Scalable end-to-end slice embedding and reconfiguration based on independent DQN agents

Pavlos Doanis[1], Theodoros Giannakas[2], and Thrasyvoulos Spyropoulos[1, 3]
[1] EURECOM, France, first.last@eurecom.fr
[2] Paris Research Center, Huawei Technologies, France, theodoros.giannakas@huawei.com
[3] Technical University of Crete, Greece

***Abstract*—Network slicing in beyond 5G systems facilitates the creation of customized virtual networks/services, referred to as "slices", on top of the physical network infrastructure. Efficient and dynamic orchestration of slices is needed to ensure the stringent and diverse service level agreements (SLAs) required by different services. In this paper, we provide a model that attempts to capture the problem of dynamic slice embedding and reconfiguration supporting a multi-domain setup and diverse, end-to-end SLAs. We then show that such problems can be optimally solved, in theory, with (tabular) Reinforcement Learning algorithms (e.g., Q-learning) even under, a priori, unknown demand dynamics for each slice. Nevertheless, the state and action complexity of such algorithms is prohibitive, even for very small scenarios. To this end, we propose a novel scheme based on independent DQN agents: The DQN component implements approximate Q-learning, based on simple, generic DNNs for value function approximation, radically reducing state space complexity; the independent agents then tackle the equally important issue of exploding action complexity arising from the combinatorial nature of embedding multiple VNFs per slice, multiple slices, over multiple domains and computing nodes therein. Using realistic data, we show that the proposed algorithm reduces convergence time by orders of magnitude with minimum penalty of decision optimality.**

***Index Terms*—VNF placement, Network Slicing, 5G Networks, Reinforcement Learning, Deep-Q Network**

I. Introduction

Network slicing is considered a key enabler for beyond 5G mobile networks. It facilitates the creation of customized virtual networks/services ("slices") on top of the physical network (PN) infrastructure. There are two main goals of slicing: (i) to ensure that each slice is perceived as an *isolated* network, performance-wise, with the desired Quality of Service (QoS) metric(s) governed by a service level agreement (SLA); (ii) to ensure the flexible and dynamic utilization of the limited network resources, which is the only economically viable way to ensure the stringent and diverse SLAs of the multitude of services envisioned [1], [2].

While different types of "slicing problems" have been considered in recent cellular network literature [3], 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) [4], [5], 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 [6], [7]. 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.

In this paper, we focus on the latter problem of 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. We summarize below the main contributions of our work:

(C.1) In Section II, we provide a model that attempts to capture the problem of dynamic slice embedding and reconfiguration supporting a multi-domain setup and diverse, end-to-end SLAs . Most of the existing works on slice embedding focus on solving the "one shot" optimization problem (i.e., based on average and/or static demands) [6], [7], while others tackle the dynamic problem but for a single domain, slice or VNF per slice [8], [9], or do not account for the impact of reconfiguration in network/slice performance [10], [11].

(C.2) In Section III, we discuss how such problems can be optimally solved, in theory, with tabular Reinforcement

The research leading to these results has been supported in part by the H2020 MonB5G Project (grant agreement no. 871780) and in part by the H2020 SEMANTIC Project (grant agreement no. 861165). This research was conducted while Theodoros Giannakas was with EURECOM, Sophia-Antipolis.

Learning (RL) (Q-Learning) algorithms even under, a priori, unknown demand dynamics for each slice. While such methods are inapplicable in realistic problem sizes, they are useful in providing a baseline for approximate ones, in small enough setups, as well in grounding more advanced algorithms with good theoretical properties [12].

(C.3) To deal with the prohibitive state and action complexity of tabular RL algorithms, we propose a novel scheme based on independent DQN agents: The DQN (Deep-Q Network) component implements approximate Q-learning, based on simple, generic Deep Neural Networks (DNNs) for value function approximation, radically reducing state space complexity; the independent agents then tackle the equally important issue of exploding action complexity arising from the combinatorial nature of embedding multiple VNFs per slice, multiple slices, over multiple domains and computing nodes therein (Section III). Using realistic data, we show that the proposed algorithm reduces convergence time by orders of magnitude with minimum penalty of decision optimality (Section IV). A key novelty of our work is that the proposed multi-agent DQN scheme is validated both in terms of scalability and optimality, as well as tested under a real-traffic dataset.

II. System Model

In this section, we present our system model. We start with the physical and virtual network models, and related KPIs; we then explain the state, action, and rewards involved in the online optimization problem.

A. Physical Network and Slices

As is common in related literature, we represent both the underlying physical network (PN), and the VNF chains ("slices") to be deployed on top of it, by graphs (e.g. [6], [7]).

Physical Network is a weighted undirected graph $G = (\mathcal{V}, \mathcal{E})$, possibly comprising multiple (technological or admin) domains (e.g., Cloud RAN (CRAN), Multi-access Edge Computing (MEC), Core Network (CN)).

Node capacity b_v is the capacity of (physical) node v to host VNFs of that domain (it could also be 0 for some nodes, e.g. for routers). This capacity could be resource blocks, cores, containers, etc., depending on the domain.

Link capacity b_e is the link capacity of edge (or path) e between two PN nodes (e.g., bandwidth).[1]

In the example of Fig. 1, there are 3 domains (CRAN, MEC, CN), with 3, 2, and 3 nodes where VNFs could be executed respectively and related capacities b_v. Nodes are connected through physical links (of capacity b_e), or paths comprising multiple links and nodes of the network.

Network Slices. We assume that a set of slices $\mathcal{K}$ (K in total) must be hosted on top of the PN. We slightly generalize the common view of a slice, in related 5G literature, and depict a slice k as follows:

A directed graph ("VNF chain") $H_k = (\mathcal{N}_k, \mathcal{L}_k)$ of VNFs (set $\mathcal{N}_k$) that model the various processing tasks required by flows of this slice, and directed (virtual) links (set $\mathcal{L}_k$) indicating the order of how these tasks are applied. Our model is fairly generic, allowing for both *loops* (e.g., flows passing by the same VNF multiple times), as well as probabilistic routing of flows (e.g., to capture the scenarios where not all flows of a slice require all VNFs in the same order). An example of such a chain is depicted in Fig. 1, where for slice 2, a percentage of flows from VNF1 proceed to VNF2 directly, while the rest must pass through VNF3, possibly going back to VNF1 as well. A simplified example that maps this slice modeling to a real service is the following video streaming slice: the traffic must first receive some baseband processing at the CRAN and then traverse a Firewall VNF at the MEC. Only a sample of (possibly malevolent) packets will also go through a Deep Packet Inspection VNF at the MEC for further processing before the billing VNF at the CN, while the rest will go straight for billing [10].

VNF demands $d_{k,n}$***:*** Each VNF n of slice k is associated with a resource demand $d_{k,n}(t)$ at each time t, that will be imposed on the PN node where the VNF is executed.

Virtual link (VL) demands $d_{k,l}$***:*** Similarly, each (virtual) link l of slice k has resource demand $d_{k,l}(t)$.[2]

Remark: These demands are often unknown, stochastic, non-stationary (correlation between VNFs of the same slice is also common); the main reason why we require a learning-based optimization algorithm to tackle this problem.

Demand D_t***:*** Let vector D_t denote the VNF and VL demands of all slices at time t.

Service Level Agreements q_k***:*** Each slice k comes with some slice-specific requirement q_k, which defines a maximum (or minimum) value for an end-to-end KPI metric.

Control variables $x_{k,i,j}$***:*** equal to 1 when VNF/VL i of slice k is hosted by node/link j, and 0 otherwise.

Configuration C_t***:*** Let vector C_t denote the configuration (embedding) of all VNFs to physical nodes at time t.

As an example, in Fig. 1, VNF 1 of slice 1 is hosted by node 2, hence $c_{1,1} = 2$, and the configuration is $C_t = (c_{1,1}, c_{1,2}, c_{2,1}, c_{2,2}, c_{2,3})$.

B. Operational Costs of Physical Network Infrastructure

Given the (usually unknown) demands D_t and the configuration C_t at time t, we assume that the system suffers an instantaneous cost related to both the network performance (i.e. direct cost to the operator) and slice performance (e.g., indirect cost related to SLA violations). We choose to consider the following cost quantities in this work (other components can be straightforwardly added to the framework):

Type 1 cost : Node utilization. Accounts for energy consumption expenses and it is equal to the number of active servers. The idle servers can be set to sleep mode and save energy, while minimizing this cost also facilitates the admission of

[1] W.l.o.g., we assume that the routing path between any two nodes is predetermined, to simplify our discussion. Routing variables could be easily included in our framework.

[2] Note that this load will be added to all physical links along the PN path between the execution nodes hosting the two VNFs connected by virtual link l.

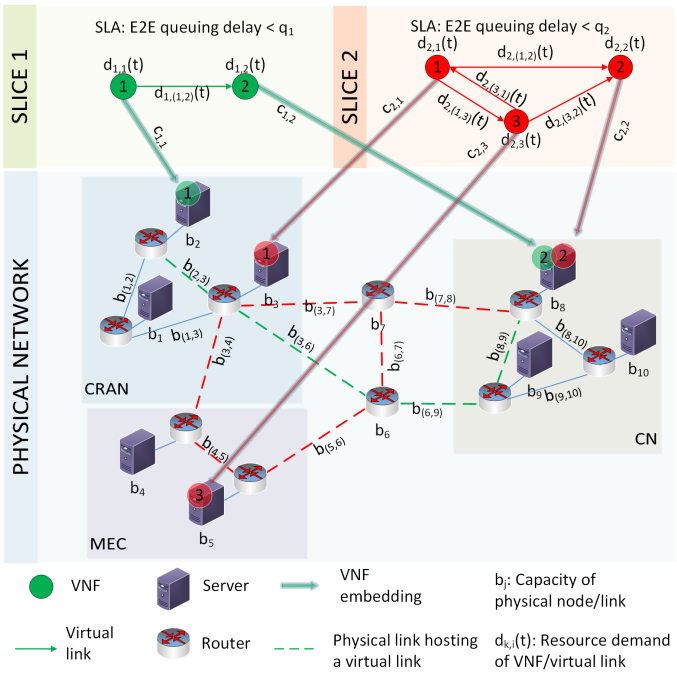


Fig. 1. Graphical illustration of the system model. The physical network consists of three technological domains, CRAN, MEC, and CN. The embedding of 2 slices onto the network is depicted.

new slices by maximizing the free space of resources [13]. We define it as:

$$r_1(C_t) = \sum_{v \in \mathcal{V}} \beta_v, \text{ with } \beta_v = 1 \text{ if } \sum_{k \in \mathcal{K}} \sum_{n \in \mathcal{N}_k} x_{k,n,v}(t) \geq 1 \tag{1}$$

Type 2 cost: Reconfiguration. The cost for migrating VNFs from their host servers to other servers in the PN. It relates to the overhead generated for reassigning all VNFs and the delays incurred by this action, which may lead to penalties for SLA violations [4], [9]. We define it as:

$$r_2(C_t, C_{t+1}) = 1/2 \cdot \sum_{k \in \mathcal{K}} \sum_{n \in \mathcal{N}_k} \sum_{v \in \mathcal{V}} |x_{k,n,v}(t+1) - x_{k,n,v}(t)| \tag{2}$$

Type 3 cost: SLA violation. When the maximum value q_k defined by the SLA is exceeded, a penalty is paid to the slice tenant. This penalty may take any form that is appropriate to model the impact of violating the corresponding KPI (e.g., linear, quadratic, etc.). For example the linear form is:

$$r_3(C_t, D_t) = \sum_{k \in \mathcal{K}} (f_k(C_t, D_t) - q_k) \tag{3}$$

where $f_k(C_t, D_t)$ gives the corresponding end-to-end KPI metric (performance) of slice k as a function of the configuration and the demand.

We give two examples of end-to-end KPIs:

a) Queuing delay: Assuming an M/G/1/Processor Sharing (PS) type of scheduler (an M/G/1/PS processor with classes has been shown to be a good approximation for many proportionally fair wireless schedulers [14]), we can calculate the mean delay experienced by a VNF/VL on the host node/link j, by [15] [3,4] :

$$f_j(C_t, D_t) = \frac{1}{b_j - z_j(C_t, D_t)}, \tag{4}$$

$$\text{where } z_j(C_t, D_t) = \sum_{k \in \mathcal{K}} \sum_{i \in \mathcal{N}_k \cup \mathcal{L}_k} d_{k,i}(t) \cdot x_{k,i,j}(t) \tag{5}$$

Then, in the case of a simple chain slice, the end-to-end queuing delay $f_k(C_t, D_t)$ is the sum of the delays on the traversed nodes and links. As an example, in Fig. 1, the delay experienced by slice 1 is the sum of the delays in node 2, link (2,3), link (3,6), link (6,9), link (8,9), and node 8. In the case of a more complex slice, a Jackson network type of analysis could be applied to calculate the delay [15]. In the remainder we focus on simple chain slices without loops. Note that this modeling captures the resource allocator scheduler impact.

b) Underprovision: A penalty is paid when the aggregate demand of the VNFs/virtual links embedded on a physical node/link exceeds its nominal capacity [5]:

$$f_j(C_t, D_t) = \begin{cases} z_j(C_t, D_t) - b_j & \text{if } z_j(C_t, D_t) > b_j \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

It can be used as a proxy for slice performance on a physical node/link when details about queuing delay are not available.

C. Reinforcement Learning Formulation

Our goal in the dynamic embedding problem is therefore to decide the configuration C_t at every time t, (i) towards optimizing the total system cost (consisting of the various cost components), while (ii) not knowing a priori how demands D_t evolve over time. This is an online learning and control problem, for which Reinforcement Learning (RL) schemes are a natural candidate. Below, we first define the main components of any RL scheme, namely its state and action space, and the rewards (often referred to as "Instantaneous Cost" in minimization problems [12]). Then, in Sec. III we present various RL algorithms to maximize these rewards (or minimize the costs) over an infinite time horizon.

Definition 1 (State $S_t = (C_t, D_t)$)**.** *We will assume that the state S_t of the system at any time t consists of (a) the slices' configuration C_t on top of the physical network, and (b) the currently observed resource demand D_t.*

Definition 2 (RL agent action A_t)**.** *We assume that at time t, the action A_t that our (RL) algorithm needs to decide is a good (re-)configuration C_{t+1} (without knowing the future demand D_{t+1})*

Definition 3 (Reward r_t)**.** *Given some observed state S_t, an action A_t taken by the RL agent, and the next state S_{t+1}, the reward obtained is the weighted sum of (1), (2), (3):*

$$r_{t+1} = -(w_1 \cdot r_1(S_{t+1}) + w_2 \cdot r_2(S_t, A_t) + w_3 \cdot r_3(S_{t+1})) \tag{7}$$

[3]We assume that two VNFs of the same slice cannot be executed in the same node.

[4]In order to avoid degenerate cases where the formula (4) would give negative values when total demand exceeds capacity, in our simulations we linearize the part of $b_j - z_j(C, D) \in [0.01, -\infty]$ with a straight line equal to the slope at $b_j - z_j(C, D) = 0.01$

III. RL Algorithms

In this section, having defined the basic RL components, we discuss how different RL schemes could be applied to our problem; we begin with a theoretically optimal yet practically inapplicable standard approach, then build towards our own proposal which attempts to have close-to-optimal performance yet with greatly improved convergence and scalability properties for practical setups.

A. State and Action Complexity: an Example

Given the RL formulation $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \gamma)$, where $\mathcal{S}, \mathcal{A}, \mathcal{R}$ are the sets of all possible states, actions, rewards respectively and γ is the discount factor, we can apply a tabular version of Q-learning with epsilon-greedy policy [12]. In such setups, a $Q(S, A)$ is maintained *for all* state-action pairs, and is updated as follows:

$$(1-\alpha)Q(S_t, A_t) + \alpha(r(S_t, A_t, S_{t+1}) + \gamma \max_{A \in \mathcal{A}} Q(S_{t+1}, :)) \tag{8}$$

Such schemes provably converge to the global optimal solution of the problem, without any a priori knowledge of the demands D_t [16] [5]. The important downside of such "tabular" algorithms is that *every possible state-action pair* must be encountered on training enough times each, in order to ensure convergence to a good estimate of the respective Q-value.

As an example, consider the moderate case of a single domain, with $K = 10$ slices. Each slice consists of only one VNF (assume no VLs); we also have $V = 5$ servers, and each VNF resource demand can take one out of $B = 4$ distinct values. Even in such a single domain scenario, the state space already explodes to $|\mathcal{S}| = 5^{10} \cdot 4^{10}$. The first component of the product corresponds to all the possible placements of the 10 VNFs in 5 computation nodes, while the second part to all possible states of the demand vector D_t. The same holds for the action space which is already of size $|\mathcal{A}| = 5^{10}$, over which the max operation in (8) must be taken *at every training step!* This simple example reveals the notorious curse of dimensionality standard RL schemes exhibit even in relatively small sized problems of combinatorial nature.

B. Approximation 1: Deep Q-Network

We, thus, need to radically improve both the state complexity and the action complexity of the above scheme. A recent and successful approach [17] is to learn a parameterized function $Q(S, A) = f(S; \theta)$ (with the function commonly being a DNN), instead of the entire Q-table previously.

The agent observes a state S_t, and computes $f(S_t; \theta)$; for *given* θ, the DNN outputs an estimate for the Q value of every available action A_t. Observe that unlike tabular Q-learning, where the state space must be discrete, we can now feed f with continuous resource demands and not use discretized levels. The DQN comes with some additional components. In every step, we do a forward pass over the so-called policy network, that dictates which action to take. We observe a reward $r_{t+1}(S_t, A_t, S_{t+1})$ and the system evolves to S_{t+1}. We store experiences $(S_t, A_t, r_{t+1}, S_{t+1})$ to a replay buffer; from this we sample data points for which we perform the gradient step. Such use of data improves efficiency (we need fewer samples to learn). Then, we update the policy network via stochastic gradient descent using samples from the buffer. For one sample, the update would be:

$$\theta^{k+1} = \theta^k + \nabla_\theta (Q_\theta(S_t, A_t) - \text{target}(S_{t+1})), \tag{9}$$

where $\text{target}(S_{t+1}) = r_{t+1} + \gamma \max_{A \in \mathcal{A}} Q_{\theta'}(S_{t+1}, :)$; the latter quantity is computed by a DNN (target network) which is updated every T (a tuning parameter) for stability reasons.

Drawbacks. With DQN, we learn a vector θ (a vector of size as equal to the DNN trainable parameters) instead of learning $|\mathcal{S}| \cdot |\mathcal{A}|$ parameters. However, DQN was designed for control problems with small action spaces (in [17], there are less than 20 actions always), and does not scale well for very large action spaces. In our case, larger problem size means (a) (combinatorially) more outputs/actions to be computed of $f(S_t; \theta)$, and (b) harder argmax operations

C. Approximation 2: Independent DQN agents

While state space has been the main issue in mainstream Deep RL problems (e.g., Atari [17]), action space is often the bottleneck in such combinatorial nature problems arising in networking setups. A natural approach for such problems where the action space can be decomposed is multi-agent: in our case, the $\mathcal{A}$ naturally decomposes into action subspaces per VNF (or per slice). We therefore apply an IDQN approach, where each VNF operates as an independent agent.

Each agent maintains a DNN and has its own set of parameters θ_i, its policy and target networks, and replay buffer (as above). When system state S_t is observed (all agents view the global state), each agent/slice i decides to which server it must migrate next. After each agent has taken an action using its policy network; the global action is formed, a reward and a new state are observed. These are broadcast to all agents' buffers, and then each agent makes a gradient update as

$$\theta_i^{k+1} = \theta_i^k + \nabla_{\theta_i} (Q_{\theta_i}(S_t, A_t) - \text{target}_i(S_{t+1})). \tag{10}$$

The take-away is that we have bypassed computationally heavy step $\max_A Q_\theta(S, :)$ of (9) for so many actions; in fact, for each agent the available actions now became linear $O(V)$ on the number of servers, a significant improvement! And moreover, we can keep less parameters (θ_i) as we no longer have to learn the $Q(S, A)$ for so many actions.

Remark: These agents, while they can be distributed on the infrastructure (e.g., collocated with the respective VNF, something that has latency advantages in 5G networks) is first and foremost an algorithmic concept, to decompose (and reduce) the action complexity; multi-agent solutions for (possibly centralized) MDP problems have resurged recently [12].

IV. Simulation Results

In this section, we have two main goals: first, to establish that approximate solutions DQN and IDQN are able to con-

[5] In the case of known and Markovian traffic D_t, our problem is a standard Markov Decision Problem (MDP), and can be solved with value iteration [16]. We will use such an MDP solution as a baseline in some simple examples of Section IV.

verge close to optimal solutions (i.e. ones found by tabular Q-learning) yet with much higher convergence speed, as the problem size increases; second, to validate our proposed solution for such problems, our IDQN implementation algorithm, in realistic enough setups. As a result, the validation section consist of two main parts, the former using small enough scenarios (to obtain the optimal policy) with synthetic data (for sensitivity analysis), while in the latter we explore the performance of the IDQN solution using real data from the Milano dataset [18], in more realistic topologies with much larger state and action baseline complexity.

Policies. Below we list various algorithms that we will use in our validation, together with some key parameters for each. Note that *not* all algorithms will or can be used in every scenario (e.g., the first two take more than days to converge, except in the smaller Scenario 1).

- *Policy Iteration (PI):* Returns the optimal policy in an offline fashion, and is applicable when the traffic dynamics are Markovian and known.
- *Q-Learning (QL):* This is the RL tabular method (Section III, (8)) that returns the optimal policy (for discrete traffic demands).
- *DQN:* This is the centralized approximation method (Section III, (9)), using a DNN of 3 layers and 60 neurons per layer. We set its replay memory size to 5000 timeslots, the target update period to 500 timeslots and the minibatch to 32, since these parameters performed well in a variety of tested scenarios.
- *IDQN:* This is the proposed multi-agent approximation method (Section III, (10)). Each agent (one agent per VNF) is a DNN of 3 layers and 60 neurons per layer. It uses the same parameter values with the DQN.
- *Group-all:* A reference static policy, whose main goal is to merely minimize the node utilization cost (1); it places all VNFs in one server and does not react to changing demands (hence no reconfiguration cost either), but possibly suffering from frequent SLA violations.
- *Split-all:* A sister reference policy to Group-all which instead aims to minimize SLA violations (3) only, trying to "spread out" VNFs among all available servers as much as possible (no reconfiguration cost either).
- *Random:* It chooses randomly one of the possible configurations at each timestep.

Note that the discount factor was set to $\gamma = 0.9$ for all RL/MDP algorithms.

A. Part I: Scalability of tabular vs. approximate RL schemes

In this section, we focus on small scenarios, and synthetic Markov traffic (defined below), so that theoretically optimal algorithms (Policy Iteration and Q-learning) can be used as baselines (converge in reasonable time). Our aim is to carefully study how the increase of problem size (K: number of slices, V: number of servers) affects the convergence speed of the approximations, as well as their attained cost, compared to the optimal. For this reason, in terms of cost performance we use PI and QL as benchmarks (both find the optimal cost), and check how DQN and IDQN perform against them (i.e. how far from the optimal). And moreover, we observe the convergence speed of QL against DQN and IDQN.

Network Setup and Markov Traffic. We consider a *single domain* physical network and two scenarios (a small and a larger) with different problem sizes, i.e. V and K. Without loss of generality, we assume each slice consists of one VNF. The demand of each VNF $d_k(t) \in \mathcal{B} = \{0, 1\}$ ("ON/OFF") and evolves, independently to the other VNFs, as a Markov process with transition probability matrix

$$\mathbb{P}_k = \begin{bmatrix} 0.98 & 0.02 \\ 0.02 & 0.98 \end{bmatrix} \tag{11}$$

This captures a very simple scenario where each slice has bursty traffic periods followed by long silence periods, not necessarily coinciding, to better illustrate the optimality of the chosen actions, as well as the performance of static heuristics. Using (11) we generate a training and a testing dataset, with duration $2 \cdot 10^6$ and $8 \cdot 10^4$ time-slots respectively. The algorithms are trained and tested 10 times in the respective datasets with different initial random seeds. For the SLAs, we consider the underprovision SLA (Section II, (6)) where $q_k = 0$ for all slices, and a quadratic penalty for violations.

Scenario 1. Here, we consider a PN with $V = 2$ servers and $K = 4$ slices to be configured (4096 state-action pairs).

-Convergence Speed (Fig. 2(a)): We depict the average cost value (over 10 runs, on the y-axis) as a function of time (counted in terms of iteration, on the x-axis), for the 3 main algorithms of Section III. The two main points to observe are the following: (i) the approximate solutions, DQN and IDQN both achieve similar costs with the theoretically optimal QL (this has been confirmed for a variety of other small scenarios as well); (ii) yet, it is impressive that QL already takes quite a few more iterations to converge, even in such a small scenario (this is not so surprising since the size of the Q-value table is already 256 by 16); (iii) since the action space in this scenario is quite small, there are no additional convergence advantages by IDQN (compared to DQN), as expected.

-Cost performance (Fig. 2(b)): We depict the cost per timeslot in the testing dataset via a box plot (for all the algorithms outlined in the beginning of the Section). The main observations are: (i) the approximations (DQN and IDQN) have no problem finding the optimal solution in such a scenario; (ii) even on this tiny scenario, we get a 20% improvement compared to the simple static heuristics; and a 60% compared to the random policy.

Take-away message 1: DQN based approximate policies are able to find good quality solutions (certainly better than static reference policies), and quite faster than tabular Q-learning.

Scenario 2 (Fig. 2(c)): We increase the problem size (w.r.t. Scenario 1), and assume a physical network with $V = 3$ servers that hosts $K = 7$ slices (10^{10} state-action pairs). While this is still not a very large scenario, in practice, policy iteration and Q-learning already collapse, due to their memory and computation requirements; we therefore omit these from the respective plot. So, Fig. 2(c) is similar to Fig. 2(a), yet

we only compare DQN and IDQN. An important observation here is that IDQN now presents considerable convergence time improvements (roughly $10\times$) compared to DQN; this is reasonable as the action space in this scenario is growing large (2187 actions per state). Nevertheless the solution quality (average cost achieved) is comparable.

Take-away message 2: for realistic size scenarios, even centralized DQN will collapse in terms of convergence time.

B. Part II: IDQN vs Heuristics

Having established that IDQN agents converge reasonably fast, using synthetic data scenarios, in this last part we examine their performance in a more realistic scenario (multiple domains, more nodes, end-to-end queuing delay SLAs, and real traffic data from the Milano dataset [18] to drive the demand).

Network Setup. We consider a physical network with two domains, CRAN and CN, with $V_{\text{CRAN}} = 9$ and $V_{\text{CN}} = 3$ servers respectively. There are also 10 slices, each of them comprising two VNFs, one for each domain (for simplicity and without loss of generality we assume there are no VLs involved). We also include end-to-end metrics: SLAs concern the maximum end-to-end queuing delay (linear penalty for SLA violations, see (Section II, (4)).

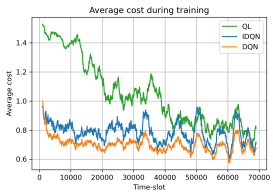
Data Preprocessing. The dataset contains 10K timeseries of base stations (BS) (with unknown statistics), each with 8926 demand values. In particular, we map each VNF to the normalized demand of a BS; and to make the simulation more realistic, we choose highly correlated timeseries for VNFs that belong to the same slice. So we pick 20 timeseries, one for each VNF. We use half of the dataset data points for training and the rest for testing.

Scenario 3 (Fig. 2(d)): We depict the cost per time-slot in the testing dataset via a box plot (for IDQN, Split-all, Group-all, and random policy) as produced after 10 runs of training-testing with different initial random seeds. DQN is omitted since it is not possible to apply it in practice for this scenario size (it would require a DNN with 2×10^{14} output neurons). The main observations are: (i) the cost achieved by IDQN is 43% lower compared to the Split-all policy, which was the best of our baselines; (ii) even the worst policy obtained by IDQN in the 10 runs performs much better than the baselines.

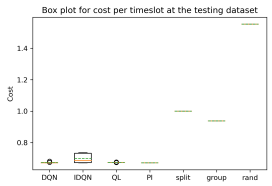
Take-away message 3: IDQN is scalable and provides significantly better quality policies compared to static baselines.

V. Conclusion

In this paper, we gave a flexible model for the slice embedding and reconfiguration problem, suitable for multi-domain setups and diverse end-to-end SLAs. We proposed a novel DRL scheme based on independent DQN agents that radically reduces both the state and action complexity to speed up convergence, and improves cost performance against static policies. Possible future work includes multi-agent schemes that consider coordination among different agents to increase robustness.

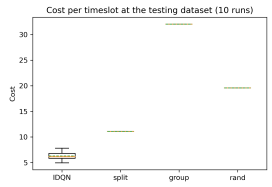


(a) Convergence plot (scenario 1)



(b) Testing stage cost (scenario 1)

(c) Convergence plot (scenario 2)



(d) Testing stage cost (scenario 3)

Fig. 2. Simulation results for the three tested scenarios

References

[1] I. Afolabi, T. Taleb, K. Samdanis, A. Ksentini, and H. Flinck, "Network slicing and softwarization: A survey on principles, enabling technologies, and solutions," *IEEE Comms. Surveys Tutorials*, 2018.

[2] X. Foukas, G. Patounas, A. Elmokashfi, and M. K. Marina, "Network slicing in 5g: Survey and challenges," *IEEE Comms. Magazine*, 2017.

[3] 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.

[4] 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.

[5] 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.

[6] 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 Comms. Magazine*, 2017.

[7] 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.

[8] M. Leconte, G. S. Paschos, P. Mertikopoulos, and U. C. Kozat, "A resource allocation framework for network slicing," in *IEEE INFOCOM*, 2018.

[9] 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 TNSM*, 2020.

[10] 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 TON*, 2019.

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

[12] D. Bertsekas, *Reinforcement Learning and Optimal Control*. Athena Scientific, 2019.

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

[14] T. Bonald and A. Proutière, "Wireless downlink data channels: User performance and cell dimensioning," in *MobiCom*, 2003.

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

[16] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. The MIT Press, 2018.

[17] V. Mnih et al., "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015.

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