transition $(o_n, a_n, r_{n'}, \tau_{n'}, f)$ involves two nodes: the forwarding node n and the next hop n'. More precisely, when a node n forwards a data packet to the next hop n', the corresponding observation-action pair (o_n, a_n) is stored in the *replay memory* $\mathcal{M}_n$. Once the data packet arrives at n', a *replay*

memory update packet is generated and sent back to n. This packet contains the tuple $(r_{n'}, \tau_{n'}, f)$ where $r_{n'}$ corresponds to the reward observed at the node n', which is the next hop delay for the packet to go from the node n to the next hop n'. $\tau_{n'}$ is the estimate of *the remaining end-to-end delay* computed by the next hop agent n' at the reception of a packet at n' as Equation (2.10). f indicates whether the packet arrived at its final destination. These next node information are required to build the current entry in the *replay memory*. This communication is needed during the training phase, adding a moderate overhead per data packet.

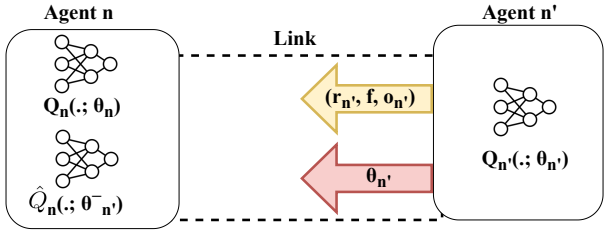
5.2.5.2 *Model sharing*

Model sharing is a term coined by us in (Alliche, da Silva Barros, et al., 2022), but it was first used in *DQRC* (You et al., 2022). In this technique, instead of collecting the target value $\tau_{n'}$ from the neighbor n', the node n computes it locally as in the Equation (2.10). For that, the node n needs a copy of the next hop agent DNN $Q_{n'}(\cdot, o_{n'}; \theta_{n'})$. We refer to this copy as *target network*, and we denote it as $\hat{Q}_{n'}(\cdot; \theta^{-}_{n'})$. Thus, the model weights $\theta_{n'}$ have to be sent from n' to n periodically to update this *target network*, adding a heavier *control overhead* with respect to the one induced by the *replay memory update packets*. We call the packets carrying the model weights *target update packets*. The time between two consecutive target updates is the *target update period* U, which controls the overhead level of the *model sharing* method. Additionally, the node n needs the observation $o_{n'}$ at node n'. This is added to the *replay memory update packets* by replacing the $\tau_{n'}$ values used in *value sharing*.

Figure 5.2 depicts the control packets sent for the two techniques when a data packet is received by the next hop node n' coming from a node n. The *replay memory update packets* are represented in yellow and are triggered at the reception of each data packet during the training phase. The *target update packets* are represented in red and are triggered at each U during the training phase.



(a) *Value sharing*



(b) *Model sharing*

Figure 5.2 – Control packets for *model sharing* and *value sharing* techniques. *replay memory update packets* and *target update packets* are represented in yellow and red, respectively.

Finally, we conclude this section with Algorithm 1 that depicts the pseudocode of the training routine of our approach. We highlight in red and blue italics the lines corresponding to the *value sharing* and *model sharing* method, respectively.

Algorithm 1: DQN Routing training routine at a node n with value sharing (or with model sharing)

Input: Training session duration S; Gradient descent update period T; Target update period U; Weights $\theta_v^{SP}, v \in \left\{n \cup \mathcal{N}_n^{out}\right\}$
Output: Final routing model weights θ_n

Initialize experience replay memory $\mathcal{M}_n$;
Initialize $Q_n(\cdot)$ with weights $\theta_n \leftarrow \theta_n^{SP}$;
Initialize targets $\hat{Q}_v(\cdot), v \in \mathcal{N}_n^{out}$ with weights $\theta_v^- \leftarrow \theta_n^{SP}$;
while *current simulation timestamp* $t < S$ **do**
  for *each arrival packet* **do**
    Observe the current node state o_n;
    Select action $a_n = n'$, where
    $$n' = \begin{cases} \text{a random neig. } v \in \mathcal{N}_n^{out}, \text{with prob. } \epsilon \\ \text{argmax}_{v_n \in \mathcal{N}_n^{out}} Q_n(o_n, v_n), \text{ otherwise} \end{cases}$$
    Forward packet p to next hop node n';
    Receive back from n' next hop estimate τ and reward r_n;
    Receive back from n' next hop state $o_{n'}$ and reward r_n;
    Set packet destination flag f;
    Store transition $(o_n, a_n, r_{n'}, \tau_{n'}, f)$ in memory $\mathcal{M}_n$;
    Store transition $(o_n, a_n, r_{n'}, o_{n'}, f)$ in memory $\mathcal{M}_n$;
    if $t \mod T$ **then**
      Sample random batch $\mathcal{B} \overset{i.i.d.}{\sim} \mathcal{M}_n$;
      Set target values Y_n^Q as (2.9) for $\mathcal{B}$;
      Update weights θ_n by gradient descent on TD error for $\mathcal{B}$;
    if $t \mod U$ **then**
      Get neighgbors' weights: $\theta_v, v \in \mathcal{N}_n^{out}$;
      Update target weights: $\theta_v^- \leftarrow \theta_v, v \in \mathcal{N}_n^{out}$;

5.2.6 Experiments

In this section, we present the experimental approach to answer quantitatively the following research questions:

1. How does the cost and overhead of *value sharing* and *model sharing* techniques evolve when modulating the target update period U?
2. How much overhead is required to make the *model sharing* solution close to the Oracle cost solution?

5.2.6.1 Experiment Settings

Hardware and Software settings.
We ran the tests in a Dell Precision 7920 workstation equipped with an Intel Xeon Gold 6230R Dual CPU (26 Cores, 2.1-4.0GHz Turbo, 128 GB RAM) with 2 NVIDIA RTX A5000 GPUs, running Linux Ubuntu 20.04. The DRL agent model is implemented in `TensorFlow` (*TensorFlow: An end-to-end open source machine learning platform*, s. d.) version 2.8 as described in 5.2.4. The implementation of the MA-DRL method is based on the OpenAI™ `Baselines` library (*OpenAI Baselines: high-quality implementations of reinforcement learning algorithms*,

s. d.). The PRISMA tool (*PRISMA tool: An open MARL framework for packet routing.*, s. d.) is used as Reinforcement Learning (RL) simulation environment.

Competitors.

We compare the routing optimality of the two versions of the *DQN routing* in Algorithm 1 (*value sharing* and *model sharing*) to the *Shortest Path routing* (Bellman, 1958), and the *Oracle routing* from LP model (5.1). The *SP routing* and *Oracle routing* solutions represent performance benchmarks used to assess the quality of the DRL solution. The *DQN routing* with *value sharing* is, in practice, an extension of the original *DQN routing* (Mukhutdinov et al., 2019), where buffers' occupancy is added to packet destination in the node observation o_n. We cannot compare our *DQN routing* directly to the literature algorithms (You et al., 2022 ; L. Chen et al., 2021 ; Manfredi et al., 2021) since they are based on different assumptions. In *DQRC* (You et al., 2022), there is a unique buffer at each node, which constraints the outgoing node throughput since, at each time, only one packet can be transferred from the buffer head to one of the forwarding ports. In the works (L. Chen et al., 2021 ; Manfredi et al., 2021), the training is centralized in a node collecting all the past experiences, in contrast to us, where the training is done on the distributed node agents.

Evaluation metrics.

To measure the performance of each routing policy, we use the *average cost per packet*, calculated as the accumulated reward along the simulation divided by the number of generated packets. We point out that we can reward each packet arrival during a network simulation using the same reward definition as in Section 5.2.3. Thus, the *average cost per packet* represents an average end-to-end delay per packet, where packet losses are also accounted as *worst-case* delays (see Section 5.2.3).

Topology and traffic.

Simulations are performed over the Abilene network ($|\mathcal{N}|$=11) (*SNDlib: Library of test instances for Survivable fixed telecommunication Network Design*, s. d.). We fix link propagation delay and link rate C to 1 ms and 500 $Kbps$, respectively. Four traffic matrices $\boldsymbol{H}$ are generated by sampling each element h_{sd} from a uniform distribution $U(0,1)$. We scale up these matrices by multiplying by a coefficient α. We increase α up to the largest value α_{max} for which an optimal routing with no packet loss can still be found by the LP model (5.1). The matrix $\alpha_{max} \cdot \boldsymbol{H}$ is associated to a load factor $\rho = 1$. Data packets are generated as UDP over IP datagrams. Their payload is 512 B long. The UDP and IP headers are $8B$ and $20B$ long, respectively. Then, the total packet size is 540 B. Packet traces are produced assuming that packet *inter-arrival time* follows an exponential distribution with mean $540B/h_{sd}$. And finally, the output buffer size B is fixed to $16,200B$ (i.e., 30 data packets of $540B$ long).

Control signalling packets.

As explained in Section 5.2.5, DRL agents share information in the form of control packets. For the *replay memory update packets*, we encode each float or integer data type unit in $8B$. Then, a pair $(r_{n'}, \tau_{n'})$ (value sharing) is $16B$ long and a pair $(r_{n'}, o_{n'})$ (model sharing) is $16 + 8 \cdot |\mathcal{N}_n^{out}|B$ long. Control packets are also encapsulated into UDP over IP datagrams. For a *target update*, the size of a DRL agent model (as described in Section 5.2.4) is around $36KB$, which we split in *target update packets* of $512B$ long. Those packets are sent through a dedicated channel (off-band) in order to avoid network saturation.

Training procedure.

The training is performed in two phases: a *supervised pre-training phase* followed by the *main reinforcement learning phase*.

Supervised pre-training is done to improve the convergence of the DQN routing, as authors in (Mukhutdinov et al., 2019) demonstrated. They pre-trained the Q-network $(Q_n(\cdot;\theta_n)$ using a

dataset of tuples $\{d, L_n^{SP}(d, n')\}$, where d is the packet destination, n' is the next hop node and $L_n^{SP}(d, n')$ is the length of the shortest path between n and d, which contains the next hop node n'. The Q-network is trained till the square loss is minimized. The so-obtained weights are denoted as θ_n^{SP}.

Main reinforcement learning (see Section 5.2.3). The weights θ_n are initialized with θ_n^{SP}. The model is trained using *ADAM* optimizer with a learning rate of 0.001, batch size of 512, and γ of 1. The total duration of the training session S is $1\ minute$ (in `ns-3` simulation time), which is sufficient to cancel the agents' TD error. The gradient descent is launched every $T = 10\ ms$ (in `ns-3` simulation time). Moreover, we use an ϵ-greedy approach (ϵ decays from 1 to 0.1) to trade between exploration and exploitation. We execute different *training sessions* for each traffic matrix, information sharing technique (*value sharing* and *model sharing*), *replay memory size*, and *target update period*. We consider several values for the *replay memory size* (512, 1024, 2500, 5000, 10000, and 15000 experiences), and for the *target update period* U (from $1s$ to $9s$ with $1s$ step). Each session corresponds to a packet trace generated using a traffic matrix scaled to the load factor ρ of 0.4. After each training session, the corresponding model weights are saved.

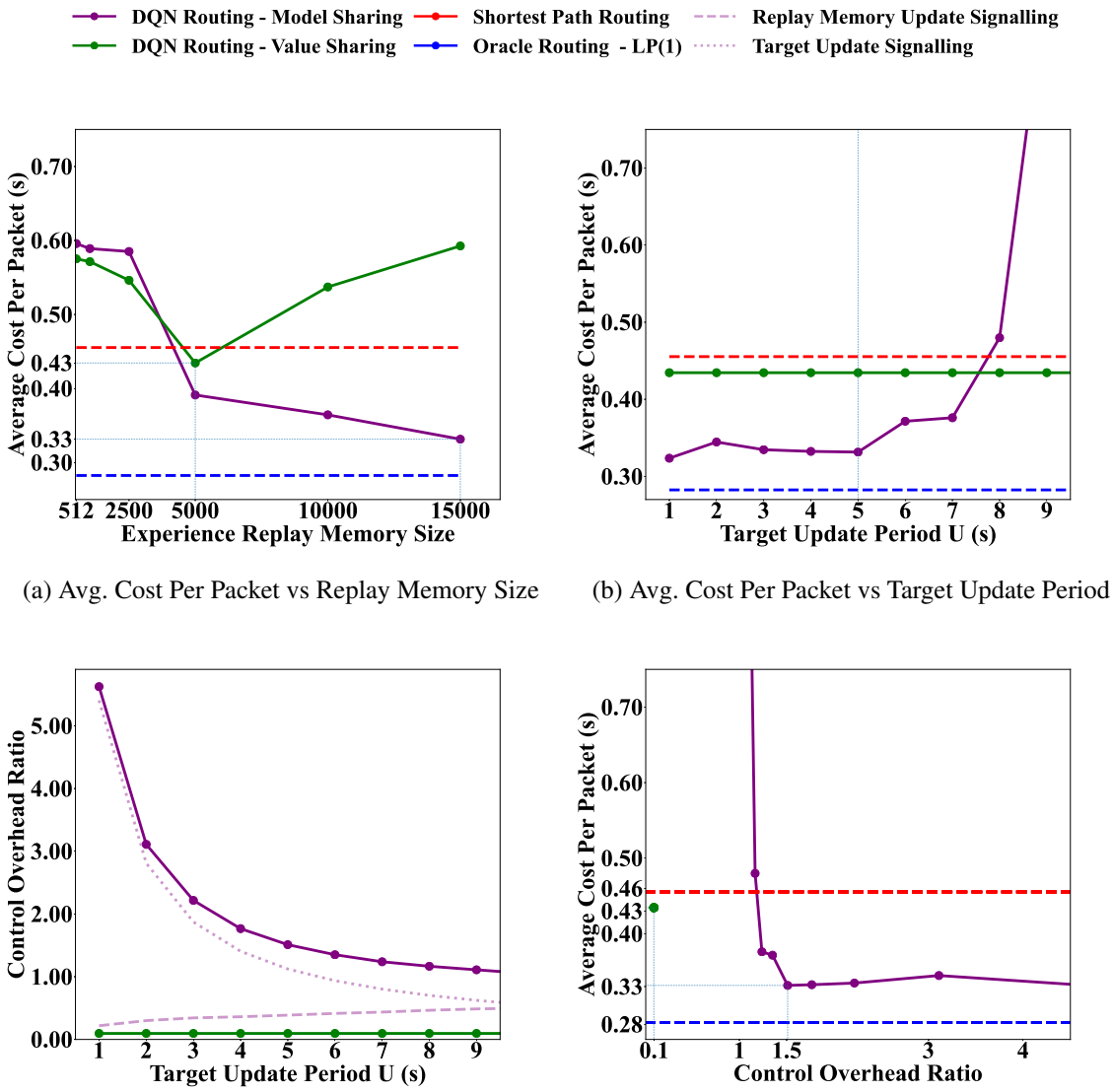
Testing procedure.

A test phase is performed for each trained model corresponding to the tuple *{traffic matrix, sharing technique, replay memory size, target update period}*. We ran *nine* simulations for load factors ρ from 0.6 up to 1.4 with 0.1 step. The rationale behind testing for traffic loads higher than 1.0 is to evaluate the model in highly saturated scenarios, where buffer delays become huge. The test packet traces are generated using the same traffic matrix as training but scaled to the corresponding load factor ρ. In other words, with respect to the training procedure, we test each DNN model with the same traffic distribution among nodes but with higher loads. The duration (in simulation time) of each *testing* simulation was $20\ s$, sufficient to reach a stationary state in the simulation.

5.2.6.2 Tradeoff between Overhead and Optimality

In this subsection, we evaluate the impact of the control signalling on the learning performance by studying the tradeoff between *average cost per packet* and *control overhead ratio*. During the training, the *control overhead ratio* is computed as the ratio between the total byte count of signalling packets and the total byte count of useful data packets. The results presented are computed as an average over the four traffic matrices and the nine load factors.

Figures 5.3a and 5.3b show the *average cost per packet* versus the *replay memory size* and the *target update period* U, respectively. From their observation, first, we see that *DQN routing with value sharing* has a slightly better average cost than SP when choosing the experience replay memory size of 5000 samples. We recall that the original *DQN routing* algorithm presented in (Mukhutdinov et al., 2019) uses value sharing. However, in (Mukhutdinov et al., 2019), the routing decisions are taken based only on the packet destination, which yields routing policies close to the SP routing since features depending on traffic dynamics, as the buffers' occupancy, are not provided to the neural network. Interestingly, although, in the current chapter, we also add the buffers' occupancy to the neural network input, our enriched version of the original *DQN routing* in (Mukhutdinov et al., 2019) does not manage to improve significantly over the SP policy, obtaining even worse results for any replay memory size value except 5000, which seems large enough to store the states' diversity, and small enough to not consider too many outdated experiences. In contrast, *model sharing* outperforms the SP routing and gets close to the Oracle routing cost when the replay memory is large enough (more than $5,000$ experiences) and the target update



(a) Avg. Cost Per Packet vs Replay Memory Size

(b) Avg. Cost Per Packet vs Target Update Period

(c) Control Overhead vs Target Update Period

(d) Avg. Cost Per Packet vs Control Overhead

period U is small enough (less than $8s$). This result suggests that routing agents need to share their DNN models to be able to learn a routing policy close to an Oracle routing. Previous literature has shown the importance of conditioning the learning of each agent on an estimate of the other agents' policies to alleviate the *non-stationarity* effect of MA-DRL. Since other agents' policies are continuously updated during the training, their *Q-value* (and, then τ) estimates become obsolete with time. For value sharing, that means that the replay memory can be populated with outdated $\tau_{n'}$ conducting to have different estimates of the *end-to-end delay* from next hop n' to destination for the same state transition $(o_n, a_n, o_{n'})$. *Model sharing* overcomes this problem by sharing the model $Q_{n'}(\cdot, o_{n'}; \theta_{n'})$. Since $\tau_{n'}$ estimates are computed using Equation (2.10) at the gradient descent update, two identical transitions $(o_n, a_n, o_{n'})$ stored at the replay memory will be associated to the same estimate $\tau_{n'}$ of the *end-to-end delay* from n'.

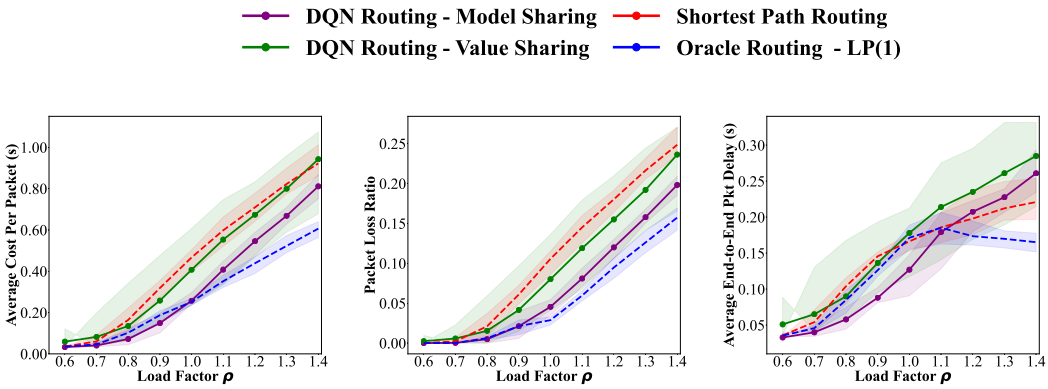
Figure 5.3c depicts the relation between *target update period* U and the *control overhead ratio*. In the *model sharing* technique, the control overhead strongly decreases with U, since the number of *target update packets* diminishes as well. For value sharing, only the much lighter *replay memory update packets* are sent, giving a much smaller control overhead (0.1). This overhead is shown as constant since it corresponds to only one point (value sharing does not depend on U). Analyzing the overhead for *model sharing* in detail, we see two different behaviors for the signalling packets. The overhead due to the *target model update packets*, the main overhead source, decreases with the target update period, but the *replay memory update packets* overhead increases linearly. This indicates a growth in packet hops during the training, which is related to a worse training performance for larger U values: the agents do not complete to find short routing policies (see Figure 5.3b).

Figure 5.3d depicts the tradeoff between *average cost per packet* and *control overhead ratio* for *model sharing* and *value sharing* techniques. Replay memory sizes are set to the best values in Figure 5.3a for each sharing technique: $5,000$ and $15,000$ for value and *model sharing*, respectively. Each point in the curve for *DQN routing* with *model sharing* corresponds to a given target update period value U, which is associated with a given cost per packet (see Figure 5.3b) and a given overhead (see Figure 5.3c). For value sharing, we have a unique point corresponding to the cost per packet for the best memory size $(5,000)$ and a constant value sharing overhead (0.1). Two observations can be made from this figure: (i) *value sharing* incurs in a very small overhead (0.1) but cannot improve significantly the SP routing performance, and (ii) *model sharing* can outperform SP routing and approach the Oracle performance but with much higher overhead (for $U = 5s$, an overhead of 1.5 and a cost 28 % smaller than SP routing and *model sharing*). These observations provide an important insight: *for the presented settings, the price to pay to become close to the Oracle performance is a control overhead larger than the useful data volume during training*.

5.2.6.3 Packet Cost Performance in Detail

We now analyze the performance of *DQN routing* for both signalling techniques in terms of: (i) *average cost per packet* (Figure 5.4a), (ii) *packet loss ratio* (Figure 5.4b), and (iii) *average end to end packet delay* (Figure 5.4c). The *x-axis* represents the load factor, ranging from 0.6 up to 1.4 with 0.1 step, and the *y-axis* represents the average, the minimum, and the maximum value of the corresponding performance metric over the four traffic matrices. A solid line is used to depict the average value, whereas the area between the minimum and the maximum value is presented with a shaded color. In these figures, the replay memory sizes are fixed to the best values found in the previous subsection: $5,000$ and $15,000$ for *value sharing* and *model sharing*, respectively. Similarly, the target update period for *model sharing* is set up to the selected value of $5s$.

From the observation of Figure 5.4a, we confirm the remarks stated in the previous subsection: in terms of average cost per packet, the *model sharing* is always better than value sharing and SP routing, and close to the Oracle routing performance. We recall that the average cost per packet is computed as an accumulated reward during a simulation session. Then, it accounts for both buffer delays and packet losses, but giving more importance to the latter to minimize it in the first place (see Section 5.2.3) as the LP model of the Oracle routing also does (see Section 5.2.2). Consequently, Figure 5.4b, which shows the packet loss performance, presents the same trends.
Figure 5.4c shows the average end-to-end packet delay for the different loads. Interestingly, *DQN routing* manages to outperform the Oracle routing for medium loads (from 0.7 to 1.1). This can be



(a) Average Cost Per Packet. (b) Packet Loss Ratio (c) Avg. End-to-End Delay.

partly explained by superior performance in terms of packet loss (if present, the main contributor to the average cost per packet) of the Oracle, namely for the loads 1.0 and 1.1, which leads the Oracle to have poorer behavior in terms of delay. But, for the loads between 0.7 and 0.9, *DQN routing* outperforms the Oracle in delay and matches its packet loss, yielding a better average cost per packet in Figure 5.4a. This better behavior of *DQN routing* can be explained by the flow-based nature of the LP model of the Oracle routing policy. This model has a coarse-grained view of the packets' flows where the notion of packets' buffers is absent: the LP model works with flows defined by an average value in traffic units (i.e., *bps*). On the contrary, the *DQN routing* has a fine-grained view of the flows, being aware of individual packets. The packets do not arrive regularly and periodically but randomly follow an exponential distribution, giving rise to situations where the instantaneous value of flow in traffic units (i.e., *bps*) exceeds the average flow. In this case, buffers allow absorbing the temporary traffic peaks. Particularly, the packet's buffers can have a bigger impact on the network performance when the average traffic becomes close to the nominal network capacity (medium loads between 0.7 and 0.9) since buffers are mostly empty for lower loads, and saturated for higher loads. Therefore, the *DQN routing* can benefit from the observation of the buffers' occupancy in these medium loads to overcome the Oracle routing, which is unaware of the buffers.

5.2.7 Discussion

The results presented in this study, especially the ratio between the signalling packets and useful data packets, may change if we change the experimental settings, such as using a larger link rate. Doing so will increase the number of data packets per second and hence the number of *replay memory update packets*. The ratio between *replay memory update signalling* and useful data will remain the same, and so the overhead for *target value sharing*. However, when keeping the same *target update period*, the ratio between *target update signalling* and useful data will decrease, and so the gap in overhead between *model sharing* and *value sharing* will narrow, but the gradient step period T may need to be reduced to maintain the number of experiences per gradient step, and hence possibly have a smaller *target update period*. When changing network topology, the overhead, particularly the one due to *target update signalling*, will scale with the number of nodes, and so if we add more nodes in the network, the overhead will increase. The relationship between the overhead, the performances, and the topology settings will be studied in

future works. To conclude this analysis, when changing the network configurations, DRL based methods need to be re-evaluated in a simulation environment taking into account the overhead of control packets, to have a precise estimation of the net gains of such an MA-DRL routing policy.

5.3 Novel methods to reduce communication overhead between the agents

As shown by the previous results, to get the best performance, the control overhead needed for the *model sharing* technique is around 150% of the injected data packet size. This significantly impacts the network if we choose to send the control packets in-band. *Target update packets* represent the most part of this information exchange, which is due to the size of the DNN weights and the frequency of sending those weights. The objective of this section is to present two ways to reduce this communication overhead.

— We first aim to reduce the communication overhead by removing the *target update packets*. For that, we present a novel sharing technique called *logit sharing*. It is inspired by both *model sharing* and *value sharing* and offers a low communication overhead and stable performance.

— Then, to still reduce the communication overhead, we explain how to reduce the amount of *replay memory update packets* by proposing an adaptive method that reduces the frequency of the neighbor's exchanges depending on the training progress.

— Finally, we evaluate both methods and compare them to *model sharing* and *value sharing*.

5.3.1 Removing the target update overhead: *logit sharing*

The *target update packets* introduce a significant burden on the agent's communication. In the following, we propose a new sharing technique, named *logit sharing*. **This new technique aims to bring the best of the two previous methods: the lower communication overhead of *value sharing*, and the higher performance of *model sharing***.

The idea is to locally train the *target network* $\hat{Q}_{n'}(\cdot;\theta^{-}_{n'})$ instead of updating it from the original neighbor network $Q_{n'}(\cdot;\theta_{n'})$. During the training phase, the node n collects from the next hop n' the observed state o'_n and the output vector of the DNN, i.e., the *logit vector* of neighbor n' for this observation $Q_{n'}(o_{n'},.;\theta_{n'})$. This information is sent along with the reward $r_{n'}$ and the flag f as a *replay memory update packet*. The observed state and the *logit vector* will be used by the node n to construct a dataset containing the inputs and the outputs of the DNN of that neighbor, namely the tuple $(o_{n'}, Q_{n'}(o_{n'},.;\theta_{n'}))$. At each time step T, the agent uses the database to train the *target network* in a supervised manner. The objective is to keep the weights $\theta^{-t}_{n'}$ of the *target network* at time t as close as possible to the weights $\theta^{t}_{n'}$ of the DNN of the neighbor n'. The computation of the target value $\tau_{n'}$ is done the same way as *model sharing* using the *target network* (see Equation (2.10)).

Logit sharing technique presents a low communication overhead since the agents are sending only the *replay memory update packets* and are not sharing the weights anymore. However, to work correctly, the weights of the *target network* and the neighbor DNN should have the same initialization (in our experiments, we initialize to the shortest path policy). Also, the supervised training time step T should be set properly: a too-big value will cause a divergence between the weights of the *target network* and the neighbor's DNN, whereas a too-small value will result in

small batches of training data and induce instability in the supervised learning (overfitting to the last period). In our experiments, we set this period T to the *target update period* of the best *model sharing* model running off-band.

We present, in Figure 5.5, a summary of the control packets when adopting the *logit sharing* technique.

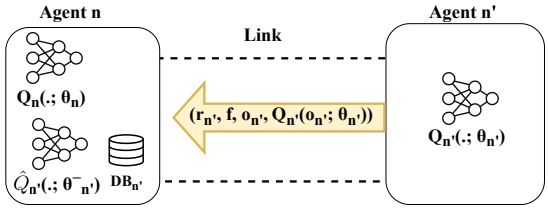


Figure 5.5 – Control packets for the *logit sharing* technique. *replay memory update packets* are represented in yellow.

5.3.2 Reducing the replay memory update overhead

In this subsection, we study how to reduce even more the communication overhead by sending *replay memory update packets* only when they are relevant to the training process. During the training phase and in all the previously-mentioned techniques, the agents need to send *replay memory update packets* as a response to each incoming data packet. Even though these packets are small compared to a data packet, they can represent a significant overhead, especially in the continual learning setting.

5.3.2.1 Evaluation of *replay memory update packets* overhead for the three techniques

— The *value sharing* introduces the lowest overhead since the agents share only three values $(r_{n'}, \tau_{n'}, f)$: two floating points $(r_{n'}, \tau_{n'})$, and one binary (f). This overhead scales with the traffic load and the number of hops.

— For *model sharing*, we have to send $(r_{n'}, o_{n'}, f)$, i.e., two values (one floating point $r_{n'}$, and one binary f) and a vector $(o_{n'})$, whose length is the number of neighbors of the node n'. The overhead, in that case, scales with the degree of the overlay network, the traffic load, and the number of hops.

— We can do the same reasoning with *logit sharing* technique. In *logit sharing*, a *replay memory update packet* carries the tuple $(r_{n'}, o_{n'}, f, Q_{n'}(o_{n'}, .; \theta_{n'}))$. In comparison with *model sharing*, we have an additional vector (the logit vector $Q_{n'}(o_{n'}, .; \theta_{n'})$), whose size is also the number of neighbors of the node n'. Thus, the overhead here scales, as in *model sharing*, with the degree of the overlay network, the traffic load, and the number of hops.

5.3.2.2 Dynamic sending of *replay memory update packets*

To reduce the overhead of this communication, we need to answer these two questions:

1. Are all the replay memory entries relevant for the agent to learn?
2. How to evaluate the importance of each *experience replay memory* entry $(o_n, a_n, r'_n, o'_n, f)$?

To answer the above questions, we propose to use the *Temporal Difference (TD) error* defined in Equation (2.11) to evaluate the importance of an entry. A high value of this loss means that the entry is relevant, and the agent needs to include it in its *experience replay memory*. In contrast, a small value of the loss means that the entry has little contribution to the learning of the agent, so it can be discarded. From that reasoning, we define the *experience relevancy* ER as the TD-error normalized by the target value Y_n^Q, as in Equation (2.9), as follows:

$$ER = \left\| \frac{L_{DQN}}{Y_n^Q} \right\| = \left\| \frac{Q_n(o_n, a_n; \theta_n)}{r_{n'} + \gamma \cdot \tau'_n \cdot (1 - f)} - 1 \right\| \tag{5.2}$$

where τ'_n is computed as in Equation (2.10). The *experience relevancy* ER hence represents a normalized deviation of the two models Q_n and $Q_{n'}$ for the transition $(o_n, a_n, r_{n'}, o_{n'}, f)$. We can compute this value in the next hop node n', and use it to decide whether to send back to n the *replay memory update packet* or not, based on a predefined threshold on it, denoted as ER_{thr}. To compute the *experience relevancy* ER at n', the only information missing is the DNN output $Q_n(o_n, a_n; \theta_n)$ of the node n for the transition. Thus, the node n needs to send this value with the data packet, corresponding to one extra floating point per data packet. When the next hop node receives the data packet with $Q_n(o_n, a_n; \theta_n)$, it computes ER and compares it to the predefined *experience relevancy threshold* ER_{thr}. If the deviation exceeds the threshold, the next hop node n' sends back a *replay memory update packet*. Otherwise, it discards the transition.

The presented mechanism applies to all the above information-sharing techniques, and it can significantly reduce the *replay memory update packets* overhead. It makes continual learning possible since the agents will have reduced communication after convergence and only share transitions when necessary.

5.3.3 Experiments

5.3.3.1 Experimental settings

To evaluate *logit sharing* and the dynamic control packet mechanism, we hold the same experiments as the first part of this chapter (Section 5.2.6). We consider the same four randomly generated traffic matrices and the same DNN architecture with the same observation, action, and reward. The training is done at each $50ms$ on a load factor of 40% and lasts for 60 seconds. After that, the trained models are evaluated in a session of 25 seconds on seven load factors (from 60% to 120% by a step of 10%). The batch size and the learning rate are set to 512 and 0.0001 respectively. The *experience replay memory* size is set to 15000, and the exploration follows an epsilon-greedy decaying from 1.0 to 0.01. The main difference between this experiment and the previous one is that we send the *replay memory update packets* in the link (in-band) since we have the capability to do it under **prisma-v2** (see Section 4.3). The high overhead of *model sharing* due to the *target update packets* forced us to keep it off-band to avoid overloading the network. We set the *target update period* to its best value from the previous experiments, which is $U = 5s$. We set the *logit sharing* training step to the same value ($T = 5s$). The training step for the *logit sharing* local copy is 100 smaller than the actual models' learning rate. We reevaluate the sizes of the *replay memory update packets* by adding the data packet ID that triggers the control packet, which results in the following:

— $25B$ for *value sharing* ($8B$ for the reward, and packet id, target value, respectively, and $1B$ for the flag).

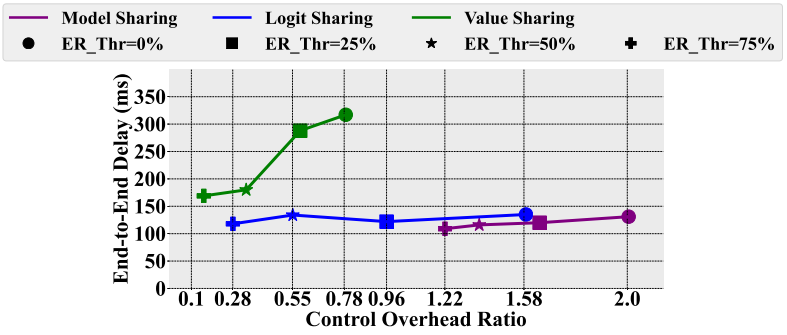
— $17 + 8 \cdot (|\mathcal{N}_n^{out}| + 1)B$ for *model sharing* ($8B$ for the reward, and packet id, respectively, and $8 \cdot (|\mathcal{N}_n^{out}| + 1)B$ for the next hop observation and $1B$ for the flag).

— $17 + 8 \cdot (2 \cdot |\mathcal{N}_n^{out}| + 1)B$ for *logit sharing* (same as *model sharing* with an additional vector representing the output of the neighbor's DNN).

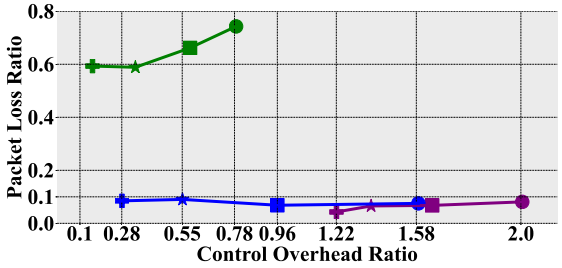
This results in a more accurate measure of the control overhead, which is higher than the values presented in the previous study.

5.3.3.2 Experimental results

We train and test the three techniques and vary the *experience relevancy threshold* ER_{thr} from 0% to 75% by a step of 25%. Figure 5.6 depicts the results obtained after the training represented in terms of average end-to-end delay per arrived packet (Figure 5.6a) and average loss packet rate (Figure 5.6b). The average is computed over all the test loads and all the traffic matrices. The results show that in terms of performance, *logit sharing* is similar to *model sharing* in both delay (around $100ms$) and loss rate (around 7%). *Value sharing*, as shown by the previous study, struggles to converge and thus presents a very high loss, which is due to the outdated experiences phenomenon. In terms of control overhead, the use of the *logit sharing* technique reduces the overhead compared to *model sharing* from 205% to 158% when $ER_{thr} = 0$. This effect is amplified when activating the dynamic sending of the control packets, for example, for $ER_{thr} = 75\%$, the overhead ratio decreases from 122% with *model sharing* to 28% with *logit sharing*.



(a) End-to-end delay vs. communication overhead



(b) Packet loss ratio vs. communication overhead

Figure 5.6 – Evaluation of *logit sharing* technique and dynamic control packet method in Abilene topology

5.3.4 Discussion

The experiment shows that coupling *logit sharing* with the dynamic sending of control packets significantly reduces the communication overhead while keeping the performance at the same level as *model sharing*. However, this effect is verified for a static traffic matrix evaluation. In this case, the transitions will become redundant throughout the training and thus will be filtered out by the dynamic sending mechanism. This also helps *logit sharing* since the neighbors's model does not drastically change after the convergence of the models, which keeps the local copy following the same trajectory as the neighbor's model. These presented two techniques are also beneficial when adopting a continual learning approach. The communication overhead will be high in the beginning of the training when all the models are learning, and then it settles to a reasonable value when the models converge.

Finally, we show that those two techniques work well in the general non-overlay setting when the agents have access to their outgoing buffer. However, this is not verified for the cloud overlay setting. This will be investigated in the next chapter.

5.4 Conclusion

In this chapter, we have presented an experimental framework to investigate, for the first time to the best of our knowledge, the control signalling mechanisms of distributed learning-based packet routing in the general non-overlay scenario. We have considered two main control exchange techniques brought by the literature, *value sharing* and *model sharing*. Both techniques have been evaluated in terms of routing cost (combining packet delay and loss rate) and the overhead packets due to control. We have shown that *model sharing* yields routing policies able to approach the optimal one provided by an oracle. However, to obtain such results, the model presents a control overhead of 150% of extra traffic.

In the second part of this chapter, we provide two novel methods to address the problem of high overhead of *model sharing*. First, a new communication technique called *logit sharing*, which brings the best of *model sharing* and *value sharing*. The second method is a dynamic control packet-sending mechanism, which enables the agents to send only relevant control packets.

We finally discussed the impact of the proposed methods and showed that combining them significantly reduces the overhead of *model sharing* from 200% to 28% while maintaining its performance.

In the next chapter, we will tackle the more challenging overlay scenario. We will adapt the proposed methods and propose a novel MA-DRL framework to handle this case.

CHAPTER 6

Distributed Learning-based Routing in Cloud Overlay Networks

This chapter addresses the problem of Distributed Packet Routing (DPR) in cloud overlay networks using a fully decentralized Multi-Agent Deep Reinforcement Learning (MA-DRL). Cloud overlay networks are built by having a virtual topology on top of an Internet Service Provider (ISP) underlay network, where those nodes are running a fixed, single path routing policy decided by the ISP. In such a scenario, the cloud overlay network's underlay topology and traffic are unknown. In this setting, we propose O-DQR*, which is a MA-DRL framework working under Distributed Training Decentralized Execution (DTDE), where the agents are allowed to communicate only with their immediate cloud overlay neighbors during both training and inference. We address three fundamental aspects for deploying such a solution: (i)* performance *(delay, loss rate), where the framework can achieve near-optimal performance , (ii)* control overhead*, which is reduced by enabling the agents to send control packets only when needed dynamically; and (iii)* training convergence stability*, which is improved by proposing a guided reward mechanism for dynamically learning the penalty applied when a packet is lost. The overhead management allows the framework to have minimal overhead, paving the way for deploying the models in continuous learning scenarios. The guided reward mechanism significantly improves the convergence rate and gives guarantees under mild assumptions. Finally, we evaluate our solution in a realistic network simulation, which allows transmitting actual control packets and measuring their impact on the performance in a cloud overlay traffic configuration.*

6.1 Introduction

The internet has seen a massive increase in usage over the last few decades, with more than 4.7 billion users worldwide in 2020 (Ritchie et al., 2023), and it is expected to reach a third of the global population by 2023 (*Cisco Annual Internet Report - Cisco Annual Internet Report (2018–2023) White Paper*, s. d.). To handle this high demand, Content Delivery Networks (CDNs) have emerged as a solution by providing data caching to servers near the end user. Video streaming accounts for a significant portion of internet traffic, so CDNs are essential for data-heavy services like YouTube and Netflix. Doan et al. (Doan et al., 2020) showed that caching reduces throughput by up to three times. However, some services are non-cacheable or cacheable for only short periods. In such cases, the requested content is downloaded from the origin server via a cloud overlay network operated by the CDN provider.

Cloud overlay networks are virtual or logical networks built over physical networks (called underlay networks). They provide flexible and dynamic traffic routing between nodes that are not directly connected by a physical link (i.e., as CDN servers that are placed at the Edge of the Internet). Instead, they are connected by tunnels traversing different Autonomous Systems (AS) operated by different Internet Service Providers. These tunnels are virtual or logical links corresponding to paths in the underlying network.

Routing traffic between the cloud overlay nodes is a challenging problem due to the lack of information about the underlay network. This yields the existence of Triangle Inequality Violations (TIV) (Lumezanu et al., 2009), which means that it is possible to find another path relayed by cloud overlay nodes that has a lower delay than following the direct path connecting them (usually the shortest path over the underlay topology). To illustrate this phenomenon, consider the example shown in Figure 6.1. Traffic from cloud overlay node A to C has two possible paths: a direct path (in red) and an alternative path (in green). If we assume high congestion in the hidden underlay node *5*, the "default path", i.e., the direct path, becomes suboptimal.

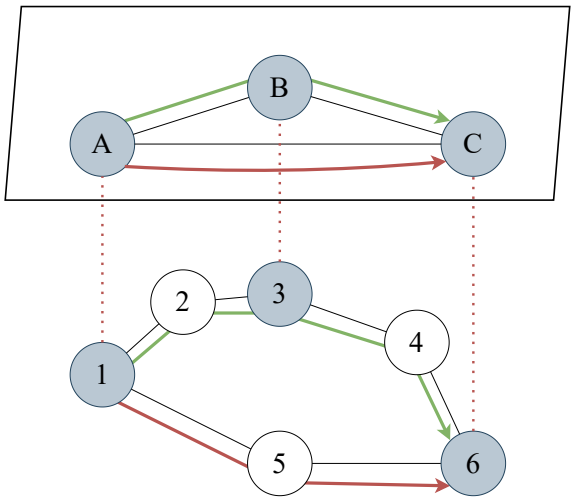


Figure 6.1 – Cloud overlay network architecture: cloud overlay topology is on top; underlay topology is on bottom. Underlay nodes $1, 3, 6$ are mapped to cloud overlay nodes A, B, C. Underlay nodes $2, 4, 5$ are hidden from the cloud overlay topology. The shortest paths in the underlay correspond to virtual links (tunnels) in the cloud overlay.

The TIV holds on the Internet since its infrastructure consists of an increasing number of ASs (*CIDR REPORT for 15 Sep 23*, s. d.). Each AS can correspond to an ISP, optimizing its local routing, ignoring the end-to-end routing performance, which impacts the performance of the Content Distribution Network (CDN).

Therefore, instead of taking the tunnel between two cloud overlay nodes (i.e., the default routing proposed by the ISPs), it may be advisable to use other routing policies. Traditionally, CDN providers, as Akamai (Sitaraman, Kasbekar, Lichtenstein, & Jain, 2014), apply a two-phase methodology to find these routing policies. First, they estimate the unknown and uncertain problem input data, such as traffic demands, using data collected from the nodes. Then, they update the routing between the cloud overlay nodes using a classical optimization algorithm (i.e., linear programming) fed by these estimates. However, such approaches are highly dependent on (i) the quality and periodicity of the estimates of the first phase, and (ii) the trade-off between optimality and solving time of the classical optimization techniques of the second phase. This two-phase approach may lead to a slow and outdated adaptation to changes in traffic and topology. For instance, the traffic in novel Internet services, such as Internet of Things (IoT) communications, multimedia streaming, or cloud computing, is characterized by becoming highly dynamic (*Cisco Annual Internet Report - Cisco Annual Internet Report (2018–2023) White Paper*, s. d.) and difficult to predict using these two-phase methods (S. Yang & Kuipers, 2014).

As a consequence, the new scenarios require solutions that can rapidly adapt to sudden traffic changes, motivating researchers to adopt Machine Learning (ML) techniques, like Deep Reinforcement Learning (DRL) (Sutton & Barto, 2018 ; Bengio, 2009). Those methods have proven their effectiveness by achieving breakthrough results in various complex tasks (Mnih et al., 2015). MA-DRL, which is an extension of DRL, has become popular in solving communication network problems like the Distributed Packet Routing (DPR) problem (Mukhutdinov et al., 2019 ; You et al., 2022). Here, the DRL agents collaborate to optimize a global objective. The training of MA-DRL agents can be done either in a centralized node Centralized Traning Decentralized Execution (CTDE) or distributed through the nodes Distributed Training Decentralized Execution (DTDE)(Gronauer & Diepold, 2022). DTDE is suitable for the packet routing problem, where each agent (i.e., network node) exploits the locally collected data to adapt its routing policy in real time. This allows a fast response to changes in the network and provides scalability since each agent is autonomous. However, referring to previous studies such as (Gronauer & Diepold, 2022 ; Matignon et al., 2012 ; You et al., 2022), implementing the DTDE approach can be challenging for two main reasons. First, the agents need to communicate to overcome the non-stationarity of the environment, resulting in a high communication overhead between the agents during the training. This has been demonstrated in our previous work (Alliche, da Silva Barros, et al., 2022) and presented in the previous chapter. Second, agents can suffer from stability issues due to the shaping of the reward, especially when optimizing more than one metric (such as end-to-end delay and loss rate). As a consequence, different questions arise when designing a DTDE approach: (i) how to ensure a good trade-off between performance and communication overhead, (ii) how to send only data that are relevant to the training, and (iii) how to design a suitable reward function to ensure a stable convergence. Additionally, other questions arise in a cloud overlay network context, such as what data an agent can collect and exploit to overcome the lack of information about the underlay network and the dynamic traffic flowing across it. Can the agents handle multiple overlapping cloud overlay links in the underlay topology?

In the previous chapter, we analyzed and addressed the challenges of communication between the agents by proposing *logit sharing* strategy and dynamic sending of control packets. In this

chapter, we provide answers to the remaining challenges, and we propose a distributed MA-DRL framework called *O-DQR* that works under the DTDE scheme. Our framework is designed to address the problem of *multihop* routing (formulated as DPR) in cloud overlay networks without any information about the underlay topology. To our knowledge, *O-DQR* is the first MA-DRL framework that tackles the *multihop* routing (as DPR) problem in this context. We focus on three main aspects in our design: *performance*, *communication overhead*, and *convergence*. We confirm, in the cloud overlay case, the results obtained in the general non-overlay case by *logit sharing* and the dynamic sending of control packets. Based on the latest research on constrained policy optimization, namely RCPO (Tessler et al., 2018), we integrate the loss rate and the delay in the reward function, ensuring fast convergence and stability during model training. We validated the performance, stability, and communication overhead of our framework through extensive experimentation on a realistic network simulator **prisma-v2** (Alliche, Barros, et al., 2022). Our framework achieved near-optimal results, improving the default ISP routing policy by more than 66% on average in the loss rate. In high loads, *O-DQR* cuts the loss rate by up to five times. The model achieves those performances while maintaining low communication overhead.

Our work differs from previous studies that predict the load weights for a set of pre-computed shortest paths (Kamri et al., 2021 ; Barzegar, Ruiz, & Velasco, 2023) in several aspects. Firstly, we assume no knowledge about the underlay topology and cannot choose relevant cloud overlay paths. Secondly, we consider the possibility of having multiple cloud overlay links that overlap in the underlay topology. Finally, we simulate cloud overlay data packets, underlay data packets, and control packets used for the communication between the agents and cloud overlay link measurement.

The summary of our contributions, along with the corresponding sections, are listed below.

— We present *O-DQR*, which is the first MA-DRL solution, working under DTDE, to tackle the *multihop* routing (as DPR) problem in cloud overlay networks without any knowledge about the underlay topology. (Section 6.2)

— We present different possibilities to integrate the lost packets into the reward function and design a guided reward mechanism to improve stability and convergence speed. (Section 6.3)

— We evaluate the *O-DQR* in terms of performance, communication overhead, and convergence through extensive experimentation. (Section 6.4)

The rest of this chapter is structured as follows. The MA-DRL framework is described in Section 6.2. The convergence and stability of the framework are studied in Section 6.3, where the guided reward mechanism is presented. At the end of Section 6.3, we illustrate the final algorithm (Algorithm 2) describing *O-DQR*. Finally, we analyze the simulation results in Section 6.4, provide a discussion of the results in Section 6.5, and conclude the chapter in Section 6.6.

6.2 Framework Description

In this section, we first formulate the multihop DPR problem in the overlay case and propose an oracle optimal solution to solve it under some assumptions. Then, we write the problem as Partially Observable Markov Decision Process (POMDP). Finally, we detail our proposed framework, namely *O-DQR*, which follows a fully decentralized MA-DRL approach.

6.2.1 Problem Statement

Let $\mathcal{G}^u(\mathcal{N}, \mathcal{L})$ be the directed underlay network representing the physical topology, where $\mathcal{N}$ is the set of nodes, and $\mathcal{L}$ is the set of unidirectional physical links. Each link has a fixed capacity C (i.e., $Kbps$). The incoming and outgoing links of the node $n \in \mathcal{N}$ are denoted as $\mathcal{L}_n^{in}$ and $\mathcal{L}_n^{out}$, respectively. Let $\mathcal{G}^o(\mathcal{N}^o, \mathcal{P})$ be the directed overlay network implementing the virtual topology onto the physical one, where $\mathcal{N}^o \subset \mathcal{N}$ denotes the set of overlay nodes, and $\mathcal{P}$ is the set of virtual links connecting the nodes $n \in \mathcal{N}^o$. A virtual link $p \in \mathcal{P}$ is an overlay tunnel starting and terminating at an overlay node pair. At the starting node a_p, packets are encapsulated into overlay packets destined to b_p. Then, the overlay packets pass through a sequence of physical links (i.e., a path) $l \in \mathcal{L}_p$. The subset of tunnels (paths) traversing a link l is called $\mathcal{P}_l$. The incoming and outgoing tunnels (virtual links) of the node $n \in \mathcal{N}^o$ are denoted as $\mathcal{P}_n^{in}$ and $\mathcal{P}_n^{out}$, respectively. The incoming and outgoing neighbor nodes of the node $n \in \mathcal{N}^o$ in the $\mathcal{G}^o(\mathcal{N}^o, \mathcal{P})$ are denoted as $\mathcal{N}_n^{in}$ and $\mathcal{N}_n^{out}$, respectively.

Each packet originates from a source node $s \in \mathcal{N}$ and is targeted to a destination node $d \in \mathcal{N}$. The traffic matrix $\boldsymbol{H} = \{h_{sd}, (s,d) \in \mathcal{N} \times \mathcal{N}$ counts up the average volumes h_{sd} in traffic units (i.e., $Kbps$) of packet flows between the node pairs $(s,d) \in \mathcal{N} \times \mathcal{N}$. A packet flow between any node pair $(s,d) \in \mathcal{N} \times \mathcal{N}$ can be routed onto the underlay network topology $\mathcal{G}^u(\mathcal{N}, \mathcal{L})$ via the *underlay routing policy* r_l^{sd}: link l is used in the route between (s,d) if r_l^{sd}=1. The *underlay routing policy* is an implicit input parameter to our problem but assumed as unknown by the cloud overlay nodes $\mathcal{N}^o$. The flows between node pairs (s,d) where one of the nodes is not an overlay node ($s \notin \mathcal{N}^o$ or $d \notin \mathcal{N}^o$) are routed using the *underlay routing policy* r_l^{sd}. We call these flows *underlay flows*. On the contrary, the flows between node pairs (s,d) where both nodes are cloud overlay nodes ($s \in \mathcal{N}^o$ and $d \in \mathcal{N}^o$) are routed via the *cloud overlay routing policy* x_p^{sd} over the overlay virtual topology $\mathcal{G}^o(\mathcal{N}^o, \mathcal{P})$. We call these flows *overlay flows*. In $\mathcal{G}^o(\mathcal{N}^o, \mathcal{P})$, a tunnel $p \in \mathcal{P}$ connecting a cloud overlay node pair (s,d) traverses the physical links $l \in \mathcal{L}_p$ defined by the *underlay routing policy* r_l^{sd}. We draw the attention that the *overlay routing policy* can set up multi-hop routes between cloud overlay nodes: a cloud overlay pair can be connected by a direct tunnel (i.e., via the underlay routing) or by a sequence of tunnels (each one defined by the underlay routing). The second option avoids using the underlay routing if a better routing can be found as a sequence of tunnels. Namely, we aim to find the *cloud overlay routing policy* x_p^{sd} that minimizes the total end-to-end delay for all the delivered packets and minimizes the number of dropped packets.

Each node $n \in \mathcal{N}$ (cloud overlay or not) is a router equipped with $|\mathcal{L}_n^{out}|$ outgoing network interfaces. Each interface $i \in |\mathcal{L}_n^{out}|$ has its own buffer queue of size B. The queue follows a `FIFO` (First-In-First-Out) policy and stores all cloud overlay and underlay packets outgoing from the network interfaces. Hence, if a packet arrives at a full buffer, it is rejected (i.e., lost). Otherwise, the packet is admitted, eventually getting to the buffer head, where it is forwarded to a next-hop link $l \in \mathcal{L}_n^{out}$. This procedure is repeated at each hop until the packet reaches its final destination d. z_{sd} represents the fraction of rejected packets per flow (s,d). In this scenario, the packet will travel in the network until it reaches its destination or is dropped due to a full outgoing buffer queue. This means that there is no limit in terms of hops or time in the network (ignored Time-To-Live).

Table 6.1 gathers the above-mentioned notation.

Name	Description
$x_p^{sd} \in [0,1]$	Decision variable. Fraction of the *cloud overlay flow* between the node pair $(s,d) \in \mathcal{N}^o \times \mathcal{N}^o$ to be forwarded via the virtual link (cloud overlay tunnel) $p \in \mathcal{P}$.
$y_l^{sd} \in [0,1]$	Decision variable. Fraction of the *underlay flow* between the node pair $(s,d) \in (\mathcal{N} \times \mathcal{N}) \setminus (\mathcal{N}^o \times \mathcal{N}^o)$ to be forwarded via the physical link $l \in \mathcal{L}$.
$z_{sd} \in [0,1]$	Decision variable. Fraction of the flow between the node pair $(s,d) \in (\mathcal{N} \times \mathcal{N})$ that is rejected.
$r_l^{sd} \in [0,1]$	r_l^{sd}=1, if the physical link $l \in \mathcal{L}$ belongs to the underlay route (tunnel) between the node pair $(s,d) \in (\mathcal{N} \times \mathcal{N}) \setminus (\mathcal{N}^o \times \mathcal{N}^o)$; r_l^{sd}=0, otherwise.
$h_{sd} \in \mathbb{R} \geq 0$	Average flow volume between the node pair (s,d) in traffic units (i.e., Kbps).
$C \in \mathbb{R} \geq 0$	Link capacity in traffic units (i.e., Kbps)
$B \in \mathbb{R} \geq 0$	Buffer size in data units (i.e., Bytes)
$M \in \mathbb{R} \geq 0$	Large number to penalize a loss (i.e., loss penalty)

Table 6.1 – Cloud overlay model notation

6.2.2 Oracle Routing Policy

To provide a benchmark solution to the scenario defined above, we propose an Oracle policy corresponding to the centralized version of the multihop DPR problem, where an Oracle observer (different from the routing nodes) has a full knowledge of the underlay network topology $\mathcal{G}^u(\mathcal{N},\mathcal{L})$, the traffic matrix $\boldsymbol{H}$ and the *underlay routing policy* r_l^{sd}. For this solution, we assume that (i) a packet belonging to a flow (s,d) follow a path decided at the source node (flow routing); (ii) the routers can reject new incoming packets at the source node, following a rejection probability of z_{sd}; and (iii) the buffer queues are ignored. This centralized version is formulated as the Minimum Cost Multi-Commodity Flow (MCF) problem solved by a Linear Programming (LP) model. The optimal solution of this model provides a lowest-cost routing policy, which we call *oracle routing* in the remainder of this paper. We formulate the MCF problem in the cloud overlay setting as follows:

$$\min_{\{x,y,z\}} \sum_{\substack{s\in\mathcal{N}^o \\ d\in\mathcal{N}^o \\ p\in\mathcal{P}}} h_{sd} x_p^{sd} + \sum_{\substack{(s,d)\in(\mathcal{N}\times\mathcal{N}) \\ \setminus(\mathcal{N}^o\times\mathcal{N}^o) \\ l\in\mathcal{L}}} h_{sd} y_l^{sd} + \sum_{\substack{s\in\mathcal{N} \\ d\in\mathcal{N}}} M h_{sd} z_{sd} \tag{6.1a}$$

$$\text{s.t} \sum_{p \in \mathcal{P}_n^{in}} x_p^{sd} - \sum_{p \in \mathcal{P}_n^{out}} x_p^{sd} = \begin{cases} -1 + z_{sd}, \text{if } n = s \\ 1 - z_{sd}, \text{if } n = d \\ 0, \text{otherwise} \end{cases}$$

$$(s,d) \in \mathcal{N}^o \times \mathcal{N}^o, n \in \mathcal{N}^o \tag{6.1b}$$

$$\sum_{l \in \mathcal{L}_n^{in}} y_l^{sd} - \sum_{l \in \mathcal{L}_n^{out}} y_l^{sd} = \begin{cases} -1 + z_{sd}, \text{if } n = s \\ 1 - z_{sd}, \text{if } n = d \\ 0, \text{otherwise} \end{cases}$$

$$(s,d) \in (\mathcal{N} \times \mathcal{N}) \setminus (\mathcal{N}^o \times \mathcal{N}^o), n \in \mathcal{N} \tag{6.1c}$$

$$\sum_{\substack{s \in \mathcal{N}^o \\ d \in \mathcal{N}^o \\ p \in \mathcal{P}_l}} h_{sd} x_p^{sd} + \sum_{\substack{(s,d) \in (\mathcal{N} \times \mathcal{N}) \\ \setminus (\mathcal{N}^o \times \mathcal{N}^o)}} h_{sd} y_l^{sd} \leq C, \qquad l \in \mathcal{L} \tag{6.1d}$$

$$x_p^{sd} \in [0,1], (s,d) \in \mathcal{N}^o \times \mathcal{N}^o, p \in \mathcal{P} \tag{6.1e}$$

$$y_l^{sd} \in [0, r_l^{sd}], (s,d) \in (\mathcal{N} \times \mathcal{N}) \setminus (\mathcal{N}^o \times \mathcal{N}^o), l \in \mathcal{L} \tag{6.1f}$$

$$z_{sd} \in [0,1], s \in \mathcal{N}, d \in \mathcal{N} \tag{6.1g}$$

In the above MILP formulation 6.1, x_p^{sd} represents the fraction of the *cloud overlay flow* between the node pair $(s,d) \in \mathcal{N}^o \times \mathcal{N}^o$ to be forwarded via the virtual link (overlay tunnel) $p \in \mathcal{P}$. y_l^{sd} represents the fraction of the *underlay flow* between the node pair $(s,d) \in (\mathcal{N} \times \mathcal{N}) \setminus (\mathcal{N}^o \times \mathcal{N}^o)$ to be forwarded via the physical link $l \in \mathcal{L}$. Equation (6.1a) is the objective function, which gathers the terms to minimize, namely: (i) the total capacity used by the overlay traffic (which is correlated to the end-to-end delay for cloud overlay packets if link capacities are fixed); (ii) the total capacity used by the underlay traffic (which is correlated to the end-to-end delay for underlay packets if link capacities are fixed); (iii) the total lost traffic in the network, where M is the loss penalty. In practice, we fix M to a high value (i.e., 1000). Equations (6.1b, 6.1c) are the cloud overlay and underlay flow conservation constraints, respectively. Equation (6.1d) describes the capacity constraint for each link $l \in \mathcal{L}$. Finally, the constraints (6.1e, 6.1f, 6.1g) define the lower and upper bounds of the variables. We impose that the *underlay flows* y_l^{sd} comply with the *underlay routing policy* r_l^{sd} by means of the constraint (6.1f).

6.2.3 Overlay-Deep-Q-Routing (O-DQR)

The *oracle routing* is the solution of the problem considering a centralized full view of the environment (the underlay network topology $\mathcal{G}^u(\mathcal{N}, \mathcal{L})$, the traffic matrix $\boldsymbol{H}$ and the *underlay routing policy* r_l^{sd}), under the assumptions described before. However, such a *centralized full view* is not always available. In that case, only a *decentralized* version of the problem with a *partial view* of the environment is accessible: the so-called Distributed Packet Routing (DPR) problem.

This DPR problem can be formulated as Multi-Agent Partially Observable Markov Decision Process (Littman, 1994): a Markov Decision Process where each agent can only observe its environment and receive its reward based on its action taken. We adopt the Deep-Q-Routing framework, as in (Mukhutdinov et al., 2019 ; You et al., 2022) and (Alliche, da Silva Barros, et al., 2022), which is based on the Q-Routing approach (Boyan & Littman, 1993) and considers a neural network to approximate the policy.

The choice of the Deep-Q-Routing framework is justified in the context of DTDE since it provides a simple and easy way to implement a control algorithm in this MA-POMDP problem. For this scheme, the agents are installed on the cloud overlay nodes. We consider the underlay layer (underlay nodes, underlay flows, and underlay routing) and the traffic demand for the cloud overlay flows as a part of the environment unknown by the agents.

In the following, we refer to this *decentralized MA-DRL framework* for the *cloud overlay routing problem* as Overlay-Deep-Q-Routing (O-DQR). In O-DQR, a "node" refers to a "cloud overlay node," and a "packet" refers to a "cloud overlay packet".

Let the set of the cloud overlay nodes $\mathcal{N}^o$ be the set of the agents, where each agent $n \in \mathcal{N}^o$ has its own local observation space $\mathcal{O}_n$ and its own action space $\mathcal{A}_n$. When a new packet arrives at time t at node n, the node performs an observation $o_n \in \mathcal{O}_n$ from the environment and takes an action $a_n \in \mathcal{A}_n$ based on it. This action corresponds to choosing an outgoing tunnel to another node $n' \in \mathcal{N}^o$. Consequently, the node n receives from the node n' a reward $r_{n'}$. The reward must represent the cost of sending the packet to the next hop. It can be defined as the time taken by the packet to arrive at the next hop, which corresponds to the *next-hop packet delay* (i.e., *waiting time at node n + transmission time in the tunnel*). A transition is made to a new state each time a packet enters a node, and so a routing decision needs to be made. The action a_n is selected to minimize an estimate of the expected *end-to-end packet delay* from the node n to its final destination. Under the DRL scheme, this estimate, denoted as $Q_n(o_n, a_n; \theta_n)$ (the Q-value), is the output of the Deep Neural Network (DNN) with weights θ_n. This DNN is trained to fit the target value Y_n^Q below:

$$Y_n^Q = r_{n'} + \gamma \cdot \tau_{n'} \cdot (1 - f) \tag{6.2}$$

where f indicates if the next hop is the packet destination, $\gamma \in [0, 1]$ is a discount factor, $r_{n'}$ is the *next-hop packet delay* and $\tau_{n'}$ is *the remaining end-to-end delay* from the next hop n' to the destination computed as follows:

$$\tau_{n'} = \min_{a_{n'} \in \mathcal{A}_{n'}} Q_{n'}(o_{n'}, a_{n'}; \theta_{n'}) \tag{6.3}$$

where $Q_{n'}(\cdot; \theta_{n'})$ is the *Q-value* estimate of the next hop agent.

To approach the estimated $Q_n(\cdot; \theta_n)$ to the optimal Q_n^*, at each agent n, the weights θ_n are updated by stochastic gradient descent when minimizing the next square loss called *Temporal Difference (TD) error*:

$$L_{DQN} = \left(Q_n(o_n, a_n; \theta_n) - Y_n^Q\right)^2 \tag{6.4}$$

In the next paragraphs, we detail (i) the *node observation* representation, (ii) the *node action*, and (iii) the *reward* definition.

Node observation.

The *node observation* $o_n \in \mathcal{O}_n$ is represented as $o_n = (d, \{\delta_p, p \in 1 \dots |\mathcal{P}_n^{out}|\})$. This is the concatenation of *two* components. The first one is *the current packet destination* $d \in \mathcal{N}^o$, which is

the index of the destination node. The second one is *the tunnel delay* $\delta_p \in \mathbb{R}$, in $seconds$, of each outgoing tunnel $p \in \mathcal{P}_n^{out}$ at the node n. In practice, this delay can be estimated from instantaneous measurements collected by sending ping packets through each tunnel at each time step P. Then, a moving average is computed over these measurements by sliding a window corresponding to the last W samples. A timeout is set to detect a lost ping packet due to high congestion in the tunnel.

Node action.

The *node action* $a_n \in \mathcal{A}_n$ is the choice of the outgoing tunnel $p \in \mathcal{P}_n^{out}$ to forward the packet to the next hop node n'. Compared to the *Oracle routing*, the agents cannot decide to drop a packet.

Reward.

The *reward* $r_{n'}$ is the *next-hop packet delay*, defined as the time required by the packet to travel from the buffer tail of n up to the buffer tail of the next hop node n'. Then, it is computed as $r_{n'} = q + l$, where: (i) q refers to *the queuing delay*, the time spent by the packet in the outgoing buffer of node n taking to the next hop node in the tunnel; (ii) l denotes *the tunnel transmission delay*, the transmission latency in the tunnel connecting n and n'. However, if the packet is lost in the tunnel, the reward can be ignored, fixed to a high value, or set dynamically. This will be detailed in the Section 6.3.

Lost packet's treatment.

If the packet is lost in the tunnel, we have multiple options to consider

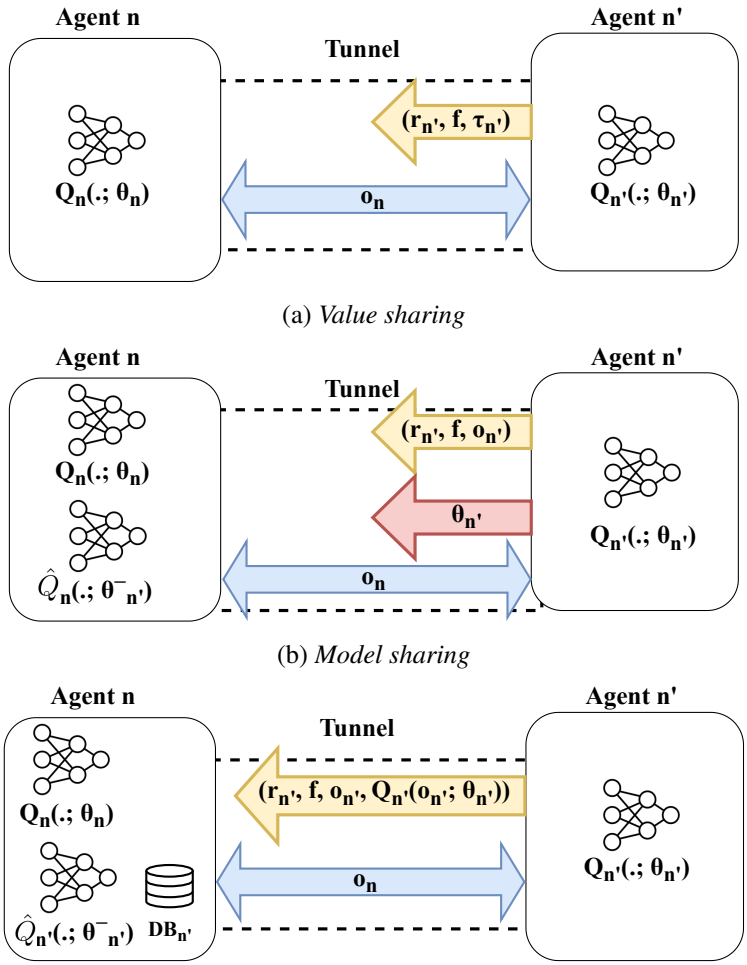
— *No loss*: ignore lost packets and do not include the transition of lost packets in the training of the DNN.

— *Fixed loss penalty*: $r_{n'}$ is considered as infinity since the packet did not arrive at n'. In practice, we use the *worst-case* end-to-end delay, which we refer to as *loss penalty* $l_{pen} = \frac{|\mathcal{N}^o| \times (B+1)}{C}$, that is, the delay of traversing over the $|\mathcal{N}^o|$ underlay nodes when all the output buffers are full. Thus, the reward can be measured in time units (typically, in $seconds$). This reward definition allows the *end-to-end delay* estimate $Q_n(\cdot; \theta_n)$ to account for both buffer delays and packet losses, but giving more weight to the packet losses to favor its minimization in priority, similarly to the LP model of the *oracle routing* (see Section 6.2.2).

— *Guided reward*: a more advanced way to consider lost packets in the reward. This will be explained in the Section 6.3.

Tunnel delay estimate packets.

Along with the control packets discussed in the previous chapter (Section 5.2.5), we add a new type of control packet specific to the overlay case. These packets, implemented as *ping packets*, are required to construct the observation o_n as stated above. The agents need to send them periodically into each outgoing tunnel to estimate the tunnel's delay. This information is used to construct the observation o_n, and so to provide continual monitoring of the outgoing interface state. Thanks to this communication, the agents can detect potential changes in the underlay network topology and the traffic load impacting the tunnel's delay. This communication is required during both the training and the execution phase and represents an overhead that depends on the time interval between two *ping packets*, denoted as P. Figure 6.2 depicts the control packets sent for *model sharing*, *value sharing*, and *logit sharing* when a data packet is received by the next hop node n' coming from a node n. The *replay memory update packets* are represented in yellow and are triggered at the reception of each data packet during the training phase. The *target update packets* are represented in red and are triggered at each U during the training phase. The *ping packets* are represented in blue and are triggered at each P during the training and execution phases.



(a) *Value sharing*

(b) *Model sharing*

(c) *Logit sharing*

Figure 6.2 – Different information sharing strategies. *Tunnel delay estimate packets*, *replay memory update packets*, and *target update packets* are represented in blue, yellow, and red, respectively.

6.2.4 Neural Network Architecture

The architecture of the neural network $Q_n(s_n, a_n; \theta_n)$ chosen is depicted in Figure 6.3. This architecture is inspired by (You et al., 2022) and (Alliche, da Silva Barros, et al., 2022) and confirmed through experimentation. The observation o_n is split into two parts: (i) *the packet destination*, which will be transferred to a *one-hot encoding* layer (i.e., the position corresponding to the destination is set to one, the rest to zero) resulting in a $|\mathcal{N}^o|$-element vector; and, (ii) *the delay for each outgoing tunnel at node* n, as an $|\mathcal{P}_n^{out}|$-element vector, whose values are layer normalized. The outputs of each block are fed to a 32-neuron fully connected layer. Afterward, their outputs are concatenated before feeding two 32-neuron fully connected layers. This split architecture is useful for efficiently treating the different types of data (i.e., the destination is a binary vector, and the tunnel delays are floating points) before merging them, where each block acts as a preprocessing block. Finally, the output layer is fully connected with as many neurons as the node

action space (i.e., out-degree $|\mathcal{P}_n^{out}|$ of n). The value of each output neuron estimates the *Q-value* $Q_n(s_n, a_n; \theta_n)$. All the activation functions are ELUs (Exponential Linear Units), which avoids the "dead neurons" problem of RELUs (Rectified Exponential Linear Units) (Lu, Shin, Su, & Karniadakis, 2019).

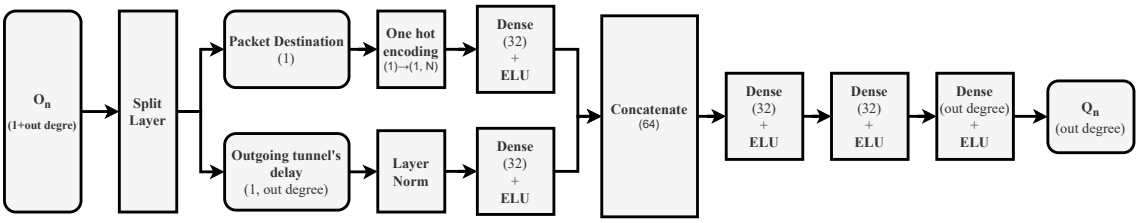


Figure 6.3 – DNN architecture

6.2.5 Training of the model

The training of the model and the DNN architecture are kept the same as our previous study (Alliche, da Silva Barros, et al., 2022). The objective is to update the neural network weights using gradient descent in order to minimize the loss given by Equation (6.4). To compute this loss, it is necessary to know the observation o_n, the action a_n, the reward $r_{n'}$, the flag f and the target value $\tau_{n'}$ (defined by Equation 6.3). The latter can be obtained using three techniques: *value sharing*, *model sharing* and *logit sharing*. The first two techniques were covered in Section 5.2.5. The latter, *logit sharing*, was introduced in Section 5.3.1. It is a novel technique that brings the best of *model sharing* and *value sharing* while preserving a low communication overhead.

6.3 Stabilizing convergence by a guided reward

In this section, we propose a guided reward technique to integrate the lost packet into the reward optimally. We want to answer the research question: *How to define the reward when a packet is dropped in the link ?*
This section is structured as follow:

1. Present a motivation behind the proposed approach.
2. Introduce RCPO (Tessler et al., 2018), a guided reward technique based on a Lagrangian relaxation.
3. Explain how we integrate the idea of RCPO to our framework.

6.3.1 Motivation

Generally, in Reinforcement Learning (RL) systems, the goal is to maximize the accumulated reward. This reward can be computed from a single signal, like in Atari games (Bellemare, Naddaf, Veness, & Bowling, 2013 ; Schwarzer et al., 2023), in which the reward is the game score. However, the problem becomes complex when dealing with multiple rewards. In our case, we have two signals: the delay and the loss. Following a multi-objective approach (Mannor & Shimkin, 2004), we define a penalty coefficient for each signal. Depending on the choice of these coefficients, there exists a different optimal solution known as Pareto optimality (Censor, 1977). In some studies, the

authors considered only the delay and ignored the loss like in (Mukhutdinov et al., 2019; You et al., 2022). In other studies, the weights are defined and fixed before the training. They can be obtained using an optimization algorithm like in (Kaviani et al., 2021), or fixed manually to a suitable value as a hyperparameter like in our previous study (Alliche, da Silva Barros, et al., 2022), where a fixed value is considered for the loss penalty l_{pen}. In practice, setting the weights to the best value can be tedious and time-consuming. Also, selecting a fixed value of the penalty weights can lead to the convergence to sub-optimal solutions (Tessler et al., 2018); when the coefficient related to the loss is too big compared to the delay, it is plausible that at the beginning of training, the agents will only focus on avoiding losing packets, which can results in conservative policy, and thus getting stuck in a local minimum.

Tessler et al. (Tessler et al., 2018) bring a solution to this problem named Reward Constrained Policy Optimization (RCPO). The idea is to formulate the problem as a constrained Markov Decision Process (MDP) and convert it into an unconstrained problem by adopting Lagrangian relaxation, where the penalty weights are the Lagrangian coefficients.

This approach is similar, in its conception, to primal-dual optimization: inside an episode, the agent optimizes its DNN weights while fixing the reward coefficients, and after the episode, the agent optimizes the reward coefficients while fixing the DNN weights. Under mild assumptions, RCPO is proven to provide convergence guarantees and safety with respect to the violation of the constraints. However, RCPO was originally designed and fitted for a single agent actor-critic based approach, like Proximal Policy Optimization (PPO) (Schulman et al., 2017). Our contribution is to adapt this method to our framework, which is a fully decentralized MA-DRL approach based on a Deep Q-Network (DQN) model. The challenges of such an adaptation are: (i) defining the constraint to consider in the model. Intuitively, the link capacity can be considered as such, but in our case, the cloud overlay agents do not have access to the underlay links, where this violation occurs; (ii) making sure to run the data collection and the update of the reward coefficients in a decentralized way (inside an agent and not in a centralized controller).

6.3.2 Reward Constrained Policy Optimization (RCPO)

In classical MDP, the goal is to optimize the expectation of discounted reward J_R^π, given the initial state distribution μ and the discount factor γ. In our case, we aim to minimize the end-to-end delay, which yields the next optimization problem:

$$\min_{\pi \in \Pi} J_R^\pi, \text{ where } J_R^\pi = \mathbb{E}_{s_0 \sim \mu}^\pi \left[\sum_{t=0}^{\infty} \gamma^t r_t \right] = \sum_{s \in S} \mu(s) V_R^\pi(s) \tag{6.5}$$

$V_R^\pi(s)$ represents the value due to following a policy π started from the state s. This value function solves the recursive Bellman equation:

$$V_R^\pi(s) = \mathbb{E}^\pi[r(s,a) + \gamma V_R^\pi(s') | s] \tag{6.6}$$

Constrained Markov Decision Process (Altman, 2021) extends MDPs by introducing a penalty signal $c(s,a)$, a constraint $C(s_t) = F(c(s_t,a_t), \ldots, c(s_N,a_N))$, and a threshold $\alpha \in [0,1]$. The function F could be a discounted sum or an average sum. The objective in this case is to optimize the expectation of the discounted reward with respect to the expectation over the constraint J_C^π:

$$\min_{\pi \in \Pi} J_R^\pi \text{ , s.t. } J_C^\pi \leq \alpha, \text{ where } J_C^\pi = \mathbb{E}_{s_0 \sim \mu}^\pi [C(s)] \tag{6.7}$$

The Lagrangian relaxation of this MDP gives the following unconstrained problem:

$$\max_{\lambda \geq 0} \min_{\theta} L(\lambda, \theta) = \max_{\lambda \geq 0} \min_{\theta} [J_R^{\pi_\theta} + \lambda.(J_C^{\pi_\theta} - \alpha)] \tag{6.8}$$

where L is the Lagrangian, and $\lambda \geq 0$ are the Lagrange multipliers, which are the penalty coefficients. The solution of this problem is done on a two timescale approach: on a faster timescale (inside an episode), we update θ with temporal-difference learning considering the penalized reward function $r'(\lambda, s_t, a_t) = r(s_t, a_t) + \lambda c(s_t, a_t)$, and the new discounted penalized reward $V^{'\pi}(\lambda, s_t) = V_R^{\pi}(s_t) + \lambda V_C^{\pi}(s_t)$, with λ being fixed; on a slower timescale (between two episodes), we update λ using gradient descent (see Equation (6.9)), with θ being fixed.

$$\lambda_{k+1} = \lambda_k + v_2 \nabla_\lambda L(\lambda, \theta) \tag{6.9}$$

where v_2 is the learning rate and $\nabla_\lambda L(\lambda, \theta)$ is the gradient for the coefficient, which is given by:

$$\nabla_\lambda L(\lambda, \theta) = \mathbb{E}_{s \sim \mu}^{\pi_\theta}[C(s) - \alpha] \tag{6.10}$$

6.3.3 Adapting the RCPO technique to the cloud overlay DPR problem

In our case, we consider the end-to-end delay as the primary goal, which we have to minimize with respect to the capacity constraint (described in Equation (6.1d)). The capacity constraint concerns the underlay links (physical links). So, unlike (Kamri et al., 2021), we don't have access to the buffers of the underlay network interfaces. Thus, we are not able to measure directly the constraint signal $c(s, a)$ to define its violation. Thus, a research question arises: How can the constraint penalty be properly set in this complex case?

We propose answering the above question by measuring the constraint violation with an overlay lost packet signal. In this solution, the constraint signal corresponds to a packet loss during its transition to the next hop. For each cloud overlay node n, we define a set of λ_k which corresponds to each outgoing tunnel $k \in \mathcal{P}_n^{out}$. When a node n sends a data packet to a next hop n' and the packet is lost, the penalty signal $c(s, a)$ is set to 1, and the delay reward $r(s, a)$ is set to 0. If the packet was successfully delivered, we ignore the penalty signal, that is, $c(s, a)$ is set to 0, and consider only the reward $r(s, a)$. So, the final reward is defined as follows:

$$r_{n'} = \begin{cases} r(s, a), \text{ if next hop was reached} \\ \lambda_i, \text{ if packet was dropped} \end{cases} \tag{6.11}$$

The constraint $C(s_t)$ is the average sum of the penalties $c(s, a)$. We set the constraint threshold α to 0 since we aim to have zero packet loss when training in low to average traffic loads.

We have designed this solution to make sure that all the information needed to compute the constraint $C(t)$ along with the coefficient gradient $\nabla_\lambda L(\lambda, \theta)$ are available locally on the node, and thus do not introduce additional interaction between the nodes or the need to have a centralized controller. However, this solution requires the detection of dropped packets, which can be done by establishing a packet acknowledgment mechanism.

Algorithm 2 depicts the pseudo-code of the training routine of *O-DQR* on a node n, namely MA-DRL with *logit sharing*, dynamic sending of *replay memory update packets*, and *guided reward* mechanism.

Algorithm 2: Training *O-DQR* on a node n

Input: Total number of training episodes N_{Ep}, Episode duration S; Gradient descent update period T; target network supervised training period T; Learning rate to update DNN weights ν_1; Learning rate for guided reward coefficients update ν_2; tunnel delay estimate period P; Weights $\theta_v^{SP}, v \in \left\{n \cup \mathcal{N}_n^{out}\right\}$

Output: Final routing model weights θ_n

Initialize experience replay memory $\mathcal{M}_n$;
Initialize $Q_n(\cdot)$ with weights $\theta_n \leftarrow \theta_n^{SP}$;
Initialize targets $\hat{Q}_v(\cdot), v \in \mathcal{N}_n^{out}$ *with weights* $\theta_v^- \leftarrow \theta_n^{SP}$*;*
while *current episode index* $i_{ep} < N_{Ep}$ **do**
  while *current simulation timestamp* $t < S$ **do**
    if $t \mod P$ **then**
      for *neighbor* **do**
        Send a ping packet to the neighbor;
    for *each arrival packet* **do**
      Observe the current node state o_n;
      Select action $a_n = n'$, where
      $n' = \begin{cases} \text{a random neig. } v \in \mathcal{N}_n^{out}, \text{with prob. } \epsilon \\ \text{argmax}_{v_n \in \mathcal{N}_n^{out}} Q_n(o_n, v_n), \text{ otherwise} \end{cases}$ Forward packet p to next hop node n' with the output value $Q_n(o_n, a_n; \theta_n)$;
      Receive back from n' next-hop state $o_{n'}$, the logit vector $Q_{n'}(o_{n'}, .; \theta_{n'})$,the reward r_n and the flag f;
      Store transition $(o_n, a_n, r_{n'}, o_{n'}, f)$ in memory $\mathcal{M}_n$;
      Store the state o_n and the output $Q_{n'}(o_{n'}, .; \theta_{n'})$ in the supervised database for this neighbor n $\mathcal{DB}_{f\,n}$;
      Receive back from n' next hop state $o_{n'}$, output of the neighbor's DNN $Q_{n'}(o_{n'}, .; \theta_{n'})$, the reward r_n and the flag f;
      observe the penalty value $c(o_n, a_n)$;
      store the penalty value in the memory $\mathcal{M}_n$ along with the transition;
      if $t \mod T$ **then**
        Sample random batch $\mathcal{B} \overset{i.i.d.}{\sim} \mathcal{M}_n$;
        Set target values Y_n^Q as (6.2) for $\mathcal{B}$;
        Update weights θ_n by gradient descent on TD error for $\mathcal{B}$;
      if $t \mod T$ **then**
        Update target weights: $\theta_v^- \leftarrow \theta_v, v \in \mathcal{N}_n^{out}$ *based on the data stored in the supervised database* $\mathcal{DB}_{f\,v}$*;*
        clear the database $\mathcal{DB}_{f\,n}$*;*
  for *neighbor* **do**
    Retrieve the penalties $c(s, a)$ stored for this neighbor i from $\mathcal{M}_i$;
    Aggregate the penalties $c(s, a)$ to compute the constraint $C(s)$;
    Update loss penalty for this neighbor i λ_i following the Equation 6.9;

6.4 Experiments

6.4.1 Experimental Settings

In this section, as pointed out above, we assume that *O-DQR* refers to the *MA-DRL framework* for the *cloud overlay routing problem* as described in Algorithm 2, i.e., with *logit sharing*, dynamic sending of *replay memory update packets*, and *guided reward* mechanism. In this subsection, we describe the experimental approach to answer the following research questions:

— Q1: What is the performance of O-DQR compared to taking the direct cloud overlay tunnel (the by-default ISP configuration) and to the *Oracle routing policy* 6.2.2?

— Q2: What is the effect of having a guided reward penalty on the convergence of O-DQR compared to ignoring the packet loss or applying a predefined packet loss penalty?

— Q3: How much overhead can we reduce, compared to *model sharing*, by using *logit sharing* and enabling the dynamic sending of *replay memory packets*?

Hardware and software settings.
We ran the training and testing of the different models locally in a Dell Precision 7920 workstation equipped with an Intel Gold 6230R (26 Cores, 2.1-4.0GHz, 128GB RAM) with 2 NVIDIA RTX A5000 GPUs, running Linux Ubuntu 20.04. The DNN agent model is implemented in TensorFlow (*TensorFlow: An end-to-end open source machine learning platform*, s. d.) version 2.8. The implementation of the MA-DRL method is based on the OpenAI™ Baselines library (Brockman et al., 2016). The *prisma-v2* tool (*PRISMA tool: An open MARL framework for packet routing.*, s. d.) (Alliche, Barros, et al., 2022) is used as a multi-agent network simulation environment, which can emulate the data and the control packets.

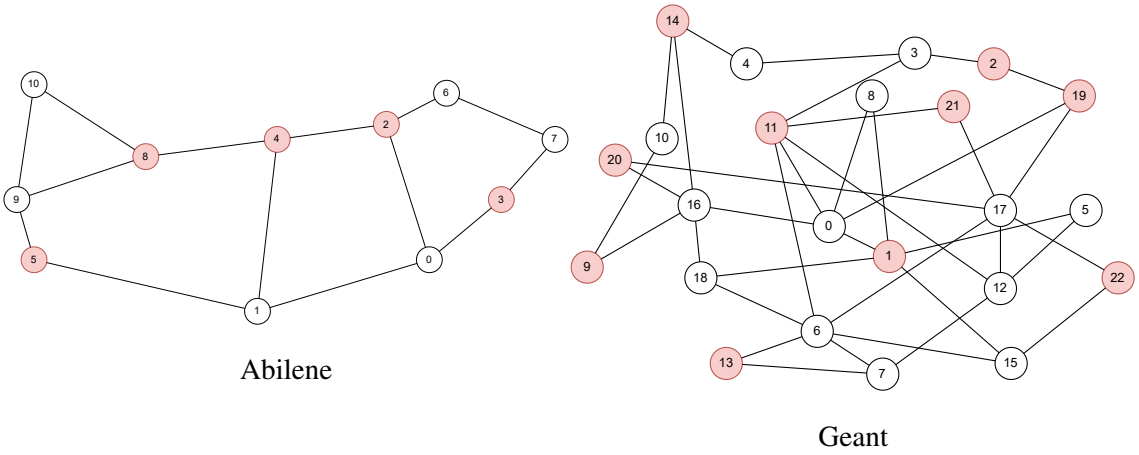
Results reproduction.
The simulator used, *prisma-v2*, which is described in Section 4.3 of Chapter 3.

Evaluation metrics.
We evaluate the performance of *O-DQR* by considering the network metrics collected over all the cloud overlay nodes during the test period: average end-to-end packet delay and loss ratio for cloud overlay packets. For the control packet overhead, we consider the total size of sent control packets divided by the total size of sent data packets over the training period. We refer to the latter metric by *overhead ratio*.

Competitors.
We compare the routing optimality of *O-DQR* in Algorithm 2 to the *ISP default path*, and the *Oracle routing* from LP model in Section 6.2.2. The *ISP default path* and *Oracle routing* solutions represent performance benchmarks used to assess the quality of our DRL solution. The *ISP default path* in our full mesh setting consists of routing the traffic between two cloud overlay nodes by taking the direct tunnel connecting them. We assume in this work that the underlay routing policy used by the ISPs is the shortest path routing. However, we cannot compare directly *O-DQR* to the DPR literature algorithms (You et al., 2022 ; L. Chen et al., 2021 ; Manfredi et al., 2021) since they do not support the cloud overlay setting. The other track of works treating cloud overlay routing under Software-Defined Networking (SDN), like (Kamri et al., 2021 ; Botta et al., 2023), considers the problem as a load balancing over a set of predefined paths, which is different from our conception, where we consider multihop distributed packet routing. Also, the training is centralized in a controller node, collecting all the past experiences. In contrast to our settings, the training is done on the distributed node agents.

Network topologies.
We test *O-DQR* in two different cloud overlay topologies. Those topologies are built on top of real-world networks. The first topology consists of a five-node full mesh cloud overlay network built on top of **Abilene** ($|\mathcal{N}|$=11) (*SNDlib: Library of test instances for Survivable fixed telecommunication Network Design*, s. d.). The second topology consists of a 10-node full mesh cloud overlay network built on top of **Geant** ($|\mathcal{N}|$=23) (*SNDlib: Library of test instances for Survivable fixed telecommunication Network Design*, s. d.) network. Both topologies are presented in the Figure 6.4. We opted for a full mesh cloud overlay topology to allow any cloud overlay node pair

to be connected by a direct tunnel. We assume no knowledge about the underlay topology. We fix link propagation delay and link rate C for the underlay topology to one millisecond and $500Kbps$, respectively.



Abilene

Geant

Figure 6.4 – Network topologies. In red, the cloud overlay nodes

Traffic generation.
We consider 10 traffic matrices for both topologies. Each traffic matrix $\boldsymbol{H}$ is generated by sampling each element h_{sd} from a uniform distribution $U(0, 1)$. Then, we multiply *element-wise* this traffic matrix by a generated matrix $\boldsymbol{A}$ to have a configuration that maximizes the gap between the *Oracle routing* and *ISP default path*. The values of this matrix, $\boldsymbol{A}$, are found using a heuristic that finds the cloud overlay flows with the maximum number of alternative paths in the underlay. For each source/destination pair, (i) get the k_{max} shortest paths in the overlay network (where k_{max} is a fixed parameter); (ii) compute their equivalence in the underlay topology; (iii) remove the redundant paths and the path using the direct underlay shortest path; (iv) set the elements $a_{ij} \in \boldsymbol{A}$ proportional to the number of alternative underlay paths found for the flow connecting the node pair (i, j).

We scale up the resulting traffic matrices ($\boldsymbol{H} \odot \boldsymbol{A}$) by multiplying it by a coefficient α. We increase α up to the largest value α_{max} for which an optimal routing with no packet loss can still be found by the LP model (6.1). The matrix α_{max} ($\boldsymbol{H} \odot \boldsymbol{A}$) is associated with a load factor $\rho = 1$. Data packets are generated as `UDP` over `IP` datagrams. Their payload is 512 B long. The `UDP` and `IP` headers are 8 B and 20 B long, respectively. Then, the total packet size is 540 B.

We do not explicitly consider the additional header needed to distinguish cloud overlay packets since it depends on the tunneling technology. Cloud overlay packets have cloud overlay nodes as sources and destinations. The rest of the traffic packets are considered underlay packets, which are not controlled by the cloud overlay nodes and, thus, do not trigger actions from the agent. Packet traces are produced assuming that packet *inter-arrival time* follows an exponential distribution with mean $540B/h_{sd}$. Finally, the output buffer size is fixed to $16,200B$ (i.e., 30 data packets of $540B$ long).

Control signalling packets.
As explained in the previous chapter (Sections 5.2.5 and 5.3), DRL agents share information through control packets that we want to quantify. For the *replay memory update packets*, we encode each float or integer data type unit in $8B$, and we ignore the flag f since it is a binary value ($1bit$).

Then, a pair $(r_{n'}, o_{n'})$ (*model sharing*) is $16+8\cdot|\mathcal{N}_n^{out}|$ B long, and a pair $(r_{n'}, o_{n'}, Q_{n'}(o_{n'}, .; \theta_{n'}))$ (*logit sharing*) is $16 + 16 \cdot |\mathcal{N}_n^{out}|B$ long. Control packets are also encapsulated into `UDP` over `IP` datagrams, which adds a header size of $28B$. For a *target update*, the size of a DRL agent model is around $36KB$, which we split into *target update packets* of $512B$ long. Since the cloud overlay topology is full mesh, the communication overhead for *target update packets* becomes significant and will strongly impact the model's training. For this reason, we decided to send those packets in an off-band control channel different from the data channel, which allows us to access the performance of *model sharing* without overloading the network. For the *tunnel delay estimate packets*, the size of the payload is $8B$, which is encapsulated into `UDP` over `IP` datagrams.

Training procedure.

The weights θ_n are initialized with supervised pre-training on the shortest path routing onto the cloud overlay topology or randomly sampled. The model is trained using *ADAM* optimizer with a learning rate of $1e-4$, batch size of 512, and γ of 1. The training is divided into 20 episodes of 20 seconds each, which results in a total training duration of 400 seconds (in `ns-3` simulation time). The gradient descent is launched every $T = 10$ milliseconds (in `ns-3` simulation time). Moreover, we use an ϵ-greedy approach (ϵ decays from 1 to 0.01) for each episode to trade between exploration and exploitation. The moving average window for the tunnel delay estimate is set to $W = 5\ packets$. We consider a replay memory size of $10K$ experiences. The local training period in *logit sharing* is done every five seconds, with 50 epochs and the same learning rate and batch size as DRL. We clear the local learning database after each training. The update of the constraint penalties is done between two episodes, and its learning rate (v_2) is set to 100 times lower than the DRL learning rate. We fix the random seed for all the experiences. Based on previous studies and experimental search, we set the training load to 0.4, the target update period for *model sharing* to $U = 5s$, and the *tunnel delay estimate packets* period P to $P = 100ms$).

We execute different training sessions for each traffic matrix, information-sharing technique (*model sharing* and *logit sharing*), reward definition technique (ignoring the packet loss, fixing a packet loss penalty, or considering the *guided reward* as in Section 6.3), *experience relevancy* threshold ER_{Thr} (0%, 25%, 50%, 75%). After each episode, the corresponding model weights are saved for all the agents.

Testing procedure.

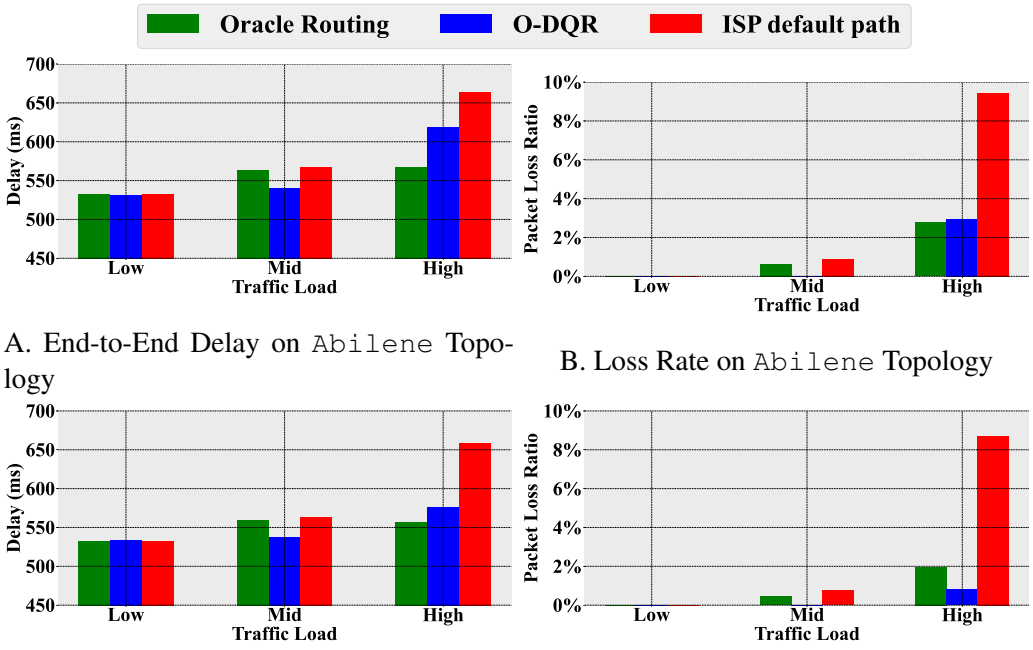
For each trained model, a test phase is performed on the tuple *{episode number, traffic matrix, sharing technique, reward definition technique, experience relevancy threshold}*. We ran *three* simulations for load factors $\rho = [0.6, 0.9, 1.2]$. The rationale behind testing for traffic loads higher than 1.0 is to evaluate the model in highly saturated scenarios, where buffer delays become huge. The test packet traces are generated using the same traffic matrix as training but scaled to the corresponding load factor ρ. In other words, with respect to the training procedure, we test each DNN model with the same traffic distribution among nodes but with higher loads. Each *testing* simulation lasts 20 seconds. This duration is sufficient to reach a stationary state in the network. We disable the *delay estimate packets* for the competitors since they do not exploit the tunnel delay.

6.4.2 Experiments results

In the following section, we present the results of the experiments to answer the above-mentioned research questions.

6.4.2.1 Answering Q1: performance of *O-DQR*

We answer the first question by evaluating the trained *O-DQR* in different test loads. We chose $\rho = 0.6$ as a low load, $\rho = 0.9$ as a mid load, and $\rho = 1.2$ as a high load. Figure 6.5 compares the performance (in terms of average end-to-end delay and loss rate) of *O-DQR* with the *ISP default path routing* and the *Oracle* routing. We run the experiments on all ten traffic matrices for Abilene topology and four matrices for Geant topology. Each bar shows the average over all the traffic matrices. The supervised training time step T is set to five seconds. In the low-load scenario, all the



A. End-to-End Delay on `Abilene` Topology

B. Loss Rate on `Abilene` Topology

C. End-to-End Delay on `Geant` Topology

D. Loss Rate on `Geant` topology

Figure 6.5 – Performance Metrics vs Load

models perform equally well, with no packet loss and an average delay of 540 milliseconds. This is because the best policy in this case is to follow the direct tunnel (i.e., *ISP default path routing*) adopted by both *O-DQR* and *Oracle routing*. However, in the mid-load scenario, the nodes experience packet loss when using the *ISP default path* or *Oracle* routing policies. *O-DQR* outperforms these models by maintaining a zero loss rate and a lower average end-to-end delay. This shows that *O-DQR* discovers the tunnels traversing the congested underlay links, even though the underlay configuration is unknown to the cloud overlay agents. In particular, O-DQR outperforms Oracle in delay and loss in the mid-load scenario. This can be explained by the flow-based nature of the LP model of the *Oracle routing*. The Oracle has a coarse-grained view of the packets' flows. Then, it cannot correctly handle temporary traffic peak scenarios (instantaneous flow values exceeding average ones), where disposing of an estimate of the current tunnel congestion (like the *tunnel delay* estimate in *O-DQR*) can improve the routing policy.

In the high load scenario, the loss ratio for the *ISP default path* reaches almost 9% as the direct tunnel becomes saturated with packets. *O-DQR* significantly reduces the loss rate in this scenario by up to 66% compared to the direct tunnel. *O-DQR* performs better in the `Geant` topology since

it has a greater degree of freedom. Moreover, *O-DQR* matches, and even slightly overcomes, the performance of the *Oracle routing*, even though the latter can reject packets at the source and "knows" the traffic matrix, the underlay topology and the underlay routing.
Conclusion for Q1: *O-DQR* beats the *ISP default path* in mid and high-load scenarios and achieves a performance close to the *Oracle routing*.

6.4.2.2 Answering Q2: convergence and stability study.

In this part, we study the effect of the guided reward strategy on the convergence of *O-DQR* by comparing it to two approaches to introduce the packet loss in the reward: (i) the *loss blind* approach, which ignores the lost packet and (ii) the *fixed loss-pen* approach that considers a fixed penalty. To evaluate the convergence, we retrieve the model at each episode and run an evaluation on it. Figure 6.6 illustrates the evolution of the performance of each loss penalty method at different states of the training phase for the two topologies. Each point represents an average over all the loads and traffic matrices. Regarding loss rate, we observe that guided reward significantly improves convergence speed compared to *loss blind* and *fixed loss-pen*. It reaches a steady state after the fourth episode, whereas the other methods need at least twice this duration to converge. The end-to-end delay continues to improve slightly with time for all the methods. The results confirm the importance of the guided reward in stabilizing and speeding up the training and convergence of the agents.

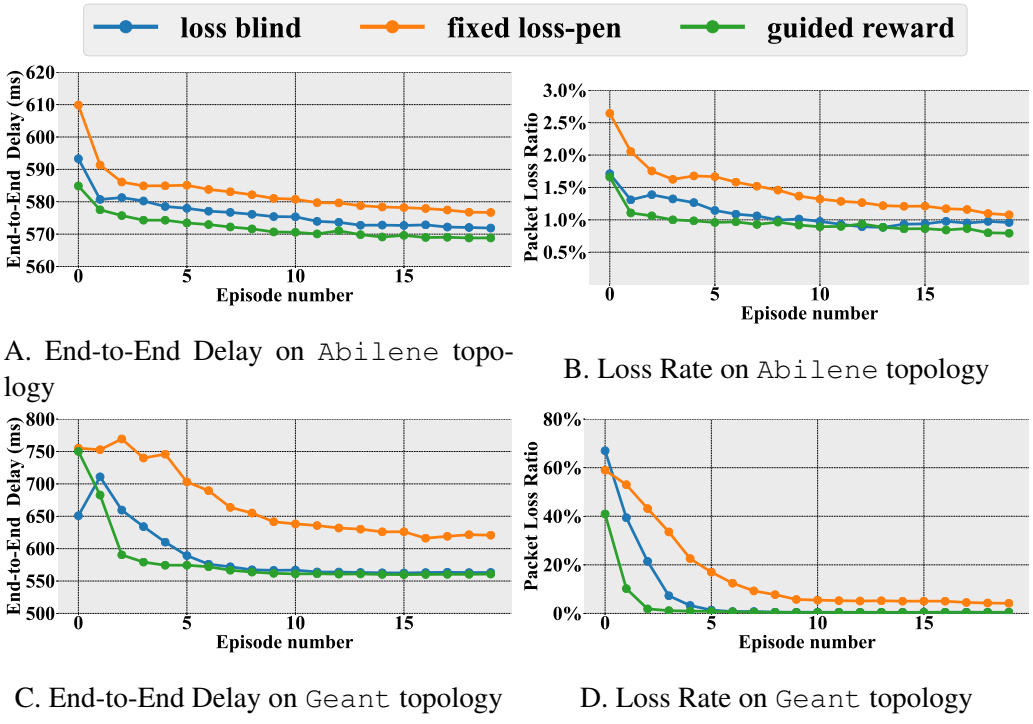
Conclusion for Q2: Furthermore, the utilization of the "guided reward" approach not only enhances performance but also improves the model's convergence and stability. Models employing the "guided reward" strategy typically achieve convergence approximately twice as quickly as those employing "loss blind" and "fixed loss-pen" strategies. Additionally, they demonstrate smoother convergence curves for increased stability.

6.4.2.3 Answering Q3: communication overhead study.

The control communication overhead is evaluated in this part. To do that, the overhead of our *logit sharing* strategy, denoted here as *O-DQR*, is benchmarked against the *model sharing* strategy. For both strategies, the training period of the local copies of the neighbors' models is fixed at five seconds.

Since the *target update packets* (only used by *model sharing*) introduces significant overhead, the *model sharing* agents overload the network links with control packets, preventing the *O-DQR* convergence. Thus, we are forced to send the *target update packets* in an off-band channel different from the channel used for the data. In the remainder of the section, we assume off-band *target update packets*. Then, their overhead contribution in *the model sharing* method is estimated, in contrast to the other control packets, whose contribution is measured from the simulation. For *logit sharing*, we vary the *experience relevancy threshold* ER_{thr} from 0% to 75% with a step of 25%.

— **Control packets overhead and performance trade-off.** Plots A to D of Figure 6.7 illustrate the trade-off between the performance of the model in the evaluation phase and its communication overhead ratio during the training phase. This trade-off depends on the *experience relevancy threshold* ER_{thr} for *logit sharing*. The number of *replay memory update packets* in *logit sharing* (and, then, the overhead) decrease when the value of ER_{thr} augments. In particular, increasing ER_{thr} reduces the overhead from 54% to 26% in Abilene and from 79% to



A. End-to-End Delay on `Abilene` topology

B. Loss Rate on `Abilene` topology

C. End-to-End Delay on `Geant` topology

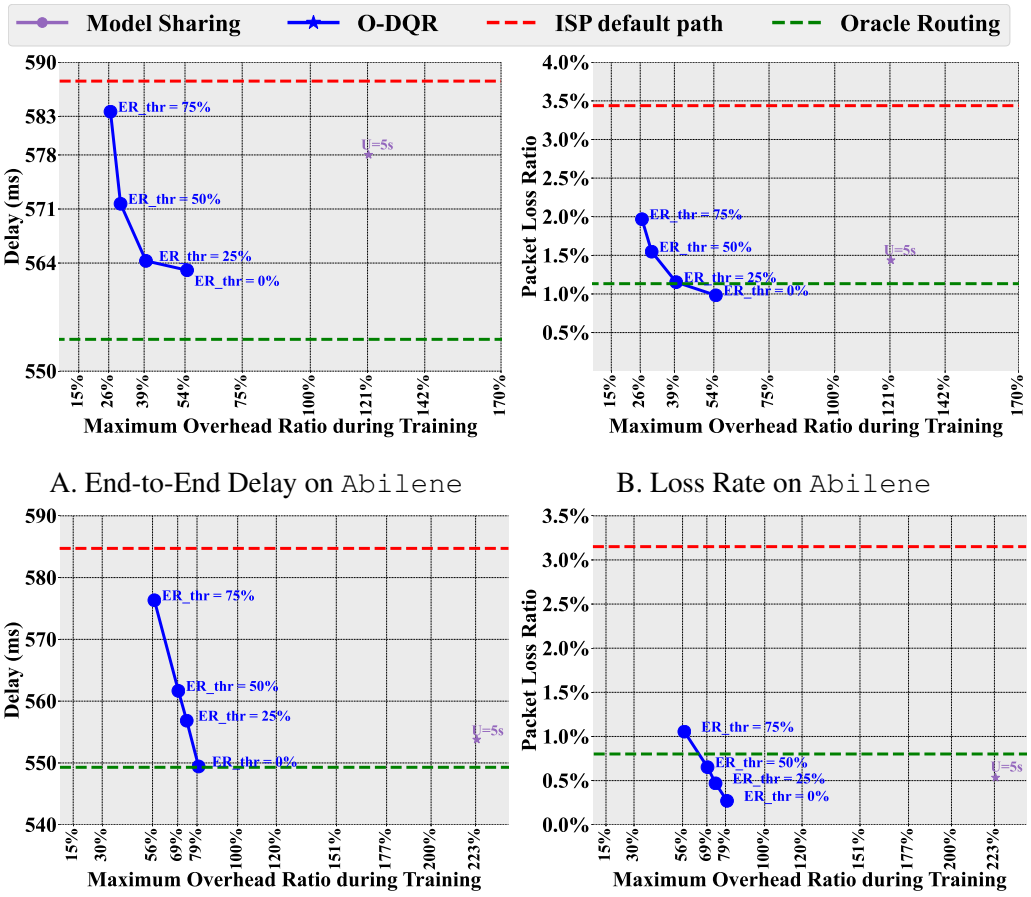
D. Loss Rate on `Geant` topology

Figure 6.6 – Performance metrics vs episode number

56% in Geant for *logit sharing*, but at the cost of degrading the performance. However, even for the lowest overhead ($ER_{thr} = 75\%$), *O-DQR* obtains, with respect to the *ISP default path*, a better performance in loss rate (42% improvement in Abilene and 66% in Geant for the worst case) and a similar one in delay. When we compare the two sharing techniques, without dynamic sending of replay memory update packets ($ER_{thr} = 0\%$), *logit sharing* (i.e., *O-DQR*) outperforms *model sharing* in loss rate (over 31.62% in Abilene and 49.33% in Geant), while keeping the same end-to-end delay and requiring significantly fewer control packets (over 55% reduction in Abilene and 64% in Geant).

— **Control packets overhead evolution during the training** Figure 6.8 illustrates the evolution of the overhead ratio per episode during the training phase when enabling the dynamic sending of *replay memory update packets* in *O-DQR*. We then observe that the control overhead decreases with the training time until reaching a minimum. At the beginning of the training, the *experience relevancy* is high since the policies are not optimal. Then, when the models get better, the *experience relevancy* lowers, and the agents send fewer and fewer control packets.

Conclusion for Q3: The results show that *O-DQR* with *logit sharing* outperforms the *model sharing* version. This is because it offers stability in the changes of the target model weights, helping the convergence: the weights are updated progressively in a supervised way, which is smoother than changing all the weights between two *target update* periods as done in *model sharing*. Moreover, by tuning the dynamic sending of *replay memory update packets* with the *experience relevancy threshold* ER_{thr}, *O-DQR* can achieve delay and loss rate performances superior to the *ISP default path* and close to the *Oracle routing*, with a moderate overhead in comparison with the *model sharing* version. On the other hand, in terms of overhead, we can control how often agents



A. End-to-End Delay on `Abilene`

B. Loss Rate on `Abilene`

C. End-to-End Delay on `Geant`

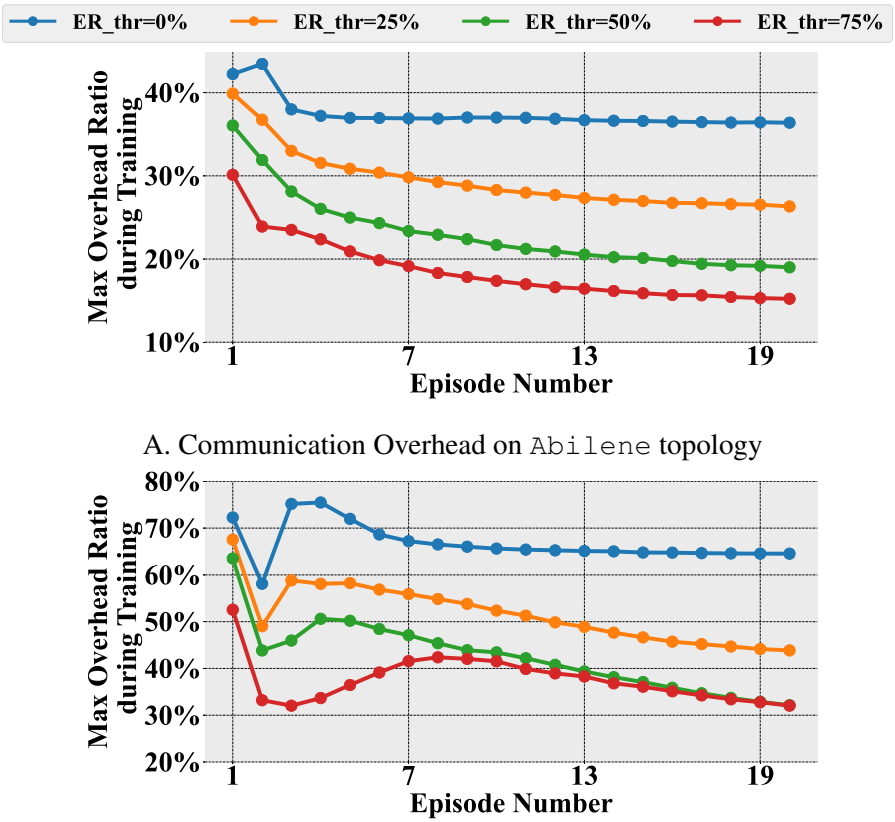
D. Loss Rate on `Geant`

Figure 6.7 – Study of the communication overhead of our model. Plots A to D represent the trade-off between the performance and the overhead

share their experiences based on their relevancy. The *experience relevancy threshold* parameter allows tuning the system sensitivity to the experience relevancy. Depending on its value, we can reduce the maximum training overhead ratio from 54% to 26% in Abilene and 79% to 56% in Geant.

6.5 Discussion

First, the *logit sharing* strategy implemented in *O-DQR* relies only on the dynamic sending of *replay memory update packets*, which can be controlled by the *experience relevancy threshold* ER_{thr}. The mechanism allows tuning the control signalling overhead. For a fixed ER_{thr}, the overhead diminishes with the progression of the training. By changing ER_{thr}, we can trade the performance of the learned routing policy with the overhead required for learning it. In a *continual learning* scenario, these features imply that (i) most of the overhead will take place at the moments when significant changes (e.g., in the traffic matrix or the underlay network) happen, (ii) the cloud



A. Communication Overhead on `Abilene` topology

B. Communication Overhead on `Geant` topology

Figure 6.8 – Evolution of the communication overhead during the training when varying the experience relevancy threshold

overlay operator can select the performance-overhead trade-off point required for satisfying their clients.

Second, the *guided reward mechanism* is based on an episodic run. The loss penalty coefficients are updated between each two episodes. To ensure the convergence of this method, the learning rate of updating the coefficients should be significantly lower than the learning rate of updating the DNN weights. Also, it is necessary to keep the initial conditions, exploration policy, and network settings (traffic and topology) static between two episodes. This condition was preserved in our simulations since the network is reset at the start of each episode, but it can be hard to ensure in a *continual learning* case. In this case, the *guided reward mechanism* should be adapted to define an episode.

The *continual learning* scenario is more relevant to cloud overlay operators, as CDNs. Consequently, future works will focus on adapting the proposed framework to this scenario. That means making the traffic matrix evolve along with time, and continuously training the models, in contrast to our experiments where we evaluated the framework in an offline setting with a static traffic matrix. We then plan to evaluate the model in a dynamic traffic situation and address the issues pointed out in the previous paragraphs.

6.6 Conclusion

In this chapter, we have presented a complete MA-DRL framework to address the problem of multi-hop packet routing in cloud overlay networks. It is, to the best of our knowledge, the first fully decentralized solution under the DTDE scheme, in which the control communication overhead is evaluated and controlled. This framework includes three novelties in order to reduce the overhead and improve the convergence speed and stability: (i) DNN agents exchange their logits (instead of all the DNN weights as in the State-of-the-Art) to tackle the non-stationary of multi-agent environments, (ii) DNN agents can tune how often to share their experiences based on their relevancy for the training process, and (iii) a guided reward penalty mechanism is used to prevent the convergence issues from learning a multi-objective function (end-to-end delay and packet loss). The results of the different experiments that we ran show that our framework achieved near-optimal performance on a small or large network topology while having a reasonable and controlled overhead.

In the next chapter, we present two other applications of DRL to networks that have been held during the thesis: (i) DRL for cloud slices reconfiguration and (ii) DRL for virtual link allocation in quantum networks.

CHAPTER 7

Other Applications of Reinforcement Learning to Network Control

This chapter focuses on the application of Deep Reinforcement Learning (DRL) to two challenging scenarios. The first scenario involves the management of cloud slicing from the perspective of 5G network operators. This research was conducted in collaboration with the COATI team at INRIA and was part of Adrien Gausseran's Ph.D. thesis (Gausseran, 2021). Our findings were presented at the IEEE 11th International Conference on Cloud Networking (CloudNet) (Gausseran et al., 2022).
The second endeavor concerns the generation of quantum virtual links, approached from the standpoint of quantum network operators. Our work on this frontier was presented at the 23rd International Conference on Transparent Optical Networks (ICTON 2023) (Aparicio-Pardo et al., 2023).
This chapter unfolds in two sections, each dedicated to elucidating each study.

7.1 Introduction

Up to this point, we have explored network optimization using DRL within the context of Content Distribution Network (CDN), showcasing compelling results in cloud overlay configurations. Expanding our investigation, this chapter delves into two additional complex scenarios: 5G network slicing and quantum network entanglement management.

The emerging 5G landscape brings a multitude of challenges, characterized by an increasing number of connections and throughput and imposing stringent quality of service requirements such as low latency and request isolation. Network Function Virtualization (NFV) and Software Defined Networking (SDN) introduce network slicing to address these growing demands. However, efficiently provisioning network and cloud resources for a myriad of applications with fluctuating user demands remains a challenging task. The dynamic nature of these demands necessitates regular reconfiguration of network slices to maintain efficient provisioning. Balancing the frequency of reconfigurations poses a significant challenge, as it involves a trade-off between mitigating network congestion and minimizing additional costs incurred by reconfiguration.

In the first section of this study, we tackle the problem of deciding the optimal moment for reconfiguration, considering this trade-off. By coupling DRL for decision-making and a Column Generation algorithm for reconfiguration computation, we propose an innovative approach, named `Deep-REC`. Our approach identifies optimal reconfiguration timings, maximizing network operator profit while minimizing resource utilization and network congestion. Moreover, by selecting the best moment for reconfiguration, our method reduces the number of required reconfigurations compared to periodic approaches, thereby enhancing overall network efficiency.

In the context of quantum networks, the utilization of quantum entanglement enables the development of novel applications such as quantum key distribution and distributed quantum computation. However, managing quantum entanglement effectively presents significant challenges due to its probabilistic nature and dependency on the characteristics of quantum devices. The management of entanglement, crucial for maintaining high-quality connections, poses a stochastic control problem that can be modeled as a Markov Decision Process (MDP) and addressed using Reinforcement Learning (RL).

In the second section of this study, we apply a DRL framework to learn an entanglement management policy, surpassing existing methodologies, particularly in scenarios where precise models of the quantum devices are unavailable. By harnessing DRL, we optimize entanglement management, enhancing the performance and reliability of quantum networks in diverse operational environments.

Finally, we conclude this chapter with Section 7.4

7.2 Reconfiguring Network Slices at the Best Time With Deep Reinforcement Learning

To meet the growing and increasingly diverse demands of users and companies, networks have evolved, adopting new technologies to make their management more efficient and more responsive to dynamic changes in traffic. The first fundamental evolution in the management of modern networks is Software-Defined Networking (SDN). SDN is a network paradigm that decouples the control plane from the data plane and enables centralized network control. The network becomes

programmable, and routing rules can be adjusted in real-time, leading to better responsiveness of network management in case of traffic changes.

Network Function Virtualization (NFV) comes with SDN as a paradigm allowing the decoupling of network functions from the hardware. Functions can be virtualized on generic servers located in data centers dispersed all over the network. NFV allows, first of all, the reduction of capital expenditure (Capex) by avoiding the purchase of dedicated equipment for each function. It also reduces operational costs (Opex) since a function can be easily stopped when not used. Finally, NFV allows more flexible network management since functions can be instantiated on demand anywhere in the network when needed due to traffic dynamics.

By combining SDN and NFV technologies, network management thus is greatly enhanced by making the network programmable, dynamic, and flexible and by allowing for controlled sharing of resources between different services and users. The increasing importance of wireless networks and the emergence of 5G bring out new needs such as massive device connectivity, high mobility, and a great diversity in the quality of service (QoS) requirements. Network slicing has been proposed to meet this challenge and to satisfy these diversified service needs. By dividing the network infrastructure into multiple logical isolated networks, network slicing allows the support of a wide range of communication scenarios with a diversified set of service demands, requirements, and performance. To meet a demand, a slice needs to fulfill an end-to-end service that requires the joint allocation of different types of resources. A slice must be deployed in real-time, and thus, the corresponding provisioning of network, computing, and storage resources has to be done dynamically.

In 5G networks, traffic is highly dynamic, and network requests may be subject to frequent changes such as arrivals and departures. This dynamicity may fragment the slice resource usage and make the use of network resources less efficient. To counter this effect, network operators need to reconfigure the network slices regularly. Indeed, thanks to SDN and NFV, the routing of flows and the allocation of the network functions can be easily modified. Thus, the slice allocation can be adjusted to reduce resource utilization and minimize operational costs (e.g., software licenses, energy consumption, and Service Level Agreement (SLA) violations).

In this work, *we propose a method to efficiently reconfigure the network without severe interruptions of traffic* (as it is done in (Gausseran, Giroire, Jaumard, & Moulierac, 2020)). We focus on the case where 5G slices correspond to service chains. We use a *make-before-break* mechanism, in which we first allocate the resources for a secondary route while keeping the first one intact (i.e., two redundant routes are reserved in parallel). Then, we migrate the flow to the new route and release the resources of the old one. There is no traffic disruption, severely impacting the quality of service (QoS) of the slices. The computations of the new routes are done by using a *scalable optimization decomposition method making use of a column generation approach*. Even when using such a mechanism to avoid degrading the QoS, network operators do not want to reconfigure their network too frequently, which may lead to additional management costs. On the other hand, reconfiguring too rarely during the day may lead to sub-optimal network usage. A simple policy with low computational cost is to reconfigure regularly after a fixed number of minutes. However, reconfiguring in response to traffic variation can reduce the number of reconfigurations required each day without impacting the overall improvement obtained.

In this section, we propose a *reconfiguration management agent* that chooses when to initiate reconfiguration as a function of different parameters, such as the traffic dynamics and the network congestion level. We use a *Deep Reinforcement Learning technique* (DRL) by implementing it with Tensorflow (*TensorFlow: An end-to-end open source machine learning platform*, s. d.) and

their Deep Q-learning Network (DQN) agent. We then show that our agent improves the efficiency of reconfigurations by performing fewer reconfigurations while still minimizing the network operational costs compared to doing periodic and frequent reconfigurations.

The rest of this section is organized as follows. In Section 7.2.1, we discuss related work. Section 7.2.2 presents the formal definition of our problem, 7.2.3 the column generation model, and 7.2.4 the optimization model `Deep-REC` based on reinforcement learning. In Section 7.2.6, we validate `Deep-REC` by various numerical results.

7.2.1 Related Work

Routing and provisioning of slices. The VNF allocation and SFC provisioning problems have recently been widely studied.

Some works focus on static scenarios such as (Huin, Jaumard, & Giroire, 2018; Tomassilli, Giroire, Huin, & Pérennes, 2018) in which the authors develop efficient methods coupling chained allocation of VNFs and traffic routing within SFCs. The dynamic nature of network traffic raises a range of problems concerning the acceptance of incoming slices, resource management, and Service Level Agreement compliance. Cheng *et al.* (Cheng, Wu, Min, Zomaya, & Fang, 2020) use method to deploy and manage slice provisioning using deep learning and Lyapunov stability theories. In (Harutyunyan, Fedrizzi, Shahriar, Boutaba, & Riggio, 2019), the authors present a Mixed Integer Linear Program and a heuristic to add new slices by minimizing the bandwidth consumption and the slice provisioning cost while considering the VNF migrations. In (Sharma, Gumaste, & Tatipamula, 2020), a method is presented to manage the creation, modification, or deletion of slices by adapting to the traffic. They aim to minimize the number of slices while having enough bandwidth to serve the traffic.

Reconfiguration using standard techniques. The reconfiguration of SFCs and/or slices aims to maintain a near-optimal state of the network over time to optimize the network usage and the acceptance of demands. Wang *et al.* (Wang et al., 2019) develop an algorithm that manages two types of reconfiguration to maximize the operator's profits. First, a reconfiguration to adapt the slices to the current traffic. Second, a reconfiguration modifies the flows traversing the slices. The algorithm then schedules the reconfigurations and reserves resources for future traffic to reduce the number of potential future reconfigurations. Each reconfiguration includes the service interruption and resource usage as costs. In (Pozza et al., 2020), authors proposed a slice reconfiguration technique in which the new state of the network is pre-computed. The reconfiguration is done in several steps, during which the VNFs and routes are modified while considering capacities and delays. In (Gausseran, Tomassilli, Giroire, & Moulierac, 2021), authors proposed an integer linear program and a heuristic to efficiently reconfigure Service Function Chains using a *make-before-break* strategy. In (Gausseran, Giroire, Jaumard, & Moulierac, 2021), column generation techniques are used to optimize the reconfiguration of hundreds of network slices in only a few seconds.

Learning-based reconfiguration Some recent works use reconfiguration techniques based on reinforcement learning and try to predict the dynamicity of the network. Liu *et al.* (Liu, Feng, Chen, Qin, & Zhao, 2020) propose a VNF migration strategy based on Double-Deep Q-Network. They aim to equally place VNFs between Mobile Edge clusters and core clouds to avoid congestion at the Edge. The migration considers future traffic and tries to reduce the number of migrations while minimizing the number of congested links. In (Troia, Alvizu, & Maier, 2019), the authors use reinforcement learning to perform dynamic SFC resource allocation in optical networks. They

define an agent that decides for each SFC when and which reconfiguration to perform (migration, scaling-up, scaling-down) to meet the QoS criteria. Each SFC has a bound on the maximum number of authorized reconfigurations, and the optimization objective is determining the total number of reconfigurations. Each reconfiguration has a penalty for service interruption.

In (Wei et al., 2020), the authors use DRL to predict when to reconfigure to minimize the resources consumed. Unlike our work, the authors focus on intra-slice reconfigurations with a fixed number of requests throughout the experiment and with unchanging service sources and destinations. Their algorithm optimizes slices only locally and uses a pre-computed set of paths using a Depth-first search algorithm.

Guan *et al.* (Guan, Zhang, & Leung, 2020) use a Markov Renewal Process to predict changes in the resource occupancy of slices and reserve resources for slices that obtain higher revenues at lower cost. They use a deep dueling neural network combined with Q-Learning to choose whether to reconfigure each slice. Their goal is to maximize long-term revenue: the increased user utility minus the cost of resource utilization and service interruptions. Like the last two works mentioned, we establish an agent based on DRL to choose when to reconfigure the slices.

To the best of our knowledge, we are the first to propose a methodology for reconfiguring a dynamic network with incoming and outgoing slices without being dependent on a fixed number of slices throughout the day.

Our deep reconfiguration learning algorithm adapts its behavior based on the variations of the whole network traffic and not on a fixed set of flows within a slice. Moreover, we do not fix a limit on the reconfigurations per slice or day. The agent determines the best time to reconfigure and performs the needed number of reconfigurations depending on the traffic variations. The reconfiguration is computed independently using a column generation algorithm based on a *make-before-break* reconfiguration that chooses which slices to reconfigure. This allows our method to deal with a large number of slices, taking only a few seconds to compute the reconfiguration.

7.2.2 System Model and Problem Formulation

We consider the network a directed capacitated graph $G = (V, L)$ where V represents the node set and L is the link set. Using the resources available in this network, we must allocate a set of slice requests D. A network slice request $d \in D$ is modeled as a Service Function Chain (SFC) (as in (N. Zhang et al., 2017)) with a quintuplet: (i) the source v_{SRC}, (ii) the destination v_{DST}, (iii) the required bandwidth BW_d in traffic units, (iv) the delay requirement γ_d, and, (v) the ordered sequence of network functions c_d that need to be performed, where $f_i^{c_d}$ is the $i-th$ function of chain c_d. Each network function instance $f \in F$ has an installation cost c_f accounting for all the VNF usage costs (licenses, energy consumption, etc). Each slice $d \in D$ provides a revenue u per bandwidth unit.

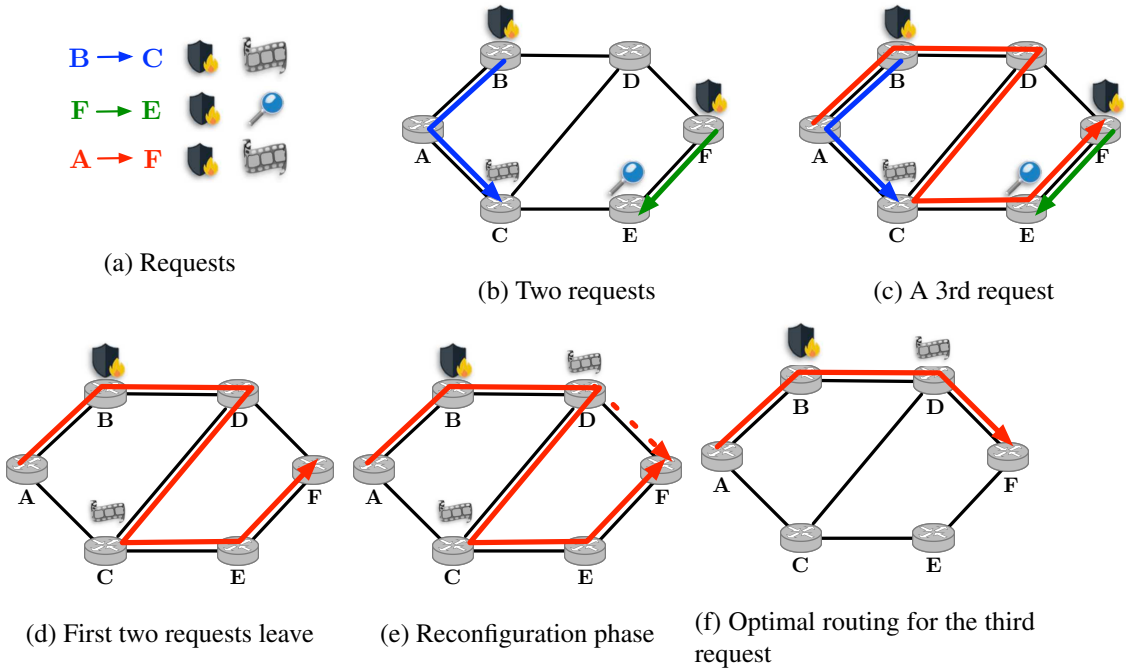
We aim to find an allocation of the slice requests such that the network operator profit accumulated over a certain time window is maximized. This cumulative profit is the sum of the instantaneous profits p_t at each observation time (i.e., minute). The profits p_t are computed as the difference between the overall revenue of the allocated slices and the overall cost of the deployed VNFs at time t:

$$p_t = \sum_{d \in D_t} u \cdot \text{BW}_d - \sum_{f \in F_t} c_f \tag{7.1}$$

where D_t and F_t are the set of slices and function instances allocated at observation time t, respectively.

In a dynamic scenario with no information on future traffic, the impact of recently arrived requests on the cumulative profit (at the operator time horizon) is still unknown: slices routed using long paths will consume too many resources, preventing the allocation of future requests. To consider that, at each time, we target placing new requests on paths so that resources (i.e., bandwidth at each link and network functions at each node) are minimized. Therefore, in the method detailed in Section 7.2.3, the slice revenue maximization (first term in Equation 7.1) is obtained by minimizing the resources used by the slices (see Equation 7.2).

Due to the lack of information about the future, the trivial mechanics of requests coming and leaving over time will bring the network to a global sub-optimal state. Hence, we are forced to reconfigure the network periodically. To do that, we use the *make-before-break* mechanism (Gausseran et al., 2020) that avoids network service disruption due to traffic rerouting. We detail the process of reconfiguring a request in the following example of Figure 7.1.



(a) Requests

(b) Two requests

(c) A 3rd request

(d) First two requests leave

(e) Reconfiguration phase

(f) Optimal routing for the third request

Figure 7.1 – An example of the reconfiguration of a request using a *make-before-break* approach with one step.

Example.

Two requests, B to C and F to E are routed during step (b). Four VNFs have been installed in B, C, E, and F to satisfy the needs of these requests. To avoid the usage cost of new VNFs, the route from A to F with minimum cost is a long 5–hops route, and the VNF already installed in node B is shared by the two slices (step (c)). When requests from B to C and from F to E leave, the request is routed on a non-optimal path (step (d)), which uses more resources than necessary. We compute one optimal 3-hop path and reroute the request on it (step (f)) with an intermediate *make-before-break* step (step (e)) in which both routes co-exist. During this intermediate step, traffic can follow both paths and resources are accordingly reserved. The old path is removed when all the allocation and provisioning are ready to be used. In doing so, no packet losses occur, and the traffic is not

interrupted. In this example, the reconfiguration can be done in only one step of reconfiguration, but we will consider the following up to 3 steps of reconfiguration.

7.2.3 Column Generation Optimization models

In this section, we first describe the main principles of the column generation algorithm to solve the problem of reconfiguration of network slices (first proposed in (Gausseran et al., 2021)), and then we show how we will use this for our deep reinforcement learning algorithm.

Column generation (CG) (Desaulniers, Desrosiers, & Solomon, 2005) is a model allowing to solve an optimization model without explicitly introducing all variables. It thus allows solving larger problem instances than a compact model with an exponential number of variables. In this work, the master problem (MP) seeks a possible global reconfiguration for all slices with a path formulation. This means that the problem decision variables (columns) correspond to paths on the layered graph. In the restricted master problem (RMP), only a subset of potential paths is used for each slice. At the initialization, the set of paths is the one used before reconfiguration. Each pricing problem (PP) then generates a new path for a request, together with the placement of the VNFs. During a reconfiguration, slices are migrated from one path to another. Note that, as the execution of each pricing problem is independent of the others, their solutions can be obtained in parallel.

7.2.3.1 Layered graph

In order to model the chaining constraint of demand, we associate to each demand d a layered graph $G^L(d)$ with $|c_d|$ layers where $|c_d|$ denotes the number of VNFs in the chain of the demand. Each layer is a duplicate of the original graph, and the capacities of both nodes and links are shared among layers.
A path on the layered graph starts at layer 0 and ends at layer c_d and corresponds to an assignment of both a path and the locations where functions are being run (the links between layers).

Representing the original graph as a layered graph is a *modeling trick* first proposed in (Dwaraki & Wolf, 2016). It allows for simplification of the problem by reducing it to a routing problem with shared capacities. This allows a drastic reduction of computation time compared to usual strategies using a large number of binary variables due to the ordering constraints of VNFs in the slice.

7.2.3.2 Master Problem

The Master Problem of the column generation algorithm is described in this section.
Parameters:
• δ_l^p is the number of times the link ℓ appears on path p (considering all the layers in the layered graph).
• $\theta_{i,u}^p$ = 1 if node u is used as a VNF on path p on layer i.
Variables:
• $\varphi_p^{d,t} \in [0,1]$ is the amount of flow of demand d on path p at time step t.
• $y_p^{d,t} \in [0,1]$ is the maximum amount of flow of demand d on path p between time step $t-1$ and t.
• $z_{u,f} \in [0,1]$, is equal to 1 if function f is activated on Node u at time step T in the final routing.
We assume an initial configuration is provided with fixed values for $\varphi_p^{d,0}$. The optimization model

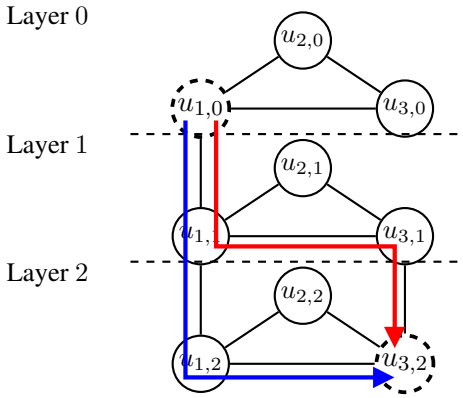


Figure 7.2 – The layered network $G^L(d)$ associated with a demand d such that $v_s = u_1$, $v_d = u_3$, and $c_d = f_1, f_2$, within a triangle network. f_1 is allowed be installed on u_1 and f_2 on u_1 and u_3. Source and destination nodes of $G^L(d)$ are $u_{1,0}$ and $u_{3,2}$. Two possible slices that satisfy d are drawn in red (f_1 is in u_1, f_2 in u_3) and blue (f_1 and f_2 are in u_1).

is written as follows.

Objective: minimize the amount of network resources consumed during the last reconfiguration time step T, which is the sum of the bandwidth used (BW) added to the sum of the costs of the deployed VNFs multiplied by a factor β.

$$\min \sum_{d \in D} \sum_{p \in P_d} \sum_{\ell \in E} \text{BW}_d \, \varphi_p^{d,T} \, \delta_\ell^p + \beta \sum_{u \in V^{\text{VNF}}} \sum_{f \in F} c_{u,f} \, z_{u,f} \tag{7.2}$$

Note that maximizing the accepted bandwidth in Equation 7.1 implies minimizing the link bandwidth used by the paths (first term in Equation 7.2); whereas the second term in Equation 7.2 represents the cost of the VNFs deployed at time t in Equation 7.1. Since the actual cost of the bandwidth and the VNFs can vary depending on the network operator, we considered a value of β for which the bandwidth and the VNFs have the same weight in the objective optimization.

Constraints:

One path constraint. For $d \in D$, time step $t \in \{0, ..., T\}$.

$$\sum_{p \in P_d} \varphi_p^{d,t} = 1 \tag{7.3}$$

Path usage over two consecutive time periods. For $d \in D$, $p \in P_d$, $t \in \{1, ..., T\}$.

$$\varphi_p^{d,t} \leq y_p^{d,t} \text{ and } \varphi_p^{d,t} \leq y_p^{d,t-1} \tag{7.4}$$

Make Before Break - Node capacity constraints. The capacity of a node u in V is shared between each layer and cannot exceed C_u considering the resources used over two consecutive time periods. Δ is the amount of computational units required by function $f \in F$ per unit of bandwidth processed. For $u \in V^{\text{VNF}}$, $t \in \{1, ..., T\}$.

$$\sum_{d \in D} \sum_{p \in P_d} \sum_{i=0}^{|c_d|-1} y_p^{d,t} \cdot \theta_{i,u}^p \cdot \text{BW}_d \cdot \Delta_{f_i^{c_d}} \leq C_u \tag{7.5}$$

Make Before Break - Link capacity constraints. The capacity of a link $\ell \in E$ is shared between each layer and cannot exceed C_ℓ considering the resources used over two consecutive time periods. For $\ell \in E$, $t \in \{1, ..., T\}$,

$$\sum_{d \in D} \sum_{p \in P_d} \text{BW}_d \, y_p^{d,t} \, \delta_\ell^p \leq C_\ell. \tag{7.6}$$

Function activation. To know which functions are activated on which nodes in the final routing. For $u \in V$, $f \in F$, $d \in D$, $i \in \{0, ..., |c_d| - 1\}$,

$$y_p^{d,T} \, \theta_{i,u}^p \leq z_{u,f_i^{c_d}}. \tag{7.7}$$

7.2.3.3 Pricing Problem

The pricing problem searches for a possible placement for the slice. Since a reconfiguration can be done in several steps, a pricing problem is launched for each demand, at each time step. The objective of the pricing problem for each demand d at time t is called the reduced cost and is expressed using the Equation in (Desaulniers et al., 2005).

Parameters:

• μ are the dual values of the master's constraints. The number written in upperscript is the reference of the master's constraints.

Variables:

• $\varphi_{\ell,i} \in \{0, 1\}$ is the amount of flow on link ℓ in layer i.

• $\alpha_{u,i} \in \{0, 1\}$ is the amount of flow on node u in layer i.

Objective: minimize the amount of network resources consumed for the demand d at time t.

$$\min \sum_{\ell \in E} \sum_{i=0}^{|c_d|} \varphi_{\ell,i} \, \text{BW}(1 + \mu_{l,t}^{(7.6)}) + \text{BW} \sum_{u \in V^{\text{VNF}}} \mu_{u,t}^{(7.5)} \sum_{i=0}^{|c_d|-1} \Delta_{f_i^{c_d}} \, \alpha_{u,i} - \mu_{d,t}^{(7.3)} + \beta \sum_{u \in V^{\text{VNF}}} \sum_{f \in F} c_{u,f} \, z_{u,f} \, \mu_{d,u,f}^{(7.7)} \tag{7.8}$$

where $\mu_{d,u,f}^{(7.7)} = 0$ when $t \neq T$, see constraints 7.7.

Constraints:

Flow conservation constraints for the demand d. For $u \in V^{\text{VNF}}$.

$$\sum_{\ell \in \omega^+(u)} \varphi_{\ell,0} - \sum_{\ell \in \omega^-(u)} \varphi_{\ell,0} + \alpha_{u,0} = \begin{cases} 1 \text{ if } u = v_s \\ 0 \text{ else} \end{cases} \tag{7.9}$$

$$\sum_{\ell \in \omega^+(u)} \varphi_{\ell,|c_d|} - \sum_{\ell \in \omega^-(u)} \varphi_{\ell,|c_d|} - \alpha_{u,|c_d|-1} = \begin{cases} -1 \text{ if } u = v_d \\ 0 \text{ else} \end{cases} \tag{7.10}$$

$$\sum_{\ell \in \omega^+(u)} \varphi_{\ell,i} - \sum_{\ell \in \omega^-(u)} \varphi_{\ell,i} + \alpha_{u,i-1} - \alpha_{u,i-1} = 0 \qquad 0 \leq i < |c_d| \tag{7.11}$$

Delay constraints. The sum of the link delays Γ_ℓ of the flow must not exceed the delay requirement of demand d.

$$\sum_{i=0}^{|c_d|} \varphi_{\ell,i} \, \Gamma_\ell \leq \gamma_d \tag{7.12}$$

Function activation. To know which functions are activated on which nodes. For $u \in V^{\text{VNF}}$, $f \in F$, layer $i \in \{0, ..., |c_d| - 1\}$

$$\alpha_{u,i} \leq z_{u,f_i^{c_d}} \tag{7.13}$$

Location constraints. A node may be enabled to run only a subset of the virtual network functions. For $u \in V^{\text{VNF}}$, $i \in \{0, ..., |c_d| - 1\}$, if the $(i+1)^{\text{th}}$ function of c_d cannot be installed on u, we have

$$\alpha_{u,i} = 0. \tag{7.14}$$

7.2.3.4 Motivation for `Deep-REC`

Our algorithm reconfigures a given set of network slices from an initial routing and placement of network functions to another solution that improves the usage of the network resources (both in terms of link bandwidth and VNFs). This reconfiguration is done with a make-before-break approach to avoid interruptions of the flows.

In a dynamic scenario, due to the frequent arrival and departure of slices, the network is regularly in a sub-optimal state. Nevertheless, the frequency to run this reconfiguration algorithm was found in an empirical manner. Indeed, works (Gausseran et al., 2020; Gausseran et al., 2021) showed that reconfiguring every 15 minutes allowed a good ratio between cost reduction, quality of reconfigurations, acceptance rate, and computation time.

A fixed frequency is easy to set up, but in practice, at some specific time of the day, reconfiguration may not be needed as traffic remains stable and the network is already in an optimal state. A new reconfiguration at this time won't bring any gain. On the opposite, during high-dynamic traffic periods, more frequent reconfigurations may be suitable to maintain an acceptable state of the network with efficient network resource usage. Therefore, a network operator might be interested in adapting the reconfiguration frequencies depending on the congestion of the network and the nature of the traffic. This is the main goal of this work.

Indeed, even if we use a *make-before-break* reconfiguration model, which allows us to reconfigure without disrupting the traffic, reconfiguring generates network management costs to migrate VNFs and to compute and instantiate the intermediate and the new paths (Noghani, Kassler, & Taheri, 2019).

We present in the new section our DRL model named `Deep-REC` to choose when to reconfigure in order to optimally adapt to the evolving network state. Our objective is to maximize the cumulative profit as presented before: the sum of the instantaneous profits p_t (See 7.1).

7.2.4 Deep Reinforcement Learning (DRL) Algorithm: `Deep-REC`

The reinforcement learning paradigm formalizes a discrete time stochastic control process (as our networking problem) where an *agent* interacts with an *environment* (in our case, the network). At each time step t, the *agent* interacts with its *environment* by (i) observing from the *environment* the current state s, (ii) accordingly, taking a decision (an *action*) a, (iii) receiving a reward $r(s, a)$, and, (iv) observing a new state s' (the network has transitioned from s to s'). The agent can repeat this process for a potentially infinite number of time steps, giving rise to a *trajectory*. The sum of the discounted rewards over a trajectory from time t, or *discounted return*, is calculated as:

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$

where $\gamma \leq 1$. The expectation of G_t over all possible trajectories initiated at a state s after taking an action a is the so-called *Q-value function* $Q(s, a)$. We aim to take, at each state s, the action a maximizing the *Q-value function*. Then, we need to estimate $Q(s, a)$.

In (C. J. Watkins & Dayan, 1992), authors proposed the *Q-learning* algorithm to learn $Q(s, a)$ from a sequence of agent interactions with the environment. Unfortunately, when the state and action spaces are huge, Q-learning needs a prohibitive computation time. To overcome that, Deep Q-learning Network (DQN) (Mnih et al., 2015) makes use of a deep neural network to approximate the *Q-value function* for high-dimensional state-space problems, as in our case. Finally, we also opt for DQN since, conversely to other reinforcement learning algorithms, it can learn efficiently from past experiences without introducing bias.

Description. For the implementation, we use the DQN agent from *tf_agents.agents.dqn.dqn_agent*, and the neural network from *tf_agents.networks.q_network* (*TensorFlow: An end-to-end open source machine learning platform*, s. d.). The network comprises a pre-processing layer from Keras (Chollet, s. d.) used for batch normalization and 3 layers of 64, 64, and 2 neurons, respectively. The batch size is 288, which is large enough to normalize properly. We use Adam optimizer with a learning rate of 1e-3, and we update the network every 16 states. The discount factor γ is set to 0.9, a value large enough to show the importance of future actions. We use an epsilon-greedy policy, where ϵ is set to 0.99 and decays to 0 in 200 instances. The replay buffer is 50 instances, and we train the agent on 250 instances.

Context. A 24-hour day consists of 1440 minutes. We decided to discretize it into 288 periods of 5 minutes to optimize the training of our agent. It is recalled that the objective is to maximize the profit while reconfiguring as efficiently as possible. The agent can potentially choose to reconfigure 288 times. To make the agent aware of the implicit trade-off between reconfiguring now or later, an artificial cost per reconfiguration v_R is introduced. Reconfiguring a network will never decrease the profit, but the agent has to learn when a reconfiguration is worth it.

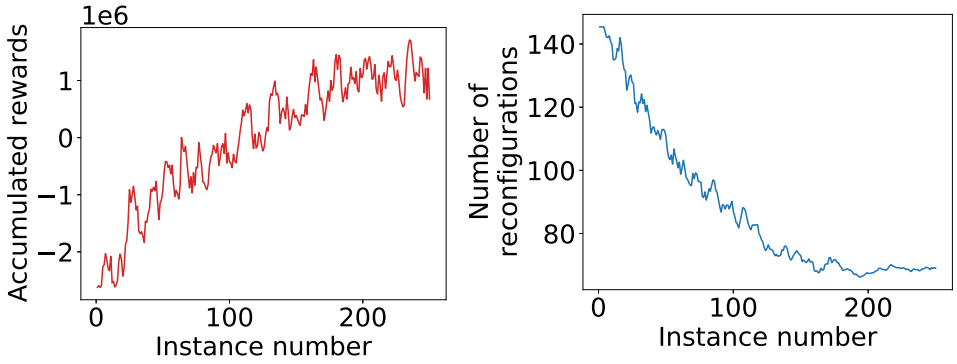
The agent will then learn the optimal number of reconfigurations to maximize the profit with this artificial cost. This cost can be real (management cost) or it can be fixed to get a given number of reconfigurations per day. The advantage of this technique compared to having a maximum number of reconfigurations allowed is that allows the agent to make more or less reconfigurations, adapting its behavior to the current period of the day.

State and Action Spaces. The network state can be described based on the next five quantities: (i) the number of minutes since the last reconfiguration ΔT, (ii) the number of slices added since the last reconfiguration λ, (iii) the number of slices released since the last reconfiguration μ, (iv) the current profit p_t 7.1 and, (v) the current time t. ΔT represent the current allocation oldness, λ and μ estimate the current network load.

The action space consists of two actions: to perform or not a reconfiguration at the current time based on the agent's decision. Thus, our agent decides when or not to reconfigure.

Reward Function. If the agent has chosen not to perform a reconfiguration, the reward is 0. Otherwise, the agent selects the reconfiguration, and two scenarios are possible:

1. *The reconfiguration was worth it.* A reconfiguration at time t is computed. To have a long-term vision, we simulate the network behavior (slices arrivals and departures) with the new network configuration for the next three time slots. This presents a good balance between accuracy and computational overhead (in training, we can simulate future requests). Finally,



(a) Accumulated rewards

(b) Number of reconfigurations

Figure 7.3 – Learning Curves

we estimate the accumulated profit gained with the reconfiguration as

$$\Delta p_R = \{\sum_{k=t}^{t+3} p_k | \text{reconf at } t\}$$

2. *The reconfiguration was not worth it.* The reward is estimated differently. We suppose that no reconfiguration was performed at t and we also simulate the network behavior (slices arrivals and departures) for the next three time slots. We estimate the accumulated profit gained without the reconfiguration as

$$\Delta p_{NR} = \{\sum_{k=t}^{t+3} p_k | \text{no reconf at } t\}$$

Finally, we compute the reward as follows:

$$r = \Delta p_R - \Delta p_{NR} - v_R.$$

We, therefore, have a positive reward when the profit increase Δp_R (if *reconfiguring* was the good decision) compensates both the profit increase Δp_{NR} (if *not reconfiguring* was the good decision) and the reconfiguration cost.

Training. We are now studying the efficiency of the learning of our agent trained on 250 instances. In Figure 7.3a, the agent's return on the training environment increases during the 250 trained instances, which implies that it learns to maximize the accumulated rewards on each instance.

We should not reconfigure too often during a day, so in Figure 7.3b, we study the variation of the number of reconfigurations made by the agent on each instance. The agent starts by reconfiguring randomly: 1 time out of 2 and thus about 144 times per instance, and it learns that it must reduce the number of reconfigurations to maximize the accumulated reward. When the agent has trained on around 200 instances, the epsilon reaches 0, and the number of reconfigurations converges to around 70 reconfigurations per instance.

7.2.5 Experimental settings

Topology. We conduct simulations on a real-world topology from SNDlib (Orlowski, Wessäly, Pióro, & Tomaszewski, 2010), `ta1` (24 nodes, 55 links), which includes 6 datacenters on which all VNFs can be instantiated. The cost of VNF c_f equals 2000 times the revenue u of a megabyte served.

Slice demands Each slice is composed of a chain of up to 5 VNFs, requires a specific amount of bandwidth and has latency constraints. We consider four different types of demands corresponding to four services: Video Streaming, Web Service, VoIP, and online gaming. The characteristics of each service are reported in Table 7.1 and have already been used by (Savi, Tornatore, & Verticale, 2015). The bandwidth usage was chosen according to the Internet traffic distribution described in ("Cisco Visual Networking Index: Forecast and Methodology, 2014–2019", 2015). The latency requirements are expressed in milliseconds and represent the maximum delay between the source and destination.

Each minute, 1 to 5 slice requests arrive (uniform random distribution), and slices that have reached the end of their life are removed from the network. By varying the lifetime of the slices, we can vary the maximum number of slices present at the same time on the network so that the load on the network follows a curve representing a real distribution of traffic measured on a dedicated network operator. We divided this traffic into five different periods, where D1 is a low-traffic period, and in D5, the network is highly congested. There are between 30 and 180 slices present at each moment on the network and in 24 hours, there are about 4320 arrivals of slices.

Reconfiguration Cost. To train `Deep-REC`, we define a fixed and artificial cost to the reconfiguration v_R. This cost can be adapted to reconfigure more or less. In our study, it is equal to the cost of deploying a VNF for 15 minutes, which means that a reconfiguration is considered useful if it allows to shut down a VNF for at least 15 minutes. To be usable in practice, a reconfiguration must be done quickly. Thanks to the column generation, we can reduce the computation time of each reconfiguration up to 30 times less (Gausseran et al., 2021) without affecting its efficiency.

Slice Types	VNF chain	Latency	bw (Mbps)
Web Service	NAT-FW-TM-WOC-IDPS	10ms	100
Video Streaming	NAT-FW-TM-VOC-IDPS	5ms	256
VoIP	NAT-FW-TM-FW-NAT	3.5ms	64
Online Gaming	NAT-FW-VOC-WOC-IDPS	2.5ms	50

Table 7.1 – Characteristics of network slices

7.2.6 Numerical Results

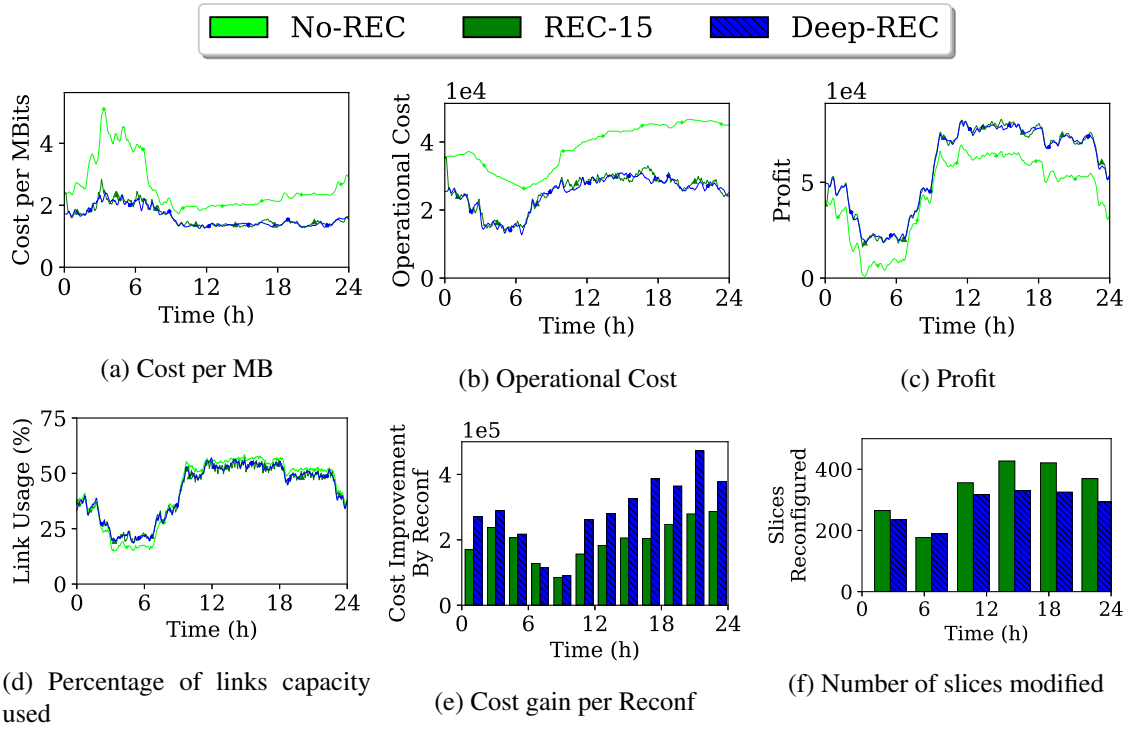
We compare the results obtained with three solutions in this section:

— `No-REC`: the slices are added in and removed from the network over time, and no reconfiguration is performed,

— `REC-15`: the reconfiguration is carried out every 15 minutes using a *make-before-break* strategy,

— `Deep-REC`: our deep-learning *make-before-break* reconfiguration proposal.

We first show that reconfiguring the network leads to significant profit gains. We then discuss the importance of selecting the best moments to carry out the reconfigurations, allowing them to perform fewer reconfigurations while achieving similar gains.



(a) Cost per MB

(b) Operational Cost

(c) Profit

(d) Percentage of links capacity used

(e) Cost gain per Reconf

(f) Number of slices modified

7.2.6.1 Improved network operational Cost

Figure 7.4a presents the network cost ((second term in Equation 7.2)) per megabyte of data sent over the network throughout the day. We observe that the costs achieved by `REC-15` and `Deep-REC` are very similar with a clear improvement compared to `No-REC`: `REC-15` allows an improvement of 36.82% when `Deep-REC` performs a little better with 38.05%. We also see that the cost is rather stable throughout the day, while with `No-REC`, the cost increases strongly during the low-congestion period (periods D1 and D2, between 2 am and 7 am). The global cost improvement through a day (not presented in this paper due to lack of space) of `REC-15` is 34.18% versus 35.55% with `Deep-REC`.

Figure 7.4b presents the network operational cost for the three strategies in terms of VNF costs (second term in Equation 7.2). This shows that both `Deep-REC` and `REC-15` reduce the network operational cost compared to `No-REC`. This justifies the necessity of reconfiguring. Moreover, the two reconfiguring strategies are comparable for this parameter.

7.2.6.2 Improved Profit and link utilization

Figure 7.4c shows the achieved profit, whose maximization is the objective of the reconfiguration: `Deep-REC` and `REC-15` have similar performance and improve the profit compared to `No-REC`. Indeed, the profit improvement of `REC-15` is 32.75% versus 32.53% for `Deep-REC`.

Finally, Figure 7.4d shows that even when minimizing VNF costs, reconfiguring the network does not lead to an increase of congestion: we observe a slightly higher utilization of links during periods D1-D2, but when the network is heavily loaded (periods D4-D5), there is a reduction of the congestion of the network.

As a conclusion, reconfiguring the network reduces congestion while reducing costs. Moreover, we validate with these results the performance of `Deep-REC` as it leads to similar profit as a periodic reconfiguration strategy such as `REC-15`. We show in the following that `Deep-REC`, by performing reconfigurations at the ideal moment, achieves this efficiency while reducing the total number of reconfigurations through a day compared to a regular and fixed reconfiguration strategy such as `REC-15`.

7.2.6.3 Number of reconfigurations

Figure 7.4e presents the cost improvement divided by the number of reconfigurations over periods of two hours. This shows that `Deep-REC` performs more efficient reconfigurations than `REC-15`. Each reconfiguration leads to a better improvement in terms of network costs.

Figure 7.4f presents the number of slices modified during the reconfiguration. Our algorithm `Deep-REC` modified approximately 20% less slices than `REC-15`, and thus, there is less impact on the network (less modifications, less computation).

Finally, Figure 7.5 shows the distribution of the number of reconfigurations during a day over two-hour periods. The green line on the figure represents `REC-15`, which does a constant number of reconfigurations, namely 8 (a reconfiguration every 15 minutes). In contrast, `Deep-REC` adapts its actions to the network load and does not carry out reconfigurations when they are not necessary, leading to a reduction of their number during the majority of the day, see the period between 10 am and 4 pm. Moreover, `Deep-REC` performs more reconfigurations than `REC-15` during the ascending phase (between D1 and D5) in order to react to the rapid change of the network and, thus, to maintain a good profit. With 96 reconfigurations during a day (against 73.2 on average for `Deep-REC`), `REC-15` has 31.15% more reconfigurations, for only 0.22% profit improvement.

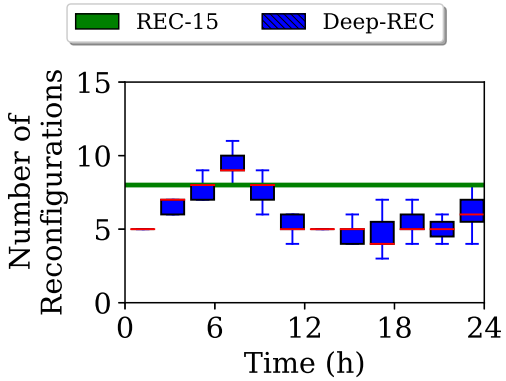


Figure 7.5 – Reconfiguration distribution for `Deep-REC` compared to `REC-15`.

7.3 Quantum virtual link generation via reinforcement learning

In the last years, the application of quantum physics principles to computer networks is gaining momentum among the research and industry communities, as shown by the first attempts of standardization of a so-called "Quantum Internet" (Kozlowski et al., 2023) (Wang, Rahman, Li, Aelmans, & Chakraborty, 2023) by the Internet Engineering Task Force (IETF). Amongst these principles, the *quantum entanglement* has been identified as a fundamental resource for Quantum Communication (Kozlowski et al., 2023), since it enables the Quantum Internet applications, as secure cryptographic key distribution and, distributed quantum computing (Wang et al., 2023). But, *quantum entanglement* is a probabilistic process strongly dependent on the features of the involved communication devices. Consequently, the entanglement management constitutes a stochastic control problem that can be formulated as a MDP (Khatri, 2021). In this preliminary work, we investigate the capacity of DRL to solve these problems, in particular, when a quantum entanglement is set up between two remote communication nodes not directly connected by a link. In the paragraphs below, we will introduce the required background.

Qubit and entanglement. In quantum communication and quantum computing, the counterpart of a classical bit is the *quantum bit (or qubit)*. But, whereas the classic bit can take either the “0" state or the “1" state, the *qubit* can be in a *superposition* of the two, with a certain probability to be at one of the states. The *qubit* exists in this superposition until its eventual measurement. Afterward, it will take the “0" or “1" value according to the corresponding probability. When two qubits are *entangled*, their states cannot be described separately: a state change, i.e., a qubit reading measurement, in one of them implicitly comes with a change in the other, regardless of the physical distance between them. Thus, the measurements at the two entangled qubits exhibit non-classical correlations used to design new applications that are not possible with classical communication, such as quantum key distribution or distributed quantum computation.

Quantum network. A set of nodes able to exchange qubits and distribute entangled states amongst themselves is defined as a *quantum network* in the RFC (Kozlowski et al., 2023). These *quantum nodes* are connected by *optical fiber* or *satellite laser* links. In this section, we assume fiber links. When an entanglement is set up between two qubits located at two adjacent quantum nodes connected by a direct link (e.g., between nodes A and B in Figure 7.6), the entanglement constitutes an *elementary quantum link* (Kozlowski et al., 2023). Its success probability P_e exponentially decreases with distance, which means that short-distance entanglements (like A-B, in Figure 7.6) are more likely to succeed than long-distance entanglements (like A-C, in Figure 7.6). To overcome this issue, we can create a *virtual link* (Kozlowski et al., 2023) over two elementary links via the so-called *entanglement swapping* (Kozlowski et al., 2023 ; Gyongyosi & Imre, 2022). This process allows the creation of long-distance entangled pairs by consuming the previously generated elementary links on the path between two further end-points. In Figure 7.6, the elementary links A-B and B-C are consumed to create a longer virtual link A-C. Quantum nodes (as B in Figure 7.6) that create long-distance entangled pairs via entanglement swapping are called quantum repeaters (Kozlowski et al., 2023), and they must store intermediate elementary links on the so-called *quantum memories* (Kozlowski et al., 2023) to be consumed later.

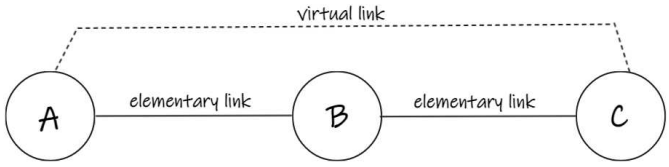


Figure 7.6 – *Elementary link vs virtual link.*

Quantum memory lifetimes. The probability that a qubit stored in a quantum memory is still, after a certain time, in its original state (e.g., an entangled state) decreases with time (Ortu, Holzäpfel, Etesse, & Afzelius, 2022). This probability is referred to as the memory efficiency η_m (Ortu et al., 2022), and its decay is known as decoherence. This process is the consequence of the progressive interactions of the quantum memory with the environment since memory cannot be perfectly isolated from it. The entanglement swapping success probability P_s depends on the memory efficiency η_m of the oldest loaded quantum memory involved in the swapping (Sangouard, Simon, de Riedmatten, & Gisin, s. d.).

This section, as far as we know, is one of the first works modeling a quantum virtual link generation process as a classical MDP and using a DRL algorithm to find an optimal generation policy tracking the age of the elementary links. *This supposes an innovative contribution with respect to the related works, where this age of the elementary is not used in the generation procedures.*

Related works are presented in Section 7.3.1. The MDP modeling of the virtual link generation along with the DRL approach used to solve it are described in Section 7.3.2. Numerical results and experiment settings are shown in Section 7.3.3.

7.3.1 Related Works

The idea that the management problem of *quantum elementary and virtual links* can be mathematically formalized as a quantum generalization of an MDP was developed in the Khatri's PhD dissertation (Khatri, 2021). This dissertation assumes the Quantum Decision Process (QDP) framework, (Barry, Barry, & Aaronson, s. d.), the quantum analog of a MDP, where states are quantum and actions are quantum operators, which implies the usage of a quantum computer. In contrast to this work, we model the decision problem as a classical, MDP with states described by measured physical properties and actions taken on a macroscopic level. We think that many of Khatri's ideas can be adapted to the current state-of-the-art without waiting for the development of a quantum computer. Then, in this section, we model as a classical MDP the *QDP with entanglement swapping* presented in Appendix D of (Khatri, 2021). This process aims to generate a *virtual link* from two *elementary links* via *entanglement swapping* as explained before. The virtual link generation has been recently studied in two contexts: (i) quantum repeaters chains (B. Li, Coopmans, & Elkouss, 2021) and (ii) quantum entanglement routing (C. Li, Li, Liu, & Cappellaro, 2021). In both of them, we aim to set up long-distance entanglements between non-adjacent communication nodes when the topologies are Daisy chains and mesh networks, respectively. In these works, the "history" of the links is ignored, i.e., the timestamps at which the elementary (and virtual) link generation processes succeed are not used. Besides, the virtual link generation always follows an *infinity memory cutoff-time policy* when, once the early elementary link is successfully set up, we keep it till the late also succeeds, regardless of the impact of larger decoherence of the oldest one onto the swapping probability P_s.

7.3.2 Reinforcement Learning For Virtual Link Generation

7.3.2.1 Problem Statement

As aforementioned, the management of a virtual link generation over two elementary links via entanglement swapping can be formulated as a MDP (Khatri, 2021). In this decision process, we aim to maximize the number of successful entanglement swaps per time unit, i.e., the virtual link generation rate. To generate the virtual link, two elementary links must have been successfully created before attempting the entanglement swapping with probability P_s. The older the first generated elementary link is, the more likely the swapping will fail. Then, after a cutoff time t_c, discarding the oldest elementary link and trying to generate it again (i.e., resetting it) becomes beneficial since links freshly reset have always a higher swapping probability. Unfortunately, this reset (a new elementary link generation) comes with a cost, as it is also a stochastic process with success probability P_e, which must be repeated till success, delaying the eventual swapping attempt. Thus, the cutoff time t_c of the elementary link has to be carefully selected to reduce the time between two successful swaps and, hence, maximize the virtual link generation rate.

Now, we describe the process as a MDP. At each time step t, a control agent applies a certain action a_t after observing the current state s_t. The execution of this action will trigger a transition into a new state s_{t+1} with a certain probability $p(s_t, a_t, s_{t+1})$. The agent receives a reward r_{t+1} based on the "quality" of the pair (s_t, a_t) to maximize a long-term objective. Then, the *agent* observes the new state s_{t+1} and repeats these steps. Assuming an initial state s_0, the MDP gives rise to a *trajectory*: $s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, , \ldots$. This *trajectory* is generated based on an agent *policy* $\pi(s, a)$ denoting the probability that action $a_t = a$ is taken at state $s_t = s$. In our case, the system state (action) consists of the concatenation of the states (actions) of the two elementary links and the virtual link. The state of each link is a vector $s = [x, m]$, where x is *1* if entanglement is active (*0*, otherwise); and, m is the entanglement age (*-1*, if entanglement is inactive). Two actions can be taken over each link: either the link is *re(set)*, i.e., a link generation is (re)tried; or the link *waits*, i.e., no link generation is (re)tried. The former provokes a stochastic transition with probability P_e (P_s) to a state $s = [1, 0]$ for an elementary (virtual) link generation. The latter leaves the current link state unchanged. Here, we assume a unique virtual link to be set up by swapping two elementary links, i.e., a space action of size $2^3 = 8$. The reward is simply *1* if the entanglement swapping succeeds; and *0*, otherwise.

Finally, we conclude this subsection by detailing the assumptions about the quantum environment model. In our work, entanglements are created in a "heralded" fashion, i.e., we know when the entanglement has been successfully established (therefore, cutoff times t_c can be measured). Once created, entanglements must be stored in quantum memories to eventually be swapped to create virtual links. Amongst the different methods of entanglement generation, the DLCZ-based protocols (Sangouard et al., s. d.) are an option satisfying these requirements. When DLCZ-based protocols are used the time becomes slotted with time slots L_0/v long, where L_0 is the length of the elementary link (fiber) and v is the light propagation speed at the fiber. This slot duration represents the time required to know if an elementary link generation has succeeded: the duration of the time step before observing a new state and taking a new action (retry or not a link generation). The elementary link generation success probabilities P_e exponentially decrease with fiber distance L_0 according to the work (Sangouard et al., s. d.). Whereas, the memory efficiency η_m, the main factor defining the swapping success probability P_s, falls with the storage time following the Mims' model described in (Ortu et al., 2022). The exact values of these probabilities depend

on the precise characterization of the optical fiber and quantum memories involved. We assume that we do not know them.

7.3.2.2 Reinforcement Learning based approach

If we do not possess models characterizing precisely the elementary link generation probability P_e and the memory efficiency η_m (and, thus, the entanglement swapping success probability P_s), the state transition probabilities $p(s_t, a_t, s_{t+1})$ are unknown. In this case, we can apply Deep Reinforcement Learning (DRL) (Bengio, 2009 ; Mnih et al., 2015) to find a policy maximizing the virtual link generation rate. In Reinforcement Learning (RL), we define a *Q-value function* $Q^{\pi}(s, a)$ as the *expected discounted return* from a given state-action pair (s, a) when following a policy π thereafter. The discounted return is the sum of the discounted rewards the agent receives over the future in a trajectory: $sum_{k=0}^{\infty}\gamma^k r_{t+k+1}$, with $\gamma \in [0, 1]$. Therefore, an agent solving a RL problem searches an optimal policy π^* maximizing this *Q-value function*. Here, this policy consists of finding the best cutoff time t_c for the oldest elementary link (see Section 7.3.2.1). When state transition probabilities are unknown (our case), this *Q-value function* can be estimated by using statistical learning with the Deep Q-Network (DQN) algorithm (Mnih et al., 2015). In the present work, the learning routine simply follows the classical DQN (Mnih et al., 2015) but is adapted to our virtual link generation problem.

7.3.3 Experiments

7.3.3.1 Experimental Settings

We use OpenAI™`Gym` (*Gym: Gym toolkit for creating reinforcement learning environments*, s. d.), one of the most popular toolkits to define RL environments in `Python`, to program a quantum environment modeling the MDP as described in Section 7.3.2.1. We consider the length of the fiber links L_0 and the light speed are 100 km and $200,000$ km/s, respectively, which yields a time step duration of $0,5$ ms. We assume fiber losses of 0.2 dB/km achievable around $1,550$ nm. We use the Mims' equation corresponding to the DD sequence 'XX' in study (Ortu et al., 2022), but considering a zero-time efficiency of 1.

The agent is implemented in `TensorFlow` (*TensorFlow: An end-to-end open source machine learning platform*, s. d.) as a three-layer neural network with two dense layers of 32 neurons, each having a `tanh` activation function, and an output layer without activation with as many neurons as actions (8, here since we consider two elementary links and one virtual link). The neural network is trained using the DRL algorithm called DQN (Mnih et al., 2015) provided by the OpenAI™ `Baselines` library (*OpenAI Baselines: high-quality implementations of reinforcement learning algorithms*, s. d.).

Figure 7.7a depicts the episode reward evolution along the training time. An episode reward is the sum of the rewards r produced during all the steps in a training episode. We observe that the average episode reward increases with time, stabilizing after 600 episodes, where an episode is $10,000$ steps long.

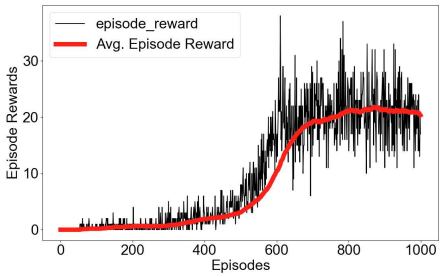
7.3.3.2 Numerical Results

In Figure 7.7b, we compare the policy learned by the DRL agent with two benchmarks:

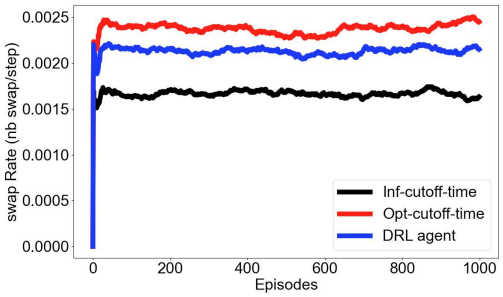
— ***Inf-cutoff-time* policy**: the cutoff time of the oldest link is set to infinity, then, when the first elementary link to succeed is set up, we keep it till the second one also succeeds regardless of the decoherence level of the first one. This is the by-default policy considered by the State-of-the-Art (B. Li et al., 2021 ; C. Li et al., 2021). It represents a *lower bound* on the optimal policy.

— ***Opt-cutoff-time* policy**: the optimal cutoff time of the oldest link is found by *brute force*. This process cannot scale up with larger problem instances. It trivially represents an *upper bound* on the policy learned by the DRL agent.

We test the policies by simulating the MDP process 1000 times. An *episode* is a simulation instance. Each episode again consists of $10,000$ steps. We observe that the DRL agent outperforms the *Inf-cutoff-time* policy and is close to the *opt-cutoff-time* policy. The found cutoff times are 146.0 steps and 108 steps for the DRL policy and *opt-cutoff-time* policy, respectively.



(a) *Episode reward evolution during training*



(b) *Swap rate of DRL agent vs. benchmarks during the test*

Figure 7.7 – DRL evaluation during training and test

7.4 Conclusion

In conclusion, this chapter has introduced innovative applications of Deep Reinforcement Learning in addressing critical challenges within network optimization, spanning both 5G network slicing and quantum network entanglement management.

We have showcased `Deep-REC`, a novel DRL framework tailored for efficient network slice reconfiguration. Through adaptive decision-making, `Deep-REC` optimizes reconfiguration timings, minimizing network congestion while maximizing operator profit. Our results demonstrate significant reductions in the number of reconfigurations required, without compromising network performance or operational costs.

Moreover, our exploration into utilizing DRL for learning entanglement management policies in quantum networks has yielded promising initial results. By surpassing existing State-of-the-Art *Inf-cutoff-time policy*, we have demonstrated the potential of DRL in maximizing virtual link generation rates, especially in scenarios where a model of the entanglement success probabilities is not previously known.

In the next chapter, we provide a closing discussion and present future directions to conclude this Ph.D. thesis manuscript.

CHAPTER 8

Conclusion

This Ph.D. thesis focuses on studying and developing Deep Reinforcement Learning (DRL) solutions, especially Multi-Agent Deep Reinforcement Learning (MA-DRL) to cloud network control. Throughout this thesis, we propose innovative improvements in performance, communication overhead, and stability. We fulfilled the goals of the thesis and showcased promising results that constitute an important milestone towards adopting those models in production.

As follows, we summarize the main contributions:

— First, realistic network platforms for experimenting MA-DRL solutions in the general non-overlay and cloud overlay cases (`PRISMA` and **prisma-v2** respectively) were developed and introduced in Chapter 4. `PRISMA` fills a crucial gap in the field by providing the first dedicated framework for developing and testing MA-DRL algorithms for the Distributed Packet Routing (DPR) problem in general non-overlay scenarios. Before its development, researchers had to rely on their own simplified Reinforcement Learning (RL) simulation environments, hindering reproducibility and realism. `PRISMA` overcomes these challenges by realistically modeling the communication process, explicitly considering the distributed nature of nodes, and providing a modular code design with real-time training visualization interfaces, facilitating the development and comparison of MA-DRL proposals. **prisma-v2** expands upon the capabilities of `PRISMA` by incorporating features such as the simulation of cloud overlay network topologies and the ability to simulate control packets, enabling a more comprehensive evaluation of network protocol overhead. Furthermore, **prisma-v2** integrates all modules with the core (`ns-3`) into a docker container, ensuring ease of deployment across different platforms. By providing a realistic simulation environment, **prisma-v2** serves as a valuable playground for testing and evaluating new network protocols based on the MA-DRL framework in the context of overlay networks.

— Next, we point out the overhead problem due to the agents' communication in the general non-overlay scenario. In this regard, the first part of Chapter 5 was devoted to the experimental investigation of control signaling impact for distributed learning-based packet routing, facilitated by the `PRISMA` simulator. We illustrated two types of signalling mechanisms adopted by the literature: *model sharing* and *value sharing*. Our results show the existence of a tradeoff between the communication overhead and the performance, and thus, communication overhead should be taken into account when implementing MA-DRL solution for the DPR problem. Motivated by these findings, the second part of the chapter proposed two novel improvements in order to reduce this communication overhead. The first one is a new signalling mechanism, called *logit sharing*, which substantially reduces the communication overhead while maintaining performance comparable to *model sharing*. The second is a dynamic control packet sending between agents, optimizing network efficiency by minimizing unnecessary control packet exchange.

— Chapter 6 adapted MA-DRL to cloud overlay networks. We proposed a novel framework, called *O-DQR*, suited for this case. O-DQR addresses three fundamental aspects for deploying such a solution. First, the performance of the network (delay, loss rate), where the framework can achieve near-optimal performance. Second, the control overhead is reduced by enabling the agents to send control packets only when needed dynamically and adopting the *logit sharing* strategy. Third, the training convergence and stability are improved thanks to a guided reward mechanism that dynamically learns the penalty applied for a packet loss.

— Finally, in Chapter 7, we studied applications of DRL to network control, addressing two challenging scenarios. Firstly, we tackled the problem of deciding the optimal moment for cloud network slice reconfiguration, considering the tradeoff between mitigating network congestion and minimizing additional costs incurred by reconfiguration. Our proposed approach successfully identifies optimal reconfiguration timings, maximizing network operator profit while minimizing resource utilization and network congestion. Moreover, by selecting the best moment for reconfiguration, our method reduces the number of required reconfigurations compared to periodic approaches, thereby enhancing overall network efficiency. Subsequently, we shifted our focus to the novel quantum networks, namely on an optimal entanglement management policy using DRL. The proposed method surpassed existing methodologies, particularly in scenarios where precise models of the quantum devices are unavailable, thereby enhancing the performance and reliability of quantum networks.

As a future extension to the proposed framework, *O-DQR*, which only considers end-to-end delay and packet loss, other Quality of Service (QoS) metrics can be regarded. The thesis considers the global optimization of network congestion by assuming all data packets are treated identically. Future improvements to our framework could include classes of service and adjust routing depending on the packet type (some packets require low latency, whereas others require low jitter). Also, packets could have different priorities. Finally, a notion of "intent" could be integrated into the model. It describes how an operator interacts with the agents and changes their behavior. This started to be explored by tuning the model's hyperparameters, like the relevancy threshold, used to balance between performance and communication overhead.

Furthermore, many other aspects could be explored, such as the security of the communication between the agents. Besides adding an additional security layer to the control packet in order to avoid sniffing information, it is important to include a data integrity verification process on each node to make it robust. Since the system is fully decentralized, the agents are vulnerable to security attacks like:

— Byzantine attacks (Young, Kate, Goldberg, & Karsten, 2012): when a node behaves in an unexpected or malicious way, such as sending incorrect information to other nodes.

— Sybil attacks (Mohaisen & Kim, 2013): caused by the effect of churn (Trifa & Khemakhem, 2014) when nodes join and leave the networks because of power or link failures.

A *zero trust* based approach could be adopted where the agents verify the incoming data before adding it to their experience replay buffers and databases.

Finally, we might explore the integration of the framework into Software-Defined Networking (SDN) environment. In this context, many challenges arise, such as how to exploit the existing statistics collected by the SDN node and how to handle heterogeneous devices, where some routers are legacy or non-IA SDN controllers. The framework should be able to communicate with such devices and exploit the information they give, as well as how to handle the exploration in the continual learning settings.

References

Altman, E. (2021). *Constrained markov decision processes*. Routledge.

Andersen, D., Balakrishnan, H., Kaashoek, F., & Morris, R. (2001). Resilient overlay networks. In *Proc. of the 8th acm symposium on operating systems principles* (pp. 131–145).

Baker, B., Kanitscheider, I., Markov, T., Wu, Y., Powell, G., McGrew, B., & Mordatch, I. (2019). Emergent tool use from multi-agent autocurricula. *arXiv preprint arXiv:1909.07528*.

Barry, J., Barry, D. T., & Aaronson, S. (s. d.). Quantum partially observable markov decision processes. *Physical Review A*, *90*(3), 032311. Consulté le 2023-04-19, sur https://link.aps.org/doi/10.1103/PhysRevA.90.032311 (Publisher: American Physical Society) doi: 10.1103/PhysRevA.90.032311

Barzegar, S., Ruiz, M., & Velasco, L. (2023). Autonomous flow routing for near real-time quality of service assurance. *IEEE Trans. on NSM*.

Bates, T., Chandra, R., & Chen, E. (2000). *Bgp route reflection-an alternative to full mesh ibgp* (Rapport technique).

Bellemare, M. G., Naddaf, Y., Veness, J., & Bowling, M. (2013). The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, *47*, 253–279.

Bellman, R. (1957). A markovian decision process. *Journal of mathematics and mechanics*, 679–684.

Bellman, R. (1958). On a routing problem. *Quarterly of applied mathematics*, *16*(1), 87–90.

Belzarena, P., Sena, G. G., Amigo, I., & Vaton, S. (2016). Sdn-based overlay networks for qos-aware routing. In *Proceedings of the 2016 workshop on fostering latin-american research in data communication networks* (pp. 19–21).

Bengio, Y. (2009). *Learning deep architectures for ai*. Now Publishers.

Benzekki, K., El Fergougui, A., & Elbelrhiti Elalaoui, A. (2016). Software-defined networking (sdn): a survey. *Security and communication networks*, *9*(18), 5803–5833.

Bertsekas, D. (2012). *Dynamic programming and optimal control: Volume i* (Vol. 4). Athena scientific.

Bhatnagar, S., & Lakshmanan, K. (2012). An online actor–critic algorithm with function approximation for constrained markov decision processes. *Journal of Optimization Theory and Applications*, *153*, 688–708.

Bono, G., Dibangoye, J. S., Matignon, L., Pereyron, F., & Simonin, O. (2019). Cooperative multi-agent policy gradient. In *Machine learning and knowledge discovery in databases: European conference, ecml pkdd 2018, dublin, ireland, september 10–14, 2018, proceedings, part i 18* (pp. 459–476).

Botta, A., Canonico, R., Navarro, A., Stanco, G., & Ventre, G. (2023). Scalable reinforcement learning for dynamic overlay selection in sd-wans. In *2023 ifip networking* (pp. 1–9).

Boyan, J., & Littman, M. (1993). Packet routing in dynamically changing networks: A reinforcement learning approach. *Advances in neural information processing systems*, *6*.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., & Zaremba, W. (2016, juin). OpenAI Gym. *arXiv:1606.01540 [cs]*. (arXiv: 1606.01540)

Busoniu, L., Babuska, R., & De Schutter, B. (2008). A comprehensive survey of multiagent reinforcement learning. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, *38*(2), 156–172.

Censor, Y. (1977). Pareto optimality in multiobjective problems. *Applied Mathematics and Optimization*, *4*(1), 41–59.

Chang, Y.-H., Ho, T., & Kaelbling, L. (2003). All learning is local: Multi-agent learning in global reward games. *Advances in neural information processing systems*, *16*.

Chen, L., Hu, B., Guan, Z.-H., Zhao, L., & Shen, X. (2021). Multiagent meta-reinforcement learning for adaptive multipath routing optimization. *IEEE Transactions on Neural Networks and Learning Systems*, 1–13. doi: 10.1109/TNNLS.2021.3070584

Chen, X., Li, B., Proietti, R., Liu, C.-Y., Zhu, Z., & Yoo, S. B. (2019). Demonstration of distributed collaborative learning with end-to-end qot estimation in multi-domain elastic optical networks. *Optics express*, *27*(24), 35700–35709.

Cheng, X., Wu, Y., Min, G., Zomaya, A. Y., & Fang, X. (2020). Safeguard network slicing in 5g: A learning augmented optimization approach. *IEEE JSAC*. doi: 10.1109/JSAC.2020.2999696

Chu, T., Wang, J., Codecà, L., & Li, Z. (2019). Multi-agent deep reinforcement learning for large-scale traffic signal control. *IEEE Transactions on Intelligent Transportation Systems*, *21*(3), 1086–1095.

Cisco visual networking index: Forecast and methodology, 2014–2019 [Manuel de logiciel]. (2015).

Claus, C., & Boutilier, C. (1998). The dynamics of reinforcement learning in cooperative multiagent systems. *AAAI/IAAI*, *1998*(746-752), 2.

Das, A., Gervet, T., Romoff, J., Batra, D., Parikh, D., Rabbat, M., & Pineau, J. (2019). Tarmac: Targeted multi-agent communication. In *International conference on machine learning* (pp. 1538–1546).

Desaulniers, G., Desrosiers, J., & Solomon, M. M. (Eds.). (2005). *Column Generation* (N° 978-0-387-25486-9). Springer. Consulté sur https://ideas.repec.org/b/spr/sprbok/978-0-387-25486-9.html doi: 10.1007/b135457

Doan, T. V., Bajpai, V., & Crawford, S. (2020). A longitudinal view of netflix: Content delivery over ipv6 and content cache deployments. In *Ieee infocom 2020* (pp. 1073–1082).

Dwaraki, A., & Wolf, T. (2016). Adaptive service-chain routing for virtual network functions in software-defined networks. In *Proceedings of the 2016 workshop on hot topics in middleboxes and network function virtualization* (pp. 32–37).

Farquhar, C., Kafle, S., Hamedani, K., Jagannath, A., & Jagannath, J. (2023). Marconi-rosenblatt framework for intelligent networks (mr-inet gym): For rapid design and implementation of distributed multi-agent reinforcement learning solutions for wireless networks. *Computer Networks*, *222*, 109489.

Finn, C., & Levine, S. (2017). Meta-learning and universality: Deep representations and gradient descent can approximate any learning algorithm. *arXiv preprint arXiv:1710.11622*.

Foerster, J., Assael, I. A., De Freitas, N., & Whiteson, S. (2016). Learning to communicate with deep multi-agent reinforcement learning. *Advances in neural information processing systems*, *29*.

Foerster, J., Nardelli, N., Farquhar, G., Afouras, T., Torr, P. H. S., Kohli, P., & Whiteson, S. (2017, 06–11 Aug). Stabilising experience replay for deep multi-agent reinforcement learning. In D. Precup & Y. W. Teh (Eds.), *Proc. intl. conf. on machine learning (icml)* (Vol. 70, pp. 1146–1155). PMLR.

Fulda, N., & Ventura, D. (2007). Predicting and preventing coordination problems in cooperative q-learning systems. In *Ijcai* (Vol. 2007, pp. 780–785).

Gausseran, A. (2021). *Algorithmes d'optimisation pour le network slicing pour la 5g* (Thèse de doctorat non publiée). Université Côte d'Azur.

Gausseran, A., Giroire, F., Jaumard, B., & Moulierac, J. (2020). Be scalable and rescue my slices during reconfiguration. In *Ieee icc.* doi: 10.1109/ICC40277.2020.9148871

Gausseran, A., Giroire, F., Jaumard, B., & Moulierac, J. (2021, 08). Be Scalable and Rescue My Slices During Reconfiguration. *The Computer Journal*, *64*(10), 1584-1599. Consulté sur https://doi.org/10.1093/comjnl/bxab108 doi: 10.1093/comjnl/bxab108

Gausseran, A., Tomassilli, A., Giroire, F., & Moulierac, J. (2021). Don't interrupt me when you reconfigure my service function chains. *Computer Communications*.

Gawłowicz, P., & Zubow, A. (2019, November). ns-3 meets OpenAI Gym: The Playground for Machine Learning in Networking Research. In *Proc. ACM International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWiM).*

Gronauer, S., & Diepold, K. (2022). Multi-agent deep reinforcement learning: a survey. *Artificial Intelligence Review*, 1–49.

Guan, W., Zhang, H., & Leung, V. C. M. (2020). Slice reconfiguration based on demand prediction with dueling deep reinforcement learning. In *Ieee globecom.* doi: 10.1109/GLOBECOM42002.2020.9322180

Gyongyosi, L., & Imre, S. (2022). Advances in the quantum internet. *Communications of the ACM*, *65*(8), 52–63. doi: 10.1145/3524455

Harutyunyan, D., Fedrizzi, R., Shahriar, N., Boutaba, R., & Riggio, R. (2019). Orchestrating end-to-end slices in 5g networks. In *15th international conference on network and service management (cnsm).* doi: 10.23919/CNSM46954.2019.9012732

Hasan, M. Z., Al-Rizzo, H., & Al-Turjman, F. (2017). A survey on multipath routing protocols for qos assurances in real-time wireless multimedia sensor networks. *IEEE Communications Surveys & Tutorials*, *19*(3), 1424–1456.

Houidi, O., Bakri, S., Zeghlache, D., Lesca, J., Quang, P. T. A., Leguay, J., & Medagliani, P. (2023). Amac: Attention-based multi-agent cooperation for smart load balancing. In *2023 ieee/ifip network operations and management symposium (noms 2023).*

Houidi, O., Zeghlache, D., Perrier, V., Quang, P. T. A., Huin, N., Leguay, J., & Medagliani, P. (2022). Constrained deep reinforcement learning for smart load balancing. In *2022 ieee 19th annual ccnc* (pp. 207–215).

Huin, N., Jaumard, B., & Giroire, F. (2018). Optimal network service chain provisioning. *IEEE/ACM Transactions on Networking*.

Jiang, J., Dun, C., Huang, T., & Lu, Z. (2018). Graph convolutional reinforcement learning. *arXiv preprint arXiv:1810.09202*.

Jiang, J., & Lu, Z. (2018). Learning attentional communication for multi-agent cooperation. *Advances in neural information processing systems*, *31*.

Jones, N. M., Paschos, G. S., Shrader, B., & Modiano, E. (2014). An overlay architecture for throughput optimal multipath routing. In *Proc. of the 15th acm int. symp. on mobihoc* (pp. 73–82).

Kamri, A. Y., Quang, P. T. A., Huin, N., & Leguay, J. (2021). Constrained policy optimization for load balancing. In *2021 17th drcn conf.* (pp. 1–6).

Kaviani, S., Ryu, B., Ahmed, E., Larson, K., Le, A., Yahja, A., & Kim, J. H. (2021). Deepcq+: Robust and scalable routing with multi-agent deep reinforcement learning for highly dynamic networks. In *Milcom 2021-2021 ieee milcom* (pp. 31–36).

Khatri, S. (2021). *Towards a general framework for practical quantum network protocols* (Thèse de doctorat non publiée). Louisiana State University and Agricultural & Mechanical College.

Kim, G., Kim, Y., & Lim, H. (2022). Deep reinforcement learning-based routing on software-defined networks. *IEEE Access*, *10*, 18121–18133.

Kozlowski, W., Wehner, S., Meter, R. V., Rijsman, B., Cacciapuoti, A. S., Caleffi, M., & Nagayama, S. (2023, mars). *Architectural Principles for a Quantum Internet* (N° 9340). RFC 9340. RFC Editor. Consulté sur https://www.rfc-editor.org/info/rfc9340 doi: 10.17487/RFC9340

Lanctot, M., Zambaldi, V., Gruslys, A., Lazaridou, A., Tuyls, K., Pérolat, J., ... Graepel, T. (2017). A unified game-theoretic approach to multiagent reinforcement learning. *Advances in neural information processing systems*, *30*.

Lauer, M., & Riedmiller, M. A. (2000). An algorithm for distributed reinforcement learning in cooperative multi-agent systems. In *Proceedings of the seventeenth international conference on machine learning* (pp. 535–542).

Leibo, J. Z., Zambaldi, V., Lanctot, M., Marecki, J., & Graepel, T. (2017). Multi-agent reinforcement learning in sequential social dilemmas. *arXiv preprint arXiv:1702.03037*.

Leyton-Brown, K., & Shoham, Y. (2022). *Essentials of game theory: A concise multidisciplinary introduction*. Springer Nature.

Li, B., Coopmans, T., & Elkouss, D. (2021). Efficient optimization of cutoffs in quantum repeater chains. *IEEE Transactions on Quantum Engineering*, *2*, 1–15.

Li, C., Li, T., Liu, Y.-X., & Cappellaro, P. (2021). Effective routing design for remote entanglement generation on quantum networks. *npj Quantum Information*, *7*(1), 1–12. Consulté le 2021-09-16, sur https://www.nature.com/articles/s41534-020-00344-4 doi: 10.1038/s41534-020-00344-4

Littman, M. L. (1994). Markov games as a framework for multi-agent reinforcement learning. In *Proc. intl. conf. on machine learning (icml)* (p. 157-163).

Liu, Y., Feng, G., Chen, Z., Qin, S., & Zhao, G. (2020). Network function migration in softwarization based networks with mobile edge computing. In *Ieee icc.* doi: 10.1109/ICC40277.2020.9148714

Lowe, R., Wu, Y. I., Tamar, A., Harb, J., Pieter Abbeel, O., & Mordatch, I. (2017). Multi-agent actor-critic for mixed cooperative-competitive environments. *Advances in neural information processing systems*, *30*.

Lu, L., Shin, Y., Su, Y., & Karniadakis, G. E. (2019). Dying relu and initialization: Theory and numerical examples. *arXiv preprint arXiv:1903.06733*.

Lumezanu, C., Baden, R., Spring, N., & Bhattacharjee, B. (2009). Triangle inequality and routing policy violations in the internet. In *Passive and active network measurement: 10th international conference, pam 2009, seoul, korea, april 1-3, 2009. proceedings 10* (pp. 45–54).

Malkin, G. (1998). *Rfc2453: Rip version 2*. RFC Editor.

Manfredi, V., Wolfe, A. P., Wang, B., & Zhang, X. (2021). Relational deep reinforcement learning for routing in wireless networks. In *2021 IEEE 22nd international symposium on a world of wireless, mobile and multimedia networks (WoWMoM)* (pp. 159–168). doi: 10.1109/WoWMoM51794.2021.00029

Mannor, S., & Shimkin, N. (2004). A geometric approach to multi-criterion reinforcement learning. *The Journal of Machine Learning Research*, *5*, 325–360.

Masoudi, R., & Ghaffari, A. (2016). Software defined networks: A survey. *Journal of Network and computer Applications*, *67*, 1–25.

Matignon, L., Laurent, G. J., & Le Fort-Piat, N. (2012). Independent reinforcement learners in cooperative markov games: a survey regarding coordination problems. *The Knowledge Engineering Review*, *27*(1), 1–31.

Mestres, A., Rodriguez-Natal, A., Carner, J., Barlet-Ros, P., Alarcón, E., Solé, M., ... Cabellos, A. (2017, sep). Knowledge-defined networking. , *47*(3), 2–10. doi: 10.1145/3138808.3138810

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... Hassabis, D. (2015, février). Human-level control through deep reinforcement learning. *Nature*, *518*(7540), 529–533. doi: 10.1038/nature14236

Mohaisen, A., & Kim, J. (2013). The sybil attacks and defenses: a survey. *arXiv preprint arXiv:1312.6349*.

Moreschini, S., Pecorelli, F., Li, X., Naz, S., Hästbacka, D., & Taibi, D. (2022). Cloud continuum: The definition. *IEEE Access*, *10*, 131876–131886.

Mukhutdinov, D., Filchenkov, A., Shalyto, A., & Vyatkin, V. (2019). Multi-agent deep learning for simultaneous optimization for time and energy in distributed routing system. *Future Generation Computer Systems*, *94*, 587-600. doi: https://doi.org/10.1016/j.future.2018.12.037

Noghani, K. A., Kassler, A., & Taheri, J. (2019). On the cost-optimality trade-off for service function chain reconfiguration. In *Ieee cloudnet.*

Oliehoek, F. A., Spaan, M. T., & Vlassis, N. (2008). Optimal and approximate q-value functions for decentralized pomdps. *Journal of Artificial Intelligence Research*, *32*, 289–353.

Orlowski, S., Wessäly, R., Pióro, M., & Tomaszewski, A. (2010). SNDlib 1.0–survivable network design library. *Wiley Networks*.

Ortu, A., Holzäpfel, A., Etesse, J., & Afzelius, M. (2022). Storage of photonic time-bin qubits for up to 20 ms in a rare-earth doped crystal. *npj Quantum Information*, *8*(1), 1–7. doi: 10.1038/s41534-022-00541-3

Peterson, L., & Davie, B. (2012). *Computer networks: A systems approach*. Elsevier.

Pineau, J., Vincent-Lamarre, P., Sinha, K., Lariviere, V., Beygelzimer, A., d'Alche Buc, F., ... Larochelle, H. (2021). Improving reproducibility in machine learning research(a report from the NeurIPS 2019 reproducibility program). *Journal of Machine Learning Research*, *22*(164), 1–20.

Pozza, M., Nicholson, P. K., Lugones, D. F., Rao, A., Flinck, H., & Tarkoma, S. (2020). On reconfiguring 5g network slices. *IEEE JSAC*. doi: 10.1109/JSAC.2020.2986898

Quang, P. T. A., Gong, X., Liu, C., Li, K., Leguay, J., Zhang, X., ... Ye, K. (2022). Routing and qos policy optimization in sd-wan. *arXiv preprint arXiv:2209.12515*.

Rabinowitz, N., Perbet, F., Song, F., Zhang, C., Eslami, S. A., & Botvinick, M. (2018). Machine theory of mind. In *International conference on machine learning* (pp. 4218–4227).

Rekhter, Y., Li, T., & Hares, S. (2006). *Rfc 4271: A border gateway protocol 4 (bgp-4).* RFC Editor.

Ritchie, H., Mathieu, E., Roser, M., & Ortiz-Ospina, E. (2023). Internet. *Our World in Data*. (https://ourworldindata.org/internet)

Sangouard, N., Simon, C., de Riedmatten, H., & Gisin, N. (s. d.). Quantum repeaters based on atomic ensembles and linear optics. *Reviews of Modern Physics*, *83*(1), 33–80. Consulté le 2022-04-14, sur https://link.aps.org/doi/10.1103/RevModPhys.83.33 (Publisher: American Physical Society) doi: 10.1103/RevModPhys.83.33

Savi, M., Tornatore, M., & Verticale, G. (2015). Impact of processing costs on service chain placement in network functions virtualization. In *Ieee conference nfv-sdn.*

Schneider, S., Werner, S., Khalili, R., Hecker, A., & Karl, H. (2022). mobile-env: An open platform for reinforcement learning in wireless mobile networks. In *Noms 2022-2022 ieee/ifip network operations and management symposium* (pp. 1–3).

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

Schwarzer, M., Obando Ceron, J. S., Courville, A., Bellemare, M. G., Agarwal, R., & Castro, P. S. (2023, 23–29 Jul). Bigger, better, faster: Human-level Atari with human-level efficiency. In A. Krause, E. Brunskill, K. Cho, B. Engelhardt, S. Sabato, & J. Scarlett (Eds.), *Proceedings of the 40th international conference on machine learning* (Vol. 202, pp. 30365–30380). PMLR. Consulté sur https://proceedings.mlr.press/v202/schwarzer23a.html

Sharma, S., Gumaste, A., & Tatipamula, M. (2020). Dynamic network slicing using utility algorithms and stochastic optimization. In *Ieee hpsr.* doi: 10.1109/HPSR48589.2020.9098981

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., ... Hassabis, D. (2017, octobre). Mastering the game of Go without human knowledge. *Nature*, *550*(7676), 354–359. Consulté le 2022-03-10, sur https://www.nature.com/articles/nature24270 (Number: 7676 Publisher: Nature Publishing Group) doi: 10.1038/nature24270

Singh, A., Jain, T., & Sukhbaatar, S. (2018). Learning when to communicate at scale in multiagent cooperative and competitive tasks. *arXiv preprint arXiv:1812.09755*.

Singh, S. K., Das, T., & Jukan, A. (2015). A survey on internet multipath routing and provisioning. *IEEE Communications Surveys & Tutorials*, *17*(4), 2157–2175.

Sitaraman, R. K., Kasbekar, M., Lichtenstein, W., & Jain, M. (2014). Overlay networks: An akamai perspective. *Advanced Content Delivery, Streaming, and Cloud Services*, 305–328.

Sun, P., Guo, Z., Li, J., Xu, Y., Lan, J., & Hu, Y. (2021). Enabling scalable routing in software-defined networks with deep reinforcement learning on critical nodes. *IEEE/ACM Transactions on Networking*, *30*(2), 629–640.

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

Tassiulas, L., & Ephremides, A. (1992). Stability properties of constrained queueing systems and scheduling policies for maximum throughput in multihop radio networks. *IEEE transactions on automatic control*, *37*(12), 1936–1948.

Tessler, C., Mankowitz, D. J., & Mannor, S. (2018). Reward constrained policy optimization. *arXiv preprint arXiv:1805.11074*.

Tomassilli, A., Giroire, F., Huin, N., & Pérennes, S. (2018). Provably efficient algorithms for placement of service function chains with ordering constraints. In *Ieee infocom.*

Tootaghaj, D. Z., Ahmed, F., Sharma, P., & Yannakakis, M. (2020). Homa: An efficient topology and route management approach in sd-wan overlays. In *Ieee infocom 2020* (pp. 2351–2360).

Trifa, Z., & Khemakhem, M. (2014). Effects of churn on structured p2p overlay networks. In *Proc. of int. conf. on acecs* (pp. 164–170).

Trimponias, G., Xiao, Y., Wu, X., Xu, H., & Geng, Y. (2019). Node-constrained traffic engineering: Theory and applications. *IEEE/ACM Transactions on Networking*, *27*(4), 1344–1358.

Troia, S., Alvizu, R., & Maier, G. (2019). Reinforcement learning for service function chain reconfiguration in nfv-sdn metro-core optical networks. *IEEE Access*. doi: 10.1109/ACCESS .2019.2953498

Wang, G., Feng, G., Quek, T. Q. S., Qin, S., Wen, R., & Tan, W. (2019). Reconfiguration in network slicing—optimizing the profit and performance. *IEEE TNSM*. doi: 10.1109/TNSM .2019.2899609

Watkins, C. J., & Dayan, P. (1992). Q-learning. *Machine learning*, *8*(3-4), 279–292.

Watkins, C. J. C. H. (1989). *Learning from delayed rewards.* (Thèse de doctorat non publiée). University of Cambridge.

Wei, F., Feng, G., Sun, Y., Wang, Y., Qin, S., & Liang, Y. C. (2020). Network slice reconfiguration by exploiting deep reinforcement learning with large action space. *IEEE Transactions on Network and Service Management*. doi: 10.1109/TNSM.2020.3019248

Weiß, G. (1995). Adaptation and learning in multi-agent systems: Some remarks and a bibliography. In *International joint conference on artificial intelligence* (pp. 1–21).

Wiegand, R. P. (2004). *An analysis of cooperative coevolutionary algorithms*. George Mason University.

Wolpert, D. H., & Tumer, K. (1999). An introduction to collective intelligence. *arXiv preprint cs/9908014*.

Yang, E., & Gu, D. (2004). *Multiagent reinforcement learning for multi-robot systems: A survey* (Rapport technique). tech. rep.

Yang, S., & Kuipers, F. A. (2014). Traffic uncertainty models in network planning. *IEEE Communications Magazine*, *52*(2), 172–177.

Yang, Z., Cui, Y., Li, B., Liu, Y., & Xu, Y. (2019). Software-defined wide area network (sd-wan): Architecture, advances and opportunities. In *2019 28th international conference on computer communication and networks (icccn).*

Yin, H., Liu, P., Liu, K., Cao, L., Zhang, L., Gao, Y., & Hei, X. (2020). Ns3-ai: Fostering artificial intelligence algorithms for networking research. In *Proc. acm 2020 workshop on ns-3 (wns3)* (p. 57–64). New York, NY, USA. doi: 10.1145/3389400.3389404

You, X., Li, X., Xu, Y., Feng, H., Zhao, J., & Yan, H. (2022, février). Toward Packet Routing With Fully Distributed Multiagent Deep Reinforcement Learning. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, *52*(2), 855–868. (Conference Name: IEEE Transactions on Systems, Man, and Cybernetics: Systems) doi: 10.1109/TSMC.2020.3012832

Young, M., Kate, A., Goldberg, I., & Karsten, M. (2012). Towards practical communication in byzantine-resistant dhts. *IEEE/ACM Trans. on Networking*, *21*(1), 190–203.

Yousefpour, A., Fung, C., Nguyen, T., Kadiyala, K., Jalali, F., Niakanlahiji, A., ... Jue, J. P. (2019). All one needs to know about fog computing and related edge computing paradigms: A complete survey. *Journal of Systems Architecture*, *98*, 289–330.

Zamora, I., Lopez, N. G., Vilches, V. M., & Cordero, A. H. (2016). Extending the openai gym for robotics: a toolkit for reinforcement learning using ros and gazebo. *arXiv preprint arXiv:1608.05742*.

Zhang, K., Yang, Z., Liu, H., Zhang, T., & Basar, T. (2018). Fully decentralized multi-agent reinforcement learning with networked agents. In *International conference on machine learning* (pp. 5872–5881).

Zhang, N., Liu, Y.-F., Farmanbar, H., Chang, T.-H., Hong, M., & Luo, Z.-Q. (2017). Network slicing for service-oriented networks under resource constraints. *IEEE JSAC*.

Zubow, A., Rösler, S., Gawłowicz, P., & Dressler, F. (2021). Grgym: When gnu radio goes to (ai) gym. In *Proceedings of the 22nd international workshop on mobile computing systems and applications* (pp. 8–14).

My publications

Alliche, R. A., Barros, T. D. S., Aparicio-Pardo, R., & Sassatelli, L. (2022). Prisma: a packet routing simulator for multi-agent reinforcement learning. In *2022 ifip networking conference (ifip networking)* (pp. 1–6).

Alliche, R. A., Barros, T. D. S., Aparicio-Pardo, R., & Sassatelli, L. (2023). prisma-v2: Extension to cloud overlay networks. In *2023 23rd international conference on transparent optical networks (icton)* (pp. 1–4).

Alliche, R. A., da Silva Barros, T., Aparicio-Pardo, R., & Sassatelli, L. (2022). Impact evaluation of control signalling onto distributed learning-based packet routing. In *34th intl. teletraffic congress, itc 2022*.

Aparicio-Pardo, R., Cousson, A., & Alliche, R. A. (2023). Quantum virtual link generation via reinforcement learning. In *23rd international conference on transparent optical networks (icton 2023)*.

Gausseran, A., Alliche, R. A., Lesfari, H., Aparicio-Pardo, R., Giroire, F., & Moulierac, J. (2022). Reconfiguring network slices at the best time with deep reinforcement learning. In *2022 ieee 11th international conference on cloud networking (cloudnet)* (pp. 85–92).

Web Pages

Chollet, e. a., F. (s. d.). *Keras. github.* Consulté sur https://github.com/keras-team/keras

Cidr report for 15 sep 23. (s. d.). Consulté sur https://www.cidr-report.org/as2.0/ (15/09/23)

Cisco Annual Internet Report - Cisco Annual Internet Report (2018–2023) White Paper. (s. d.). Consulté le 2023-09-22, sur https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html

Docker, I. (2020). Docker. *lnea].[Junio de 2017]. Disponible en: https://www. docker. com/what-docker*.

Facts and Figures 2023 - Internet use. (s. d.). Consulté le 2024-03-18, sur https://www.itu.int/itu-d/reports/statistics/2023/10/10/ff23-internet-use

Gym: Gym toolkit for creating reinforcement learning environments. (s. d.). Consulté sur https://gym.openai.com (11/05/22)

nsnam. (s. d.). *Ns-3 documentation website.* Consulté le 2022-03-14, sur https://www.nsnam.org/documentation/ (11/05/22)

Openai baselines: high-quality implementations of reinforcement learning algorithms. (s. d.). Consulté sur https://github.com/openai/baselines (11/05/22)

Prisma tool: An open marl framework for packet routing. (s. d.). Consulté sur https://github.com/rapariciopardo/PRISMA/tree/master

Prisma-v2: A packet routing simulator for multi-agent reinforcement learning - extension to cloud overlay networks. (s. d.). Consulté sur https://github.com/rapariciopardo/PRISMA/tree/odqr

Sndlib: Library of test instances for survivable fixed telecommunication network design. (s. d.). Consulté sur http://sndlib.zib.de

Tensorboard: Tensorflow's visualization toolkit. (s. d.). Consulté sur https://www.tensorflow.org/tensorboard (11/05/22)

Tensorflow: An end-to-end open source machine learning platform. (s. d.). Consulté sur https://www.tensorflow.org/ (11/05/22)

Wang, C., Rahman, A., Li, R., Aelmans, M., & Chakraborty, K. (2023, 10 mars). *Application Scenarios for the Quantum Internet* (Internet-Draft N° draft-irtf-qirg-quantum-internet-use-cases-15). Internet Engineering Task Force. Consulté sur https://datatracker.ietf.org/doc/draft-irtf-qirg-quantum-internet-use-cases/15/ (Work in Progress)

Zero mq: An open-source universal messaging library. (s. d.). Consulté sur https://zeromq.org/ (11/05/22)

Contrôle du réseau cloud basé Intelligence Artificielle

Abderrahmane Redha ALLICHE

Résumé

L'explosion du nombre d'utilisateurs d'Internet et du volume de trafic constitue un défi majeur pour la gestion efficace des réseaux de diffusion de contenu (CDN). Bien que ces réseaux aient amélioré le temps de réponse en exploitant la mise en cache dans des serveurs cloud proches des utilisateurs, les services non mis en cache continuent de poser des problèmes de gestion de trafic. Pour répondre à cette problématique, les réseaux overlay cloud ont émergé, mais ils introduisent des complexités telles que les violations d'inégalités triangulaires (TIV).
Dans ce contexte, l'application du paradigme des réseaux à définition logicielle (SDN) combinée aux techniques d'apprentissage par renforcement profond (DRL) offre une opportunité prometteuse pour s'adapter en temps réel aux fluctuations de l'environnement. Face à l'augmentation constante du nombre de serveurs edge, les solutions distribuées de DRL, notamment les modèles d'apprentissage par renforcement profond multi-agent (MA-DRL), deviennent cruciales. Cependant, ces modèles rencontrent des défis non résolus tels que l'absence de simulateurs réseau réalistes, le surcoût de communication entre agents et la convergence et stabilité.
Cette thèse se concentre donc sur l'exploration des méthodes MA-DRL pour le routage de paquets dans les réseaux overlay cloud. Elle propose des solutions pour relever ces défis, notamment le développement de simulateurs de réseau réalistes, l'étude du surcoût de communication et la conception d'une solution MA-DRL adaptée aux réseaux overlay cloud. L'accent est mis sur le compromis entre la performance et la quantité d'information partagée entre les agents, ainsi que sur la convergence et la stabilité de l'entraînement.

Mots-clés : Apprentissage par Renforcement Profond Multi-Agent ; Routage de Paquets Distribué ; Réseaux Cloud Overlay ; Contrôle de Réseaux Autonomes ; Intelligence Artificielle; Optimisation

Abstract

The exponential growth of Internet traffic in recent decades has prompted the emergence of Content Delivery Networks (CDNs) as a solution for managing high traffic volumes through data caching in cloud servers located near end-users. However, challenges persist, particularly for non-cacheable services, necessitating the use of cloud overlay networks. Due to a lack of knowledge about the underlay network, cloud overlay networks introduce complexities such as Triangle inequality violations (TIV) and dynamic traffic routing challenges.
Leveraging the Software Defined Networks (SDN) paradigm, Deep Reinforcement Learning (DRL) techniques offer the possibility to exploit collected data to better adapt to network changes. Furthermore, the increase of cloud edge servers presents scalability challenges, motivating the exploration of Multi-Agent DRL (MA-DRL) solutions. Despite its suitability for the distributed packet routing problem in cloud overlay networks, MA-DRL faces non-addressed challenges such as the need for realistic network simulators, handling communication overhead, and addressing the multi-objective nature of the routing problem.
This Ph.D. thesis delves into the realm of distributed Multi-Agent Deep Reinforcement Learning (MA-DRL) methods, specifically targeting the Distributed Packet Routing problem in cloud overlay networks. Throughout the thesis, we address these challenges by developing realistic network simulators, studying communication overhead in the non-overlay general setting, and proposing a distributed MA-DRL framework tailored to cloud overlay networks, focusing on communication overhead, convergence, and model stability.

Keywords: Multi-Agent Deep Reinforcement Learning; Distributed Packet Routing; Cloud Overlay Networks; Autonomous Network Control; Artificial Intelligence; Optimization