The Curse of (Too Much) Choice: Handling combinatorial action spaces in slice orchestration problems using DQN with coordinated branches

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

***Abstract*— One of the prominent problems in envisioned 6G networks is the truly dynamic placement of multiple virtual network function chains on top of the physical network infrastructure. Reinforcement Learning based schemes have been recently explored for such problems. Yet these have to deal with astronomically high state and action spaces in this context. Using a standard Deep Q-Network (DQN) is a common way to effectively deal with state complexity. While the use of independent DQN (iDQN) agents could be further used to mitigate action space complexity, such schemes often suffer from instability and sample (in)efficiency, and their theoretical performance is hard to assess. To this end we propose a DQN-based scheme that uses a recent Deep Neural Network architecture, with a different branch responsible for the placement of each virtual network function (again reducing action space complexity), yet with (implicit) coordination among branches, via shared layers (hence avoiding iDQN shortcomings). Using a real traffic dataset, we (i) theoretically ground the proposed scheme by comparing it with an optimal online algorithm for a stateless experts environment; (ii) we demonstrate a 41% cost improvement compared the existing state-of-the-art multi-agent DQN approach (independent agents).**

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

I. Introduction

Network slicing had already been identified as a key enabler for 5G and beyond networks [1]. A slice is often seen as a "VNF chain" comprising Virtual Network Functions (VNFs) and Virtual Links (VLs). Hosting multiple slices, with different demands and Service Level Agreements (SLA) on top of shared physical resources is thus a prominent problem [2].

The use of traditional static optimization methods for network slicing is problematic due to the dynamically changing, a priori unknown, or even non stationary parameters that determine slice performance. With an eye towards 6G networks, that will heavily leverage modern Artificial Intelligence (AI) methods, recent works have explored the use of Deep Neural Networks (DNNs) to either forecast mobile traffic [3] or even directly allocate a physical node's resources to the co-located VNFs [4]. Deep Reinforcement Learning (DRL) has been also employed for dynamic resource allocation [5], when traffic dynamics are unknown, or slice embedding (mapping of VNFs and VLs to physical nodes and links respectively), when the performance function is unknown [6].

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).

While the main focus of this paper's contribution is on the dynamic slice embedding problem (continuous embedding of VNFs and VLs), the proposed framework also considers the impact of resource allocation on slice performance. Some of the remaining open challenges for RL based solutions are the following: (i) infinite state spaces, due to continuous traffic demands of 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) [6]; (iii) poor sample-efficiency, which is an important shortcoming in online settings.

While DNN-based RL schemes, e.g. a Deep Q-Network (DQN) [7], can help with challenge (i), and multi-agent DQN [8] mitigates challenge (ii), existing schemes often suffer from instability and/or sample (in)efficiency. In this work, we propose a DRL scheme based on the Branching Deep Q-Network (BDQ) architecture, which dramatically reduces action complexity by allotting the control of each VNF to a different DNN branch. Moreover, to avoid the non-stationarity issues arising in multi-agent solutions with independent agents, a DNN module that is shared between different branches is responsible for their (implicit) coordination, improving the scheme's sample efficiency and scalability properties. The outline and specific contributions of this paper are as follows:

(C.1) In Section III-A, 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) We then consider a stateful RL version of the problem, that attempts to take advantage of patterns in the VNF/VL traffic dynamics, and discuss how and where existing DQN-based schemes fail to cope with the combinatorial state and action spaces involved (Section III-B).

(C.3) In Section IV, 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 V, 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 a 41% cost improvement in a fairly large scenario.

II. System Model

Our model is based on some common assumptions on VNF embedding, originally discussed in [2], and generalized recently in [8]. We summarize below the main attributes, for completeness, but refer the interested reader to [8] 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,

$$d^{(t)} = (d_i^k(t) | \forall k \in \mathcal{K},\ i \in \mathcal{N}_k \cup \mathcal{L}_k). \quad (1)$$

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

$$c^{(t)} = (c_n^k(t) | \forall k \in \mathcal{K},\ n \in \mathcal{N}_k), \quad (2)$$

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 [9]. We refer the reader to [8] for more details on the general model. Then, the SLA violation cost is:

$$\ell^{\text{SLA}}(c, d) = \sum_{k \in \mathcal{K}} (\sigma_k + F_k^{delay}(c, d) - q_k) \cdot \mathbf{1}_{\{F_k^{delay}(c,d) > q_k\}}, \quad (3)$$

where σ_k is a fixed penalty inflicted when an SLA violation occurs, and $\mathbf{1}_{\{\text{condition}\}}$ is a binary indicator variable that is equal to 1 when the *condition* is satisfied (0 otherwise).

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 idle node could be set to sleep mode to save energy [10]. This "on" nodes cost is given by:

$$\ell^{\text{ON}}(c) = \sum_{v \in \mathcal{V}} \mathbf{1}_{\{v \in c\}}. \quad (4)$$

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 [11]). This reconfiguration cost is given by:

$$\ell^{\text{RC}}(c^{(t)}, c^{(t+1)}) = \sum_{k \in \mathcal{K}} \sum_{n \in \mathcal{N}_k} \mathbf{1}_{\{c_n^k(t) \neq c_n^k(t+1)\}}. \quad (5)$$

III. 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*.

A. 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) [12]. 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.[2]

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

$$a_t = c^{(t+1)} \in \mathcal{A}. \quad (6)$$

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:

$$\ell(a, d^{(t)}) = w^{\text{SLA}} \cdot \ell^{\text{SLA}}(a, d^{(t)}) + w^{\text{ON}} \cdot \ell^{\text{ON}}(a), \quad (7)$$

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 [13], a simple online algorithm that can learn probabilistic policies with optimality guarantees. The algorithmic steps of MW are detailed in Fig. 1.

The performance of experts algorithms is compared to an "optimal static oracle". This oracle knows *in advance all future*

[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.

[2] 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.

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:

$$p_t(a) = \frac{Q_t(a)}{\sum_a Q_t(a)} \tag{8}$$

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:

$$Q_{t+1}(a) \leftarrow Q_t(a) \cdot (1-\eta)^{\ell(a,d^{(t+1)})}, \forall a \in \mathcal{A} \tag{9}$$

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

Fig. 1. Main algorithmic steps of MW.

demands up to horizon T and chooses *one* (hence "static") configuration ($a_0 = a_1 = ... = a_{T-1} = a^*$):

$$a^* = \arg\min_{a \in \mathcal{A}} \sum_{t=0}^{T-1} \ell(a, d^{(t+1)}). \tag{10}$$

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 [13]:

Lemma 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 [14]. Exp3 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). As a result of this partial feedback, the regret of Exp3 is $O(\sqrt{|A|T \ln|A|})$ (as opposed to the $O(\sqrt{T \ln|A|})$ regret of MW). This highlights the important advantage of MW, in terms of sample efficiency, compared to bandit schemes.

Pros: (i) MW doesn't require any foreknowledge about demands; (ii) it has optimallity 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 $|A|$ 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 patterns, etc.); (iii) it does not account for reconfiguration costs (while bandit algorithms for setups with reconfiguration costs do exist [15], these go beyond the scope of this work).

B. Reinforcement Learning formulation

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). Since this is a typical RL setting, in what follows we will first provide the RL formulation of the slice embedding problem, and then discuss the approximate RL algorithms that we use as baselines for the more advanced proposed scheme.

State Space. The state of the system at t consists of the configuration vector (2) and the demand vector (1):

$$s_t = (c^{(t)}, d^{(t)}) \in \mathcal{S}. \tag{11}$$

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 [16], 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:

$$r_{t+1} = -(w^{\text{SLA}} \cdot \ell^{\text{SLA}}(s_{t+1}) + w^{\text{ON}} \cdot \ell^{\text{ON}}(a_t) + w^{\text{RC}} \cdot \ell^{\text{RC}}(s_t, a_t)) \tag{12}$$

The only difference of (12) with the cost function of the experts problem (7), 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 [17]). Q-learning is a standard "tabular" RL algorithm that is guaranteed to converge to this more "powerful oracle", in theory. 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 III-A as baselines.

RL baseline: independent Deep Q-Networks (iDQN). It

is a state-of-the-art, multi-agent, Q-learning-based algorithm in the DRL class of schemes, proposed in [8] for dynamic slice embedding. The standard single-agent DQN algorithm [7] uses a DNN with parameters θ to approximate the action value function $Q(s,a)$. It 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). These action complexity problems can be addressed by using multiple independent DQN agents and decomposing the original action space into much smaller action sub-spaces.

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 (11)). 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). We refer the interested readers to [8] for a detailed description of DQN and iDQN algorithms.

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.

IV. 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 [18] 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. 2 visualizes the branching architecture.

Action space decomposition: Each DNN branch $Q_{\theta_{nk}}(s)$, outputs the predicted action values of placing VNF n of slice k to any of the permitted physical nodes (the number of network outputs and the computational complexity of argmax operations scale linearly with the number of VNFs).

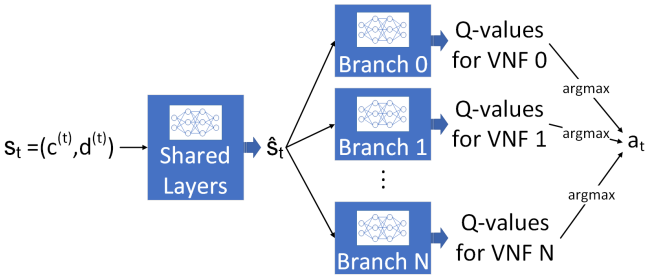


Fig. 2. Schematic representation of the branching architecture. A shared module of the DNN takes as input the state s_t and outputs a latent representation $\hat{s}_t$, which in turn is given as input to N different branches (one branch per VNF). The assignment $c^i(t+1)$ of each VNF i to a physical node in the next slot is determined by an argmax operation over the Q-value estimates of branch i. Then, the chosen action is $a_t = (c^0(t+1), c^1(t+1), ..., c^N(t+1))$.

Coordination: A DNN module that is shared among the different "cooperative" 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 the branches. This module is the key difference between iDQN and BDQ, as it can learn features that foster coordination (during the backward pass its parameters are updated based on the gradients backpropagated by all 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, 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 states visited by the agent are highly correlated). BDQ uses two standard DQN mechanisms to ensure stable learning, the experience memory replay and the target network (both are also used in iDQN). 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 the Temporal Difference (TD) target (13).

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 (15). We give the main algorithmic steps of BDQ in Fig.3.

V. 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) theoretically ground the proposed BDQ scheme both in terms of cost per time slot (compared to the static oracle), as well as in terms of sample efficiency (compared to MW), in a moderately-sized setup; (ii) validate the scalability of BDQ

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 θ' (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:

$$y_i = r_i + \gamma \frac{1}{N} \sum_{n \in \mathcal{N}_k, k \in \mathcal{K}} \max_{(a_i^{nk})' \in \mathcal{A}^{nk}} Q_{\theta'_{nk}}(s_i', (a_i^{nk})'), \tag{13}$$

where N is the number of branches.

Then, perform a gradient step:

$$\theta \leftarrow \theta - \eta \nabla_\theta L, \tag{14}$$

where

$$L = \mathbb{E}_{i \sim U(\mathcal{D})}[\frac{1}{N} \sum_{n \in \mathcal{N}_k, k \in \mathcal{K}} (y_i - Q_{\theta_{nk}}(s_i, a_i^{nk}))^2]. \tag{15}$$

Repeat steps 1 to 5 for T time slots.

Fig. 3. Main algorithmic steps of BDQ.

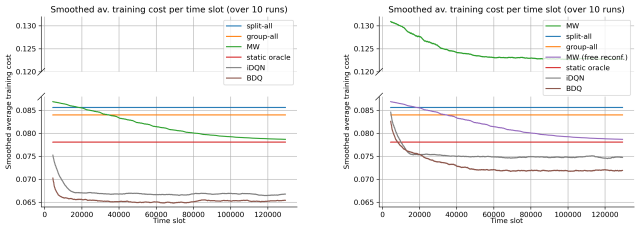
and the performance gains offered by coordination, compared to the independent agents of iDQN, in a large-scale setup. Thus, the Section is divided into two respective parts, each dedicated to one of the above objectives.

Algorithms. Here we outline all the algorithms (or policies) used in this section and any 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 III-A.
- **MW** the experts algorithm of Section III-A (Fig. 1), that has optimal regret with respect to the static oracle above.
- **iDQN** the multi-agent DRL scheme of [8], described in Section III-B.
- **BDQ** the DRL scheme of Section IV (Fig. 3).

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 γ = 0.9, as in [8]. The DNNs are multilayer perceptrons[3] (commonly used in related works, e.g. [6], [8]). Each iDQN agent or BDQ branch has 3 hidden layers with 60 neurons per layer (in BDQ this includes a single-layer module shared between branches); this size performed well in a variety of tested scenarios.

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



(a) Scenario 1 (b) Scenario 2

Fig. 4. 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.

VNF demands. We use the popular Milano dataset [16] to drive the demands. The imported timeseries consist of 8928 samples per base station (1 sample every 10 minutes), so we map the demand sequence of each VNF to the normalized "internet" traffic of a different base station. W.l.o.g., we assume that VL demands are zero.

A. 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(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 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.

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, 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.*

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(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.*

B. 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.

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!

Cost performance. We execute 10 independent training runs for each DRL agent (with different random seeds per run), as in Section V-A. We remark that, due to the vast action space of this scenario, it was not possible to apply MW here[4]. In order to examine both the mean performance and the stability of the DRL agents, we outline the results in the box plot of Fig. 5, depicting the cost of the policies obtained at the end of each training round (agents act greedily with respect to the learned action value functions). The main observations are the following: (i) BDQ demonstrates 41% and 47% better mean cost than iDQN and the static oracle respectively; (ii) BDQ has robust performance with much lower standard deviation than iDQN (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.

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

VI. Conclusion

In this paper we investigated the dynamic slice embedding problem both in an experts and an RL setting. We proposed a DQN-based algorithm that uses a recent DNN architecture with semi-independent, but coordinated branches, to improve scalabilty, and validated it using a real traffic dataset.

[4]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).

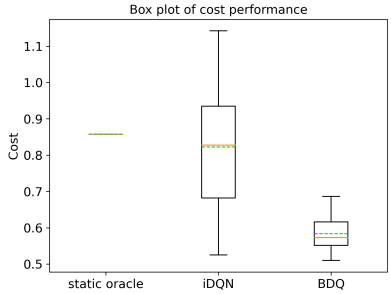


Fig. 5. Cost performance comparison in large-scale scenario ($2 \cdot 10^{14}$ actions).

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] 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.

[3] 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.

[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] F. Mason, G. Nencioni, and A. Zanella, "Using distributed reinforcement learning for resource orchestration in a network slicing scenario," *IEEE/ACM Trans. Netw*, vol. 31, no. 1, 2023.

[6] 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.

[7] V. Mnih et al., "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, 2015.

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

[9] 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.

[10] M. Shojafar, N. Cordeschi, and E. Baccarelli, "Energy-efficient adaptive resource management for real-time vehicular cloud services," *IEEE Trans. on Cloud Computing*, vol. 7, no. 1, 2019.

[11] 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, 2022.

[12] T. Lattimore and C. Szepesvári, *Bandit Algorithms*. Cambridge University Press, 2020.

[13] S. Arora, E. Hazan, and S. Kale, "The multiplicative weights update method: a meta-algorithm and applications," *Theory of Computing*, vol. 8, no. 6, 2012.

[14] P. Auer, N. Cesa-Bianchi, Y. Freund, and R. E. Schapire, "The non-stochastic multiarmed bandit problem," *SIAM journal on computing*, vol. 32, no. 1, pp. 48–77, 2002.

[15] 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*, 2019.

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

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

[18] A. Tavakoli, F. Pardo, and P. Kormushev, "Action branching architectures for deep reinforcement learning," in *Proceedings of the aaai conference on artificial intelligence*, 2018.