A Deep Reinforcement Learning Framework for Scalable Slice Orchestration in Beyond 5G Networks

Pavlos Doanis

A Deep Reinforcement Learning Framework for Scalable Slice Orchestration in Beyond 5G Networks

by

Pavlos Doanis

A dissertation submitted in partial fulfillment for the degree of

Doctor of Philosophy

at the

Ecole Doctorale Informatique, Télécommunications et Electronique

of

Sorbonne Université

Scheduled for defense on the 24th of April 2024, before a committee composed of:

Thesis supervisor:	
Thrasyvoulos Spyropoulos	EURECOM & Technical University of Crete
Reviewers:	
Michail Lagoudakis	Technical University of Crete
Christos Verikoukis	University of Patras
President of the jury:	
Dirk Slock	EURECOM
Examiners:	
Carla Fabiana Chiasserini	Politechnico di Torino
Marios Kountouris	EURECOM

Acknowledgements

First of all, I would like to thank my supervisor Akis, who guided me through this challenging journey in the best way. Each of our meetings helped me look ahead with more optimism and rationality than before, which was key to making it through. Also, I want to thank Theodoros Giannakas, my informal co-supervisor during the first year of the PhD, who was another cornerstone for both my work and social life. I will always remember our long discussions (either accompanied by a few beers or not), which I truly enjoyed. In addition, I would like to thank my parents for supporting all my decisions that brought me here. Of course, these three and a half years would not be the same without all the good friends I made, inside and outside of Eurecom, with whom we shared some great moments that served as fuel for me to keep going. Finally, I would never be able to accomplish all this without Katerina, my partner in life, who brings love and joy into my life and supports me unconditionally in all my endeavors.

Contents

List of Figures

List of Tables

Acronyms and Abbreviations

This is a list of all the Acronyms and Abbreviations used throughout the text, in order of appearance.

QoS	Quality of Service
5G	5^{th} Generation of mobile networks
CAPEX	Capital Expenditure
OPEX	Operational Expenditure
RAN	Radio Access Network
MME	Mobility Management Entity
NFV	Network Function Virtualization
SDN	Software Defined Networking
VNF	Virtual Network Function
OTT	Over-The-Top
SLA	Service Level Agreement
3GPP	3rd Generation Partnership Project
eMBB	enhanced Mobile Broadband
URLLC	Ultra-Reliable and Low Latency Communications
mMTC	massive Machine Type Communications
MEC	Multi-access Edge Computing
NR	5G New Radio
5GC	5G Core
NPN	Non Public Network
AI	Artificial Intelligence

ML	Machine Learning
NSA	Non Stand Alone
SA	Stand Alone
ITU	international Telecommunication Union
NGMN	Next Generation Mobile Networks
VL	Virtual Link
DNN	Deep Neural Network
RL	Reinforcement Learning
DRL	Deep Reinforcememt Learning
KPI	Key Performance Indicator
B5G	Beyond 5G Networks
DQN	Deep-Q Network
iDQN	independent Deep-Q Network
QL	Q-learning
MW	Multiplicative Weights
BDQ	Branching Deep-Q Network
C-RAN	Cloud Radio Access Network
TD	Temporal Difference

Chapter 1

Introduction

Future mobile networks are envisioned to support a large number of different services tailored to the specific needs of Vertical sectors and complying with stringent and diverse Quality of Service (QoS) requirements [1]. Network slicing is a key enabler of this vision, facilitating the creation of multiple independent logical networks ("slices"), running on top of the physical network and sharing its resources. Each slice can be dedicated to a different service, offering isolation and an agreed upon QoS [2]. However, considering the dynamicity of traffic demand (e.g. diurnal variations, peaks of traffic during big sports events, etc.), dynamic slice orchestration is essential to ensure the QoS requirements of different slices in a cost-efficient way [3]. This Thesis focuses on data-driven optimization methods for dynamic slice orchestration in Beyond 5G Networks.

1.1 Network Slicing in Beyond 5G Networks

In this section we provide some background on Network Slicing, starting with the evolution from a Network Sharing paradigm in 3G and 4G networks to the modern network slicing concept introduced in 5G. Then, we give an overview of the current standardization and deployment status of 5G, as well as the vision for Beyond 5G (B5G) networks (with a focus on slicing). Finally, we outline some key algorithmic challenges that still remain open in 5G+ network slicing and explain how our work comes into the picture.

Network Sharing. With a view to accommodate the ever-increasing mobile traffic demand in an economically viable way, Network sharing was introduced in mobile networks reducing the capital and operational expenditure costs (CAPEX/OPEX) for Mobile Network Operators (MNOs) [4]. First, with passive sharing and network roaming in 3G networks (e.g. sharing of sites, masts, etc.), and then with active Radio Access Network (RAN) sharing in 4G networks (e.g. sharing base stations, antennas, spectrum resources, backhaul equipment, etc.), including also scenarios of core network component sharing (e.g. the Mobility Management Entity (MME)) [1]. However, the 5G and beyond vision to support on-demand services tailored for Vertical sectors with stringent and diverse QoS requirements, requires a more advanced paradigm that extends beyond static network sharing solutions.

From Network Sharing to Network Slicing. The key to flexible service provisioning in 5G networks, was the advent of Network Function Virtualization (NFV) and Software Defined Networking (SDN) technologies [5, 6]. With NFV, traditional Network Functions (e.g. baseband processing, firewalls, load balancers, etc.), that were typically running on dedicated proprietary hardware, are now virtualized. Thus, Virtual Network Functions (VNFs) can be flexibly deployed as software on commercial off the self servers throughout the network. On the other hand, SDN provides flexible connectivity between VNFs by decoupling the control and data planes, meaning that the routing of traffic from VNF to VNF can be directly programmed via open interfaces (e.g., OpenFlow [7]). Network slicing leverages on both of these technologies to create virtual networks ("slices") on top of the Physical Network Infrastructure, offering isolation (both performance-wise and privacy-wise) and flexible resource sharing [8, 9]. It is the enabler of multi-tenancy in 5G and beyond systems, where Verticals, Over-The-Top (OTT) service providers, and Virtual MNOs can flexibly lease slices of the network from the Infrastructure Provider (typically an MNO), with custom functionalities and QoS that is governed by a Service Level Agreement (SLA) [1]. The benefits of Network slicing are the following:

- It expedites the provisioning of new services (no need to purchase new dedicated hardware equipment and re-train technicians to operate it).
- It offers isolation between slices (performance-wise and privacy-wise).

- It offers flexible utilization of the network resources, significantly reducing operational and capital expenses for MNOs (the network capacity provisioned to any slice can be dynamically managed according to traffic demands, avoiding unnecessary use of resources).

However, the above benefits of network slicing are offered in the expense of higher complexity in resource management, and therefore, challenging slice orchestration problems have emerged, that require algorithmic innovation.

5G Standardization Status. Network Slicing was introduced in Release 15 (Rel-15) of the 3rd Generation Partnership Project (3GPP) specifications, which was the first set of specifications for 5G networks. It was aiming to enable three main use case scenarios [10]:

- *enhanced Mobile Broadband (eMBB):* High data rates for accessing multimedia content and data (e.g. mobile video streaming, cloud applications, etc.).
- *Ultra-Reliable and Low Latency Communications (URLLC):* Stringent requirements on latency, throughput, and availability, targeting to mission-critical applications (e.g. remote surgeries, transportation safety, etc.).
- *massive Machine Type Communications (mMTC):* Supports a massive number of connected devices that transmit small volumes of data with high delay tolerance (e.g. smart meters, sensors, etc.).

While Rel-15 laid the foundations for 5G networks by introducing the main key enabling technologies (e.g. network slicing, Multi-Access Edge Computing (MEC), 5G New Radio (NR), 5G Core (5GC), etc.) to drive the above use cases, it focused mostly on the eMBB service. The second phase of 5G specifications (Rel-16 and Rel-17), termed as 5G-Evolution, included enhancements to support (among others) the Industrial Internet of Things (URLLC, Time Sensitive Networks) and Non Public Networks (NPNs) [11]. The third phase of 5G standardization, coined 5G-Advanced, started with Rel-18 that was concluded at the end of 2023, and is currently continuing with the on-going Rel-19, paving the way towards 6G networks. Some of the 5G-Advanced era features are the enhanced support of (new) services (e.g. cloud gaming, immersive reality, indoor positioning, and industrial sensor networks), as well as initial steps towards using Artificial Intelligence (AI)

and Machine Learning (ML) for network automation (to reduce energy consumption and improve QoS) [12].

5G Deployment Status. The first commercial rollout of 5G networks started in 2019 and concerned the Non-Stand Alone (NSA) version, meaning that only the 5G New Radio part was deployed, coexisting with the 4G radio (Long Term Evolution), and using the 4G core network (Evolved Packet Core). 5G NSA was deployed with a view to enhance network capacity and support the eMBB use case, however, it does not support network slicing, which requires a fully-fledged 5G network (5G radio *and* 5G core). The progress of the rollout has been significant, as more than 81% of the population in European Union is now covered by a 5G network [13].

Mobile Network Operators (MNOs) around the world are currently putting significant efforts to upgrade their 5G networks to the Stand Alone (SA) version, which will enable the full extend of 5G features, using 5G radios along with a cloud-native, service-based 5G core network. According to the latest European 5G Observatory report [13], at least 36 operators in 25 countries have already deployed 5G SA networks, however, their wide deployment is expected to take time as it requires careful planning and large investments.

Network slicing is supported by the 5G SA deployments, and is expected to address the needs of various Vertical sectors and applications in the future. To this end, MNOs have started conducting initial trials, with an example being the successful pilot of Vodafone and Ericsson in August 2023, that deployed a network slice for cloud gaming at Coventry University, achieving a significant QoS improvement [14]. It is noteworthy that network slicing is considered as the 5G feature with the highest potential impact in economic growth and sustainability, as its industrial applications will optimize production processes and reduce resource consumption (aligned with European Union's goals) [13]. Therefore, there is a great interest in addressing the remaining research challenges related to network slicing, that will allow its wide use in future mobile networks.

The Vision for Beyond 5G Networks. The need for a new generation of mobile networks is driven by the predicted exponential growth of mobile traffic [15], and the emergence of new disruptive services with more stringent QoS requirements. Discussions regarding the road to 6G had already started since 2019 with the creation of a focus group within the International Telecommunication

Union (ITU), intending to set the vision for mobile networks of 2030 and beyond and to identify the main drivers [16]. Similar projects were undertaken later on by other organizations, e.g. the white paper of the Next Generation Mobile Networks (NGMN) [17]. Some of the envisioned new use cases for 6G networks are holographic-type communications, extended reality, tactile internet, and Pervasive Intelligence [18]. One of the main drivers will be the extensive use of MEC, which will provide distributed computing, storage, and memory resources in close proximity to the connected devices, with a large number of "small" servers located at the Edge of the network [19]. Therefore, it will offer computation off-loading capabilities (computational intensive AI tasks can be off-loaded by a mobile device with limited resources to an edge server), and facilitate time-sensitive AI tasks by avoiding the transfer of data all the way to the Cloud for processing. Another key enabler will be end-to-end network automation, facilitated by data-driven slice orchestration based on AI/ML techniques [20].

Algorithmic and Modeling Challenges in Network Slicing. When it comes to algorithms for slice orchestration, or "squeezing as many virtual slices into a common physical network without anyone knowing it", two important performance considerations arise: (i) the fulfilment of the desired QoS metrics (defined by Service Level Agreements (SLAs)); (ii) the efficient utilization of the limited network resources. Since the demand for resources (by the hosted slices) fluctuates over time due to traffic variations, data-driven algorithms for dynamic slice orchestration are necessary to accomplish the aforementioned goals [3].

While different types of "slicing problems" have been considered in recent cellular network literature [21], the main flavors appear to be either: (i) the problem of *allocating resources* of physical nodes among slices (and users of that slice) sharing that node, e.g. allocating resource blocks in the Radio Access Network (RAN) [22–24], or (ii) the problem of *slice embedding*; the latter represents slices as graphs ("VNF chains") of Virtual Network Functions (VNFs) and a set of virtual links (VLs) that need to be mapped among physical nodes and links, while satisfying each slice's demands [25, 26]. Despite interesting initial attempts to tackle such problems, a number of challenges arising from the vision of 5G+ slicing remain: these relate both to the existence of generic yet useful models, as well as algorithmic efficiency.

Challenge 1: the majority of the parameters that affect the performance of each

network component (and thus hosted VNFs) are often unknown a priori, dynamically changing, or even non stationary, rendering traditional static and centralized optimization methods (whether discrete, continuous, or stochastic) problematic, if not altogether inapplicable. To this end, different works have applied Supervised ML using Deep Neural Networks (DNNs) to either forecast mobile traffic or even forecast directly the allocated resources [27–29]. Moreover, Deep Reinforcement Learning (DRL) has been employed to find effective slice orchestration policies by learning either the unknown traffic dynamics and/or the unknown performance functions [30, 31]. The reconfiguration cost involved when migrating VNFs from one server to another [32] makes far-sighted policies essential for slice orchestration, while the ability of DRL to obtain such policies makes it an ideal candidate for this task [33].

Challenge 2: B5G networks will involve slices whose VNFs will be spread across multiple technological (and administrative) domains (e.g. RAN, Edge, Core, Cloud, etc.); they will also be governed by end-to-end performance KPIs (key performance indicators) that often depend on the performance along the entire VNF chain (e.g. queuing delay across an end-to-end, possibly non-loop free path of VNFs and links). This not only complicates the modeling of such KPIs in a tractable manner, but immensely increases the optimization complexity due to the combinatorial nature of placing multiple (correlated) VNFs, for multiple slices, among multiple computation nodes. So, most works addressing slicing problems focus on simple setups (e.g. single domain, single slice, simple VNF chains and performance metrics), while solutions based on modern reinforcement learning theory have to deal with astronomically high action spaces, when one considers multi-VNF, multi-domain, multi-slice problems.

Challenge 3: the training of the algorithms is not particularly data-efficient, which can hinder their practical application given the scarce(r) availability of cellular network related data (as opposed to standard machine learning problems).

In this Thesis we focus on the problem of dynamic slice embedding (reconfiguration) in the context of inter-slice orchestration, assuming that resource allocation per node is performed by a given scheduling algorithm (e.g., proportionally fair sharing) whose impact is captured by our model. We attempt to tackle all the aforementioned shortcomings in one common RL framework.

1.2 Related Work and Novelty of Our Approach

In what follows we outline some key related work papers, which are close to our work either from a modeling or algorithmic perspective, and highlight the novelty of our approach.

In [34], the authors tackled the joint problem of placement and resource allocation for single-tenant scenarios, introducing a queuing-based system model (focusing on the end-to-end delay KPI). Their solution was based on decoupling the two problems but accounted for their mutual impact. So, they proposed to first place VNFs to physical nodes, based on a heuristic, and then allocate resources to them by solving a convex optimization problem with strict constraints on the end-to-end delay (the traffic demands are known - no statistical multiplexing). The common ground with our approach is that we also use queuing theory to model the end-to-end delay of slices, however our framework is very different, considering unknown traffic demands (statistical multiplexing) and using RL to address the placement problem (accounting also for the resource allocator impact through our queuing model). Moreover, our goal is to find a good trade-off between slice and network performance (taking into account the reconfiguration cost), while in [34] the authors try to merely minimize the SLA violations (user-centric perspective).

The resource allocation problem, in every node and link of the physical network, was recently addressed in an inter-slice setting with fully distributed deep RL agents (partial observability) in [31]. That work shares a number of features with ours, as it considers an inter-slice, multi-domain, end-to-end setup, and proposes a multi-agent DRL solution. However, there are two key differences in our work: (i) the work in [31] does not consider the VNF placement problem, assuming the placement is predetermined and fixed. Its focus instead is on the allocation problem among flows on that node; (ii) unlike [31], we propose and use an analytical (queuing) model to estimate the end-to-end delay for each slice (that also captures the competition of traffic among collocated VNFs); as a result, the (mean) reward at a given step is quickly available to the DRL algorithm, rather than having to be measured for every flow of every slice.

The two papers we consider to be the closest to our work, as they both tackle slice placement with deep RL algorithms, are [30] and [33]. In [30] the authors proposed the Deep Deterministic Policy Gradient algorithm, enhanced by a heuristic to speed up convergence (explore more efficiently the large action space). One

of the main reasons for employing RL was to learn the unknown and nonlinear performance functions (stressing the scarcity of available models). In [33], the reconfiguration cost was additionally taken into account, which is the reason why far-sighted policies are required, and the branching Deep Q-Network was proposed to deal with the large action space. While both of the above works manage to address a very challenging problem, considering the action space complexity hurdle, they ignore the end-to-end performance aspect (no slice-specific end-to-end SLAs), which is an important limitation nowadays with the type of complex slices envisioned for B5G networks (which should go with specific end-to-end performance guarantees). The inclusion of end-to-end KPIs introduces two additional complications: (i) it requires more sophisticated (queuing) models to properly capture end-to-end metrics like delay (e.g. probabilistic routing of flows through VNF chains, loops, etc.); (ii) the action space complexity, when considering multiple VNFs per slice does not simply scale additively (as would be the case if we simply require that the performance of every VNF is "good enough"); it rather scales multiplicatively, due to the "coupling" induced by the aforementioned end-to-end performance metrics, quickly giving rise to astronomically high action spaces, induced by the combinatorial nature of the problem.

Apart from the aforementioned differences, there are two additional aspects of our work that distinguish it from the above (as well as additional, but of course not all, related work): (i) we make a conscious effort to slowly build up to our final algorithm, starting from small, theoretically tractable scenarios (to use as benchmarks), and attempting an informed justification of every new building block along the way (this is not something common in related literature, and we believe it helps the reader understand/dissect the potential impact of each algorithmic component on the observed performance benefits); (ii) we also validate our algorithm using a real traffic dataset [35] (it is crucial to confirm that the proposed solution is still effective in realistic non-Markovian scenarios).

In Table 1.1 we present a table of (desirable) features, problem, and algorithmic characteristics that each of these discussed schemes (including ours) cover, in order to provide a more compact picture of similarities and differences.

TABLE 1.1: Comparison with related work. The checkmarks indicate which features are supported by each work (enclosed within parentheses when a feature is partly supported).

Features		Related work papers [34]	[31]	[30]	[33]	This Thesis
Problem	slice placement	✓		✓	✓	✓
	resource allocation	✓	✓			(✓)
Optim. goals	e2e SLAs	✓	✓			✓
	network performance		✓	✓	✓	✓
	reconfigurations				✓	✓
Model	statistical multiplexing		✓	✓	✓	✓
	multiple slices		✓			✓
	complex slices	✓	✓	✓		✓
	multiple domains		✓			✓
Validation	Markov traffic	✓	✓			✓
	real traffic					✓

1.3 Outline of Thesis and Contributions

In this section we provide an outline of the Thesis and specify, separately for each chapter, the corresponding technical contributions as well as the limitations of existing works they address. The final goal is to tackle all the modeling and algorithmic challenges highlighted in Section 1.1 in one common RL framework, facilitating dynamic slice embedding in B5G networks. Therefore, each chapter builds on top of the previous to provide all the different components required, starting from the RL environment in Chapter 2, and continuing with the RL agents in Chapters 3 and 4.

1.3.1 Chapter 2: RL Environment for End-to-End Inter-Slice Orchestration

Limitations of Existing Works

Most works addressing the problem of dynamically placing VNF chains upon a physical network topology with RL-based methods focus on single-domain setups, or single/simple network slices, or simple performance metrics (no end-to-end SLAs with diverse and slice-specific KPIs). However, Beyond 5G networks are envisioned to be multi-tenant systems, hosting a large number of different services (slices) that span multiple domains (e.g. Edge, RAN, Core, Cloud), have widely diverse QoS requirements, and are governed by end-to-end SLAs. Moreover, there is lack of analytical (but still realistic) RL models capturing the above characteristics. Such analytical models could be potentially used to speed up the (offline) training of RL agents, since simulating the end-to-end performance of every flow of every slice can be computationally intensive in large scale scenarios.

Technical Contributions

We first introduce a generic queuing-based system model that can capture the co-existence of multiple VNF chains residing upon the same physical network and sharing its resources (Section 2.2). To this end, (i) we model the physical network as a queuing network of M/G/1/PS queues that correspond to physical nodes and links (the per node resource allocation approximates a proportionally fair scheduler); (ii) we model network slices as directed graphs of VNFs and VLs that may have arbitrarily complex topologies (allowing for probabilistic routing of flows and loops). Finally, we provide analytical expressions for the per slice end-to-end delay (as a function of the assignment of VNFs to physical nodes and the traffic demand).

Based on the above system model, we formulate the RL problem of dynamic slice embedding (Section 2.3). Therefore, (i) we define the corresponding state and action spaces and discuss their scaling properties; (ii) we define the reward function as a weighted sum of three different cost terms: SLA violations cost, reconfigurations cost, and active nodes cost.

This Chapter's contributions appear in:

- P. Doanis, T. Giannakas, and T. Spyropoulos, "Scalable end-to-end slice embedding and reconfiguration based on independent DQN agents," in IEEE Global Communications Conference (GLOBECOM), pp. 3429–3434, Dec. 2022.
- (*prelim version*) P. Doanis and T. Spyropoulos, "Scalable slice orchestration with DQN agents in Beyond 5G networks," in IEEE 27th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD), Nov. 2022.

1.3.2 Chapter 3: Multi-Agent DQN with Independent Sample-Efficient Agents

Limitations of Existing Works

While Chapter 2 introduced the RL environment, laying the ground for an RL-based solution that will be inline with the key characteristics dictated by the B5G vision (multiple complex slices, multiple domains, end-to-end SLAs), it didn't propose any specific DRL algorithm. It is evident that, since such an environment further exacerbates the state and action complexity, the proposed algorithmic solutions should be scalable and sample efficient. Some additional limitations frequently encountered in related works are: (i) the performance of the proposed DRL schemes (that have no performance guarantees whatsoever) is not compared against any theoretically grounded scheme (at least in smaller more tractable scenarios); (ii) the proposed schemes are not validated under real traffic datasets.

Technical Contributions

Due to the inherent complexity of the problem at hand, stemming from the combinatorial state and action spaces, we first attempt a theoretical analysis of the scalability properties of various existing RL approaches (Section 3.2). We use as a starting point the standard (tabular) Q-learning method, and gradually build on top of that, justifying each additional component along the way. We end up with a multi-agent scheme of independent DQN agents (iDQN), that will be the basis of our proposed algorithm for this chapter. This scheme decomposes the combinatorial action space into smaller sub-spaces, by allotting the control of each VNF to a different agent. Therefore, the DQN component tackles the state complexity, while the use of multiple agents tackles the action complexity.

To further improve the (sample) efficiency of standard DQN agents (single or multi), we introduce two additional mechanisms that optimize how the "experience replay" is used (Section 3.2.4): (i) a priority mechanism to access the samples in the experience replay (often referred to as a prioritized experience replay [36]); and (ii) the storage of some additional information per experience to reduce the number of computations during updates. These mechanisms aim to further speed up the convergence of our multi-agent algorithm and facilitate its practical applicability. We denote by DQN+ and iDQN+ the "enhanced" DQN and iDQN versions respectively.

In Section 3.3, we use simulation both with synthetic (Markovian) and real traffic to confirm in practice the observations of Section 3.2 (that were based on theoretical computational complexity arguments). First, in Section 3.3.2, using small scale scenarios and synthetic traffic, we show that Q-learning based approximate algorithms, either single-agent or multi-agent, are able to obtain close to optimal solutions (i.e. ones found by exact Q-learning), yet with much higher convergence speed. We also show that as the problem size increases, iDQN converges orders of magnitude faster than standard single-agent DQN, with minimum penalty of decision optimality. Then, in Section 3.3.3, using a real traffic dataset to drive the demand, we confirm the convergence speed gains offered by the proposed speed up heuristics on top of DQN and iDQN, while in a fairly large multi-domain scenario (Section 3.3.4) we show that the proposed algorithm outperforms static heuristic policies by at least a 3× factor.

This Chapter's contributions appear in:

- P. Doanis, and T. Spyropoulos, "Multi-agent DQN with sample-efficient updates for large inter-slice orchestration problems," in IEEE International Conference on Computing, Networking and Communications (ICNC), Feb. 2024.
- P. Doanis and T. Spyropoulos, "Sample-efficient multi-agent DQNs for scalable multi-domain 5G+ inter-slice orchestration," IEEE Transactions on Machine Learning in Communications and Networking (TMLCN), 2024.

1.3.3 Chapter 4: Scalable DQN with Coordinated Branches

Limitations of Existing Works

Although the multi-agent DQN-based scheme proposed in Chapter 3 proved to be a much more scalable solution compared to its single-agent counterpart, the use of *independent* agents can potentially still be problematic in large scale scenarios. The lack of coordination among agents results to a non-stationary environment, which might negatively affect sample efficiency, quality of the obtained policies, or even lead to stability problems. Moreover, in the validation section of the previous chapter, the policies obtained by the DRL schemes were grounded to the optimal policies of Q-learning only under synthetic (Markovian) traffic, while in real traffic scenarios no theoretically grounded scheme was used as a benchmark (Q-learning is not applicable due to the infinite state space).

Technical Contributions

First, we formulate the dynamic slice embedding problem as a (stateless) "experts" problem and propose a state-of-the-art algorithm, called Multiplicative Weights (MW), that is theoretically optimal in the experts context (Section 4.3.1). While "stateless", this experts algorithm can in fact explore all actions - often a huge number - in parallel, unlike our scheme which operates in a "bandit-like" setup, exploring one action at a time. As a result, this baseline can be seen as a non-implementable oracle, that even a stateful scheme might not be able to match, and therefore, we use it as a benchmark in real traffic scenarios.

Then, in Section 4.4, we propose a DQN-based scheme with a different DNN architecture, called the Branching Deep Q-Network (BDQ). According to this, the control of each VNF is allotted to a different branch of the DNN, dramatically reducing action complexity compared to vanilla single-agent approaches (similar action decomposition advantages with iDQN). On top of that, to avoid the non-stationarity issues arising in multi-agent solutions with independent agents, a shared DNN module between different branches is responsible for their (implicit) coordination, improving the scheme's sample efficiency and scalability properties.

Finally, in Section 4.5, using real traffic to drive the demands, we demonstrate by simulations that the proposed BDQ scheme outperforms (i) the experts baseline, both in terms of cost performance and sample efficiency (theoretically "grounding" the proposed DRL scheme); and (ii) the existing state-of-the-art multi-agent DQN approach of Chapter 3, showing up to 45% cost improvement in a fairly large scenario.

This Chapter's contributions appear in:

- P. Doanis, and T. Spyropoulos, "The Curse of (Too Much) Choice: Handling combinatorial action spaces in slice orchestration problems using DQN with coordinated branches," in IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), May 2024.

Chapter 2

RL Environment for End-to-End Inter-Slice Orchestration

2.1 Introduction

As we discussed in Chapter 1, while different types of "slicing problems" have been considered in recent cellular network literature [21], the main flavors appear to be either: (i) the problem of *allocating resources* of physical nodes among slices (and users of that slice) sharing that node, e.g. allocating resource blocks in the Radio Access Network (RAN) [22–24], or (ii) the problem of *slice embedding*; the latter represents slices as graphs ("VNF chains") of Virtual Network Functions (VNFs) and Virtual Links (VLs) that need to be mapped among physical nodes and links respectively, while satisfying each slice's demands [25, 26]. Despite interesting initial attempts to tackle such problems, a number of challenges arising from the vision of 5G+ slicing remain: these relate both to the existence of generic yet useful models, as well as algorithmic efficiency.

First, beyond 5G networks will involve slices whose VNFs will be spread across multiple technological (and administrative) domains; they will also be governed by end-to-end performance KPIs (key performance indicators) that often depend on the performance along the entire VNF chain (e.g. queuing delay across an end-to-end, possibly non-loop free path of VNFs and links). This not only complicates the modeling of such KPIs in a tractable manner, but immensely increases the optimization complexity due to the combinatorial nature of placing multiple (correlated) VNFs, for multiple slices, among multiple computation nodes.

Second, the majority of the parameters that affect the performance of each network component (and thus hosted VNFs) are often unknown a priori, dynamically changing, and often non stationary, rendering traditional static and centralized optimization methods (whether discrete, continuous, or stochastic) problematic, if not altogether inapplicable.

This Thesis focuses on the problem of dynamic slice embedding assuming that resource allocation per node is performed by a given scheduling algorithm (e.g., proportionally fair sharing), whose impact is captured by our model. The above problem is characterized by (i) unknown future resource demands; and (ii) delayed rewards (e.g. if the resource demand of a VNF is predicted to increase soon and stay high for a while, paying now a cost to migrate this VNF to a less busy server could potentially lead to high future rewards). Since this is the standard "playground" of RL, in this Chapter we introduce a generic queuing-based B5G RL environment that attempts to tackle the aforementioned modeling complexity challenges and supports multiple technological domains, complex slices, and diverse end-to-end SLAs (the algorithmic complexity challenges will be addressed later on, in Chapters 3 and 4).

2.2 System Model

First, we will define a B5G system model that facilitates slicing and VNF chains co-existence. We introduce the model components gradually as follows:

- the *physical network model*: the network components to be shared by various tenants/slices (Subsection 2.2.1).
- the *virtual network model*: the (virtual) components of a "slice" (VNF chain), their demands for network resources, and their placement upon the physical network (Subsection 2.2.2).
- the *queuing model* that captures the competition for the same resources by different slices assigned on the same physical component (Subsection 2.2.3).

Since there is plenty of notation to keep track of, we will use the toy example of Fig. 2.1 to explain the various quantities involved throughout this section. We also provide a notation table (Table 2.1).

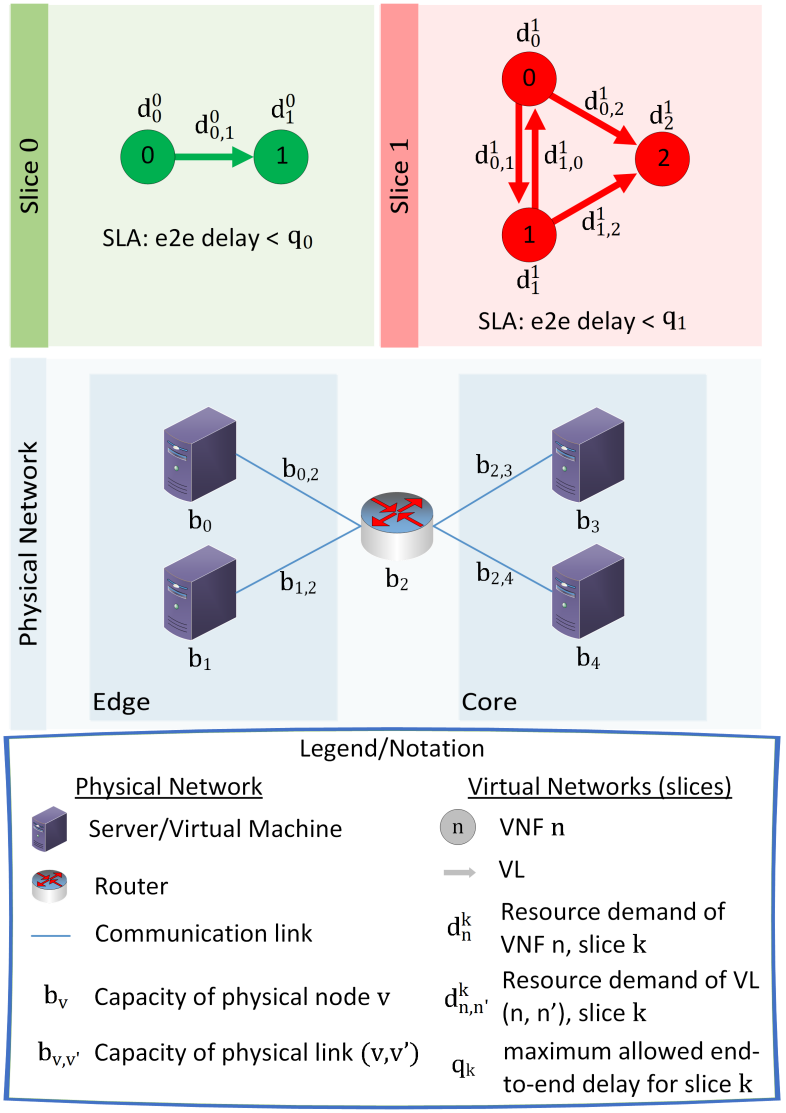


FIGURE 2.1: A toy example depicting some of the features of our model for inter-slice orchestration. The complex topology of slice 1 is an example that showcases the probabilistic routing of flows supported by our model, e.g., if a flow has just been processed by VNF 0, it will next move either to VNF 1 or to VNF 2 with respective probabilities (there could be also some probability to exit the system).

2.2.1 Physical Network

The physical network is represented as a weighted undirected graph $G = (\mathcal{V}, \mathcal{E})$ of physical nodes and the links connecting them, as is common in related literature [25], [26].

Physical nodes: They are either nodes that can process and forward user generated traffic (e.g. servers, virtual machines, base stations) or nodes responsible only for traffic forwarding (routers). Each node, $v \in \mathcal{V} = \{0, 1, ..., V-1\}$, is

characterized by some capacity b_v, which may correspond to its mean service rate, number of CPU cores, memory, storage, resource blocks, etc., depending on the domain and the underlying modeling assumptions[1]. It is important to note that these nodes might belong to different technological or administrative domains, e.g. Cloud Radio Access Network (C-RAN), Multi-access Edge Computing (MEC), Core network, etc.

Physical Paths: We assume that any two physical nodes v and v' are connected by a path of physical links and routers/switches with (bottleneck) capacity $b_{v,v'}$.[2]

In the toy example of Fig. 2.1, the physical network comprises two different domains, Edge and Core, with two servers per domain and a router that is responsible for forwarding traffic between servers.

2.2.2 Network Slices

A set $\mathcal{K} = \{0, 1, ..., K-1\}$ of virtual networks, called network slices (or VNF chains), are hosted on top of the physical network. Each slice $k \in \mathcal{K}$ is represented by a directed graph $H_k = (\mathcal{N}_k, \mathcal{L}_k)$, comprising a set $\mathcal{N}_k = \{0, 1, ..., N_k - 1\}$ of Virtual Network Functions and a set $\mathcal{L}_k$ of Virtual Links, and is associated with some SLA q_k (a maximum or minimum value for an end-to-end KPI metric). This is a standard way to model slices in related work literature [25], [26].

Virtual Network Functions (VNFs): They are processing tasks running on physical nodes (e.g. baseband processing at the C-RAN [38], analytics at the MEC [39], access and mobility management at the Core [40]). Assuming that time is separated into time windows (referred to as *slots* hereafter), each VNF $n \in \mathcal{N}_k$ of slice $k \in \mathcal{K}$ requires an amount of resources $d_n^k(t)$ during slot t, where $t \in \{0, 1, ..., T\}$ (in the remainder, we drop the time notation when clear from the

[1] We assume that the capacity restrictions of routers are already included in the path capacity (to be defined shortly).

[2] W.l.o.g. we assume that routing paths are predetermined and known, for any pair of physical nodes. Hence, the main impact of including path capacity in our model is that, an algorithm might choose to migrate a VNF in different nodes, even if the current node is not congested, because it predicts that the path capacity will become congested soon. However, our algorithm could easily be extended to scenarios with multiple alternative paths to choose from, for each node pair (e.g., SDN-based implementation of per slice routing, for load-balancing, as in [37]).

context). These demands fluctuate over time slots, and their transition dynamics are assumed to be unknown initially (possibly non-stationary).[3]

Virtual Links (VLs): They are directed links $l = (n, n') \in \mathcal{L}_k$, where $n \neq n'$, that capture some communication demand from a VNF n to another VNF n' and are mapped to physical paths. Thus, VLs indicate the sequence of VNFs a user flow must go through. They are associated to a time-varying resource demand $d^k_{n,n'}(t)$ (similarly to VNFs).

Slice topology (VNF chain): Our model is fairly generic allowing for *probabilistic routing* of flows through the chain: i.e., not all user flows must go through the same VNFs sequence (akin to a *Jackson queuing network* [41]; we will elaborate on this shortly). In Fig. 2.1, we show two simple VNF chain examples. Slice 0 is a simple "tandem" of two VNFs that all flows go through. Slice 1 contains 3 VNFs, but a flow will go through given VNFs "paths" with some probability, or possibly even "loop" through the same VNF (e.g. a flow might go from VNF 0 to VNF 1 and then back to VNF 0 again).

A motivating example of such probabilistic routing could be that of a video streaming slice consisting of 4 VNFs: firewall, deep packet inspection, transcoder, and billing [34]. The traffic of this slice must first traverse the firewall VNF and then only a sample of (possibly malevolent) packets will also go through deep packet inspection before proceeding to the transcoder and finally billing (the rest will go straight for transcoding-billing).

Slice orchestration: Our main goal is to "orchestrate" the various VNF chains of different slices on top of the shared resources, which mainly means to decide on which physical node to place each VNF of each slice (see Fig. 2.2 for an example). This placement decision must take place at each time slot, and hence a VNF might change its location from slot to slot ("migrate") if this will benefit the slice and/or the network performance (we will elaborate on this placement problem shortly).

Following, we define two of the most important slice-related quantities, the configuration and demand vectors, which we will be frequently using throughout the thesis:

[3] We assume that this corresponds to the mean aggregate demand (e.g. arrival rate) for all flows served by that VNF, whose number might also fluctuate from slot to slot. While mean traffic demand between time slots might change, to facilitate our analysis, we'll assume that it is constant during a slot (it is long enough for standard queuing models to reach the steady state).

Configuration vector ($c_t \in \mathcal{C}$)**:** the assignment of all VNFs to physical nodes at time slot t,

$$c_t = (c_n^k(t) | \forall k \in \mathcal{K},\ n \in \mathcal{N}_k), \tag{2.1}$$

where $c_n^k(t)$ indicates the host node of VNF n (slice k) at t; e.g., in Fig. 2.2, the configuration vector is $c = (c_0^0, c_1^0, c_0^1, c_1^1, c_2^1) = (0, 3, 0, 1, 3)$.

Demand vector ($d_t \in \mathcal{D}$)**:** denotes the demands of all slices at time slot t,

$$d_t = (d_i^k(t) | \forall k \in \mathcal{K},\ i \in \mathcal{N}_k \cup \mathcal{L}_k). \tag{2.2}$$

As an example, in Fig. 2.1 the demand vector is $d = (d_0^0, d_1^0, d_{0,1}^0, d_0^1, d_1^1, d_2^1, d_{0,1}^1, d_{1,0}^1, d_{0,2}^1, d_{1,2}^1)$.

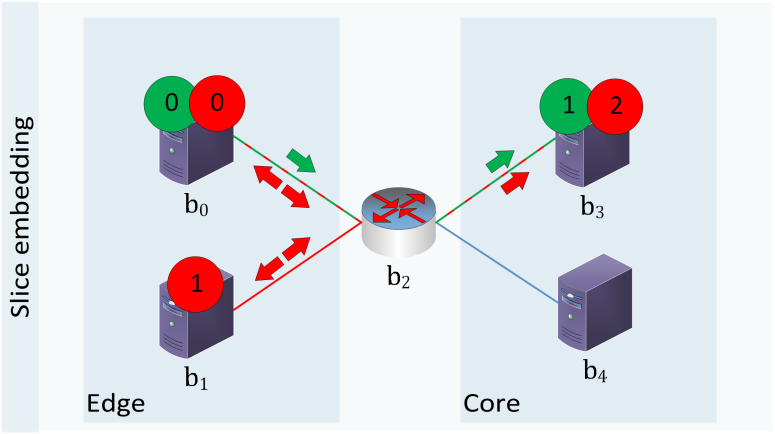


FIGURE 2.2: Slice embedding example corresponding to the toy-scenario of Fig. 2.1. VNFs are assigned to physical nodes and VLs to physical paths. In this example we assume that the resource demand of VNF 1 (red) is currently very high (e.g., training of a Machine Learning model under way), and thus it has been placed on a dedicated server to avoid congestion/SLA violations. Regarding the rest of the VNFs, the capacity of a single server is adequate to host two of them if paired properly (e.g. a high-demand VNF is placed together with a low-demand VNF). Due to fluctuating demands, this placement might become highly sub-optimal withing the next few time slots; in that case a reconfiguration might be required to maintain optimal system performance.

2.2.3 Queuing model.

Given a configuration decision (assignment of VNFs to nodes), the (unknown) traffic demands of co-located VNFs will compete for that node's resources. We are therefore in need for a model that captures:

1. Scheduling: the competition of VNF demands on the same node and its impact on the VNF/node performance.

TABLE 2.1: Notation table

Notation	Description
$\mathcal{G} = (\mathcal{V}, \mathcal{E})$	Physical network (weighted undirected graph)
$\mathcal{V}$	Set of physical nodes (cardinality V)
$\mathcal{E}$	Set of physical links (cardinality E)
b_v	Capacity of physical node v
$b_{v,v'}$	Capacity of physical link/path (v, v')
$\mathcal{K}$	Set of network slices (cardinality K)
$\mathcal{H}_i = (\mathcal{N}_i, \mathcal{L}_i)$	Weighted directed graph representing slice $k \in \mathcal{K}$
$\mathcal{N}_k$	Set of VNFs of slice k (cardinality N_k)
$\mathcal{L}_k$	Set of VLs of slice k (cardinality L_k)
d_n^k	Resource demand of VNF n of slice k
$d_{n,n'}^k$	Resource demand of VL (n, n') of slice k
d	Demand vector: $d = (d_i^k \vert \forall k \in \mathcal{K},\ i \in \mathcal{N}_k \cup \mathcal{L}_k)$
c_n^k	Indicates the host node of VNF n of slice k
$c_{n,n'}^k$	Indicates the host link of VL (n, n') of slice k
c	Configuration vector: $c = (c_n^k \vert \forall k \in \mathcal{K},\ n \in \mathcal{N}_k)$
s	State of the system
a	Agent action
r	Reward

2. End-to-end performance: how the performance of all (physical) nodes and paths on which a given VNF chain is placed upon is combined into an end-to-end (slice-wise) KPI.

Per node performance model (Scheduling). We model each physical node/path as a single server queue. In the simplest setup, we assume that the traffic demands of all VNFs/VLs assigned on that physical component are multiplexed using a simple "Processor Sharing" (PS) scheduling. It is known that PS schedulers (with traffic classes) can be a good approximation for a number of real life schedulers (e.g. LTE) [42]. Given that the actual flow demands can be generically distributed, this gives rise to an M/G/1/PS queuing system [41] for each physical node/path i, with total arrival rate equal to λ_i (a function of c and d; we will elaborate on its derivation shortly) and service rate b_i. The mean delay per flow for that node/path is:

$$f_i^{\text{delay}}(c_t, d_t) = \frac{1}{b_i - \lambda_i(c_t, d_t)} \tag{2.3}$$

End-to-end performance. Having modeled the performance of each node/-path involved in each chain, the entire network can now be modelled as a Jackson network of M/G/1/PS servers. Thankfully, closed form expressions for the total delay of such networks are available, even in situation where VNF chain paths contain loops. In fact, these results can be greatly generalized through the BCMP model [43] that allows for different traffic classes, priorities, etc., allowing significantly more fine-tuned scheduler models to be captured as well. We give a simple example of calculating the end-to-end delay of complex slices in Fig. 2.3. Note that any other end-to-end KPI metric can be supported by our framework, e.g., considering an SLA agreement on bandwidth, if $\psi^v_{k,n}$ is the amount of bandwidth assigned to VNF n of slice k on host node v by the local scheduler, then the corresponding end-to-end KPI could be given by $F^{\text{bw}}_k(s_t) = \min_{n,v} \psi^v_{k,n}$.

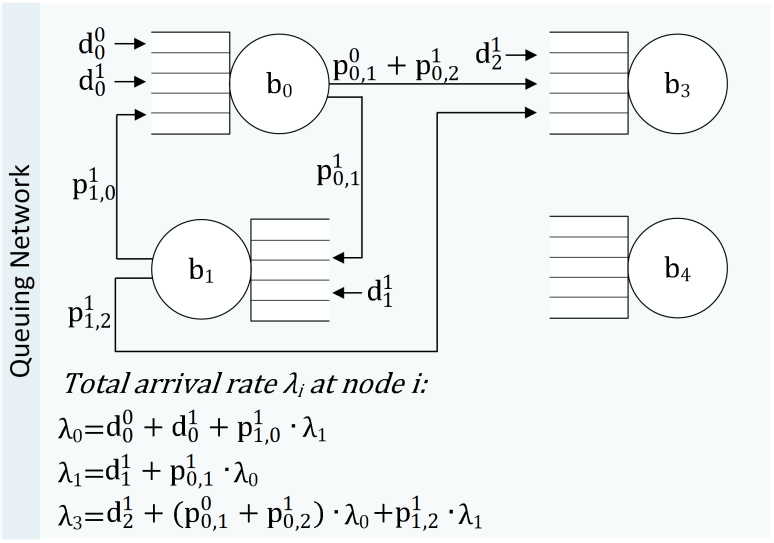


FIGURE 2.3: Queuing network corresponding to the system of Fig. 2.2 (to simplify illustration we ignored physical links). Each physical node i is an M/G/1/PS queue with mean service rate b_i and total arrival rate λ_i. We denote by d^k_n the arrival rate of jobs from outside the network for VNF n of slice k, while $p^k_{n,n'}$ is the probability that a flow of slice k will be processed by VNF n' right after getting processed by VNF n. The total arrival rate at each node can be simply calculated by solving the corresponding system of equations given in the figure. Then, the end-to-end delay of a flow is the sum of delays at the traversed nodes (given by (2.3), based on the calculated λ_i).

2.3 RL Problem Formulation

In RL problems the state of the environment is observed by an agent in discrete time slots. Based on this observation, the agent takes an action in each slot and the environment returns a feedback signal (called the reward) to indicate how good this action was. Following, we define the state space, the action space, and the reward function in our problem.

State Space $\mathcal{S}$. The state of the system at time slot t consists of (i) the configuration vector c_t (2.1); and (ii) the demand vector d_t (2.2) (both are necessary to calculate the instantaneous reward, to be elaborated shortly).

Definition 2.1 (State). $s_t = (c_t, d_t), s_t \in \mathcal{S} = \mathcal{C} \times \mathcal{D}$

We consider two cases:

-*Discrete traffic demand:* this scenario arises either in situations where traffic demand is expressed in naturally discrete quantities (e.g. Resource Blocks in OFDM, or Modulation and Coding schemes [44]) or as an approximation of continuous demand (e.g., quantized). While these scenarios can be easily handled with standard ("exact") discrete MDP based scenarios, in theory, the complexity of the state space increases combinatorially in the traffic levels. We will mainly use this scenario in simulations, for sensitivity analysis or toy examples, to better illustrate the tradeoffs involved.

-*Continuous traffic demand:* while this is a natural modeling assumption for traffic demand, it implies an infinite state space, and thus cannot be handled by standard ("tabular") methods. We will handle such scenarios with approximate RL methods only (specifically, using DNNs), and will be the main driving scenario for our more advanced, data-driven DRL solutions.

Action space $\mathcal{A}$. Given the current state of the system $s_t = (c_t, d_t)$, the agent's action a_t is a new assignment of VNFs to physical nodes c_{t+1}.

Definition 2.2 (Action). $a_t = c_{t+1},\ a_t \in \mathcal{A} = \mathcal{C}$

Remark: This implies that *both* the state space *and* the action space of this problem quickly explode in bigger scenarios. This is unlike recent video game problem setups [45], where DNNs are able to handle successfully very large state

spaces (video game images), but action spaces are rather small. While more recent successes in the field of board games (e.g. Go, Chess [46]) do need to tackle large state and action spaces, they tackle "planning" problems, where the environment and rules are fully known, unlike our problem, where environment variables (the VNF demands) are unknown and time-varying.

Reward function. We consider three different cost terms that determine the total cost performance of the system (other components can be straightforwardly added to the framework).

Type 1 cost (SLA violation): When the maximum (or minimum) KPI value defined by the SLA is infringed, a penalty is paid by the network operator to the slice tenant [29]. This is captured by a function $\Phi_k(s_t)$, for slice $k \in \mathcal{K}$, which may take any suitable form (e.g., linear, quadratic, etc.) to model the impact of violating the corresponding KPI (depending on the slice type). Then, the total SLA violation cost is given by:

$$g_1(s_t) = \sum_{k \in \mathcal{K}} \Phi_k(s_t) \tag{2.4}$$

We give as an example the linear form of the function $\Phi_k(s_t)$, assuming that the SLA defines a *maximum* value of the corresponding KPI:

$$\Phi_k(s_t) = (\sigma_k + (F_k^{KPI}(s_t) - q_k)) \cdot \mathbf{1}_{\{F_k^{KPI}(s_t) > q_k\}}, \tag{2.5}$$

where $F_k^{KPI}(s_t)$ is a function that outputs the end-to-end KPI value of slice k, σ_k is a fixed penalty paid for any SLA violation, and $\mathbf{1}_{\{\text{condition}\}}$ is a binary indicator variable that is equal to 1 when the condition inside the brackets is satisfied and 0 otherwise. In this work, we consider SLAs on the end-to-end delay, which can be calculated based on our queuing model (see (2.3), Section 2.2.3).

Type 2 cost (Reconfiguration): Migrating VNFs from their host servers causes network overhead (e.g., due to signalling, transfer of associated data), or even service downtime [29, 32, 33]. If $\mathbf{M} = (m_{v,v'})$ is a matrix that defines the cost of migrating a VNF between any two nodes v and v' of the network[4], then the reconfiguration cost is:

[4]The diagonal elements of matrix $\mathbf{M}$ are equal to zero.

$$g_2(c, c') = \sum_{k \in \mathcal{K}} \sum_{n \in \mathcal{N}_k} m_{c^{kn}, c^{kn\prime}} \tag{2.6}$$

In the remainder, w.l.o.g., we assume that all migrations are equivalent (with a cost equal to one).

Type 3 cost (Active nodes): Each of the physical nodes that are "on" (host at least one VNF) inflicts a monetary cost (or perhaps some other resource maintenance cost); e.g., the idle servers/virtual machines can be turned off or set to sleep mode in order to save energy [47]:

$$g_3(c_t) = \sum_{v \in \mathcal{V}} w_v^{\text{on}} \cdot \mathbf{1}_{\{v \in c_t\}}, \tag{2.7}$$

where w_v^{on} is the cost of node v when it is "on". In the remainder, w.l.o.g., we assume that $w_v^{\text{on}} = 1$, $\forall v \in \mathcal{V}$. Note that minimizing this cost leads to freeing up resources, which in turn facilitates admission control [48].

Reward: Given the current state of the system $s_t = (c_t, d_t)$, the respective agent action $a_t = c_{t+1}$, and the next state $s_{t+1} = (c_{t+1}, d_{t+1})$, we define the total cost as a weighted sum of the individual costs (2.5), (2.6), and (2.7):

$$g(c_t, a_t, s_{t+1}) = w_1 \cdot g_1(s_{t+1}) + w_2 \cdot g_2(c_t, a_t) + w_3 \cdot g_3(a_t), \tag{2.8}$$

where the weights w_1, w_2, w_3 are positive scalars and sum to one. Then, the corresponding reward is defined as:

Definition 2.3 (Reward)**.**

$$r_{t+1} = -g(c_t, a_t, s_{t+1}) \tag{2.9}$$

We introduce the negative sign in (2.9), as typically RL agents try to maximize the expected accumulated rewards (we want to minimize the expected accumulated cost).

2.4 Conclusions

In this Chapter we introduced a generic queuing-based B5G RL environment for dynamic slice embedding, that supports multiple technological domains, complex slices (allowing for probabilistic routing of flows and loops), and diverse end-to-end SLAs. The reward function was defined as a weighted sum of three different cost terms corresponding to SLA violations, reconfigurations, and number of active physical nodes, with a view to optimize slice performance (minimize SLAs) while also keeping network costs low (resource consumption and reconfigurations). Based on this framework, we provided analytical expressions for the per slice end-to-end delay KPI (a function of the assignment of VNFs to physical nodes and the traffic demand), but we stress that our model is generic and can support any other KPI. We also remark that the proposed environment has been implemented in Python, based on the popular Open AI Gym framework, and thus it can be easily reused and further enhanced by other researchers (we plan to make the code available in a public repository soon).

Remaining Open Questions. What is missing is to devise RL agents that can learn an effective slice embedding policy in reasonable time (withing a reasonable number of interactions with the environment), without any prior knowledge on the dynamics of traffic demand. As discussed in Section 2.3, due to the combinatorial nature of placing multiple VNFs of multiple slices over multiple nodes and domains of the physical network, *both* the state space and the action space of this problem quickly explode in realistically-sized scenarios (let alone the infinite state space when considering continuous traffic demands). Therefore, the devised RL agents should be able to cope with the algorithmic complexity challenges arising from this state/action space explosion.

Chapter 3

Multi-Agent DQN with Independent Sample-Efficient Agents

3.1 Introduction

While in Chapter 2 we introduced an RL environment for dynamic slice embedding, inline with the key characteristics dictated by the B5G vision (multiple complex slices, multiple domains, end-to-end SLAs), this Chapter focuses on devising RL agents that can address the inherent complexity of such problems, stemming from the combinatorial state and action space. We make a conscious effort to slowly build up to our final algorithm, starting from small, theoretically tractable scenarios (to use as benchmarks), and attempting an informed justification of every new building block along the way (this is not something common in related literature). The main contributions of this Chapter are:

(C.1) In Section 3.2, we attempt a theoretical analysis of the scalability properties of various existing RL approaches. We start our exposition with the standard (tabular) Q-learning method, and gradually evolve to a multi-agent DQN-based scheme that will be the basis of our proposed algorithm.

(C.2) To further improve the (sample) efficiency of standard DQN agents (single or multi) we introduce two additional mechanisms that optimize how the "experience replay" is used (Section 3.2.4): (i) a priority mechanism to access the samples in

the experience replay (often referred to as a prioritized experience replay [36]); and (ii) the storage of some additional information per experience to reduce the number of computations during updates. These mechanisms aim to further speed up the convergence of our multi-agent algorithm and facilitate its practical applicability.

(C.3) Using a real dataset to drive a fairly large, multi-domain scenario, we show that the proposed multi-agent scheme reduces convergence time by orders of magnitude with minimum penalty of decision optimality, compared to standard single-agent DQN (Section 3.3.2), while it also significantly outperforms static heuristic policies by at least a 3× factor (Section 3.3.4). To our best knowledge, validation with real data is not so common in some of the close related works.

3.2 RL Agents

The goal of any Reinforcement Learning problem like the above is to devise an *agent* that gradually (but efficiently) learns to take better actions over time, eventually maximizing the rewards per round. This learning is achieved by interacting with the *environment* of the problem. We briefly describe this interaction here using some standard RL nomenclature:

-Agent: At every round t, based on the current state of the system $s_t = (c_t, d_t)$ (consisting of the current configuration and demand vectors), it must choose a new configuration vector $a_t = c_{t+1}(s_t)$, namely the new embedding of each VNF chain on the physical network (which might involve a certain number of VNF migrations). This action is taken based on a value function $Q(s, a)$ that the agent maintains, which estimates the expected (discounted) cumulative reward starting from state s and taking action a (for all s, a)[1]. Note that the expected discounted cumulative reward starting from a state $s_t = s$ and following a policy π is defined as $V^\pi(s) = \mathbb{E}_\pi\{\sum_{k=0}^{\infty} \gamma^k r_{t+k} | s_t = s\}$, where the expectation is over the transition probabilities between states, dictated by the traffic demand dynamics, and $\gamma \in [0, 1)$ is the discount factor (for larger values of γ future costs become more important).

-Environment: After the agent has chosen the next placement vector c_{t+1}, the environment "reveals" the true current demand vector d_{t+1}. This moves the state

[1]This function can initially be arbitrarily bad, and thus the initial configuration actions. Some knowledge of problem structure could help use a better initialization of the agent.

of the problem to the new state $s_{t+1} = (c_{t+1}(s_t), d_{t+1})$. It also decides the instantaneous reward r_{t+1} (a function of the old state s_t, the action a_t, and the new state s_{t+1}), according to the details of Section 2.3.

-Agent: Given the tuple $(s_t, a_t, s_{t+1}, r_{t+1})$, often referred to as an "experience", the agent *first* attempts to "improve" its decision function $Q(s, a)$, based on the recent (and/or past experiences), and then proposes a new configuration $c_{t+2}(s_{t+1})$ for the time window.

-Environment: Again reveals the new demands d_{t+2} and respective rewards, the agent will improve its value function and proceed with a new configuration proposal, and so on and so forth, until convergence.

The key features that differentiate the vast gamut of RL algorithms are (i) how the $Q(s, a)$ is encoded, (ii) how it is updated, (iii) whether in fact this actual value function is learned as an intermediate step (known as "value-based" methods) or whether action(s) probabilities are learned directly (known as "policy-based" methods), and (iv) how new actions (i.e., configuration here) are to be chosen from this function. This section attempts to introduce and motivate all the building components and choices of the RL agent(s) we propose. We choose to introduce these, step-by-step, using the previous blueprint as a guide, starting from textbook ("tabular") Q-learning for two reasons: (i) to justify which specific obstacle each extra mechanism we add is trying to improve upon, and (ii) while we cannot claim any theoretical optimality guarantees for the (final) RL scheme we propose (as is the case for any Deep RL scheme on non-toy size problems), we hope that this will make our scheme and its pros and cons more clear to the reader, as sophisticated (Deep) RL algorithms tend to be quite complex and opaque.

3.2.1 Q-learning (QL)

A "tabular" Q-learning (QL) agent maintains a table of size $|\mathcal{S}| \times |\mathcal{A}|$, for every possible state $s \in \mathcal{S}$ and action $a \in \mathcal{A}$, where each table entry captures an estimate of the corresponding Q-value [49].

Actions Selection: Actions are chosen based on this $Q(s, a)$ table according to the well known ϵ-*greedy* algorithm [49]: With probability $1 - \epsilon$ at a given state the agent picks the configuration with the maximum predicted Q value ("exploit") and with probability ϵ it picks an action randomly ("explore").

Action Improvement: Based on the reward it observes at that round it applies a stochastic approximation step [50] that aims to improve the estimate Q(s,a) *for that specific state and action only.*[2]

We give the algorithmic steps of the QL scheme in more detail in Alg. 1.

Algorithm 1 (QL) Main algorithmic steps.

Step 1 (in agent): When at state s (Def. 2.1), take an action a (Def. 2.2) with ϵ-greedy:

$$a \leftarrow \begin{cases} \text{random } a \in \mathcal{A}, & \text{with probability } \epsilon \\ \arg\max_{a \in \mathcal{A}} Q(s,a), & \text{with probability } 1-\epsilon \end{cases} \tag{3.1}$$

Step 2 (in env): Returns the next state s' and reward r (Def. 2.3)
Step 3 (in agent): Update the corresponding Q-table entry:

$$Q(s,a) \leftarrow (1-\eta)Q(s,a) + \eta(r + \gamma \max_{a' \in \mathcal{A}} Q(s',a')), \tag{3.2}$$

where η is the *learning rate* and γ the *discount factor*.
Repeat steps 1 to 3 until convergence.

Pros: The main advantage of tabular QL is that it provably converges to the optimal Q values, under mild conditions, and thus also to the optimal actions that maximize long-term rewards for every possible state. For this reason, in scenarios small enough to run Q-learning to convergence, we will use it as an "oracle" to compare other schemes against.

Cons: Due to the combinatorial nature of states and actions, serious scalability bottlenecks arise in all of the three key features of QL (decision function, action improvement, action selection). Fig. 3.1 demonstrates how the size of the state, action, and state-action spaces scale with the number of slices, highlighting the prohibitively large number of states and actions even in toy-ish scenarios (let alone realistic ones). Hence, the scalability bottlenecks are (i) *in decision function:* the size of the Q-table, which leads to slow convergence and high memory requirements (this renders QL inapplicable in scenarios with continuous traffic demands due to the infinite state space); (ii) *in action improvement:* only the value of the visited state-action pair is updated at each timestep (3.2), meaning that multiple visits

[2] Approximate Q-learning schemes such as the ones based on Deep Neural Networks that we'll introduce shortly, attempt to improve their predictions for multiple (s,a) pairs at a single step. This is often termed "generalization".

are required over *every* state-action pair for convergence (poor sample efficiency)[3]; (iii) *in action selection:* the expensive maximization operations (3.1), (3.2), over all possible configurations (combinatorial) at *every* time slot lead to impractical runtimes (slow convergence);

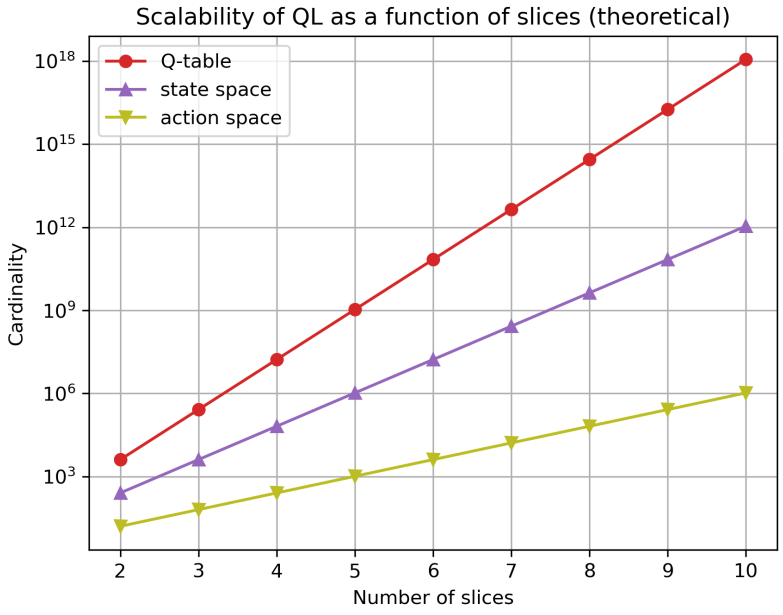


FIGURE 3.1: Number of states, actions, and state-action pairs as a function of the number of slices for "vanilla" Q-learning. This plot corresponds to a toy scenario where the physical network is similar to that of Fig. 2.1 (two domains with two nodes per domain), the slices are simple VNF chains similar to Slice 0 of Fig. 2.1 (one VNF for each domain), and the VNF demands are quantized to only 2 different levels.

3.2.2 Approximate QL - Step 1: Deep-Q Network (DQN)

A natural first step to tackle the shortcomings of Q-learning, related to state space explosion, is to replace the (potentially huge) $Q(s, a)$ table with a function $Q_\theta(s, a)$, parameterized by a set of variables (or parameters) θ, which attempts to approximate the (optimal) Q-table efficiently. While a number of different approximations can be tried here (see [49] for for a survey of methods that have been popular in earlier RL works), using a Deep Neural Network (DNN) has recently shown great promise [52], giving rise to the broad class of *Deep Reinforcement Learning.*

[3]There exist some recent theoretical results that improve the sample efficiency of tabular methods [51], but the main shortcomings of tabular methods largely remain.

A standard design is a DNN that takes as input the state s of the system (i.e., the current configuration and recent traffic demand), and has $|\mathcal{A}|$ outputs that "predict" the long term reward of every possible configuration we could choose[4]. We'll elaborate shortly on the advantages of this approach, compared to tabular, but intuitively, we are now able to use continuous values as (part of the) input as well as learn (potentially) much fewer parameters θ than the Q-table entries.

Action Selection: is performed as before, using the ϵ-greedy algorithm with $Q_\theta(s, a)$.

Action Improvement: Upon receiving a reward r from the environment the agent calculates the, so called, Temporal Difference (TD) error δ, meaning that it compares its prediction of the (long term) reward of the chosen configuration a (i.e., $Q_\theta(s, a)$) to an improved "estimate" based on the reward r just received and the predicted remaining rewards from the resulting next state s':

$$\delta = Q_\theta(s, a) - (r + \gamma \max_{a' \in \mathcal{A}} Q_\theta(s', a')) \tag{3.3}$$

The latter quantity of this equation is referred to as the *TD-target*, and is akin to semi-supervised learning as this target is used as a (weak) proxy of the true value in approximate RL. The agent then applies a gradient step on the parameters θ of the DNN (referred to as *backpropagation* in DNN lingo), attempting to slightly "correct" them toward reducing δ and improving its Q value prediction (and consequently the respective actions)[5]:

$$\theta \leftarrow \theta - \eta \nabla_\theta \delta^2 \tag{3.4}$$

While this alone resolves the state space issue, in theory, such methods can be extremely unstable and often fail to converge [53]. Instead, the recent seminal work of [45] has proposed some additional mechanisms on top of the DNN-based

[4]In other words, instead of maintaining a single output function $Q_\theta(s, a)$, as implied earlier, here the agent maintains a multiple output function $Q_\theta(s)$. This is common practice and is often advantageous [45].

[5]It is important to note here, that the Q value update step for the tabular method can also be cast as a gradient step on a linear function with features its table entries, and a parameter θ_i for each entry. Hence the two methods so far only differ in the function used, a table vs a DNN [49].

QL, coined *Deep Q Network (DQN)*, achieving remarkable success in solving tough problems ranging from video games to wireless networks [54].

DQN mechanisms. There are three main ideas, each targeting a specific shortcoming of the vanilla DNN-based approximation. We briefly describe them following.

Experience Replay Buffer: Each transition (s, a, s', r) experienced by the agent, consisting of a state, action, next-state, and reward, is stored in the replay buffer and can be used in a next timestep to update the DNN parameters. Hence, in (3.4), we don't use anymore the current experience but rather select randomly from the replay buffer. This is necessary to ensure that updates are performed based on i.i.d. samples, which is a theoretical requirement for convergence in Supervised Machine Learning [55] (it is easy to see that the sequence of states visited by the agent is highly correlated in the simple approximate QL algorithm described earlier).

Mini-batch updates: The key idea here is to use multiple random samples (experiences) from the buffer (instead of one) at each gradient step. This is a standard idea in stochastic gradient descent methods to reduce variance [55].

Target Network: Instead of using the current DNN $Q_\theta(s)$ for the calculation of the TD-target in (3.3), we use an older ("frozen") version of the network $Q_{\theta'}(s)$, that is only periodically updated. This is to remove correlations between $Q_\theta(s, a)$ and the TD-target in (3.3).

The algorithmic steps of DQN are summarized in Alg. 2, with the main differences from tabular QL highlighted.

Pros: A key advantage of DQN is that it can handle arbitrarily large state spaces, including continuous-valued ones. Another key advantage of approximation methods in general is their ability to "generalize" each experience: In "tabular" QL, an experience (s, a, r, s') is used to improve the Q value estimate only for that specific state-action pair (s, a). Instead, in DQN this experience is used to change the weights of the approximation, affecting (and hopefully improving) the predicted Q value for a large number of other (s, a) pairs with similar "features". Thereof stems the ability to handle much larger problems effectively (together with the extra DQN mechanisms for stability). More specifically, DQN (partly) tackles shortcomings (i) and (ii) of QL as listed in Section 3.2.1. The theoretical

Algorithm 2 (DQN) Main algorithmic steps. The important differences from QL (Alg. 1) are highlighted in red.

Step 1 (in agent): When at state s, take an ϵ-greedy action:

$$a \leftarrow \begin{cases} \text{random } a \in \mathcal{A}, & \text{with probability } \epsilon \\ \arg\max_{a \in \mathcal{A}} Q_\theta(s,a), & \text{with probability } 1-\epsilon \end{cases} \tag{3.5}$$

Step 2 (in env): Returns the next state s' and reward r
Step 3 (in agent): store transition (s, a, s', r) in the replay buffer $\mathcal{B}$
Step 4 (in agent): copy the policy network parameters θ to the target network θ' (only every T timesteps)
Step 5 (in agent): pick M samples randomly from replay buffer and calculate δ_i for each sample i:

$$\delta_i = Q_\theta(s_i, a_i) - (r_i + \gamma \max_{a'_i \in \mathcal{A}} Q_{\theta'}(s'_i, a'_i)) \tag{3.6}$$

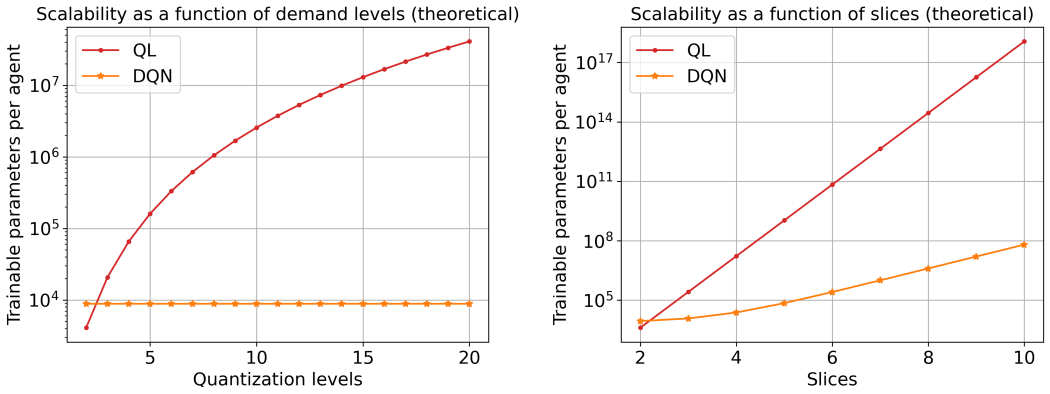
Then, perform a gradient step:

$$\theta \leftarrow \theta - \eta \nabla_\theta \mathbb{E}_{i \sim U(\mathcal{B})}[\delta_i^2], \tag{3.7}$$

where $i \sim U(\mathcal{B})$ denotes that experiences have been sampled uniformly at random from the replay buffer.
Repeat steps 1 to 5 till termination criterion.

advantage of DQN compared to QL as the state space grows larger is highlighted in Fig. 3.2(a), which confirms the fixed memory requirements regardless of the state space explosion (the ability of DQN to obtain good quality policies will be investigated later, in the simulation section).

Cons: DQN still faces two hurdles. The first is the combinatorial action space $\mathcal{A}$, which materializes in two ways: (i) the fanout of the $Q_\theta(s)$ DNN is equal to the size of the action space, meaning that the number of parameters explodes in large scale scenarios (see Fig. 3.2(b)); (ii) as it can be observed in (3.5) and (3.6), there is a max operation to be performed over this large action space twice per round (in fact, in (3.6) as many times as the size of the mini-batch). The former leads to delays due to training a large number of parameters (and high memory requirements), the latter leads to delays due to significantly more flops per round. The second hurdle is that the full potential of the replay buffer toward improving sample efficiency has not been harnessed yet (the uniformly random sampling of experiences from the buffer is often suboptimal).



(a) Varying number of demand levels (2 slices)

(b) Varying number of slices (2 levels of demand)

FIGURE 3.2: Scalability comparison QL vs DQN. The above plots correspond to a toy scenario where the physical network is similar to that of Fig. 2.1, the slices are simple VNF chains similar to Slice 0 of Fig. 2.1, and the VNF demands are quantized. The DQN's DNNs have 3 hidden layers with 60 neurons per layer (this setup performed well in both large and small scale scenarios in the simulation section).

3.2.3 Approximate QL - Step 2: independent DQN agents (iDQN)

While DQN can handle arbitrarily large state space $\mathcal{S}$, as explained earlier, the combinatorial action space $\mathcal{A}$ still hinders its scalability. In this section, we introduce the concept of *independent DQN agents* to tackle the problems stemming from the action space complexity, while maintaining the state space advantages of standard DQN. The idea is to decompose the (large) action space into (much) smaller action subspaces as follows:

Action space decomposition: Let us consider again the toy example of Fig. 2.2. Instead of having one agent choosing the entire configuration for every VNF of every slice at each round, i.e. $c = (c_0^0, c_1^0, c_0^1, c_1^1, c_2^1)$, and learning a $Q_\theta(s, c)$ for every possible configuration, we assume one separate agent (n, k) responsible for each VNF $n \in \mathcal{N}_k$ of slice $k \in \mathcal{K}$: i.e. agent (n, k) is responsible for c_n^k only, and attempts to learn/predict a rewarding next configuration for that VNF only, given the same (full) state s, but a smaller DNN $Q_{\theta_{nk}}(s, c_n^k)$.

Remarks: It is important to stress here that, while each such agent could be a separate, possibly distributed agent, residing with/close to the VNF it controls, this is not necessary and it is orthogonal to our work. The main idea here, is to decompose the DNN network (solving the large fanout problem) and the max

operations of (3.5) and (3.6), in order to largely reduce the computational complexity, even if the agents are still one piece of code residing in the same location. What is more, one could consider, w.l.o.g., other granularities of decomposition (e.g. per slice, per slice and domain, etc.).

Hence, the action selection and policy improvement are the same as in the DQN algorithm, except that now each agent acts independently and in parallel. The fully independent flavor that we employ here has been considered both for "tabular" QL in [56], and for approximate QL in [57]. Note that even though there is no explicit coordination between agents, they still get sufficient signals to co-operate harmoniously via the common state s (that they all observe) and the reward per round (which depends on the collective action of all agents). Additional levels of coordination can be explored [58] but we defer them for future work.

The algorithm's steps, with the differences from single agent DQN highlighted, are shown in Alg. 3.

Pros: The action space $\mathcal{A}^{nk}$ per agent (n, k) is orders of magnitude lower compared to the original action space $\mathcal{A}$ of DQN. Considering one agent per VNF, the action is not combinatorial anymore (the number of actions is equal to the number of candidate hosts where the VNF can be assigned on). This translates to (i) computationally cheaper max operations per agent (faster convergence); and (ii) smaller DNN fanouts (better sample efficiency, lower memory requirements). The latter is confirmed by Fig. 3.3, which demonstrates that the number of trainable parameters per iDQN agent remains constant with the number of slices, while both QL and DQN requirements quickly explode.

Cons: First, the use of independent agents can potentially lead to stability problems as the environment becomes non-stationary. This is because each agent conceives the rest of the agents as part of the environment (due to their independence) but their policies change over time (each of them tries to improve its policy). Hence, older experiences stored in the buffer may quickly become outdated and negatively affect policy updates. However, by using a small experience replay buffer (that retains the most recent experiences) and by carefully choosing the learning and exploration rates (so that agents don't change their policies very fast), this problem can be mitigated. A second shortcoming is that there is still much room for improvement regarding sample efficiency and convergence speed.

Algorithm 3 (iDQN) Main algorithmic steps. The important differences with respect to DQN (Alg. 2) are highlighted in red.

iDQN algorithm

Multiagent scheme:

one DQN agent (n,k) per VNF $n \in \mathcal{N}_k$ of slice $k \in \mathcal{K}$

Step 1: (in agent) each agent (n,k) takes an ϵ-greedy action:

$$a^{nk} \leftarrow \begin{cases} \text{random } a^{nk} \in \mathcal{A}^{nk}, & \text{with probability } \epsilon \\ \arg\max_{a^{nk} \in \mathcal{A}^{nk}} Q_{\theta_{nk}}(s, a^{nk}), & \text{with probability } 1-\epsilon \end{cases}$$

Then, the collective action is:

$$a = (a^{00}, ..., a^{N_K K})$$

Step 2 (in env): Returns the next state s' and reward r

Step 3: (in agent) $\forall (n,k)$ store transition (s, a^{nk}, s', r)

Step 4 (in agent): $\forall (n,k)$ copy the policy network parameters θ_{nk} to the target network θ'_{nk} (only every T timesteps)

Step 5 (in agent): $\forall (n,k)$ pick M samples randomly from replay buffer and calculate δ_i for each sample i:

$$\delta_i = Q_{\theta_{nk}}(s_i, a_i^{nk}) - (r_i + \gamma \max_{(a_i^{nk})' \in \mathcal{A}^{nk}} Q_{\theta'_{nk}}(s'_i, (a_i^{nk})')) \tag{3.8}$$

Then, perform a gradient step:

$$\theta_{nk} \leftarrow \theta_{nk} - \eta \nabla_{\theta_{nk}} \mathbb{E}_{i \sim U(\mathcal{B}_{nk})}[\delta_i^2], \tag{3.9}$$

Repeat steps 1 to 5 till termination criterion.

As described in Section 3.2.2, the full potential of the replay buffer toward improving sample efficiency has not been harnessed yet. Lastly, despite the massive improvement offered by iDQN, the max operations could still become expensive for very large scenarios, large minibatch sizes (the TD-error must be calculated for *every* sample), or when a coarser action decomposition is considered (e.g. one agent per slice)[6].

[6]This kind of decomposition (one DQN agent per slice) becomes necessary in scenarios where we try to optimize both VNF and VL placement. This is because the set of possible physical paths where a VL can be mapped to depends on the placement of the VNFs it interconnects (it is not possible to treat them independently in parallel).

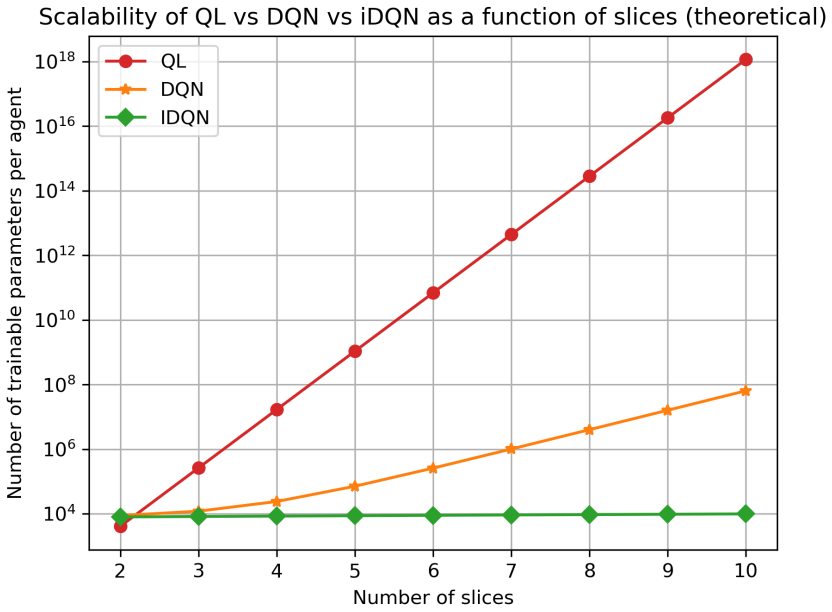


FIGURE 3.3: Number of trainable parameters for QL, DQN, and iDQN as a function of the number of slices. This plot corresponds to a toy scenario where the physical network is similar to that of Fig. 2.1, the slices are simple VNF chains similar to slice 0 of Fig. 2.1, and and the VNF demands are quantized to only 2 different levels. Moreover, DQN and iDQN agents have the same number and size of hidden layers (3 hidden layers with 60 neurons per layer).

3.2.4 DQN+/iDQN+

As we saw, iDQN manages to resolve the first major hurdle that DQN faced (for the problem at hand), the exploding action space size, by introducing a multi-agent decomposition of the action space. Together with the DNN-based encoding of the state space, the algorithm can now smoothly scale up to considerably large(r) problems. The last obstacles remaining are the experience replay buffer suboptimalities, as well as the potentially expensive computation of the TD-error in large scale scenarios (requires a max operation for *every* sample of the mini-batch). The above are both related to the action improvement mechanism, and in order to tackle them we propose respectively, (i) the use of a prioritized experience replay, to intelligently pick minibatches from the replay buffer in a way that improves sample efficiency; (ii) a "lazy" computation of the TD-target to further improve convergence speed. The above speed up mechanisms can be applied on top of either DQN or iDQN (we refer to these schemes as DQN+ and iDQN+

respectively).

Prioritized experience replay. We have observed that as the action space in our problem grows larger, actions with similar effect are over-represented in the replay buffer, while potentially more effective actions are under-represented (slowing down convergence). For example, due to the combinatorial nature of the action space, the subset of actions that utilize just a few of the available physical nodes (among which lie the optimal actions, during periods when the majority of VNFs have low traffic) is much smaller compared to the subset that assigns at least one VNF to each physical node; thus it is very rare to randomly explore an action from the former subset with the ϵ-greedy policy. As a result, due to the random selection of minibatches from the replay buffer, the over-represented transitions are replayed more frequently and their Q-value prediction by the DNN quickly stabilizes around a (suboptimal) value, meaning that the associated TD-error becomes small (slow learning). To this end, we employ a prioritized experience replay [36], which prioritizes the transitions with a larger TD-error to boost sample efficiency. The modification introduced in the DQN algorithm to incorporate this mechanism is given in Alg. 4.

Algorithm 4 (DQN+) Modification 1: Prioritized experience replay. The modified steps with respect to DQN (Alg.2) are given below.

Step 5 (in agent): Select M samples from the replay buffer, with probability $P(i)$ for each of the N experiences:

$$P(i) = p_i^{\alpha^{\text{rep}}} / \sum_{j=1}^{N} p_j^{\alpha^{\text{rep}}}, \tag{3.10}$$

where $p_i = |\delta_i| + \epsilon$, with ϵ being a small positive constant that prevents p_i from going to zero. Then, perform a gradient step:

$$\theta \leftarrow \theta - \eta \nabla_\theta \mathbb{E}_{i \sim P(\mathcal{B})}[(\frac{w_i^{rep}}{\max_j w_j^{rep}} \delta_i)^2], \tag{3.11}$$

$$\text{where } w_i^{rep} = (N \cdot P(i))^{-\beta^{\text{rep}}} \tag{3.12}$$

The performance of prioritized experience replay is affected by two important hyperparameters, α^{rep}, which determines the amount of prioritization ($\alpha^{\text{rep}} = 0$ leads to uniform sampling while $\alpha^{\text{rep}} = 1$ to full prioritization), and β^{rep}, which

similarly determines the amount of compensation applied by weighted importance sampling to balance the bias introduced by prioritization.

Lazy computation of TD-target. The maximization operation in (3.6), required for *every* sample of the minibatch, may become computationally expensive in large scale scenarios. In order to reduce the number of such computations, we introduce a second mechanism (modification), that stores the current Q value estimate of s' along with the visited (s, a, s', r) tuple and uses it as (part of) the TD-target every time the corresponding experience is sampled. The detailed mechanism can be seen in Alg. 5.

Algorithm 5 (DQN+) Modification 2: Lazy TD-target computation. The modified steps with respect to DQN (Alg.2) are given below.

Step 3 (in agent): calculate Q_{next}:

$$Q^{\text{next}} = \max_{a' \in \mathcal{A}} Q_{\theta'}(s', a') \tag{3.13}$$

and store $(s, a, s', r, Q^{\text{next}})$ in the replay buffer.
Step 5 (in agent): the TD-error for all M minibatch samples is now computed using the stored Q_{next} values:

$$\delta_i = Q_\theta(s_i, a_i) - (r_i + \gamma Q_i^{\text{next}}) \tag{3.14}$$

This trick offers important real time gains, as DQN+ performs M times less computations per timestep, compared to DQN, for TD-target calculations (in DQN+, the max operation of (3.13) is applied only on the visited transition, while in DQN on each one of the M samples). The gain of the "lazy" TD-target computation mechanism is confirmed by Table 3.1, which outlines the simulation results for a toy-ish slicing scenario (256 possible configurations). Observe that the sample efficiency may be mildly deteriorated compared to "vanilla" DQN, as expected, but convergence speed (in real time) is improved by roughly 40% ("lazy" DQN fully converges in 1025 sec, while "vanilla" DQN in 1462 sec). This gain should further increase in scenarios with larger action spaces, as the TD-target computation would constitute a larger part of the total runtime (due to more expensive max operations).

TABLE 3.1: Convergence speed comparison in scenario with 256 actions

Convergence (%)	Time slot		Real time (s)	
	DQN	"lazy" DQN	DQN	"lazy" DQN
60	18493	30829	574	663
80	32157	35304	1005	757
100	46937	47515	1462	1025

3.3 Simulation Results

So far, we have analyzed the theoretical scalability of the various solutions, using toy scenarios, and computational complexity arguments about the state and action space explosion. In this section, we will use simulation to see these phenomena in practice. To this end, we will use both synthetic traffic demands, as well as a real dataset [35] that has been used before as a benchmark in the wireless community [59].

Specifically, the three main goals of this section are: (i) to establish that Q-learning based approximate algorithms, either single-agent or multi-agent, are able to obtain close to optimal solutions (i.e. ones found by exact Q-learning), yet with much higher convergence speed as the problem size increases; (ii) to quantify the convergence speed gains offered by the speedup heuristics proposed in section 3.2.4 on top of of DQN and iDQN algorithms; and (iii) to validate our proposed enhanced multiagent solution (iDQN+) in a realistic large scale setup.

3.3.1 Simulation Setup

Algorithms. We summarize here the main algorithms we will test, as well as their shorthand we'll use to refer to them, hereafter:

- **QL** the tabular (theoretically optimal) RL algorithm of Section 3.2.1.
- **DQN** the single-agent approximate RL algorithm of Section 3.2.2.
- **iDQN** the multi-agent approximate RL algorithm of Section 3.2.3.
- **DQN+/iDQN+** these variants are the above DQN/iDQN, but with all the speedup heuristics we have proposed in Section 3.2.4.

DNN architectures: We choose simple and relatively small DNN architectures as Q-function approximators. On the one hand to demonstrate that the methodology is generic enough to work without the need for specialized, large architectures. On the other hand, because they offer better sample efficiency. To this end, all DQN-based agents utilize multilayer perceptron DNNs with 3 hidden layers and 60 neurons per layer (this architecture performed well in the whole range of tested scenario sizes). Note that multilayer perceptrons have been also employed for similar slicing problems in related work literature, e.g. in [30].

Hyperparameters: In approximate RL schemes we set the replay buffer size to 5000, the target update period to 500, the minibatch size to 32, and the learning rate to 10^{-3} (these are the typical values used in [45], adapted to our problem when necessary). In Q-learning, we set the learning rate to 10^{-1}, which is a commonly used value in related work literature [49]. Finally, we set the discount factor to $\gamma = 0.9$. In general, γ relates to the reconfiguration cost and how far ahead the agent is willing to look in order to amortize the immediate cost, and a value of 0.9 allows the agent to look "roughly" 10 timesteps ahead without making convergence too slow (the higher the γ the slower the convergence). This value is large enough to provide good policies even in scenarios involving important reconfiguration costs. Note that all the above parameters performed well in a variety of tested scenarios.

Traffic details. We consider two types of traffic in our simulations, (i) synthetic Markov traffic (quantized demands); (ii) real traffic from the well known Milano dataset (continuous demands) [35]. The former is used in small toy scenarios, so that tabular methods are applicable (not the case with continuous), and an optimal policy benchmark can be derived (in reasonable time). The latter is employed later on to validate our practical/scalable approximate schemes in large setups driven by data entailing common characteristics of real demands (e.g. diurnal patterns and other non-stationarities).

Traffic type 1 (synthetic): In these first toy scenarios, we use the simplest possible (non-IID) traffic model, driven by a 2-state Markov chain, able to capture bursts of traffic (with configurable mean duration), with (also configurable) silence periods

in between. W.l.o.g. we use the following chain for every VNF: [7]

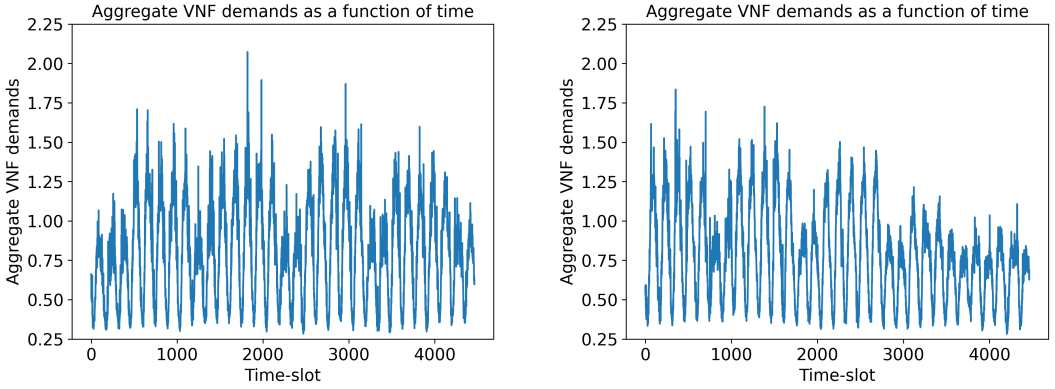
$$\mathbb{P} = \begin{bmatrix} 0.98 & 0.02 \\ 0.02 & 0.98 \end{bmatrix} \tag{3.15}$$

This kind of simple traffic model is helpful in better illustrating the performance of optimal dynamic actions against static heuristics.

Traffic type 2 (real trace): The Milano dataset is a real traffic open dataset provided by Telecom Italia [35]. It contains timeseries of 10K base stations located in the city of Milan and the Province of Trentino. Each timeseries has 8928 samples, with one sample taken every 10 minutes, meaning that the total real time duration of the dataset is 2 months. Each sample consists of different Call Detail Records (CDRs), e.g. the number of calls, SMSs, internet connections etc. Here we use only the internet connections CDR, and in particular we map the average arrival rate of jobs for each VNF to the normalized timeseries of a different base station. Moreover, to make simulations even more realistic, we choose highly correlated timeseries for VNFs that belong to the same slice. The first half of the datapoints are used for the training of the algorithms and the other half for testing, meaning that each episode consists of 4464 timeslots. It is the case that such mobile traffic datasets usually contain a limited number of samples. Thus, it is important to confirm that our agent can perform well even when encountering previously unseen states (testing episode). In Fig. 3.4 we depict an example of aggregate demand timeseries from the Milano dataset (training and testing episodes). The key observations are that (i) there are some patterns emerging in traffic due to the day/night and weekend human activities, but also plenty of unpredictable ("noisy") patterns (these plots underline the need for algorithms, like RL, that are able to deal with non-IID patterns); (ii) the magnitude of demand decreases in the second half of the testing episode in this example, so indeed the training and testing datasets may differ.

Service Level Agreements. We consider the end-to-end delay KPI, which can be calculated with our model as described in Section 2.2.3, based on (2.3). The penalty for SLA violations is incurred according to (2.5) of Section 2.3.

[7] Note that even with this simple model the burst periods do not coincide for all VNFs, and considerable complexity of states can already be produced by the environment with a few VNFs and nodes.



(a) Training episode

(b) Testing episode

FIGURE 3.4: Example of demand timeseries from the Milano dataset. The aggregate demand in the y axis corresponds to the sum of the (normalized) demands of 4 VNFs and each time-slot in the x axis corresponds to a 10 minutes period of the trace.

3.3.2 Part I: Scalability of RL schemes

In this part, our aim is to address the first main goal of the results section, namely to ground the cost performance of approximate RL schemes (DQN, iDQN) with respect to optimal policies obtained by QL and examine how the increase of problem size affects the convergence speed of the above schemes.

System Setup. To ensure the tabular Q-learning is tractable we focus on a single domain physical network and two different setups, a small one and a larger one (which we will refer to as Scenario 1 and 2 respectively), differing in the number of physical nodes and hosted slices. Regarding traffic demands, we use the synthetic Markov model of (3.15).

Training. Each algorithm is trained over 10 individual runs with different initial random seeds, meaning that the randomly initialized parameters of the Q-function (either with "tabular" or DNN representation) and the random exploration of actions differ among runs. The exploration rate during training is constant and equal to $\epsilon = 0.1$.

Scenario 1 (Fig. 3.5(a)): The system consists of 2 servers that host 4 slices, corresponding to 256 states, 16 actions, and 4096 state-action pairs. The cost as a function of time slot during training (averaged over the 10 runs), is depicted in Fig. 3.5(a) for QL, DQN, and iDQN algorithms. Two important remarks for

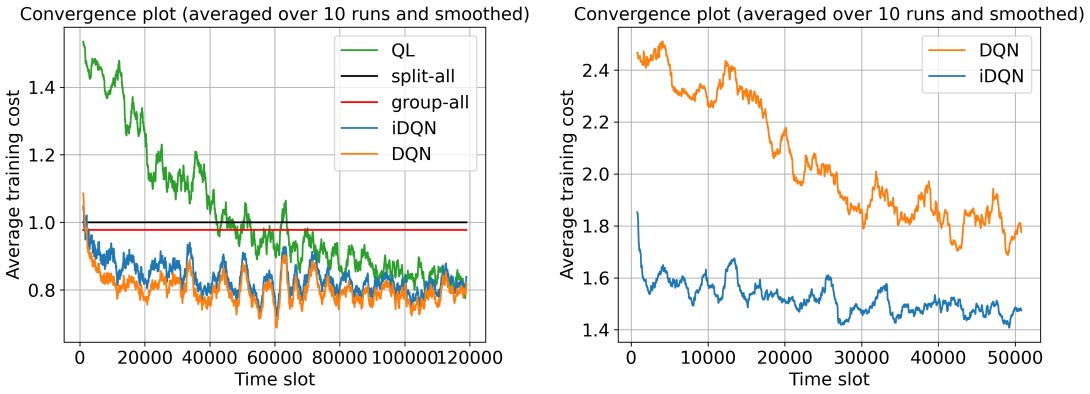
this cost metric, which we will be using throughout the simulation section, are that i) the lower the cost the better; and ii) this cost includes also the cost of the random exploration actions during training which is constant due to the constant exploration rate. The two main observation from Fig. 3.5(a) we after are:

(i) *Near-Optimality:* that the two approximate algorithms, DQN and iDQN, are able to achieve similar cost to the (optimal) QL policy. Indeed, in Fig. 3.5(a), when the QL algorithm has finally converged to an optimal policy (at timeslot 120000), it achieves roughly the same cost as DQN and iDQN (this has been confirmed for a variety of other small scenarios as well).

(ii) *Convergence Speed:* that the two algorithms, even in this very simple scenario (hence, not so computationally challenging for tabular Q-learning) still reduce convergence speed, as expected. In Fig. 3.5(a), QL takes roughly an order of magnitude more iterations to converge compared to approximate RL schemes (the performance deficit of QL only increases in larger scenarios). This gain is mainly attributed to the inherent ability of the DNN to be more sample efficient than tabular methods, and quickly generalize. This is corroborated by the fact that the number of trainable parameters for the DNN in this small scenario (the weights) is comparable to the number of trainable parameters (the Q table entries) of QL, and therefore the speed gain cannot be attributed, in this small scenario, to having to learn for example much fewer parameters. Note that the convergence speed performance of DQN and iDQN is similar, since the action space is still quite small (the advantage of iDQN will become evident in Scenario 2, to be introduced shortly).

Finally, as a "sanity check" to ensure that the scenario is not trivial, despite its simplicity, we plot the performance of two simple *static* heuristic policies: *group-all*, that aims to merely minimize the cost of active nodes by grouping all VNFs in one server, and *split-all*, that tries to minimize SLA violations by distributing VNFs equally to both servers. Given that such policies can in fact be optimal for a subset of time slots [8], we want to make sure that an optimal policy (and respective cost) outperforms both (i.e., there is a "non-trivial" policy to be learned, even in this small scenario). Indeed, looking at Fig. 3.5(a), the dynamic policies obtained by RL demonstrate roughly 20% lower cost compared to simple static

[8]The group-all policy is optimal when the traffic demand is low (avoids unnecessary node usage). The split-all policy is optimal when the traffic demand is high (avoids SLA violations). We will revisit these "extreme" policies and how optimal policies learn to live in between such extremes, in more detail, in the data-driven scenarios to follow.



(a) Scenario 1 (b) Scenario 2

FIGURE 3.5: Convergence plots for two scenarios of different size (Markov traffic). On the left, Scenario 1 is a small toy-scenario with 256 states and 16 actions, while on the right, Scenario 2 has 279936 states and 2187 actions.

heuristics (this gain significantly increases in larger setups, as we will see later on in Section 3.3.4).

Take-away message 1: *DQN based algorithms (either single or multi agent) can obtain good quality solutions in orders of magnitude fewer iterations than vanilla QL.*

Scenario 2 (Fig. 3.5(b)): The above experiment serves to confirm the theoretical advantages of the DQN agent of Section 3.2.2: near-optimality with better convergence speed. Nevertheless, in this very small scenario, action space complexity is not yet large enough to show any advantages of iDQN over DQN. To test this conjecture, we consider now a larger scenario with a physical network that consists of 3 servers and hosts 7 slices (279936 states, 2187 actions, $\sim 6 \cdot 10^8$ state-action pairs)[9]. While this is still not a very large scenario, in practice Q-learning already collapses due to memory requirements; we therefore omit this scheme and compare iDQN only with DQN.

The convergence plot of Fig. 3.5(b) depicts only the first 50000 training steps, in order to facilitate comparisons up until iDQN's convergence (DQN converges also to a similar cost policy, but much later). To better quantify the convergence speed gain, we give the full results of the training phase in Table 3.2, which indicate that

[9]Note that, while this sounds like a "slighly" larger scenario only, the number of possible actions has already reached 2187, compared to 16 in scenario 1. Note also that DQN algorithms start becoming extremely sluggish, as we will see, even for this action space size. We also remind that traffic demands in scenario 2 are Markov (Traffic type 1, Section 3.3.1), similar to scenario 1.

TABLE 3.2: Convergence speed comparison

Convergence (%)	DQN	iDQN
78	148857	**919**
84	297143	**963**
89	593224	**1198**
95	593311	**13620**
100	708700	**25956**

iDQN fully converges 27 times faster than DQN (it also reaches 44 times faster the 95% of convergence). This is reasonable, as the action space has quickly grown larger in this scenario due its combinatorial nature. The reasons for the much slower convergence of DQN, as we have discussed in Section 3.2.2, are mainly: (i) the exploding fanout (the output layer of DQN dominates the number of trainable parameters leading to a total of 142000 parameters, as opposed to 8400 parameters per iDQN agent, despite that we have set the same number of hidden layers and neurons therein for both schemes); (ii) the computational complexity of the max operations (3.5) and (3.6) (for DQN it is exponential V^K ,while for iDQN it is linear $V \cdot K$, where V the number of servers and K the number of slices).[10]

Take-away message 2: *Even for mildly realistic size scenarios, DQN quickly collapses in terms of convergence speed.*

3.3.3 Part II: Performance gains of DQN+/iDQN+

Having established that approximate schemes can increase convergence speed without significantly impacting the quality of the obtained policy, here we aim to validate that the speed up heuristics proposed in Section 3.2.4 can further boost convergence speed in a realistic setup. We will also introduce real traffic data to drive the demand, as well as a full-fledged multi-domain physical network, coupled with the end-to-end queuing delay SLAs to further induce realism into our scenarios. We will first test the prioritized experience replay mechanism, and the impact of hyperparameters α^{rep}, β^{rep} (where α^{rep} and β^{rep} roughly control the amount of prioritization and importance sampling applied respectively, see Section 3.2.4 for the exact definition). We will then proceed to compare the convergence speed of

[10]Even if we could perhaps consider a different DNN architecture for DQN in order to avoid (i), e.g., use as an input the state-action pair and output the corresponding Q-value estimate, (ii) is an inherent problem that would remain.

approximate RL schemes (DQN/iDQN), with and without the speed up heuristics. Note that since the demands take continuous values (traffic type 2, see Section 3.3.1), the state space has infinite size and only approximate algorithms can be directly employed.

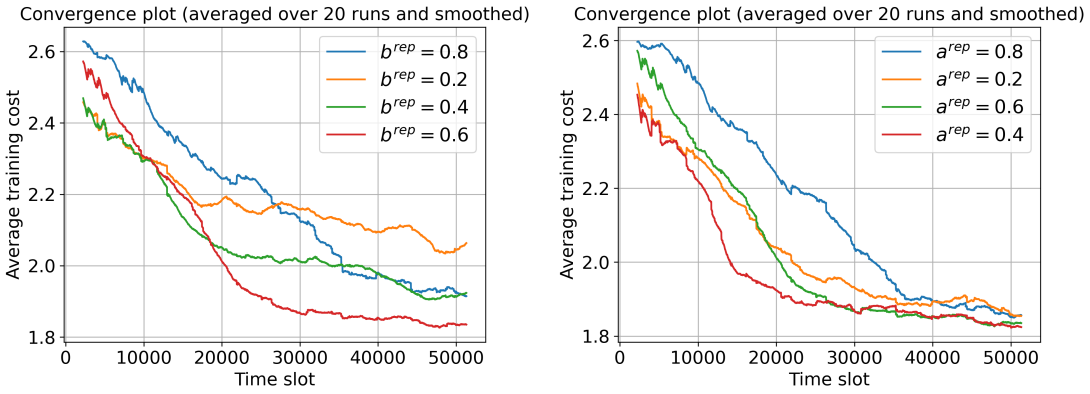
System Setup. The physical network consists of 2 domains, each of them comprising 2 nodes (servers) respectively. On top of it there are 4 slices (simple VNF chains) with 2 VNFs each (one VNF per domain). We match real traffic to each VNF of each slice, from the Milano dataset [35], as explained in Section 3.3.1. For simplicity, and without loss of generality, we assume that the demands of VLs are all zero.

Training. The training procedure is similar to Part I, only that now we execute 20 individual runs instead of 10 to increase the accuracy of the results (in Part I the performance difference between algorithms was orders of magnitude, while here the tested algorithms perform much closer).

Sensitivity Analysis. In the paper where prioritized replay was introduced [36], the corresponding hyperparameters were set to $\alpha^{rep} = 0.6$, $\beta^{rep} = 0.4$ (for the proportional variant). However, their optimal values are problem dependent. In order to adjust them for DQN+ in our setup we performed a coarse grid search, with the best performing values being $\alpha^{rep} = 0.4$, $\beta^{rep} = 0.6$. We give two representative convergence plots in Fig. 3.6. An important observation is that these parameters can affect both the sample efficiency and the quality of the obtained policy. In Fig. 3.6(a), a low b^{rep} leads to suboptimal policies (inadequate importance sampling) and a high b^{rep} to very slow convergence (excessive importance sampling); $b^{rep} = 0.6$ demonstrates roughly 11% better cost performance than $b^{rep} = 0.2$. In Fig. 3.6(b), varying a^{rep} affects mostly the sample efficiency (roughly up to a 1.6 factor speed improvement between the best and worst parameter values). Note that repeating the same analysis for iDQN resulted in similar findings, while these parameters performed well in a variety of scenarios. Thus, we use the same hyperparameters in the remainder of the section.

Take-away message 3: *Hyperparameter tuning can significantly affect the performance of prioritized experience replay.*

Impact on DQN/iDQN. In Fig. 3.7, we compare the performance of vanilla DQN/iDQN algorithms with respect to their DQN+/iDQN+ counterparts. There



(a) Sensitivity of β^{rep} ($\alpha^{rep} = 0.6$)

(b) Sensitivity of α^{rep} ($\beta^{rep} = 0.6$)

FIGURE 3.6: Sensitivity analysis of prioritized experience replay's hyperparameters for DQN+ in a real traffic scenario. Convergence plots for (a) varying β^{rep}; (b) varying α^{rep}.

are 4 main observations to take away, (i) iDQN is again confirmed to converge faster than DQN (due to the additional approximation in action space) (ii) the speedup heuristics of DQN+/iDQN+ improve their convergence speed compared to their vanilla counterparts; (iii) the speed improvement for DQN is much larger than the speed improvement for iDQN, due to its larger action space (the gain for iDQN is expected to become more prominent in larger scenarios); (iv) iDQN+ is the fastest among the tested algorithms (but only slightly faster than DQN+).

To better quantify the speed gain, we outline the full simulation results in Table 3.3, which indicates the timestep when the average cost of each algorithm went below a specified threshold values. So, Table 3.3 highlights that DQN and iDQN converge 7.8 and 0.5 times slower than iDQN+ respectively.

Take-away message 4: *the proposed speedup heuristics on top of DQN/iDQN offer significant convergence speed gains.*

3.3.4 Part III: Validation of iDQN+ in large scale scenario

We have thus far confirmed the advantages of the speedup mechanisms for both DQN schemes in relatively realistic scenarios. In this last set of experiments, we go one final step further increasing the action complexity (in fact beyond what single agent DQN can handle). Our goal is to show that this final scheme can indeed handle rather large scenarios and scale gracefully, while also seeing higher

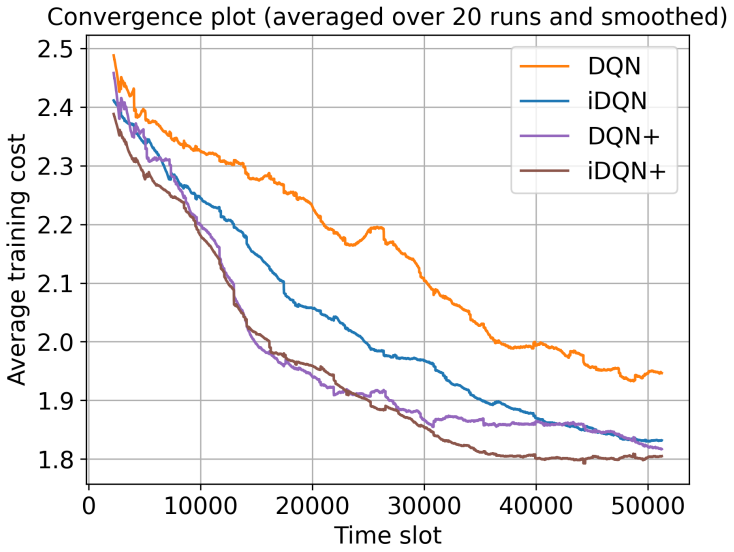


FIGURE 3.7: Convergence plot for DQN, iDQN, DQN+, iDQN+ in a real traffic scenario (after fine-tuning the prioritized experience replay hyperparameters).

TABLE 3.3: Convergence speed comparison

Average cost threshold	DQN	DQN+	iDQN	iDQN+
2.3	13832	7384	6731	**4577**
2.2	21044	**9913**	12974	9538
2.1	30233	12900	17440	**12444**
2.0	36711	**14857**	24676	16149
1.9	107555	26975	35159	**24973**
1.85	236139	44433	45361	**30465**

advantages from the smart replay buffer even for iDQN, given the larger problem size. Moreover, while the results of the previous section (3.3.3) are promising indeed, we have no hope whatsoever of knowing the optimal policy (since tabular Q-learning is impossible to run), in order to get some insights as to how far from the optimal cost we might be. Instead, in this last scenario, we go back to our static heuristics (introduced in Section 3.3.2), as another "sanity check", to ensure that we outperform them.

System Setup. The physical network consists of two technological domains, comprising 9 and 3 servers respectively. On top of it there are 10 slices (simple VNF chains) with 2 VNFs each (one VNF per domain). This results to $2 \cdot 10^{14}$ possible actions for a DQN agent. In contrast, each of the iDQN agents associated

to one of the VNFs, has only only 9 or 3 possible actions (depending on the domain). Similarly to Part II, we use real traffic VNF demands (traffic type 2, Section 3.3.1).

Training. The training procedure is similar to Part II.

Evaluation. Since this last part focuses both on convergence speed and cost performance (part II focused on the convergence speed gains of the proposed speedup heuristics), all the obtained policies are evaluated over 1 episode of the training and testing datasets (we just rollout each policy and record the average cost achieved). We evaluate the algorithms in both datasets in order to confirm that they can perform well even when applied in previously unseen states.

Convergence speed. The convergence plot of Fig. 3.8 provides a comparison between iDQN and iDQN+. There are two key observations: (i) iDQN+ is again verified to be more sample efficient than iDQN, as it achieves lower cost in the same amount of training steps (this is a standard metric in ML [55]); (ii) convergence time remains reasonable, despite the significant increase in the problem size compared to Part II (the number of timesteps to reach convergence has only doubled compared to Fig. 3.7, while the original action space has increased from 256 to $2 \cdot 10^{14}$ actions).

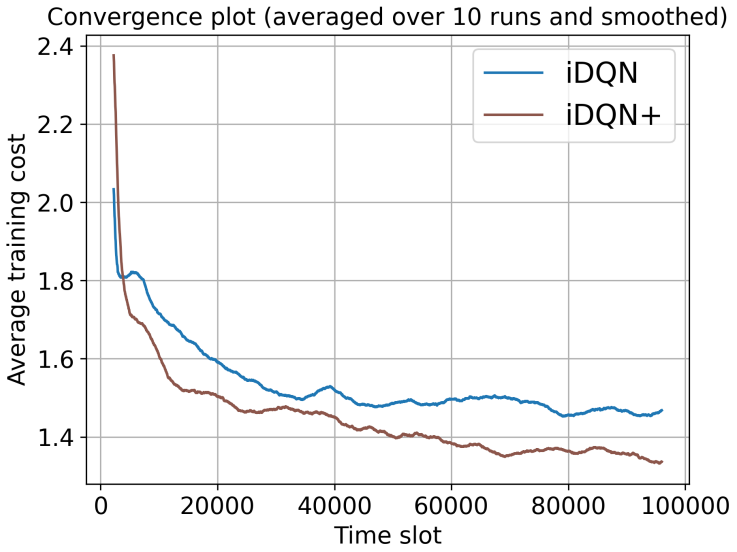
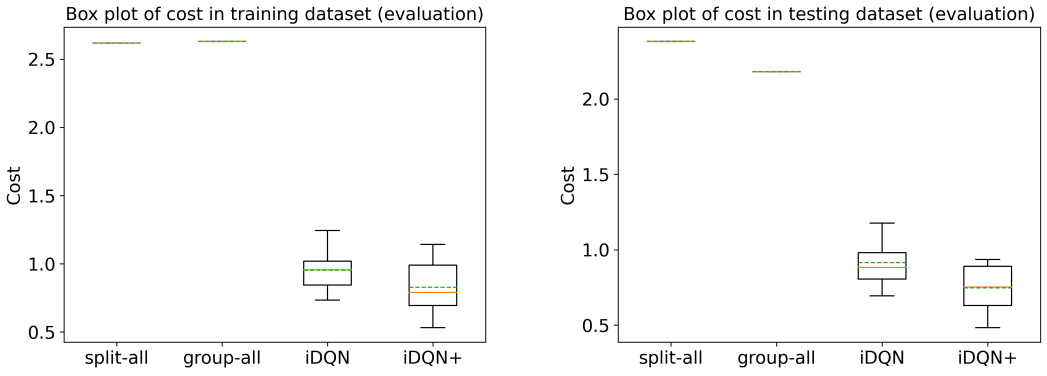


FIGURE 3.8: Convergence speed comparison between iDQN and iDQN+ (large-scale real traffic scenario).



(a) Training dataset

(b) Testing dataset

FIGURE 3.9: Cost performance evaluation. Box plot depicting the distribution of the cost achieved by each algorithm in the training and testing datasets (large-scale real traffic scenario).

Cost performance. The results of evaluating each of the obtained policies in the training and testing datasets are given in the box plots of Fig. 3.9(a) and Fig. 3.9(b) respectively, which compare iDQN, iDQN+, and the static heuristic policies group-all and split-all (introduced in Section 3.3.2). The main observations are: (i) in the training dataset iDQN+ on average outperforms iDQN by 15.2% and the static policies by a factor of 3.16; (ii) in the testing dataset iDQN+ on average outperforms iDQN by 22% and the static policies by a factor of 2.92 (the performance of iDQN+ doesn't deteriorate significantly in the testing dataset); (iii) even the worst policies obtained by iDQN (with or without the speed up extensions) in all 10 runs perform much better than the static baselines; (iv) the performance gain of iDQN+ is more prominent in this larger scenario, compared to Part II;

Analysis of obtained policies. To better understand where the lower cost of the iDQN+ policies originates from, we provide Fig. 3.10, which highlights the configurations used by different policies and the corresponding costs *per time slot*. In Fig. 3.10(a), the number of active nodes as a function of time slot is depicted (this is an important characteristic of the configuration, which can be used to justify cost performance). Then, Fig. 3.10(b) depicts the cost as a function of time slot (zooming into a specific area to aid more detailed comparisons). In what

follows, we analyze the different policies based on observations from Fig. 3.10, and having in mind that demands fluctuate over time (see Fig. 3.4):

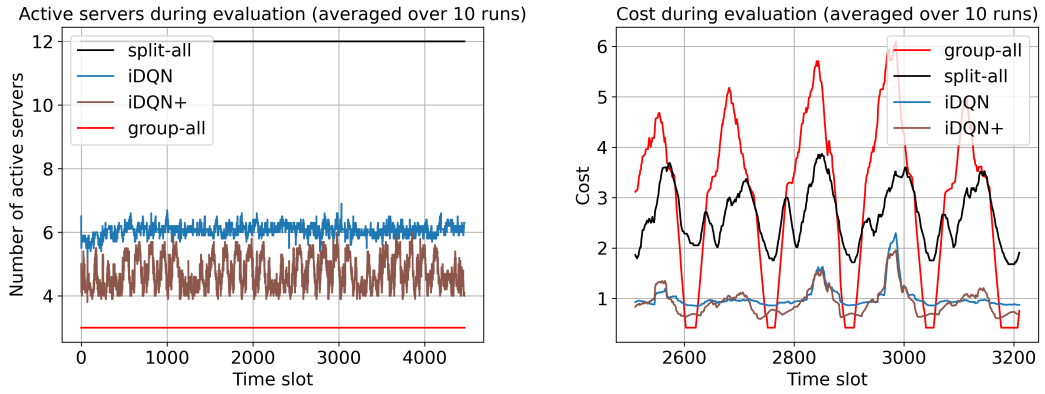
-group-all: it places all VNFs in only 3 nodes (static policy) and is characterized by large magnitude periodical fluctuations of the cost. When demands are low (no SLA violations) group-all is the best policy (uses the minimum number of nodes), but when demands are high it suffers from severe SLA violations.

-split-all: it distributes the VNFs in all 12 nodes (static policy). Thus, when demands are low split-all keeps unnecessary many nodes active (higher cost than group-all), but when demands are high it partly avoids SLA violations (lower cost than group-all). However, a more intelligent placement of VNFs is required to further reduce SLA violation penalties.

-iDQN: it intelligently distributes VNFs mainly to 6 nodes (migrating 1 VNF only every now and then, when things get really hectic). When demands are high it mostly avoids SLA violations (lower cost than split-all and group-all), but when demands are low it still uses more active nodes than necessary (lower cost than split-all but higher than group-all).

-iDQN+: it is a dynamic policy that intelligently distributes VNFs between 4 and 6 nodes (more dynamic than iDQN). When demands are high it mostly avoids SLA violations (similar cost to iDQN) and when demands are low it tries to reduce the number of active nodes by migrating VNFs (lower cost than iDQN but still slightly higher than group-all). It is probable that iDQN+ doesn't reduce the number active nodes all the way down to group-all during the low traffic periods due to the corresponding reconfiguration cost.

According to the above analysis, iDQN+ manages to obtain better, more dynamic, policies than iDQN in the same number of training timesteps (the sample efficiency gain offered by the prioritized experience replay is confirmed). The dynamic nature of the iDQN+ policies explain also why it demonstrates an increased performance gain against iDQN in the testing dataset. Looking at the demand timeseries of the training and testing datasets in Fig.3.4, it is evident that the latter is characterized by lower magnitude demands. Hence, the iDQN+ policies that are able to migrate VNFs toward reducing the number of active nodes when traffic is low have the upper hand compared to the more static iDQN policies.



(a) Number of active nodes

(b) Cost

FIGURE 3.10: Comparison of policies during evaluation in the training dataset (large-scale real traffic scenario).

Take-away message 5: *the proposed speed up heuristics of iDQN+ improve the performance of iDQN, particularly as the action space gets larger, and provide a more scalable solution.*

3.4 Conclusions

In this chapter we presented a theoretical analysis of the scalability properties of various Q-learning-based schemes in our problem, starting from a standard "tabular" Q-learning method and gradually adding extra components towards a more scalable solution (based on computational complexity arguments). We ended-up with a multi-agent scheme of independent DQN agents (iDQN), where the DQN component can tackle state space complexity and the use of multiple agents addresses action space complexity. Then, we propose two speed-up heuristic mechanisms to further improve convergence speed: (i) by selecting more efficiently the mini-batch samples (prioritized experience replay); (ii) by smartly reducing computations during parameter updates (lazy TD-target computation).

Remaining Open Questions. Although the proposed multi-agent DQN-based scheme proved to be a much more scalable solution compared to its single-agent counterpart, the use of *independent* agents can potentially still cause problems in large scale scenarios. The lack of coordination among agents results to a non-stationary environment, which might negatively affect sample efficiency, quality of the obtained policies, or even lead to stability problems. Therefore, further

investigation is required to assess if inducing coordination among agents could further improve performance.

Chapter 4

Scalable DQN with Coordinated Branches

4.1 Introduction

As discussed in Chapter 3, some of the hurdles faced by RL based solutions tackling the dynamic slice embedding problem are the following: (i) infinite state spaces, due to continuous traffic demands of the VNFs involved; (ii) astronomically high action spaces, due to the combinatorial nature of placing multiple VNFs upon multiple nodes (considering multiple slices further exacerbates this problem); (iii) poor sample-efficiency, which can be an important shortcoming in online settings. While DNN-based RL schemes, e.g. DQN [45], can help with challenge (i), multi-agent DQN (Section 3.2.3) can mitigate challenge (ii), and the speed-up heuristic mechanisms proposed in Section 3.2.4 can (up to an extent) alleviate from challenge (iii), the scalability of existing multiagent schemes, of collaborative but independent DQN agents, may be negatively affected by the emerging non-stationary environment (induced by agent independence).

In this Chapter, we propose a DRL scheme based on the Branching Deep Q-Network (BDQ) architecture [60], which dramatically reduces action complexity (with respect to single-agent approaches) by allotting the control of each VNF to a different DNN branch. Moreover, to avoid the non-stationarity issues arising in multi-agent schemes of independent agents, BDQ involves an input DNN module that is shared among all branches and is responsible for their (implicit) coordination, improving the scheme's sample efficiency and scalability properties. Finally,

in contrast to the majority of the closely related works involving DRL solutions, we benchmark the proposed scheme against an algorithm that is theoretically optimal in the (stateless) "experts" setting, even in real traffic scenarios. Note that, while "stateless", this experts algorithm can in fact explore all actions - often a huge number - in parallel, unlike our scheme which operates in a "bandit-like" setup, exploring one action at a time. As a result, this baseline can be seen as a non-implementable oracle, that even a stateful scheme might not be able to match. The outline and specific contributions of this Chapter are as follows:

(C.1) In Section 4.3.1, we formulate the dynamic slice embedding problem as a (stateless) "experts" problem. We use a state-of-the-art algorithm that is theoretically optimal (in the experts context) as a baseline for our scheme.

(C.2) Section 4.3.2 recaps on the stateful RL version of the problem (introduced in Chapter 2), which attempts to take advantage of patterns in the VNF/VL traffic dynamics. It also includes a discussion on how and where existing DQN-based schemes fail to cope with the combinatorial state and action spaces involved.

(C.3) In Section 4.4, we propose a DRL scheme based on a sophisticated DNN architecture that considers: (i) a different DNN branch for each VNF; (ii) (implicit) coordination among branches (to improve scalability).

(C.4) Using a real dataset, in Section 4.5, we demonstrate that the proposed scheme outperforms (i) the experts baseline, both in terms of cost performance and sample efficiency (theoretically "grounding" the proposed approximate RL scheme); and (ii) the existing state-of-the-art multi-agent DQN approach, showing up to 45% cost improvement in a fairly large scenario.

4.2 System Model

Our model is based on some common assumptions on VNF embedding, originally discussed in [25], and generalized in Chapter 2. We summarize below the main attributes, as a recap, but refer the reader to Chapter 2 for details.

Physical Network: a weighted undirected graph $G = (\mathcal{V}, \mathcal{E})$ of physical nodes (set $\mathcal{V} = \{0, 1, ..., V-1\}$), interconnected by a set of links $\mathcal{E}$ (physical paths). Each node, $v \in \mathcal{V}$, and link, $(v, v') \in \mathcal{E}$, is characterized by a capacity to process traffic flows.

Network Slices: virtual networks on top of the physical network. Each slice $k \in \mathcal{K}$ is a directed graph $H_k = (\mathcal{N}_k, \mathcal{L}_k)$ of VNFs (set $\mathcal{N}_k$) and VLs (set $\mathcal{L}_k$), that must be assigned to physical nodes and paths respectively. Assuming that time is slotted, each VNF n and VL (n, n') requires an amount of resources denoted by $d_n^k(t)$ and $d_{n,n'}^k(t)$ respectively, where t indicates the time slot.

(Input) Demand vector $d_t \in \mathcal{D}$***:*** denotes the demands of all slices at time slot t,

$$d_t = (d_i^k(t) | \forall k \in \mathcal{K},\ i \in \mathcal{N}_k \cup \mathcal{L}_k).$$

(Control Variables) Configuration vector $c_t \in \mathcal{C}$***:*** denotes the assignment of all VNFs to physical nodes[1] at time slot t,

$$c_t = (c_n^k(t) | \forall k \in \mathcal{K},\ n \in \mathcal{N}_k),$$

where $c_n^k(t)$ indicates the host node of VNF n (slice k) at t.

The goal of dynamic slice embedding is to choose the configuration c_t at every time slot t, before actually knowing the demands for that slot (but possibly knowing past demands and configurations), in order to ensure that (i) each slice's performance is isolated from other slices, despite sharing common resources (SLAs are fulfilled); (ii) network resources are utilized efficiently (low network-related costs).

Slice SLAs: We assume that each slice's performance is measured with an *end-to-end* KPI, and there is an agreed slice-specific worst case performance q_k (SLA). Without loss of generality, we will assume here that this KPI is the end-to-end delay of an average flow going through that VNF chain, given by a function $F_k^{delay}(c, d)$. This delay is captured by a fairly sophisticated queuing model, where resources between collocated VNFs on a node (or VLs on a link) are scheduled with a (generalized) Processor Sharing discipline, while the end-to-end delay per chain is captured with a (generalized) Jackson network (see Section 2.2.3). Then the SLA violation cost $\ell^{SLA}(c, d)$ is given by (2.5).

Network Costs: In addition to the costs associated with end-to-end delay violating the SLA q_k, we assume that there are additional network costs that an operator might pay. First, we consider a (monetary) cost related to using a node, e.g., an

[1] W.l.o.g., we assume that routing paths are predetermined and known for any pair of physical nodes. Our algorithm could be straightforwardly extended to scenarios with multiple alternative paths to choose from, for each node pair.

idle node could be set to sleep mode to save energy [47], given by (2.7). Second, we assume there is another potential cost for migrating a VNF from one node to another,e.g., network overhead due to signalling, or even service downtime [32], given by (2.6).

4.3 Optimization Baselines for VNF Chain Placement

In this section, we will discuss two popular solution frameworks for solving the previously defined high level problem. In both cases, demands d_t are assumed unknown and time-varying, so algorithms sought fall in the broad area of *online learning/optimization*.

4.3.1 Experts optimization

As a first step, we formulate and solve the problem as a standard "experts" problem. These problems are often categorized under the umbrella of Bandit optimization or Online Convex Optimization (OCO) [61]. In the experts setting, a learning agent takes actions based on a "goodness" estimate that he maintains for each configuration ("arm" or "expert"). This estimate depends only on past costs and gets updated at every time slot for all configurations. We stress here that an experts algorithm is very powerful in that, at every step, it improves the goodness estimate of *all* possible configurations, not just the chosen configuration c_t. This is in stark contrast to *bandit* environments, or online RL environments, where information only about c_t is obtained at each step. For massive action spaces, like the ones arising in slice embedding problems, this constitutes a very significant theoretical advantage in terms of sample efficiency. For this reason, we'll treat this scheme as one of our baselines, that is not possible to implement in practice, for large problems.

Action space. The agent's action at t, is the assignment of VNFs to physical nodes in $t+1$ (without knowing d_{t+1}):

$$a_t = c_{t+1} \in \mathcal{A}.$$

The action space $\mathcal{A} = \mathcal{C}$ quickly explodes, even in moderate-sized scenarios, due to the combinatorial configuration vector.

Cost function: The total cost of a configuration a at t is:

$$\ell(a, d_t) = w^{\text{SLA}} \cdot \ell^{\text{SLA}}(a, d_t) + w^{\text{ON}} \cdot \ell^{\text{ON}}(a), \tag{4.1}$$

where w^{SLA} and w^{ON} are "fixed" scalar weights that determine the importance of the respective cost terms. Note that we normalize the cost, so that $\ell(a, d_t) \in [0, 1]$ for all $a \in \mathcal{A}$ and $t \in \{0, 1, ..., T-1\}$, where T is the optimization horizon.

Baseline algorithm. To solve this problem, we consider the Multiplicative Weights (MW) algorithm [62], a simple online algorithm that can learn probabilistic policies with optimality guarantees. The algorithmic steps of MW are detailed in Fig. 6.

Algorithm 6 (MW) Main algorithmic steps.

MW algorithm

Initialize a "goodness" estimate vector $Q_t(a)$ to $Q_0(a) = 1$, for all configurations $a \in \mathcal{A}$. Set the learning rate to $\eta = \sqrt{\frac{\ln |\mathcal{A}|}{T}}$, where $|\mathcal{A}|$ is the number of configurations and T the optimization horizon.

Step 1: At time slot t, the agent selects $a_t \in \mathcal{A}$ (the configuration c_{t+1}), with probability:

$$p_t(a) = \frac{Q_t(a)}{\sum_a Q_t(a)} \tag{4.2}$$

Step 2 The demand vector d_{t+1} is revealed and the cost $\ell(a_t, d_{t+1})$ is inflicted. Also, the costs $\ell(a, d_{t+1})$ for all configurations $a \in \mathcal{A}$ become known.

Step 3: All estimates are updated according to:

$$Q_{t+1}(a) \leftarrow Q_t(a) \cdot (1 - \eta)^{\ell(a, d_{t+1})}, \forall a \in \mathcal{A} \tag{4.3}$$

Repeat steps 1 to 3 untill $t = T$.

The performance of experts algorithms is compared to an "optimal static oracle". This oracle knows *in advance all future demands up to horizon* T and chooses *one* (hence "static") configuration ($a^{(0)} = a^{(1)} = ... = a^{(T-1)} = a$):

$$a^* = \arg\min_{a \in \mathcal{A}} \sum_{t=0}^{T-1} \ell(a, d_{t+1}). \tag{4.4}$$

Regret is defined over T, as the difference between the accumulated cost achieved by the MW agent $L_{MW}^{(T)} = \sum_{t=0}^{T-1} \sum_a p_t(a)\ell(a, d_{t+1})$, and the respective cost of the optimal static oracle $L_{a^*}^{(T)} = \sum_{t=0}^{T-1} \ell(a^*, d_{t+1})$. MW has optimal (scaling-wise) regret [62]:

Lemma 4.1. *Regret*$^{(T)} = L_{MW}^{(T)} - L_{a^*}^{(T)} \leq 2\sqrt{T \ln |A|}$.

Sublinear regret implies that MW eventually catches up with the oracle, in terms of cost per slot, and hence, we cannot expect to do better (in this class of schemes). We will thus use both the performance of MW and the oracle's performance as theoretically-grounded baselines.

We stress that MW is 1-to-1 equivalent to the more well known Exp3 algorithm, with appropriate parameter changes [63]. Exp3 ("Explore and Exploit with Exponential weights") is an algorithm for the bandit setting, where only the cost of the chosen action is revealed (as opposed to the experts setting where the costs of all actions become available). This becomes clearer in Alg. 7, where the main algorithmic steps of the Exp3 are given. The most important difference compared to MW, is that in Exp3 there is a small probability of exploring a random action per round, instead of only exploiting the already learned "goodness" estimates (see (4.5)). Without any exploration the algorithm could potentially get stuck to suboptimal actions (no sublinear regret). Of course, another difference is that at each timestep only the estimate of the taken action gets updated, as there is no feedback for the rest of the actions. The result of this partial feedback is that the regret of Exp3 is $O(\sqrt{|\mathcal{A}| \cdot T \ln |\mathcal{A}|})$ (as opposed to the $O(\sqrt{T \ln |\mathcal{A}|})$ regret of MW). This highlights the important advantage of MW in terms of sample efficiency compared to bandit schemes, as $|\mathcal{A}|$ grows very fast with the scenario size.

Pros: (i) MW doesn't require any foreknowledge about demands; (ii) it has optimality guarantees on cost performance; (iii) it has an important sample efficiency advantage compared to standard bandit-like schemes.

Cons: (i) it is a very strong assumption, and computationally very intensive, to improve the estimates for *all* configurations at every step, when $|\mathcal{A}|$ is in the order of billions (not to mention the memory requirements); (ii) MW is a stateless scheme, essentially assuming an "adversary" chooses demands, hence fails to exploit any patterns intrinsic to the demands (e.g. diurnal traffic, week-weekend

Algorithm 7 (Exp3) Main algorithmic steps.

Exp3 algorithm
Initialize a "goodness" estimate vector $Q_t(a)$ to $Q_0(a) = 1$, for all configurations $a \in \mathcal{A}$. Set the learning rate to $\eta = \sqrt{\frac{\ln|\mathcal{A}|}{T\cdot|\mathcal{A}|}}$, and the exploration rate to $\gamma = |\mathcal{A}| \cdot \eta$.
Step 1: At time slot t, the agent selects $a_t \in \mathcal{A}$ (the configuration c_{t+1}), with probability:

$$q_t(a) = (1-\gamma)p_t(a) + \frac{\gamma}{|\mathcal{A}|}, \tag{4.5}$$

$$\text{where } p_t(a) = \frac{Q_t(a)}{\sum_a Q_t(a)}.$$

Step 2 The demand vector d_{t+1} is revealed and the cost $\ell(a_t, d_{t+1})$ is inflicted.
Step 3: The estimate of the taken action $a = a_t$ is updated according to:

$$Q_{t+1}(a) \leftarrow Q_t(a) \cdot e^{-\eta \cdot \frac{\ell(a,d_{t+1})}{q_t(a)}} \tag{4.6}$$

Repeat steps 1 to 3 untill $t = T$.

patterns, etc.); (iii) it does not account for reconfiguration costs (while bandit algorithms for setups with reconfiguration costs do exist [64], these go beyond the scope of this work).

4.3.2 RL optimization

We now assume that that the (unknown) demand dynamics have stateful characteristics, meaning that the current history of demands determines the probability distribution of future demands. Considering also a reconfiguration cost for migrating VNFs between consequent time slots, gives rise to a problem with delayed rewards (e.g. if the demand of a VNF is predicted to increase and stay high for long enough, its migration to a less busy server might be suboptimal in the short term, due to a high reconfiguration cost, but could pay-off in the next few time slots). This is a typical RL setting that we discussed in detail in Chapter 2. In what follows, we briefly recap on the RL formulation of the slice embedding problem and the approximate RL algorithms of Chapter 3, as we will use them as baselines for the more advanced proposed scheme of this Chapter.

State Space. The state of the system at t consists of the configuration vector (2.1) and the demand vector (2.2):

$$s_t = (c_t, d_t) \in \mathcal{S}.$$

Consequently, the state space $\mathcal{S}$ is the Cartesian product between the sets of configuration and demand vectors ($\mathcal{S} = |\mathcal{C}| \times |\mathcal{D}|$). In this work we consider continuous real traffic demands imported from the Milano dataset [35], which implies an infinite state space (due to the infinite set $\mathcal{D}$).

Action space. The agent action a_t is the same as in the experts setting (the configuration to be applied in the next time slot).

Remark: An RL algorithm can be practically applied in the slice embedding problem only if it is able to handle both the infinite state space and the combinatorial action space.

Reward function. If the system is at state s_t and the agent takes an action a_t, then in the next time slot a new state s_{t+1} is revealed and the corresponding reward is:

$$r_{t+1} = -(w^{\text{SLA}} \cdot \ell^{\text{SLA}}(s_{t+1}) + w^{\text{ON}} \cdot \ell^{\text{ON}}(a_t) + w^{\text{RC}} \cdot \ell^{\text{RC}}(s_t, a_t))$$

The only difference of the above reward with the cost function of the experts problem (4.1), is that it has an additional reconfiguration cost term and a minus sign (typically RL agents try to maximize the received rewards instead of minimizing the cost).

Q-learning. In the RL setting, the goal is to learn an optimal configuration for each possible state s of the system. This gives rise to a more "powerful" oracle than the static one, which may select a different action at every state (this optimal policy can be obtained by dynamic programming algorithms, e.g. Policy Iteration [49]). Q-learning is a standard "tabular" RL algorithm that is guaranteed to converge to this more "powerful oracle", in theory (see Section 3.2.1).

However, neither Q-learning nor dynamic programming can be (directly) applied to our problem, due to the infinite number of states (even for quantized demands, these schemes would be applicable only in very small toy scenarios due to the combinatorial state *and* action spaces). Thus, we refer to them as a motivation

for more practical DRL schemes, and instead, we use the MW algorithm and the static "oracle" of Section 4.3.1 as baselines.

Deep Q-Network (DQN). It is an approximate RL algorithm, introduced in [45], that can tackle problems with an infinite number of states, as is the case in our problem (see Section 3.2.2 for a detailed analysis). It uses a DNN with parameters θ to approximate the action value function $Q(s, a)$, which takes as input the state s and outputs the estimates $Q_\theta(s, a)$ of the expected (discounted) long-term reward, for all actions $a \in \mathcal{A}$. Learning a "good" approximation $Q_\theta(s, a)$ is equivalent to learning a "good" slice embedding policy: at any state, the agent can select the best configuration by performing an argmax operation over the action values of all possible configurations. While standard DQN can be applied in arbitrarily large state spaces, the exploding action space of our problem still poses a scalability bottleneck (exploding DNN fanout and expensive argmax operations over the combinatorial actions space).

RL baseline: independent Deep Q-Networks (iDQN). It is the multi-agent, DQN-based algorithm of Section 3.2.3. The action complexity problems of single-agent DQN can be addressed by the use of multiple independent DQN agents, that decomposes the original action space into much smaller action sub-spaces. The sample-efficiency of this scheme can be further improved with the speed-up heuristic mechanisms proposed in Section 3.2.4 (we denote this enhanced version by iDQN+).

Action space decomposition: Each independent DQN agent (n, k) is responsible only for the placement of a specified VNF n (of slice k), and thus its DNN outputs the predicted action values of placing this VNF to any of the permitted physical nodes (all agents view the same state s_t (2.1)). The new action space $\mathcal{A}^{nk}$ is *not* combinatorial anymore (much smaller fanout). Moreover, the computational complexity of the argmax operation required to choose a configuration increases linearly instead of exponentially (N argmax operations over V actions instead of one argmax over V^N actions, where V is the number of physical nodes and N the total number of VNFs).

Pros: iDQN can be applied in practical slicing scenarios (the DQN component tackles state space complexity while the use of multiple agents radically reduces action space complexity).

Cons: The lack of coordination among agents can potentially deteriorate sample efficiency and quality of the obtained policies (or even lead to stability problems), due to the induced non-stationarity. The fact that the agents are independent means that each of them conceives the rest as part of the environment, and thus, as agents try to improve their policies, the environment becomes non-stationary.

4.4 DQN with Coordinated Branches

We are now ready to delve into the details of our proposed algorithm, that attempts to overcome the different shortcomings in the baseline schemes, identified earlier. The action branching Deep Q-Network (BDQ) architecture was introduced in [60] to facilitate the application of DQN (and any other discrete-action RL algorithm) into problems with high-dimensional discrete action spaces. This method shares the same action space decomposition advantages with the iDQN scheme of the previous section, but also aims to tackle the problems stemming from the lack of coordination between agents. Fig. 4.1 visualizes the branching architecture.

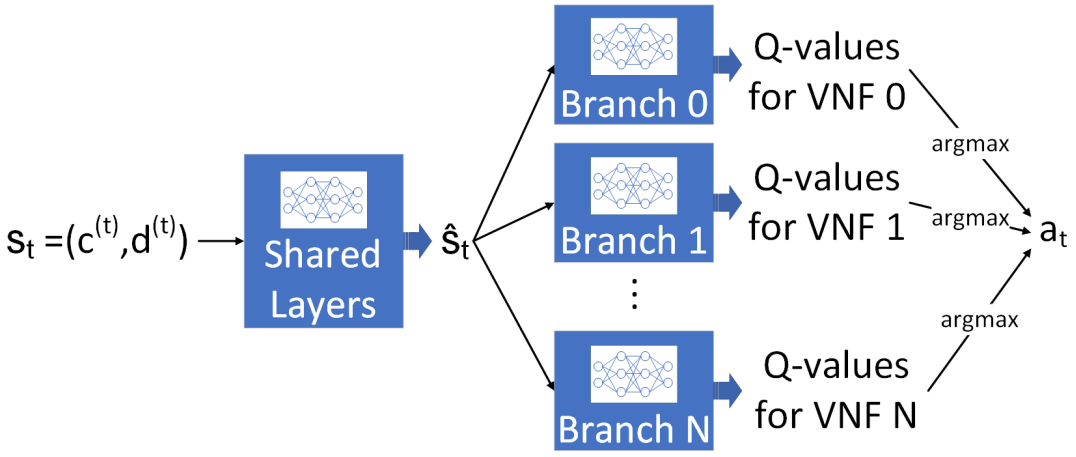


FIGURE 4.1: Schematic representation of the branching architecture. A shared module of the DNN takes as input the state s_t and outputs a latent representation $\hat{s}_t$, which in turn is given as input to N different branches (one branch per VNF). The assignment $c^i(t+1)$ of each VNF i to a physical node in the next slot is determined by an argmax operation over the Q-value estimates of branch i. Then, the chosen action is $a_t = (c^0(t+1), c^1(t+1), ..., c^N(t+1))$.

Action space decomposition: Each DNN branch $Q_{\theta_{nk}}(s)$, outputs the action value estimates of placing VNF n of slice k to any of the permitted physical nodes. This results to a linear increase of network outputs as a function of the number of VNFs, as opposed to the exponential increase of the vanilla DQN scheme ($V \cdot N$

instead of V^N, where V the number of physical nodes and N the number of VNFs). The same holds for the computational complexity of the argmax operation over all Q-value estimates, that is required in every time slot to select the best action.

Coordination: A DNN module that is shared among the different branches is responsible for their implicit coordination. It takes as input the state s and outputs a latent representation $\hat{s}$, that is in turn given as input to each one of the branches.

Action selection: An ϵ-greedy policy is used to balance exploration of new actions and exploitation of the learned Q-function. To this end, at each time slot a random configuration is chosen with probability ϵ, while each VNF is assigned to the node with the maximum Q-value estimate with probability $1 - \epsilon$.

Stability mechanisms: Using a DNN to approximate the Q-function can potentially lead to instabilities due to correlations between subsequent parameter updates (the visited states by the agent are highly correlated). BDQ uses the two standard DQN mechanisms that ensure stable learning, the experience memory replay and the target network, which where discussed in detail in Section 3.2.2. The former is a replay buffer that stores visited experiences (s, a, s', r), and enables updating the parameters θ of the DNN (policy network) based on randomly sampled mini-batches. The latter is an older "frozen" version of the policy network, with parameters θ', that is updated less frequently and is used as part of a Temporal Difference (TD) target (4.7) for semi-supervised learning.

Action improvement: At each round, a minibatch of experiences is randomly sampled from the replay buffer and the policy network's parameters are updated based on the expected value of the mean squared TD error across all branches (4.9).

The main algorithmic steps of BDQ are given in Alg.8.

4.5 Simulation Results

In this Section we employ a real traffic dataset to drive the demands in various slice embedding scenarios, with the goal to: (i) examine the theoretical optimality of the proposed BDQ scheme in terms of cost per time slot (compared to the static oracle), as well as its sample efficiency (compared to MW), in a moderately-sized setup; (ii) validate the scalability of BDQ and the performance gains offered

Algorithm 8 (BDQ) Main algorithmic steps.

BDQ algorithm
Action branching architecture: A DNN $Q_\theta(s)$, with a separate branch $Q_{\theta_{nk}}(s)$ per VNF $n \in \mathcal{N}_k$ of slice $k \in \mathcal{K}$.
Step 1: (in agent) An ϵ-greedy action is taken:

$$a^{nk} \leftarrow \begin{cases} \text{random } a^{nk} \in \mathcal{A}^{nk}, & \text{with probability } \epsilon; \\ \arg\max_{a^{nk} \in \mathcal{A}^{nk}} Q_{\theta_{nk}}(s, a^{nk}), & \text{with probability } 1-\epsilon. \end{cases}$$

Then, the collective action is:

$$a = (a^{00}, ..., a^{N_K K})$$

Step 2 (in env): Returns the next state s' and reward r.
Step 3: (in agent) store transition (s, a, s', r).
Step 4 (in agent): copy the policy network parameters θ to the target network $\theta^{'}$ (only every X timesteps).
Step 5 (in agent): pick M samples randomly from replay buffer and calculate the TD target y_i for each sample i:

$$y_i = r_i + \gamma \frac{1}{N} \sum_{n \in \mathcal{N}_k, k \in \mathcal{K}} \max_{(a_i^{nk})' \in \mathcal{A}^{nk}} Q_{\theta'_{nk}}(s_i', (a_i^{nk})'), \tag{4.7}$$

where N is the number of branches.
Then, perform a gradient step:

$$\theta \leftarrow \theta - \eta \nabla_\theta L, \tag{4.8}$$

where

$$L = \mathbb{E}_{i \sim U(\mathcal{D})}[\frac{1}{N} \sum_{n \in \mathcal{N}_k, k \in \mathcal{K}} (y_i - Q_{\theta_{nk}}(s_i, a_i^{nk}))^2]. \tag{4.9}$$

Repeat steps 1 to 5 for T time slots.

by coordination, compared to the independent agents of iDQN+, in a large-scale setup. Thus, the Section is divided into two respective parts, each dedicated to one of the above objectives.

Algorithms. Below we first outline all the algorithms (or policies) used in this section and then specify the selected values for all the algorithm-specific parameters.

- **group-all** a simple static policy that merely minimizes the number of active nodes by placing all VNFs on the largest node. Possibly suffers from SLA

violations.

- **split-all** a sister policy to group-all, which instead aims to minimize SLA violations by spreading VNFs to all available nodes. It often uses more nodes than necessary, inflicting a high "on" nodes cost.
- **static oracle** the optimal static policy of Section 4.3.1.
- **Exp3** the adversarial multi-armed bandit algorithm of Section 4.3.1 (Fig. 7), that has optimal regret (in the bandit setting) with respect to the static oracle above.
- **MW** the experts algorithm of Section 4.3.1 (Fig. 6), that has optimal regret (in the experts setting) with respect to the static oracle above.
- **iDQN** the multi-agent DRL scheme of Section 3.2.3 (denoted by iDQN+ when employing also the speed-up enhancements of section 3.2.4).
- **BDQ** the DRL scheme of Section 4.4.

Parameters of DRL schemes: We set the replay buffer size to 5000, the target update period to 500, the minibatch size to 32, the learning rate to 10^{-3}, and the discount factor to $\gamma = 0.9$, same as in Section 3.3. The DNNs are multilayer perceptrons[2] (commonly used in related works, e.g. [30], [65]). Each iDQN agent has 3 hidden layers with 60 neurons per layer (this size performed well in a variety of tested scenarios). In BDQ there is first a fully connected layer shared among the different branches, which takes as input the state and has 60 output neurons. This is followed by N different branches, each of them being an MLP with 60 input neurons, 1 hidden layer of 60 neurons, and an output that has as many neurons as the number of actions per VNF.

VNF demands. We use the popular Milano dataset [35] to drive the demands and use half of the imported datapoints for training and the rest for testing. More details can be found in Section 3.3.1 (*Traffic Type 2*).

[2]We use simple DNNs to not entangle our discussion with the additional impact of specific (fancier) DNN architectures. We defer this to future work.

4.5.1 Comparison with experts baseline

We first focus on a medium-sized setup, in order to compare our proposal to the theoretically grounded MW algorithm and the corresponding static oracle.

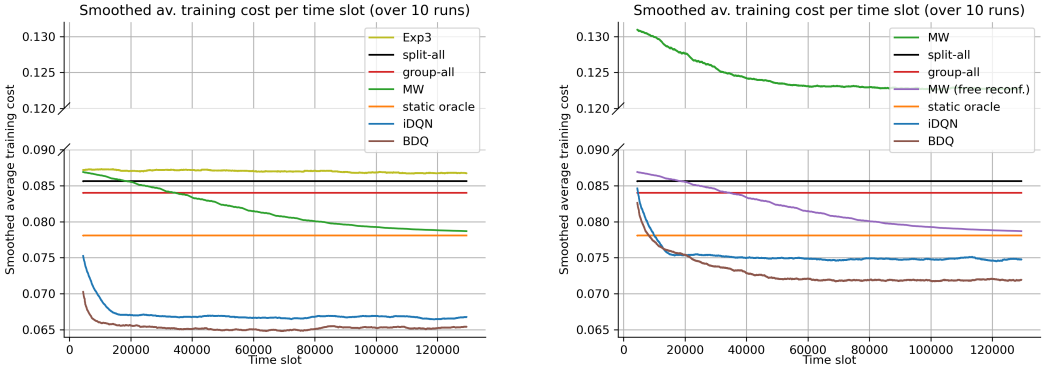
System setup. We consider a physical network with two domains, each consisting of two servers, while there are four slices comprising 2 VNFs each (one VNF per domain). This results to 256 possible configurations (we remind that the state space is infinite due to continuous traffic demands).

Scenario 1 - free reconfigurations. We first consider a scenario without reconfiguration cost ($w^{RC} = 0$), in order to assess in isolation the ability of both bandit and DRL algorithms to dynamically adapt their actions according to the changing traffic. Fig. 4.2(a) depicts the cost as a function of time slot during training (averaged over 10 independent training runs and smoothed), for the algorithms under test. Note that Exp3, MW, iDQN, and BDQ start with a random policy that they improve online at each timestep (they demonstrate a higher cost at timestep 0, which gets lower over time), while the rest of the policies have been obtained offline.

Sanity checks: (i) static oracle is indeed better than the simple static heuristics group-all and split-all; (ii) MW converges to the cost of the static oracle as expected; (iii) MW has a huge sample-efficiency advantage compared to Exp3 (it manages to drive regret close to zero in the given horizon, while Exp3 barely achieves any significant cost reduction). This advantage stems from the fact that MW receives feedback for all actions at each round in contrast to Exp3 and any other bandit-like algorithm that receive feedback only for the chosen action (as discussed in Section 4.3.1)

Key Observations: i) both iDQN and BDQ are able to not only reach the static oracle much faster than MW, but they in fact outperform any static or bandit/expert policy (i.e. they more than make up for the theoretical sample efficiency gap with MW, through increased algorithmic sophistication); (ii) already the advantages of BDQ over iDQN are visible, even in this relatively small action space setup.

Take-away message 1: *DRL schemes converge faster than the experts baseline.*



(a) Scenario 1

(b) Scenario 2

FIGURE 4.2: Convergence plots for 2 different scenarios in a setup with 256 actions. In (a) reconfigurations are free, while in (b) an additional reconfiguration cost is inflicted. Notice that the y axis is discontinuous in order to be able to depict MW in (b), where it demonstrates significantly higher costs due to reconfigurations. We remark that in (b) we also plot "MW (free reconf.)", which is MW, but with the advantage of making free reconfigurations.

Take-away message 2: *DRL schemes obtain dynamic policies with lower cost than any static or bandit/expert policy.*

Scenario 2 - costly reconfigurations. We now introduce a reconfiguration cost in the previous scenario (we increase w^{RC}). We hope that the DRL agents will be able to smartly factor this in, unlike (vanilla) experts algorithms that do not. To make this even more challenging for DRL schemes, we further compare their performance with an MW version that has been given the advantage of free reconfigurations, denoted by "MW(free reconf.)". The results are depicted in Fig.4.2(b), with the main observations being: (i) the DRL schemes are able to gracefully degrade a little bit their performance, now making some costly reconfigurations only when they predict that this can be amortized later (hence the slightly higher cost compared to Scenario 1); (ii) they are still better than MW(free reconf.), despite its advantage of free reconfigurations; (iii) without this advantage, MW's performance is severely degraded due to many unnecessary reconfigurations.

Take-away message 3: *approximate RL schemes obtain effective policies even in the presence of reconfiguration costs.*

4.5.2 Comparison with DRL baseline

Having established both the theoretical sanity and necessity for (stateful) RL policies, we now consider a more realistically sized scenario, to test our new BDQ-based policy to a state-of-the-art iDQN+ one. We execute 10 independent training runs for each DRL agent (with different random seeds per run), as in Section 4.5.1, and then we evaluate the obtained policies both in the training and in the testing datasets to check the ability of DRL agents to generalize (in the evaluation phase agents act greedily with respect to the learned action value functions). We remark that, due to the vast action space of this scenario, MW is not applicable[3].

System setup. We consider a physical network with two domains, where one domain consists of 9 servers and the other of 3, while on top of it there are 10 slices comprising 2 VNFs each (one VNF per domain). This setup already leads to an immense action space of $|\mathcal{A}| = 2 \cdot 10^{14}$ confgurations!

Convergence speed. To facilitate a speed comparison between BDQ and iDQN+, we provide a convergence plot (Fig. 4.3) depicting the cost (averaged over 10 runs and smoothed) of each algorithm as a function of time slot during training. The key observation is that BDQ is verified to be more sample-efficient than iDQN+, as it manages to achieve a lower cost policy in the same number of training steps.

Cost performance. In order to examine both the mean performance and the stability of the DRL agents, we outline the results of the evaluation phase in the box plots of Fig. 4.4. The main observations are the following: (i) in the training dataset, BDQ demonstrates 41% and 47% better mean cost than iDQN+ and the static oracle respectively; (ii) in the testing dataset, BDQ demonstrates 45% and 65% better mean cost than iDQN+ and the static oracle respectively (both DRL schemes generalize successfully in the testing dataset and increase their performance gain against the static oracle's policy, which was also obtained based on the traffic demands of the training dataset; the lower cost achieved by the DRL schemes in the testing dataset can be explained by the lower traffic demand compared to the training dataset, as it can be seen in Fig. 3.4); (iii) BDQ has robust performance with much lower standard deviation than iDQN+ in both datasets (thanks to the implicit coordination of its branches); it is noteworthy that even the worst policy obtained by BDQ is still better than the static oracle.

[3]To obtain the static oracle in this scenario we used the *surrogateopt* function of Matlab with the default parameter values (this solver performed well in a variety of tested scenarios).

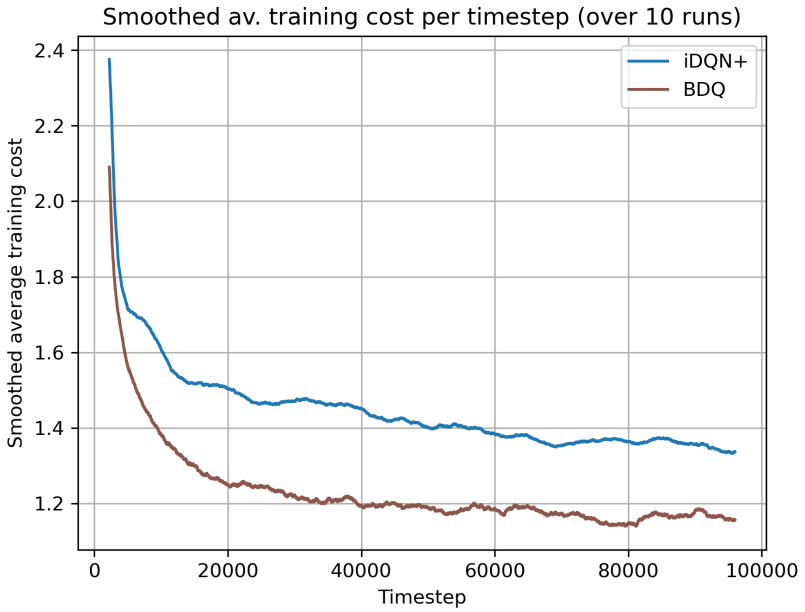
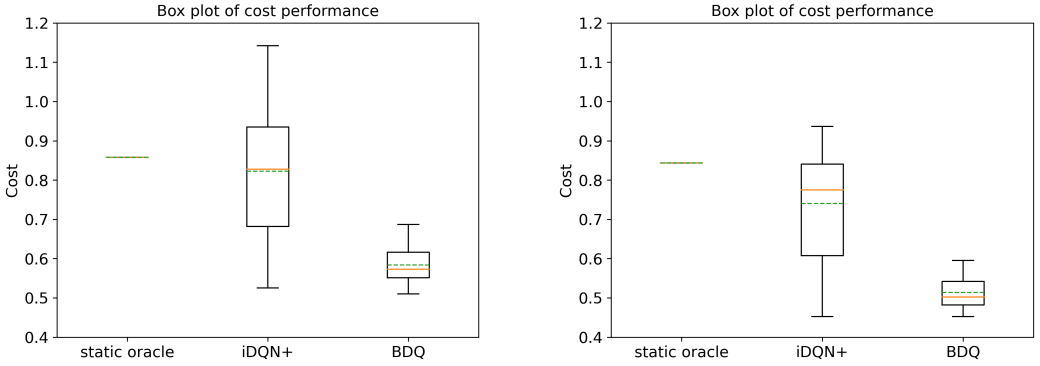


FIGURE 4.3: Convergence speed comparison between iDQN+ and BDQ (large-scale real traffic scenario).



(a) Training dataset

(b) Testing dataset

FIGURE 4.4: Cost performance evaluation in large-scale scenario ($2 \cdot 10^{14}$ actions). Box plot depicting the distribution of the cost achieved by each algorithm in the training and testing datasets.

Take-away message 4: *the performance gains of BDQ against iDQN+ and the static oracle become more prominent as the scenario size grows larger.*

Analysis of obtained policies. Here we will attempt to highlight why the cost of the obtained policies by BDQ is lower compared to the corresponding iDQN+ and static oracle policies. To this end, we provide two plots: (i) Fig. 4.5 depicts the number of active nodes per time slot for each of the above algorithms

(highlighting the dynamicity of the obtained policies); (ii) Fig. 4.6 depicts the corresponding costs as a function of time slot (we zoom into a specific area to facilitate comparisons). In what follows, we analyze the different policies based on observations from these two Figures and having in mind that demands fluctuate periodically over time (see Fig. 3.4):

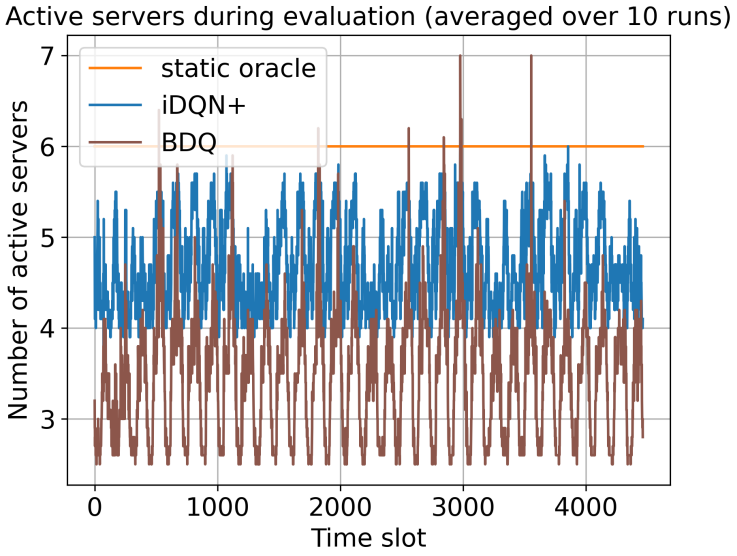


FIGURE 4.5: Number of active nodes as a function of time slot for BDQ, iDQN+, and the static oracle, during their evaluation in the training dataset (large-scale real traffic scenario).

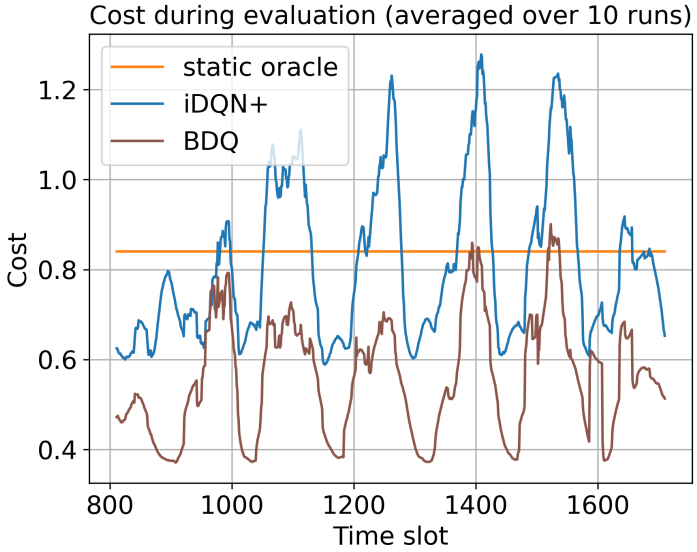


FIGURE 4.6: Inflicted cost as a function of time slot for BDQ, iDQN+, and the static oracle, during their evaluation in the training dataset (large-scale real traffic scenario).

-static oracle: a static VNF placement using 6 nodes (see Fig. 4.5). The chosen static configuration distributes VNFs in a way that avoids SLA violations (in Fig. 4.6 the cost of the static oracle is constant, equal to the cost of having 6 nodes active). We note that the oracle knows in advance all future traffic demands, and thus can choose an optimal configuration that avoids SLAs.

-iDQN+: it is a dynamic policy that distributes VNFs between 4 and 6 nodes (see Fig. 4.5). When demands are low it mostly uses 4 nodes to reduce the active nodes cost (lower cost than static oracle), but when demands are high it tries to avoid SLA violations by distributing VNFs to more nodes (at most 6). However, at traffic peaks the chosen configuration is usually worse than the static oracle's (see Fig. 4.6), inflicting higher costs due to either using a lower number of nodes than required or not distributing VNFs smartly enough.

-BDQ: it is the most dynamic policy of the three (see Fig. 4.5). When demands are low it uses less than 3 nodes to reduce the active nodes cost (lower cost than iDQN+ and static oracle), but when demands are high it intelligently distributes VNFs to more nodes and mostly avoids SLA violations, as it rarely inflicts higher cost than the static oracle (see Fig. 4.6).

According to the above analysis, the highly dynamic nature of the BDQ policies explain why it demonstrates an increased performance gain against both the static oracle and iDQN+.

4.6 Conclusions

This last Chapter focused on further improving the scalability of our DQN-based slice orchestrator, as well as on grounding its performance with respect to algorithms that are simpler, yet theoretically optimal even in real traffic scenarios (in the experts context). To this end, we proposed a DQN agent with a more advanced DNN architecture, called BDQ. It comprises one DNN branch per VNF (offering the same action decomposition advantages with the multiagent DQN scheme of Chapter 3), but also considers a number of shared layers to induce coordination among branches and avoid the non-stationarity issues arising in multi-agent schemes of independent agents. Using a real dataset, we demonstrate that the proposed scheme outperforms (i) the experts baseline, both in terms of cost performance and sample efficiency (theoretically "grounding" the proposed BDQ

scheme); and (ii) the multi-agent DQN approach of Chapter 3 (iDQN+), showing up to 45% cost improvement in a fairly large scenario.

Chapter 5

Conclusions

The current Thesis explored modeling and algorithmic approaches for the dynamic placement of multiple, concurrent, VNF chains ("slices") on top of the physical network infrastructure in Beyond 5G networks (commonly known as slice orchestration). The goal was to fulfil the QoS requirements of each slice while utilizing efficiently the limited network resources. The challenges we attempted to tackle arise from the B5G vision: (i) standard static optimization methods are not applicable, since traffic demands, and thus slice performance, are dynamically changing, a priori unknown, or even non stationary, necessitating online learning approaches; (ii) slices span multiple technological (and administrative) domains (e.g. RAN, Edge, Core, Cloud, etc.), they possibly involve loops and probabilistic routing of flows, while slice performance is determined by end-to-end KPIs along the entire VNF chain, increasing both modeling complexity and algorithmic efficiency.

The unknown traffic demands, the stateful characteristics of traffic variations (e.g. diurnal variations), and the potentially large cost of migrating a VNF from the host server to another server of the network, motivate the use of Reinforcement Learning methods to tackle the problem at hand. Therefore, as a first step, we introduced an RL environment for dynamic slice embedding in B5G networks (Chapter 2) that can support any KPI or SLA and attempts to address all the modeling challenges mentioned above. Relying on queuing theory, we provided analytical expressions for the end-to-end delay of complex slices over non loop-free paths across heterogeneous domains. Since simulating the end-to-end performance of every slice (consisting of numerous flows) can become computationally intensive

in large scale scenarios, the proposed analytical model can facilitate the faster training of RL agents.

Having formulated the RL problem, the next step was to devise RL agents that would be able to cope with the vast state and action spaces, stemming from the combinatorial nature of placing multiple VNFs of multiple slices over a large number of physical nodes across different domains. To this end, in Chapter 3 we presented a theoretical analysis of the scalability properties of various RL methods, starting from a vanilla "tabular" Q-learning algorithm, and gradually adding components on top of it to improve scalability (justifying the addition of each component along the way with computational complexity arguments). The above analysis concludes with a multiagent scheme of independent DQN agents. The DQN component tackles the problems stemming from state space complexity, using a DNN to approximate the Q-function, while the use of multiple agents mitigates the problems stemming from action space complexity (decomposing the vast original action space into much smaller per agent subspaces). Then, to further improve sample-efficiency and convergence speed of DQN-based methods (either single or multiagent), we proposed two heuristic mechanisms: (i) a prioritized experience replay that expedites learning by selecting more intelligently mini-batches from the experience replay memory (more efficient parameter updates); (ii) a novel mechanism that reduces the number of computations in parameter updates, by storing the Q-value estimate of the current state in the buffer and using it later in multiple updates. We confirmed the gains offered by the above multi-agent scheme by conducting extensive simulations, both with synthetic traffic (Markovian) in small scale scenarios (to compare with an optimal baseline) and with a real traffic dataset. We showed that the proposed scheme reduces convergence time by orders of magnitude with minimum penalty of decision optimality, compared to standard single-agent DQN (Section 3.3.2), while it also significantly outperforms static heuristic policies by at least a $3\times$ factor (Section 3.3.4).

While using multiple independent DQN agents proved to be a much more scalable solution compared to single-agent DQN, their performance may still deteriorate in very large scale realistic scenarios. This is due to the non-stationary environment arising from independent agents, since each of them conceives the rest as part of the environment, but their policies evolve over time. Therefore, a large number of co-existing independent agents can potentially lead to instabilities, very slow convergence, or convergence to a highly sub-optimal policy. To this end, the

last part of this Thesis focused on investigating if algorithmic efficiency can be improved by introducing coordination among the controlling entities of different VNFs. We proposed a DQN-based agent with an advanced DNN architecture, where each branch of the DNN is dedicated to the control of a different VNF (offering the same action decomposition benefits with the predecessor multiagent DQN scheme), while a shared DNN module among the branches offers (implicit) coordination. To theoretically ground the proposed method, we formulated dynamic slice embedding as a stateless "experts" problem, in order to use the MW algorithm, that is optimal in this context, as a baseline for our scheme. While MW disregards any stateful characteristics in the traffic variations, it has an important sample-efficiency advantage compared to bandit-like schemes (e.g. RL), by getting feedback for all possible actions at each time step (not just the chosen one). Moreover, it is guaranteed to converge to the optimal static policy. In Section 4.5, using real traffic to drive the demands, we demonstrated by simulations that the proposed BDQ scheme outperforms (i) the experts baseline, both in terms of cost performance and sample efficiency (theoretically "grounding" the proposed DRL scheme); and (ii) the predecessor multi-agent DQN approach of Chapter 3, showing up to 45% cost improvement in a fairly large scenario.

Bibliography

[1] K. Samdanis, X. Costa-Perez, and V. Sciancalepore, "From network sharing to multi-tenancy: The 5G network slice broker," *IEEE Communications Magazine*, vol. 54, no. 7, pp. 32–39, 2016.

[2] GSMA, "Network slicing use case requirements." https://www.gsma.com/futurenetworks/wp-content/uploads/2018/07/Network-Slicing-Use-Case-Requirements-fixed.pdf, 2018.

[3] C. Marquez, M. Gramaglia, M. Fiore, A. Banchs, and X. Costa-Perez, "How should I slice my network? A multi-service empirical evaluation of resource sharing efficiency," in *ACM MobiCom*, pp. 191–206, 2018.

[4] GSMA, "Mobile infrastructure sharing." https://www.gsma.com/publicpolicy/wp-content/uploads/2012/09/Mobile-Infrastructure-sharing.pdf, 2012.

[5] R. Mijumbi, J. Serrat, J.-L. Gorricho, N. Bouten, F. De Turck, and R. Boutaba, "Network function virtualization: State-of-the-art and research challenges," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 1, pp. 236–262, 2016.

[6] M. S. Bonfim, K. L. Dias, and S. F. L. Fernandes, "Integrated NFV/SDN architectures: A systematic literature review," *ACM Comput. Surv.*, vol. 51, feb 2019.

[7] A. Lara, A. Kolasani, and B. Ramamurthy, "Network innovation using openflow: A survey," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 1, pp. 493–512, 2014.

[8] I. Afolabi, T. Taleb, K. Samdanis, A. Ksentini, and H. Flinck, "Network slicing and softwarization: A survey on principles, enabling technologies, and solutions," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 3, pp. 2429–2453, 2018.

[9] X. Foukas, G. Patounas, A. Elmokashfi, and M. K. Marina, "Network slicing in 5G: Survey and challenges," *IEEE communications magazine*, vol. 55, no. 5, pp. 94–100, 2017.

[10] M. Series, "IMT vision–framework and overall objectives of the future development of IMT for 2020 and beyond," *Recommendation ITU*, vol. 2083, no. 0, 2015.

[11] A. Ghosh, A. Maeder, M. Baker, and D. Chandramouli, "5G evolution: A view on 5G cellular technology beyond 3gpp release 15," *IEEE Access*, vol. 7, pp. 127639–127651, 2019.

[12] Ericsson Blog, "The next wave of 5G – 3GPP release 19." https://www.ericsson.com/en/blog/2023/12/3gpp-release-19, Dec 15, 2023. Accessed: January 2024.

[13] European 5G Observatory, "5G observatory report 19." https://5gobservatory.eu/report-19-october-2023/, Oct 19, 2023.

[14] Vodafone Press Release, "Vodafone and Ericsson successfully trial cloud gaming on 5G standalone network slice." https://www.vodafone.co.uk/newscentre/press-release/vodafone-and-ericsson-successfully-trial-cloud-gaming-on-5g-standalone-network-slice/, 2023. Accessed: January 2024.

[15] ITU-R, "IMT traffic estimates for the years 2020 to 2030." https://www.itu.int/dms_pub/itu-r/opb/rep/R-REP-M.2370-2015-PDF-E.pdf, 2015.

[16] ITU-FG-NET-2030, "Network 2030 - a blueprint of technology, applications and market drivers towards the year 2030 and beyond." https://www.itu.int/en/ITU-T/focusgroups/net2030/Documents/White_Paper.pdf, 2019.

[17] NGMN, "6G drivers and vision v1.0." https://www.ngmn.org/wp-content/uploads/NGMN-6G-Drivers-and-Vision-V1.0_final.pdf, Apr 19, 2021.

[18] W. Jiang, B. Han, M. A. Habibi, and H. D. Schotten, "The road towards 6G: A comprehensive survey," *IEEE Open Journal of the Communications Society*, vol. 2, pp. 334–366, 2021.

[19] A. Filali, A. Abouaomar, S. Cherkaoui, A. Kobbane, and M. Guizani, "Multi-access edge computing: A survey," *IEEE Access*, vol. 8, pp. 197017–197046, 2020.

[20] K. B. Letaief, W. Chen, Y. Shi, J. Zhang, and Y.-J. A. Zhang, "The roadmap to 6G: Ai empowered wireless networks," *IEEE Communications Magazine*, vol. 57, no. 8, pp. 84–90, 2019.

[21] R. Su, D. Zhang, R. Venkatesan, Z. Gong, C. Li, F. Ding, F. Jiang, and Z. Zhu, "Resource allocation for network slicing in 5G telecommunication networks: A survey of principles and models," *IEEE Network*, vol. 33, no. 6, pp. 172–179, 2019.

[22] P. Caballero, A. Banchs, G. de Veciana, and X. Costa-Pérez, "Multi-tenant radio access network slicing: Statistical multiplexing of spatial loads," *IEEE/ACM Transactions on Networking*, vol. 25, no. 5, pp. 3044–3058, 2017.

[23] Y. L. Lee, J. Loo, T. C. Chuah, and L.-C. Wang, "Dynamic network slicing for multitenant heterogeneous cloud radio access networks," *IEEE Transactions on Wireless Communications*, vol. 17, no. 4, pp. 2146–2161, 2018.

[24] P. L. Vo, M. N. H. Nguyen, T. A. Le, and N. H. Tran, "Slicing the edge: Resource allocation for ran network slicing," *IEEE Wireless Communications Letters*, vol. 7, no. 6, pp. 970–973, 2018.

[25] S. Vassilaras, L. Gkatzikis, N. Liakopoulos, I. N. Stiakogiannakis, M. Qi, L. Shi, L. Liu, M. Debbah, and G. S. Paschos, "The algorithmic aspects of network slicing," *IEEE Communications Magazine*, vol. 55, no. 8, pp. 112–119, 2017.

[26] F. Schardong, I. Nunes, and A. Schaeffer-Filho, "NFV resource allocation: A systematic review and taxonomy of VNF forwarding graph embedding," *Computer Networks*, vol. 185, p. 107726, 2021.

[27] V. Sciancalepore, K. Samdanis, X. Costa-Perez, D. Bega, M. Gramaglia, and A. Banchs, "Mobile traffic forecasting for maximizing 5G network slicing resource utilization," in *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, pp. 1–9, 2017.

[28] D. Bega, M. Gramaglia, M. Fiore, A. Banchs, and X. Costa-Perez, "Deepcog: Cognitive network management in sliced 5G networks with deep learning," in *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pp. 280–288, IEEE, 2019.

[29] D. Bega, M. Gramaglia, M. Fiore, A. Banchs, and X. Costa-Perez, "Aztec: Anticipatory capacity allocation for zero-touch network slicing," in *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*, pp. 794–803, IEEE, 2020.

[30] P. T. A. Quang, Y. Hadjadj-Aoul, and A. Outtagarts, "A deep reinforcement learning approach for vnf forwarding graph embedding," *IEEE Transactions on Network and Service Management*, vol. 16, no. 4, pp. 1318–1331, 2019.

[31] F. Mason, G. Nencioni, and A. Zanella, "Using distributed reinforcement learning for resource orchestration in a network slicing scenario," *IEEE/ACM Transactions on Networking*, vol. 31, no. 1, pp. 88–102, 2023.

[32] K. Kaur, F. Guillemin, and F. Sailhan, "Container placement and migration strategies for cloud, fog, and edge data centers: A survey," *International Journal of Network Management*, vol. 32, no. 6, p. e2212, 2022.

[33] F. Wei, G. Feng, Y. Sun, Y. Wang, S. Qin, and Y.-C. Liang, "Network slice reconfiguration by exploiting deep reinforcement learning with large action space," *IEEE Transactions on Network and Service Management*, vol. 17, no. 4, pp. 2197–2211, 2020.

[34] S. Agarwal, F. Malandrino, C. F. Chiasserini, and S. De, "VNF placement and resource allocation for the support of vertical services in 5G networks," *IEEE/ACM Transactions on Networking*, vol. 27, no. 1, pp. 433–446, 2019.

[35] Telecom Italia, "Milano Grid," 2015.

[36] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," in *ICLR*, 2016.

[37] S. Jain et al., "B4: Experience with a globally-deployed software defined wan," *SIGCOMM Comput. Commun. Rev.*, vol. 43, p. 3–14, Aug 2013.

[38] N. Nikaein, E. Schiller, R. Favraud, R. Knopp, I. Alyafawi, and T. Braun, "Towards a cloud-native radio access network," *Advances in mobile cloud computing and big data in the 5G era*, pp. 171–202, 2017.

[39] ETSI, "Mobile-Edge Computing (MEC): service scenarios," *ETSI GS MEC-IEG 004*, 2015.

[40] Y.-i. Choi and N. Park, "Slice architecture for 5G core network," in *2017 Ninth international conference on ubiquitous and future networks (ICUFN)*, pp. 571–575, IEEE, 2017.

[41] M. Harchol-Balter, *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. USA: Cambridge University Press, 1st ed., 2013.

[42] T. Bonald and A. Proutiere, "Wireless downlink data channels: User performance and cell dimensioning," in *Proceedings of the 9th annual international conference on Mobile computing and networking*, pp. 339–352, 2003.

[43] F. Baskett, K. M. Chandy, R. R. Muntz, and F. G. Palacios, "Open, closed, and mixed networks of queues with different classes of customers," *Journal of the ACM (JACM)*, vol. 22, no. 2, pp. 248–260, 1975.

[44] D. López-Pérez, A. Ladányi, A. Jüttner, H. Rivano, and J. Zhang, "Optimization method for the joint allocation of modulation schemes, coding rates, resource blocks and power in self-organizing lte networks," in *2011 Proceedings IEEE INFOCOM*, pp. 111–115, 2011.

[45] V. Mnih et al., "Human-level control through deep reinforcement learning," *Nature*, vol. 518, pp. 529–533, Feb. 2015.

[46] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, *et al.*, "A general reinforcement learning algorithm that masters chess, shogi, and go through self-play," *Science*, vol. 362, no. 6419, pp. 1140–1144, 2018.

[47] M. Shojafar, N. Cordeschi, and E. Baccarelli, "Energy-efficient adaptive resource management for real-time vehicular cloud services," *IEEE Trans. on Cloud Computing*, 2019.

[48] M. O. Ojijo and O. E. Falowo, "A survey on slice admission control strategies and optimization schemes in 5G network," *IEEE Access*, vol. 8, pp. 14977–14990, 2020.

[49] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. The MIT Press, second ed., 2018.

[50] H. Robbins and S. Monro, "A stochastic approximation method," *The annals of mathematical statistics*, pp. 400–407, 1951.

[51] M. G. Azar, I. Osband, and R. Munos, "Minimax regret bounds for reinforcement learning," in *Proceedings of the 34th International Conference on Machine Learning* (D. Precup and Y. W. Teh, eds.), vol. 70 of *Proceedings of Machine Learning Research*, pp. 263–272, PMLR, 06–11 Aug 2017.

[52] G. Tesauro, "TD-Gammon, a self-teaching backgammon program, achieves master-level play," *Neural computation*, vol. 6, no. 2, pp. 215–219, 1994.

[53] J. Tsitsiklis and B. Van Roy, "Analysis of temporal-diffference learning with function approximation," *Advances in Neural Information Processing Systems (NIPS)*, vol. 9, 1996.

[54] S. Wang, H. Liu, P. H. Gomes, and B. Krishnamachari, "Deep reinforcement learning for dynamic multichannel access in wireless networks," *IEEE Transactions on Cognitive Communications and Networking*, vol. 4, no. 2, pp. 257–265, 2018.

[55] L. Bottou, F. E. Curtis, and J. Nocedal, "Optimization methods for large-scale machine learning," *SIAM review*, vol. 60, no. 2, pp. 223–311, 2018.

[56] S. Sen, M. Sekaran, J. Hale, *et al.*, "Learning to coordinate without sharing information," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 94, pp. 426–431, 1994.

[57] A. Tampuu, T. Matiisen, D. Kodelja, I. Kuzovkin, K. Korjus, J. Aru, J. Aru, and R. Vicente, "Multiagent cooperation and competition with deep reinforcement learning," *PloS one*, vol. 12, no. 4, p. e0172395, 2017.

[58] D. Bertsekas, "Multiagent reinforcement learning: Rollout and policy iteration," *IEEE/CAA Journal of Automatica Sinica*, vol. 8, no. 2, pp. 249–272, 2021.

[59] C. Zhang and P. Patras, "Long-term mobile traffic forecasting using deep spatio-temporal neural networks," in *Proceedings of the Eighteenth ACM International Symposium on Mobile Ad Hoc Networking and Computing*, pp. 231–240, 2018.

[60] A. Tavakoli, F. Pardo, and P. Kormushev, "Action branching architectures for deep reinforcement learning," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, 2018.

[61] T. Lattimore and C. Szepesvári, *Bandit algorithms*. Cambridge University Press, 2020.

[62] S. Arora, E. Hazan, and S. Kale, "The multiplicative weights update method: a meta-algorithm and applications," *Theory of computing*, vol. 8, no. 1, pp. 121–164, 2012.

[63] P. Auer, N. Cesa-Bianchi, Y. Freund, and R. E. Schapire, "The nonstochastic multiarmed bandit problem," *SIAM Journal on Computing*, vol. 32, no. 1, pp. 48–77, 2002.

[64] N. Liakopoulos, A. Destounis, G. Paschos, T. Spyropoulos, and P. Mertikopoulos, "Cautious regret minimization: Online optimization with long-term budget constraints," in *Proceedings of the 36th International Conference on Machine Learning,*, pp. 3944–3952, PMLR, 2019.

[65] P. Doanis, T. Giannakas, and T. Spyropoulos, "Scalable end-to-end slice embedding and reconfiguration based on independent DQN agents," in *GLOBECOM 2022-2022 IEEE Global Communications Conference*, pp. 3429–3434, IEEE, 2022.