Multi-agent DQN with sample-efficient updates for large inter-slice orchestration problems

Pavlos Doanis[1] and Thrasyvoulos Spyropoulos[1, 2]
[1] EURECOM, France, first.last@eurecom.fr
[2] Technical University of Crete, Greece

***Abstract*— Data-driven network slicing has been recently explored as a major driver for beyond 5G networks. Nevertheless, we are still a long way before such solutions are practically applicable in real problems. Reinforcement learning based solutions, addressing the problem of dynamically placing virtual network function chains on top of a physical topology, have to deal with astronomically high action spaces (especially in in multi-VNF, multi-domain, and multi-slice setups). Moreover, their training is not particularly data-efficient, which can pose shortcomings, given the scarce(r) availability of cellular network related data. Multi-agent DQN can reduce the action space complexity by many orders of magnitude compared to standard DQN. Nevertheless, these algorithms are data-hungry and convergence can still be slow. To this end, in this work we introduce two additional mechanisms on top of (multi-agent) DQN to speed up training. These mechanisms intelligently decide how to store to, and how to pick from the experience replay buffer, in order to achieve more efficient parameter updates (faster learning). The convergence speed gains of the proposed scheme are validated using real traffic data.**

***Index Terms*—Slice orchestration, Beyond 5G Networks, Reinforcement Learning, Deep-Q Network**

I. Introduction

Network slicing is a paradigm that promises to enable one of the key-characteristics envisioned for beyond 5G networks, the reliable support of a massive number of services with widely diverse Quality of Service (QoS) requirements [1]. It leverages network function virtualization and software-defined networking technologies to create virtual networks ("slices") on top of the physical network infrastructure, which can be tailored to the needs of a specific service. The two main goals of slicing are: (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 resource demands of the hosted slices are dynamically changing (due to traffic variations), dynamic slice orchestration is necessary to accomplish the aforementioned goals [2].

A slice is a "VNF chain" comprising Virtual Network Functions (VNFs) and Virtual Links (VLs). Different optimization problems for network slicing have been considered in the literature, with the main representatives being (i) the placement (embedding) of slices onto the physical network (VNFs and VLs must be mapped to physical nodes and links respectively) [3]; (ii) the allocation of a physical node's resources to the hosted slices (users) [4], [5]. Recently we have proposed in [6] a system model that tackles the former problem by capturing also the impact of the (per node) resource allocation scheduler.

Initial attempts tried to tackle slice placement as an "one-shot" optimization problem, using mainly heuristic algorithms (due to non-polynomial complexity) [7]. More recent works formulated it as a Reinforcement Learning (RL) problem to account for the (unknown) changing VNF demands and the reconfiguration cost [8], [9]. However a number of challenges still remain: (i) most works focus on single domain setups [7] or simple VNF chains [9], and/or consider simple performance metrics [8], [9] (no end-to-end slice-specific Key Performance Indicators); (ii) RL based solutions have to deal with astronomically high action spaces [8], [9], due to the combinatorial nature of placing multiple VNFs upon multiple physical nodes (considering multiple slices/domains exacerbates this problem); (iii) data-efficient algorithms are required, given the scarce(r) availability of cellular network related data [8].

In a recent work [6], we have addressed challenge (i) by introducing a generic, queuing network based model that captures complex VNF chain topologies and end-to-end performance metrics (supporting multi-domain, multi-slice, end-to-end setups), and (ii) by a multi-agent algorithm of independent Deep-Q-Network (iDQN) agents that can reduce the action space complexity by many orders of magnitude compared to standard, single-agent, Deep-Q-Network (DQN). Nevertheless, convergence can still be slow, requiring a large amount of training data. To this end, here we focus on improving the training speed of DQN agents (challenge (iii)), by introducing two mechanisms to store to and select from the experience replay buffer (for more efficient parameter updates). We summarize below the main contributions:

(C.1) We introduce a prioritized experience replay [10] on top of standard DQN (either single-agent or multi-agent) (Section III-C) and investigate its performance gains in the inter-slice orchestration problem (including a sensitivity analysis on its hyperparameters) (Section IV-A).

(C.2) We introduce a mechanism that stores some additional information per experience to reduce the number of computations during parameter updates (Section III-C).

(C.3) We validate the proposed multiagent DQN scheme with all the above speed up extensions (iDQN+) in a large scale scenario, and confirm its superior performance compared to vanilla iDQN and static baselines (Section IV-B).

The research leading to these results has been supported in part by the H2020 SEMANTIC Project (grant agreement no. 861165) and in part by the H2020 MonB5G Project (grant agreement no. 871780).

II. RL FRAMEWORK FOR INTER-SLICE ORCHESTRATION

A. The Environment: A Beyond 5G system

In this work we adopt the system model of [6] (the reader can refer to that paper for a more detailed description). Fig. 1 outlines its main components, the physical network and network slices, as well as the notation, in a toy example[1].

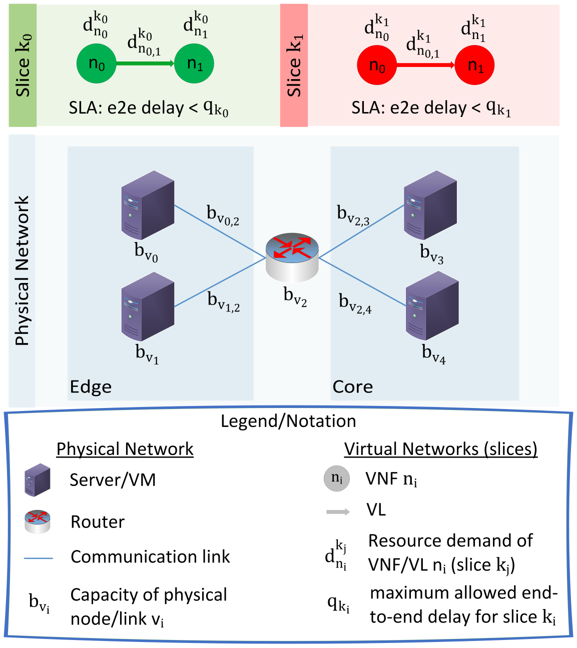


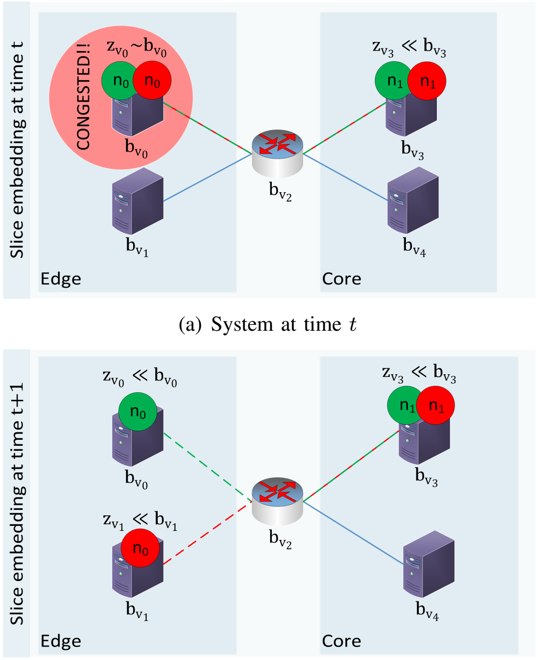
Fig. 1. Toy example exhibiting the main components of our environment.

Slice orchestration. The assignment of each VNF/VL to a physical node/link[2] (at every timestep t) determines both the slice and the network performance. Slice performance deteriorates by SLA violations while network performance by unnecessary use of network resources. Hence, a VNF might "migrate" to another node at a next time unit with a view to improve system's performance. But migration of VNFs does not come for free; a reconfiguration cost should be taken into account (e.g. management overhead, delays leading to monetary penalties [5], [9]). See Fig. 2 for an example.

Configuration $c \in \mathcal{C}$: mapping of all VNFs to physical nodes[3] at time t. (e.g. in Fig. 2(a), the configuration is: $c = (c^{k_0}_{n_0}, c^{k_0}_{n_1}, c^{k_1}_{n_0}, c^{k_1}_{n_1}) = (v_0, v_3, v_0, v_3)$, where $c^{k_j}_{n_i}$ indicates the host node of VNF n_i, slice k_j).

Demand $d \in \mathcal{D}$: denotes the resource demands of all hosted slices at time t (e.g. in Fig. 1, the demand is: $d = (d^{k_0}_{n_0}, d^{k_0}_{n_1}, d^{k_0}_{n_{0,1}}, d^{k_1}_{n_0}, d^{k_1}_{n_1}, d^{k_1}_{n_{0,1}})$).

Queuing Model. The impact on the performance of slices when multiple VNFs are assigned to the same physical node and their aggregate resource demand is close to or exceeds the node's capacity is captured using a queuing model. Each physical node/link is modeled as an M/G/1/PS queue (Memoryless arrivals (Poisson)/General distribution of service times/1 server/Processor Sharing scheduling) [11]. Such models tend to capture well the characteristics of proportionally fair schedulers, commonly used for resource scheduling [12]. Then, the average delay experienced by any VNF/VL hosted on node/link v_i is given by the function[4]:



(a) System at time t

(b) System at time $t+1$

Fig. 2. Slice orchestration example. (a) initial embedding of slices (at time t) for the toy scenario of Fig. 1. The aggregate resource demand of the hosted VNFs at physical node v_0 ($z_{v_0} = d^{k_0}_{n_0} + d^{k_1}_{n_0}$) is close to b_{v_0}; the node is congested and causes high delays (and probably SLA violations) for both slices k_0 and k_1. (b) at the next timestep ($t+1$), VNF n_0 of slice k_1 migrates to v_1 to avoid SLA violations, in exchange with i) a reconfiguration cost; ii) a node activation cost (server v_1 is turned "on" from the idle state).

$$f^{delay}_{v_i}(c,d) = \frac{1}{b_{v_i} - z_{v_i}(c,d)}, \qquad (1)$$

$$\text{where } z_{v_i}(c,d) = \sum_{k_j \in \mathcal{K}} \sum_{n_l \in \mathcal{N}_{k_j} \cup \mathcal{L}_{k_j}} d^{k_j}_{nl} \cdot x^{k_j}_{nl,v_i}. \qquad (2)$$

In (2), $\mathcal{K}$ is the set of slices, $\mathcal{N}_{k_j}$, $\mathcal{L}_{k_j}$ the sets of VNFs and VLs of slice k_j respectively, and $x^{k_j}_{nl,v_i}$ a binary variable (1 if VNF/VL n_l of slice k_j is hosted by physical node/link v_i). For complex slices the end-to-end delay can be calculated by a Jackson network type of analysis. For a simple slice (VNF chain) the corresponding end-to-end delay $F^{delay}_{k_i}(c,d)$ is the sum of delays across all host nodes and links along its path (e.g. in Fig. 2(a) the delay of slice k_0 is: $F^{delay}_{k_0}(c,d) = f^{delay}_{v_0}(c,d) + f^{delay}_{v_{0,2}}(c,d) + f^{delay}_{v_{2,3}}(c,d) + f^{delay}_{v_3}(c,d)$).

B. States, actions, and rewards

State space $\mathcal{S}$. The state of the system at timestep t consists of (i) the assignment of all VNFs to physical nodes; (ii) the

[1] We stress that the above system model goes well beyond the depicted toy example, being able to capture VNF chains with probabilistic routing between VNFs, loops (i.e. traffic potentially going through the same VNF more than once), etc. See [6] for more details.

[2] A VL can be mapped to a path (its load is imposed to each traversed link).

[3] To simplify our discussion, and w.l.o.g., we consider the mapping of VLs is predetermined. Routing variables could be easily included in our framework.

[4] When demand exceeds capacity, (1) is extended to include a large penalty.

resource demands of all VNFs/VLs (both are necessary to calculate the instantaneous reward, to be elaborated shortly).

Definition 1 (State). $s = (c, d),\ s \in \mathcal{S} = \mathcal{C} \times \mathcal{D}$

Remark: The above state space $\mathcal{S}$ grows exponentially fast with slice and VNF number. What is worse, continuous traffic demand (as will be the case for the dataset used in our simulations [13]) essentially renders vanilla RL methods (e.g. Q-learning) inapplicable, even for toy scenarios. To this end, approximate RL methods (e.g. using a Deep Neural Network (DNN) to encode the input) cannot be avoided for such problem. We are using the popular DQN architecture [14] to this end.

Action space $\mathcal{A}$. The agent's action is the configuration to be applied in the next timestep (combinatorial). Note that we use an apostrophe to denote all quantities of $t + 1$.

Definition 2 (Action). $a = c',\ a \in \mathcal{A} = \mathcal{C}$

Action complexity example. In a physical network with $V = 10$ nodes and $K = 10$ slices (one VNF per slice), the number of possible actions is $|\mathcal{A}| = V^K = 10^{10}$! Standard DQN algorithms [14] are designed for small action spaces. To deal with this problem (on top of state space complexity), we use a multi-agent DQN architecture , that can reduce action state complexity by orders of magnitude [6].

Reward function $\mathcal{R}$. We consider three individual costs that determine the total cost performance of the system. Given some observed state s, an agent action a, and the next state s', these costs are the following.

Type 1 cost: SLA violation. A penalty is paid when the end-to-end delay $F^{delay}_{k_i}(s')$ perceived by a slice k_i is higher than q_{k_i} (defined by the SLA). This may take any suitable form (linear, quadratic, etc.). We give as an example the linear form:

$$g_1(s') = \sum_{k_i \in \mathcal{K}} (F^{delay}_{k_i}(s') - q_{k_i}) \cdot \mathbf{1}_{\{F^{delay}_{k_i}(s') > q_{k_i}\}}. \tag{3}$$

Type 2 cost: Reconfiguration. Migrating VNFs from their host servers causes network overhead or even service downtime [5], [9].

$$g_2(s, a) = 1/2 \cdot \sum_{k_i \in \mathcal{K}} \sum_{n_j \in \mathcal{N}_{k_i}} \sum_{v_l \in \mathcal{V}} |(x^{k_i}_{n_j, v_l})' - x^{k_i}_{n_j, v_l}|, \tag{4}$$

where $\mathcal{V}$ is the set of physical nodes.

Type 3 cost : Active nodes. It is the number of physical nodes that are "on" (hosting at least one VNF). The idle servers/VMs can be turned off (or set to sleep mode) and save energy/free up resources [15].

$$g_3(a) = \sum_{v_i \in \mathcal{V}} \mathbf{1}_{\{\sum_{k_j \in \mathcal{K}} \sum_{n_l \in \mathcal{N}_k} x^{k_j}_{n_l, v_i} \geq 1\}}. \tag{5}$$

Definition 3 (Reward r)**.** *The reward obtained is the negative weighted sum of the individual costs*[5] *(3) - (5):*

$$r = -(w_1 \cdot g_1(s') + w_2 \cdot g_2(s, a) + w_3 \cdot g_3(a)). \tag{6}$$

III. Approximate RL schemes

As is evident by the problem model, the inter-slice orchestration problem at hand is characterized by (i) unknown future resource demands; and (ii) delayed rewards (e.g. if the demand of a VNF is predicted to increase soon and stay high for a while, paying now a reconfiguration cost for its migration to a less busy server could lead to high future rewards). While this is the standard "playground" of RL, vanilla algorithms like Q-learning [16] are unable to handle the problem at hand, due to the prohibitive state and action spaces even in relatively small setups. We will first describe here the basic DQN and iDQN (multi-agent DQN) solutions we use as our starting point to deal with state and action complexity, respectively, then proceed with the proposed experience replay buffer heuristics.

A. DQN

In order to deal with the combinatorially large (potentially infinite) state space, an approach that has found significant success recently in many applications (e.g., games) is to learn a parameterized function $Q_\theta(s, a)$ (with the function commonly being a DNN), that approximates the original Q function (using much fewer learnable parameters than a complete state-action table). The advantage of a DNN is the automatic encoding of important features that would otherwise be problem dependent and tough to track (e.g. the discretization of continuous traffic demands). However, simply adding a DNN forfeits the convergence guarantees of tabular RL algorithms, and leads to unstable learning in most practical scenarios. The recent Deep Q-Network (DQN) algorithm [14] is shown to often overcome these issues, and will be our starting point.

A DQN agent is equipped with two DNNs, the so called policy and target networks ($Q_\theta(s, \cdot)$ and $Q_{\theta'}(s, \cdot)$ respectively), which take as an input the state and output the Q values of all possible actions (configurations). Moreover, the visited transitions are stored in a replay buffer ($\mathcal{B}$). In Fig. 3 we outline the main steps of the DQN algorithm.

Drawbacks. DQN does not scale well for very large action spaces. In our case, larger problem size means (a) (combinatorially) more outputs/actions for the DNN, (b) harder argmax operations, (c) slower exploration of the action space.

B. iDQN

To deal with the above bottlenecks (a) and (b), we employ a multi-agent scheme to decompose the combinatorial action space into smaller subspaces. To this end, we consider one independent DQN agent (iDQN) per VNF, responsible only for placing the specific VNF on the physical network. The introduced modifications are outlined in Fig. 4.

With iDQN we have managed to reduce memory requirements by avoiding the combinatorial output layer of the single-agent DQN scheme. Also, we have replaced the computationally expensive maximization operations of (7), (9), with much less expensive operations over $\mathcal{A}^j$. However, there is still much room for improvement regarding convergence speed and sample efficiency (drawbacks (b) and (c) of Section III-A), which are crucial characteristics for a practical algorithm.

[5] Note that the goal of the RL agents is to maximize the accumulated rewards, and this is why we introduce a negative sign in Def. 3 (in our problem we want to minimize the accumulated cost).

DQN algorithm

Step 1 (in agent): When at state s (Def. 1), take an ϵ-greedy action (Def. 2):

$$a \leftarrow \begin{cases} \text{random } a \in \mathcal{A}, & \text{with probability } \epsilon; \\ \arg\max\limits_{a \in \mathcal{A}} Q_\theta(s,a), & \text{with probability } 1-\epsilon. \end{cases} \quad (7)$$

Step 2 (in env): Returns the next state s' and reward r (Def. 3).

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 perform a gradient step on the minibatch:

$$\theta \leftarrow \theta - \eta \nabla_\theta (\mathbb{E}_{i \sim U(\mathcal{B})}[\delta_i^2]), \quad (8)$$

$$\text{where } \delta_i = Q_\theta(s_i, a_i) - (r_i + \gamma \max_{a_i' \in \mathcal{A}} Q_{\theta'}(s_i', a_i')). \quad (9)$$

In (8) η is the learning rate and δ_i the Temporal Difference (TD)-error of transition i.

Repeat steps 1 to 5 till termination criterion.

Fig. 3. Algorithmic steps of DQN

iDQN Modifications

Multi-agent scheme: One DQN agent j per VNF.

Step 1: Each agent j takes an ϵ-greedy action:

$$a^j \leftarrow \begin{cases} \text{random } a^j \in \mathcal{A}^j, & \text{with probability } \epsilon; \\ \arg\max\limits_{a^j \in \mathcal{A}^j} Q_\theta(s, a^j), & \text{with probability } 1-\epsilon. \end{cases} \quad (10)$$

Then, the collective action is:

$$a = (a^0, a^1, ..., a^{j_{max}}).$$

Step 3: Each agent stores (s, a^j, s', r).

Steps 4-5: Applied to all agents (with $\mathcal{A}^j$ instead of $\mathcal{A}$).

Fig. 4. Modifications of iDQN algorithm, with respect to DQN (Fig. 3).

DQN+ Modification 1:

Step 5 (in agent): M samples are picked from the buffer with a probability $P(i)$ for each of the N transitions:

$$P(i) = p_i^{\alpha^{\text{rep}}} / \sum_{j=1}^{N} p_j^{\alpha^{\text{rep}}}, \quad (11)$$

where $p_i = |\delta_i| + \epsilon$ and ϵ is a small positive constant. Then, the gradient step is:

$$\theta \leftarrow \theta - \eta \nabla_\theta (\mathbb{E}_{i \sim P(\mathcal{B})}[(w_i^{rep} \delta_i)^2]), \quad (12)$$

$$\text{where } w_i^{rep} = (\frac{1}{N} \cdot \frac{1}{P(i)})^{\beta^{\text{rep}}}. \quad (13)$$

Fig. 5. Modification 1 of DQN+ algorithm, with respect to DQN (Fig. 3).

DQN+ Modification 2:

Step 3 (in agent): Calculate Q^{next},

$$Q^{\text{next}} = \max_{a' \in \mathcal{A}} Q_{\theta'}(s', a'), \quad (14)$$

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:

$$\delta_i = Q_\theta(s_i, a_i) - (r_i + \gamma Q_i^{\text{next}}). \quad (15)$$

Fig. 6. Modification 2 of DQN+ algorithm, with respect to DQN (Fig. 3).

C. DQN+/iDQN+

The last step towards a more scalable solution is to improve convergence speed by (i) smarter picking of minibatches; (ii) DNN parameter updates with fewer computations (we introduce the "lazy" computation of the TD-target). Following, we demonstrate how DQN must be modified to incorporate these two speedup tricks (DQN+), which can be readily applied in the multiagent scheme (iDQN+) to further improve its convergence speed (especially in large scale scenarios).

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). To this end, we employ a prioritized experience replay [10], which prioritizes transitions with a larger Temporal Difference error (TD-error) to boost sample efficiency. Fig. 5 outlines the modifications on top of the DQN algorithm. In (11), α^{rep} is a hyperparameter that determines the amount of prioritization ($\alpha^{\text{rep}} = 0$ leads to uniform sampling while $\alpha^{\text{rep}} = 1$ to full prioritization), while β^{rep} in (13) 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 (9), required for *every* sample of the minibatch, becomes very expensive as the action space grows larger. In order to reduce the number of such computations we introduce a second modification (Fig. 6). This trick offers important real time gains, as DQN+ performs M times less computations per timestep, compared to DQN, for the calculation of the TD-target (in DQN+, (14) is calculated only for the visited transition while in DQN for each one of the M samples).

IV. Simulation results

In this section we aim to (i) find good values for the hyperparameters of prioritized experience replay in a relatively small slicing setup (allowing for sensitivity analysis in reasonable time); (ii) verify if the proposed speedup heuristics of Section III-C can improve convergence speed, and thus scalability, of DQN/iDQN (in the same setup); (iii) verify the performance gains of iDQN+ in a large scale scenario.

Algorithms. (i) DQN (the single-agent approximate RL algorithm of Section III-A); (ii) iDQN (the multi-agent approximate RL algorithm of Section III-B); (iii) DQN+/iDQN+ (these variants are the above DQN/iDQN but with all the speedup heuristics we have proposed in Section III-C).

Considering the limited training data and computing resources available, we choose to use simple and relatively small DNNs that offer better sample efficiency and faster convergence compared to larger and more complex networks. To this end, all agents use multilayer perceptron DNNs with 3 hidden layers and 60 neurons per layer. Regarding their hyperparameters, we set replay buffer size to 5000, target update period to 500, minibatch size to 32, and learning rate to 10^{-3} (typical values [14]). Finally, we use a discount factor

$\gamma = 0.9$ that allows the agent plan "roughly" 10 timesteps ahead without making convergence too slow. We chose all the above parameters as they performed well in various scenarios.

Demand. We import VNF resource demands from the Milano dataset [13]. Due to their continuous values the state space is infinite for all RL algorithms of this section. Milano timeseries consist of 8928 samples per base station (1 sample every 10 minutes), so we map to each VNF the normalized "internet" traffic demand of a different base station. W.l.o.g., we assume that VL demands are zero. We use the first 4464 Milano samples for training and the rest for testing, hence the duration of each training/testing episode is 4464 timesteps.

A. Part I: Performance gains of DQN+/iDQN+

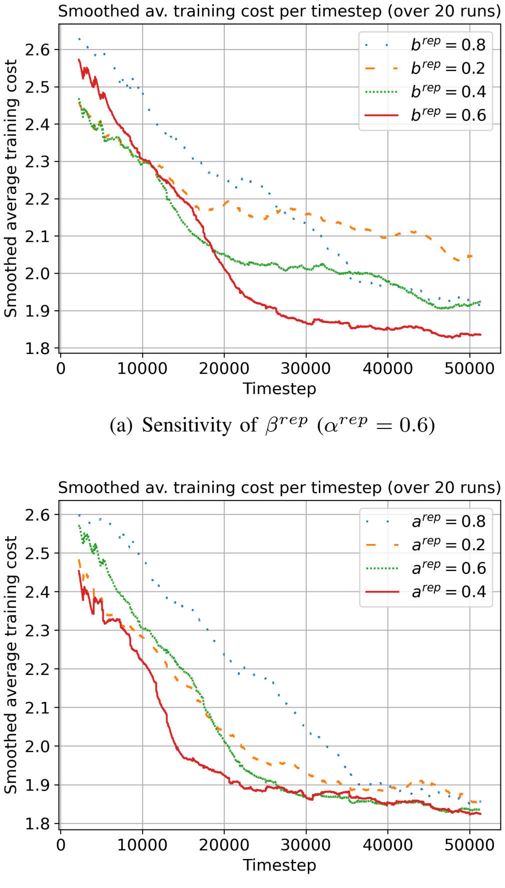
In this part we quantify the impact of the proposed speedup heuristics on the performance of DQN/iDQN algorithms, after first finding suitable values for the hyperparameters of prioritized experience replay (α^{rep}, β^{rep}) in our setup. DQN is used as a baseline, since we have tested it in many scenarios (smaller ones, where Q learning can be applied), and it tends to eventually converge to the optimal solution [6].

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). This results to 256 possible actions for single-agent DQN. The size of this setup was specifically chosen for the purpose of sensitivity analysis, as it allows for multiple runs of the algorithms in reasonable time.

Training. Each algorithm is trained over 12 episodes, while this procedure is repeated over 20 individual runs with different random seeds (to get averaged results).

Sensitivity Analysis. In the paper where prioritized replay was introduced [10], 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 plots in Figs. 7(a), 7(b). An important observation is that these parameters can affect both the sample efficiency and the quality of the obtained policy. In Fig. 7(a), a low b^{rep} leads to suboptimal policies (almost no importance sampling) and a high b^{rep} to very slow convergence (excessive importance sampling), while in Fig. 7(b), varying a^{rep} affects mostly the sample efficiency. 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 1: Hyperparameter tuning can significantly affect the performance of prioritized experience replay



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

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

Fig. 7. Sensitivity analysis of prioritized experience replay. Convergence plots for (a) varying β^{rep}; (b) varying α^{rep}.

Impact on DQN/iDQN. In Fig. 8, we compare the performance of vanilla DQN/iDQN algorithms with respect to their DQN+/iDQN+ counterparts. There are 4 main observations to take away, (i) iDQN converges 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 larger action space); (iv) iDQN+ converges faster than the rest of the tested algorithms.

To better quantify the speed gain, we outline the full simulation results in Table I, which indicates the timestep when the average cost of each algorithm goes below some specified threshold values. So, Table I highlights that DQN and iDQN converge 7.8 and 0.5 times slower than iDQN+ respectively. Note that the gain of iDQN+ is expected to be more prominent in larger scenarios, which is what we will show in the next set of experiments.

Take-away message 2: The proposed speedup heuristics on top of DQN/iDQN can offer significant convergence speed gains.

B. Part II: Validation in large scale scenario

Having validated the gain offered by the proposed speed ups in a relatively small scenario, in this last part we examine the performance of iDQN+ in a larger setup. Since DQN can not be used as a baseline here (due to the prohibitive action space size), we use some simple static policies (split-all, group-all), or even random actions, as minimum benchmarks to assess the obtained dynamic policies. Split-all aims to merely minimize

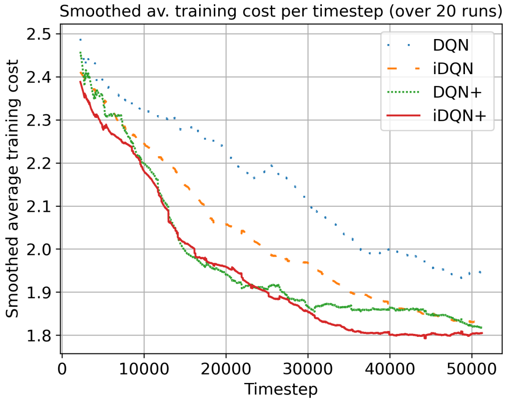


Fig. 8. Convergence plot for DQN, iDQN, DQN+, iDQN+.

TABLE I
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**

SLA violations by spreading VNFs equally among all nodes, while group-all focuses on minimizing the number of active nodes by placing all VNFs to a small subset of the largest servers. These policies turn out to be close-to-optimal for a subset of slots during the episode and have no reconfiguration cost (our scheme should outperform them as a minimum).

System Setup. The physical network consists of two technological domains, comprising 9 and 3 nodes (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 single-agent DQN.

Training and Testing. Each algorithm is trained over 22 training episodes, while this procedure is reperated for 10 individual runs with different random seeds. All the obtained policies are evaluated over 1 testing episode (we rollout the policy with no exploration and record the mean cost).

Cost performance. The results of the testing phase are given in the box plot of Fig. 9. This plot compares the performance of iDQN and iDQN+, including also the static baseline policies split-all, group-all, and random. The main observations are: (i) iDQN+ performs 2x better than the split-all policy, which was the best of the static baselines; (ii) 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; (iii) iDQN+ demonstrates 20% cost reduction compared to iDQN (converges faster as it achieves lower cost in the same amount of training steps); (iv) the performance gain of iDQN+ is more prominent in this larger scenario, compared to Part I. *Take-away message 3: iDQN+ was validated to improve the cost performance of iDQN by 20% in a large scale scenario.*

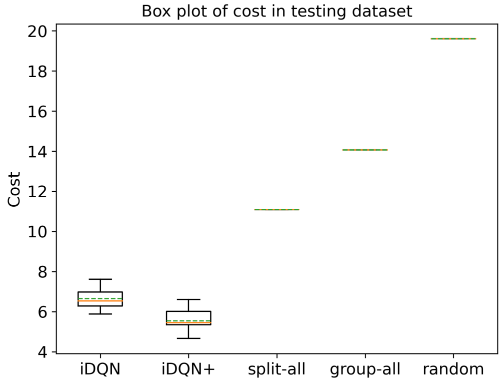


Fig. 9. Box plot depicting the distribution of the cost achieved by each algorithm in the testing dataset.

V. CONCLUSION

In this paper we examined the inter-slice orchestration problem and introduced two speedup heuristics on top of multiagent DQN to improve convergence speed and scalability. The proposed solution was validated by simulations and proved to offer significant performance gains.

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 Commun. Surv. Tutor.*, vol. 20, no. 3, 2018.

[2] 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*, 2018.

[3] 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 Commun. Mag.*, vol. 55, no. 8, 2017.

[4] 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.

[5] 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.

[6] P. Doanis, T. Giannakas, and T. Spyropoulos, “Scalable end-to-end slice embedding and reconfiguration based on independent dqn agents,” in *IEEE GLOBECOM*, 2022.

[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] 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, 2019.

[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*, vol. 17, no. 4, 2020.

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

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

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

[13] Telecom Italia, “Milano Grid,” 2015.

[14] V. Mnih et al., “Human-level control through deep reinforcement learning,” *Nature*, vol. 518, no. 7540, Feb. 2015.

[15] M. Shojafar, N. Cordeschi, and E. Baccarelli, “Energy-efficient adaptive resource management for real-time vehicular cloud services,” *IEEE Transactions on Cloud computing*, vol. 7, no. 1, 2019.

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