No Free Lunch: Balancing Learning and Exploitation at the Network Edge

Federico Mason
Dept. of Information Engineering
University of Padua
Via Gradenigo 6/b, Padua, Italy
masonfed@dei.unipd.it

Federico Chiariotti
Dept. of Electronic Systems
Aalborg University
Fredrik Bajers Vej 7C, Aalborg, Denmark
fchi@es.aau.dk

Andrea Zanella
Dept. of Information Engineering
University of Padua
Via Gradenigo 6/b, Padua, Italy
zanella@dei.unipd.it

***Abstract*—Over the last few years, the Deep Reinforcement Learning (DRL) paradigm has been widely adopted for 5G and beyond network optimization because of its extreme adaptability to many different scenarios. However, collecting and processing learning data entail a significant cost in terms of communication and computational resources, which is often disregarded in the networking literature. In this work, we analyze the cost of learning in a resource-constrained system, defining an optimization problem in which training a DRL agent makes it possible to improve the resource allocation strategy but also reduces the number of available resources. Our simulation results show that the cost of learning can be critical when evaluating DRL schemes on the network edge and that assuming a cost-free learning model can lead to significantly overestimating performance.**

***Index Terms*—Reinforcement Learning, Continual Learning, Network Slicing, Mobile Edge Computing**

I. Introduction

The orchestration of next-generation mobile networks is beyond the capabilities of human-designed algorithms, as it is characterized by multiple objectives and fast dynamics, with several classes of traffic having highly specific activity patterns and Quality of Service (QoS) guarantees [1]. In this context, machine learning is essential to allow the network protocols to dynamically adapt to different scenarios without the need to manually reconfigure the entire system [2]. In particular, the combination of Reinforcement Learning (RL) principles with deep learning, also known as Deep Reinforcement Learning (DRL) [3], is one of the most promising tools for the optimization of 5G and beyond networks [4].

The training of DRL models in complex environments is still very computationally expensive [5], and cannot be always performed in advance because of the rapid changes that characterize future mobile networks. Therefore, it is fundamental for 5G and beyond systems to support online training on the network edge, exploiting a continual learning approach [6]. In this context, the training updates are either performed directly on the edge nodes, according to the Mobile Edge Computing (MEC) paradigm, or they are offloaded to more powerful Cloud servers, reducing the consumption of local resources but requiring the transmission of large amounts of data.

This work was supported by Consortium GARR through the "Orio Carlini" scholarship 2019, and partly by the Danmarks Frie Forskningsfond (DFF) through the WATER project.

In a MEC scenario, the training cost creates a fundamental trade-off: updating a DRL model with new experience can improve the learned policy, increasing its efficiency in the target task, but also subtracts some resources from that very same task. Disregarding this trade-off and assuming that training is a *free action* may have serious consequences, degrading the performance of the system that should be optimized [7]. The machine learning research community is starting to become aware of this issue [8], focusing on model compression and lightweight learning techniques to reduce the burden on the edge hardware [9] and considering the *cost of learning* in the design of neural networks meant to operate on resource-constrained devices [10]. In particular, the Federated Learning (FL) approach can reduce the computational load of learning systems by performing the training of the target algorithms in a distributed fashion [11]. In the past years, many frameworks to reduce the computation and communication cost of FL have been proposed: for a deeper review on these topics, we refer the reader to [12], [13].

To the best of our knowledge, the resource efficiency of DRL techniques in future network scenarios is a relatively unexplored topic [14], as most researches in the literature still neglect the cost of the training, separating the learning process from the optimization even in online applications. In this work, we attempt to model the cost of learning explicitly, defining an optimization problem that balances learning and system optimization. The objective is to identify learning strategies that maximize the system performance during training, accounting for the cost of the learning itself. Our results show that adapting the number of training updates is a key factor for the optimization of MEC systems, while considering an ideal case with free learning actions may lead to a significant overestimating of the real performance.

The rest of this paper is organized as follows: Sec. II presents the cost of learning problem, which is then applied in a network slicing use case in Sec. III. The results of our online learning are presented in Sec. IV, and Sec. V concludes the paper and presents some possible avenues of future work.

II. Cost of Learning Model

We consider a typical network management application, in which a DRL agent tries to maximize system performance by allocating communication or computational resources to different users. The environment is modeled as a Markov Decision Process (MDP) defined by the 4-tuple $(\mathcal{S}, \mathcal{A}, P, R)$, where $\mathcal{S}$ is the state space, $\mathcal{A}$ is the action space, $P : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ is the state transition probability function and $R : \mathcal{S}^2 \times \mathcal{A} \to \mathbb{R}$ is the reward function. Hence, time is discretized into slots $t = 0, 1, 2, ...$ and, at any slot, the agent chooses a new action according to a policy $\pi : \mathcal{S} \to \mathcal{A}$. The function $P_{a_t}(s_t, s_{t+1})$ is the probability that action a_t in state s_t at time t will lead to state s_{t+1} at time $t+1$, while $R_{a_t}(s_t, s_{t+1})$ is the immediate reward received after a transition from s_t to s_{t+1} due to action a_t. The goal of the agent is to learn the optimal policy π^*, which maximizes the expected discounted return $G(t)$:

$$G(t) = \mathbb{E}\left[\sum_{\tau=t}^{\infty} \gamma^{\tau-t} R_{a_\tau}(s_\tau, s_{\tau+1}) \,\middle|\, \pi, s_t\right], \tag{1}$$

where $\gamma \in [0, 1)$ is the so-called *discount factor*.

We assume that the environment dynamics change over time, which makes it impossible to explore the environment offline, i.e., collect data with a pre-determined policy and perform training on this static dataset before the agent deployment. More specifically, we consider that the environment is organized into episodes $k = 0, 1, ...,$ and that each episode lasts T slots. Hence, we assume that $P(\cdot)$ and $R(\cdot)$ are stochastic processes that change over time in a step-wise manner, after a *coherence period* of K episodes. In general, we can assume that the agent is aware of the environment evolution and that it is reinitialized at the end of every coherence period. Accelerating the training might be possible by exploiting *transfer learning*, using previous experience to avoid periodically starting a new learning process from scratch [15]. However, the implementation of such techniques is out of the scope of this work.

We define the expected reward for an episode as

$$\overline{R}(k) = \mathbb{E}\left[\sum_{t=0}^{T-1} \frac{R_{a_t}(s_t, s_{t+1})}{T}\right], \tag{2}$$

and we set the goal of maximizing the expected reward $\overline{R}_K = \sum_{k=0}^{K-1} \overline{R}(k)$ during the K episodes constituting a single coherence period. In general, this is achieved by making the agent policy converge to the optimum in the shortest time possible. However, in our scenario, training the agent consumes some of the resources (computational or communication, depending on the architecture) that should be assigned to the users. Hence, each learning update has a direct impact on the system performance, and there is a trade-off between convergence speed and cost of learning.

To model this aspect, we assume that each episode is divided into two subsequent phases: first, an *exploitation* phase, in which the learned strategy is applied, then an *update* phase devoted to the agent training. In particular, the two phases last T_π and $T_\rho = T - T_\pi$ slots, respectively, so that the agent's actions are determined by the learned policy π during the first T_π slots of any episode k, while the agent follows a predetermined policy ρ during the last T_ρ slots, i.e.,

$$a_t = \begin{cases} \pi(s_t), & t \in \{0, 1, ..., T_\pi - 1\}; \\ \rho(s_t), & t \in \{T_\pi, T_\pi + 1, ..., T - 1\}, \end{cases} \tag{3}$$

where t is the slot index within the same episode.

The policy ρ ensures that all or part of the system resources are used for the agent training, making it possible to update the policy π with the new experience that the agent has gained. On the other hand, since ρ subtracts some of the resources from the users, this strategy leads to sub-optimal performance, decreasing the reward collected during the episode.

From a practical perspective, the value of T_ρ determines the amount of time and resources devoted to the learning process. As T_ρ increases, so does the number of experience samples used to train the agent after each episode. This makes it possible to accelerate the convergence speed of the agent, thus improving the immediate reward gained during the exploitation phases of the next episodes. On the other hand, increasing T_ρ also shortens those exploitation phases, because more time is spent in the update phases in which the sub-optimal policy ρ is used. In particular, the average reward during the k-th episode, considering a fixed T_ρ, is given by:

$$\overline{R}(k|T_\rho) = \left(\frac{T - T_\rho}{T}\overline{R}_{\pi_k}(k) + \frac{T_\rho}{T}\overline{R}_\rho(k)\right), \tag{4}$$

where π_k is the policy learned by the agent in the k-th episode, while $\overline{R}_{\pi_k}(k) = \mathbb{E}[\overline{R}(k); \pi_k]$ and $\overline{R}_\rho(k) = \mathbb{E}[\overline{R}(k); \rho]$ are the expected average rewards gained using policies π_k and ρ, respectively. The optimal value of T_p depends on multiple factors, including the coherence time of the underlying non-stationary MDP.

In this work, we consider two possible approaches for balancing exploitation and training. In the first, we adopt the naive assumption that the amount of resources assigned to the learning task is constant over time, and we analyze how T_ρ affects the system performance. In the second approach, instead, we assume that the amount of training resources can be adapted over time, based on the convergence speed of the learning agent, and we study the trade-off between convergence speed and effectiveness of the learned strategy. In the following, we formally define the optimization problems underlying the two approaches, which will be successively analyzed and compared in Sec. IV, considering a MEC system as a use-case scenario.

A. Constant update duration

Given the coherence period duration K, the first optimization problem aims at determining the optimal T_ρ to maximize $\overline{R}_K$, while assuming that T_ρ is constant over time:

$$T_\rho^* = \underset{T_\rho \in \{0,...,T\}}{\arg\max} \left(\sum_{k=0}^{K-1} \frac{\overline{R}(k|T_\rho)}{K}\right). \tag{5}$$

B. Adaptive update duration

The second approach assumes that the agent can determine when the policy π converges to the optimal one. In particular, after discovering the optimal policy, the agent sets T_ρ to zero, fully exploiting the system resources. If we define the number of episodes until convergence as $\eta(T_\rho)$, we have the following optimization problem:

$$T_\rho^* = \underset{T_\rho \in \{0,\dots,T\}}{\arg\max} \left(\sum_{k=0}^{\eta(T_\rho)-1} \frac{\overline{R}(k|T_\rho)}{K} + \left(1 - \frac{\eta(T_\rho)}{K}\right) \overline{R}_{\pi^*} \right). \tag{6}$$

In this case, we expect T_ρ^* to be higher, since a quicker convergence to the optimal policy allows the system to terminate the training process and fully dedicate its resources to the users.

III. Use-case

To test the benefits of our cost-aware learning framework, we consider a Network Slicing (NS) scenario, where a set of users with heterogeneous requirements transmits data through the uplink. We assume that the traffic is divided into *slices* that depend on the applications' QoS requirements, and the Base Station (BS) needs to allocate the capacity of a backhaul link to guarantee the best possible performance for each slice. The link resources are managed by a DRL agent, whose actions depend on the state of the slice buffers at the BS.

A. Communication model

We assume that time is discretized into slots $t = 0, 1, 2, \dots$ of length τ and that, in each slot t, each user $u \in \mathcal{U}$ (corresponding to a single application) transmits a vector of packets $\mathbf{x}_u(t)$ to the BS. All packets have the same length L, and we denote by $x_u(t, m)$ the m-th packet of $\mathbf{x}_u(t)$. We also assume that each application is associated with a specific slice $\sigma \in \Sigma$, according to the application requirements. We denote by $\mathcal{U}_\sigma \subset \mathcal{U}$ the set of users associated with slice σ. In particular, all the packets belonging to the same slice σ are seen by the BS as a single stream of data sharing the same communication resources.

We assume that the BS maintains a First In First Out (FIFO) buffer with a maximum size of Q packets for each slice $\sigma \in \Sigma$. The packets present in the buffer for slice σ at the beginning of slot t are collected in vector $\mathbf{q}_\sigma(t)$. We can conservatively assume that packets are added to the backhaul link buffer at the end of each slot. The buffer size condition is then $|\mathbf{q}_\sigma(t)| \leq Q$, where $|\mathbf{x}|$ represents the length of vector $\mathbf{x}$.

We assume that the backhaul link has a total capacity C_{bh}, that can be divided into N resource blocks, each of which makes it possible to transmit $\frac{\tau C_{\text{bh}}}{N}$ bits per slot. Note that packets can be transmitted over multiple subsequent slots. We now denote by $N_\sigma(t)$ the number of resource blocks assigned to slice σ during slot t. Naturally, any resource allocation scheme should comply with the condition $\sum_{\sigma \in \Sigma} N_\sigma(t) \leq N$.

The number of packets from slice σ that can be delivered in slot t, given that $N_\sigma(t)$ backhaul resources are allocated to it, is then given by:

$$\chi_\sigma(t) = \min \left(|\mathbf{q}_\sigma(t)|, \left\lfloor N_\sigma(t) \frac{\tau C_{\text{bh}}}{LN} \right\rfloor \right). \tag{7}$$

We denote by $\mathbf{y}_\sigma(t)$ the vector containing the $\chi_\sigma(t)$ packets of slice σ that are transmitted during slot t. At the next step, the queue vector $\mathbf{q}_\sigma(t+1)$ contains the remaining queued packets, as well as the newly arrived ones. However, the queue cannot contain more than Q packets, and in case of buffer overflows, the oldest $\omega_\sigma(t)$ packets are then discarded:

$$\omega_\sigma(t) = \max\left(0, |\mathbf{q}_\sigma(t-1)| - \chi_\sigma(t-1) + |\mathbf{x}_\sigma(t)| - Q\right). \tag{8}$$

We denote the vector of discarded packets by $\mathbf{d}_\sigma(t)$, so that $|\mathbf{d}_\sigma(t)| = \omega_\sigma(t)$. Once the buffer states $\mathbf{q}_\sigma(t), \forall \sigma \in \Sigma$ have been updated, the backhaul resources can be reallocated. To evaluate the system performance, we consider the queuing delay, defined as:

$$\delta(p, t) = \begin{cases} s : p \in \mathbf{x}_\sigma(t-s), & p \in (\mathbf{y}_\sigma(t) \cup \mathbf{q}_\sigma(t)); \\ \infty, & p \in \mathbf{d}_\sigma(t), \end{cases} \tag{9}$$

where the elay is computed for packet p from slice σ at time t. Therefore, the delay is the number of slots since p was delivered to the BS if p is transmitted or still in the buffer, while it is considered infinite if p is discarded.

We then define a utility function $f_u(\cdot)$ for each user, which takes the packet delay as input and return a value in $[0, 1]$. In particular, $f_u(\cdot)$ is monotonically decreasing, with $f_u(0) = 1$ and $\lim_{x\to\infty} f_u(x) = 0$, and depends on the delay requirements of the user, expressed in terms of the maximum delay Δ_u. Finally, the performance $\Phi_u(t)$ of u and the overall system performance $\Phi(t)$ at slot t are given by:

$$\Phi_u(t) = \sum_{m=1}^{\chi_\sigma(t)} \frac{f_{u(y_\sigma(t,m))}(\delta(y_\sigma(t, m), t))}{(\chi_\sigma(t) + \omega_\sigma(t))}, \tag{10}$$

$$\Phi(t) = \frac{1}{|\Sigma|} \sum_{\sigma \in \Sigma} \frac{1}{|\mathcal{U}_\sigma|} \sum_{u \in \mathcal{U}_\sigma} \Phi_u(t), \tag{11}$$

where $u(p)$ is the user that sent packet p.

B. Learning framework

In order to optimize the system in a foresighted manner, we model the resource allocation problem as an MDP, implementing a DRL agent to manage the link resources. In particular, the state $s(t)$ at time t depends on the individual requirements of the users currently being served, as well as the state of the transmission buffer for each slice. Specifically, $s(t)$ is a tuple with $4 \cdot |\Sigma|$ elements, namely:

- The number of packets contained in each slice buffer, i.e., $|\mathbf{q}_\sigma(t)|, \forall\, \sigma \in \Sigma$;

TABLE I: Agent architecture.

Layer size (inputs $\times$ outputs)	Inter-layer operations
$4 \cdot \|\Sigma\| \times 64$	ReLU activation
64×32	ReLU activation
$32 \times (1 + \|\Sigma\| \cdot (\|\Sigma\| - 1))$	Linear activation

- The average remaining time before the packets contained in each slice buffer exceed the maximum allowed delay Δ_u, which is given by

$$\sum_{m=1}^{|\mathbf{q}_\sigma(t)|} \frac{\Delta_{u(q_\sigma(t,m))} - \delta(q_\sigma(t,m))}{|\mathbf{q}_\sigma(t)|}, \ \forall \sigma \in \Sigma; \qquad (12)$$

- The minimum remaining time among the packets in each slice buffer, i.e.,

$$\min_{m \in \{1,\dots,|\mathbf{q}_\sigma(t)|\}} \left(\Delta_{u(q_\sigma(t,m))} - \delta(q_\sigma(t,m))\right), \ \forall \sigma \in \Sigma; \qquad (13)$$

- The number of packets that will be transmitted during the current slot for each slice, assuming that resource allocation scheme does not change, which is:

$$\min\left(|\mathbf{q}_\sigma(t)|, \left\lfloor N_\sigma(t-1)\frac{\tau C_{\text{bh}}}{LN}\right\rfloor\right), \ \forall \sigma \in \Sigma. \qquad (14)$$

We have $\mathcal{S} = \left\{0, \dots, \frac{Q}{L}\right\}^{2\cdot|\Sigma|} \times \mathbb{R}^{2\cdot|\Sigma|}$, while the action space $\mathcal{A}$ includes $1 + |\Sigma| \cdot (|\Sigma| - 1)$ different actions. Specifically, action 0 maintains the resource allocation constant, so that $N_\sigma(t) = N_\sigma(t-1)$, $\forall \ \sigma \in \Sigma$. Instead, each of the remaining actions is defined by the ordered tuple (i, j), $i \neq j$, and corresponds to taking one resource block from slice σ_i and assigning it to slice σ_j, so that $N_{\sigma_i}(t) = N_{\sigma_i}(t-1) - 1$ and $N_{\sigma_j}(t) = N_{\sigma_j}(t-1) + 1$. Naturally, resource allocation can never be negative, i.e., $N_\sigma(t) \geq 0 \ \forall \sigma \in \Sigma$, and the total number of allocated resources is always N.

Hence, at each slot t, the agent observes state $s(t) \in \mathcal{S}$ and selects a new action $a(t) \in \mathcal{A}$ according to its current policy π, which determines the number of blocks $N_\sigma(t)$ assigned to each slice $\sigma \in \Sigma$. Then, the agent receives an instantaneous reward $\Phi(t)$, which is given by the system utility defined in (11); we note that, by definition, $\Phi(t) \in [0, 1]$.

We adopt the Deep Q-Network (DQN) approach [3], in which the expected long-term value of each action, as given by (1) with $\Phi(t)$ as reward, is approximated by a Neural Network (NN) that, at each slot t, takes the current state $s(t)$ as input. In particular, we implement a fully connected feed-forward NN, whose input layer is formed by one neuron for each element of $s(t)$. The output of the NN is a scalar vector of size $|\mathcal{A}|$, representing the expected long-term reward of each possible action $a \in \mathcal{A}$ for the current state. In particular, when operating greedily, the agent will always pick the action corresponding to the highest output value. The main parameters of the learning architecture are reported in Tab. I.

IV. Settings and Results

In the following, we apply our meta-RL model in the NS environment described in Sec. III. Specifically, we assume that the DRL agent deployed at the BS cannot be trained locally and needs to exchange information with the core network to improve its own policy. Part of the backhaul link resources are then used to update the agent's architecture, possibly degrading the user performance. Hence, we investigate how the system utility changes when varying the resources dedicated to the training, both when the learning update is performed regularly in time, and when it is stopped after a certain period (assuming the DRL agent has converged to the optimal policy). In the rest of the section, we will describe the settings we used in our system, as well as the simulation results.

TABLE II: Application parameters.

Application	Bit rate [kb/s]	Packet delay budget [ms]
NCVO	25	100
NCVI	384	300
CVO	25 (when active)	75
CVI	384 (when active)	100

A. Scenario settings

The model described in Sec. III-A is very general and can suit multiple communication scenarios with different characteristics. In this work, for the sake of simplicity, we consider a simple case with only two slices, named *non-critical* (σ_{NC}) and *critical* (σ_{C}), respectively, with the following performance functions $f_u(\delta)$:

$$f_u(\delta) = \begin{cases} \min\left(1, \frac{\Delta_u}{\delta}\right), & \text{if } u \in \sigma_{\text{NC}}; \\ \mathbb{1}(\Delta_u - \delta), & \text{if } u \in \sigma_{\text{C}}; \end{cases} \qquad (15)$$

where $\mathbb{1}(x)$ is the limit-step function, equal to 1 if $x \geq 0$ and 0 otherwise. Critical packets have a hard deadline, i.e., delivering them after the maximum delay Δ_u gives zero performance benefits. On the other hand, the utility of non-critical packets decreases gradually as the delay grows past the deadline.

We assume that time is divided into slots of $\tau = 10$ ms and that the backhaul link has a total capacity $C_{\text{bh}} = 1$ Mb/s. The channel is divided into $N = 10$ resource blocks so that each block allows the delivery of exactly 1 kb. However, we assume that any allocation scheme remains constant over 10 slots, i.e., the agent takes a new action every 100 ms. This value is closer to the granularity of actual schedulers and avoids the *reward tampering* phenomenon [16]. Finally, we consider packets to have a constant length $L = 512$ b, and set the maximum buffer size to $Q = 100$ packets, equivalent to 64 kB.

We assume that there are 5 different users in the system, each of which randomly picks an application among the 2 critical and 2 non-critical applications at the beginning of each episode. The Non-Critical Voice (NCVO) and Non-Critical Video (NCVI) applications are associated with σ_{NC} and have constant bitrate. The Critical Voice (CVO) and Critical Video (CVI) applications are associated with σ_{C} and generate new packets following an on-off process.

More specifically, CVO and CVI behave according to a Markov Chain (MC) with two state, namely *silent* (s) and *active* (a): in the silent states, no packets are generated, which implies that $|\mathbf{x}_u(t)| = 0$; conversely, in the active states, packets are generate with constant bitrate. We assume that a critical application can switch between states at the beginning of each slot t, and we set the state transition probability to $p_{ss} = p_{aa} = 0.9$ and $p_{sa} = p_{as} = 0.1$, respectively. The bitrate and the packet delay budget of the different applications are summarized in Tab. II.

B. Learning settings

To orchestrate the backhaul link resources, we deploy the learning system presented in Sec. III-B. Our implementation of DQN is distributed: the *inference network* is installed on the BS and is used to allocate resources in the exploitation phase of each episode, while the *training network* is updated at every training step and is in a Cloud server. The transmission of the learning data and updated model occupies the backhaul link, following the general framework defined in Sec. II.

At the beginning of each episode, the slice buffers are emptied and each user is associated with a random application. During the first T_π slots of each episode, the inference network sets the resource allocation policy and saves experience samples $(s(t), a(t), s(t+1), \Phi(t), a(t+1))$ in a local memory. Instead, during the training phase, no application data are transmitted, i.e., $N_\sigma(t) = 0, \forall \sigma \in \Sigma$, and the entire link capacity is used to forward training data to the core network.

We assume that each experience sample can be encoded into $L_{\text{tr}} = 704$ b, while the NN architecture size is $L_{\text{NN}} = 92256$ b. Therefore, during the updating phase, $T_\rho \frac{\tau C_{\text{bh}}}{L_{\text{tr}}}$ transitions can be forwarded through the channel and used by the Cloud server to update the training network. Moreover, every 10 learning steps, a copy of the training network is sent back to the BS, replacing the inference network and improving the current policy ρ. In those steps, the number of transitions forwarded through the channel is limited to $\frac{T_\rho \tau C_{\text{bh}} - L_{\text{NN}}}{L_{\text{tr}}}$.

To train the agent during the update phase, we implement the *on-policy* State-Action-Reward-State-Action (SARSA) algorithm [17] with a *softmax* exploration policy. In particular, we set the discount factor $\gamma = 0.95$, and we implement the Adaptive moment estimator (Adam) algorithm to optimize the NN weights, using $\zeta = 10^{-5}$ as maximum learning rate. Finally, we set the coherence period to $K = 10000$ episodes, and we assume that each episode lasts $T = 1000$ slots.

C. Results

We now investigate different configurations of our learning system, varying the time $T_\rho \in \{1, \ldots, 5\}$ devoted to the agent training. When T_ρ is low, most of the system resources are assigned to the users, and agent training proceeds slowly; conversely, as T_ρ increases, more learning data are transmitted through the link, taking up more resources but increasing the training speed. We compare the results with an ideal system, where all the learning transitions are instantaneously transmitted through a side-channel and used to update the inference network, without impacting the users. Clearly, this represents an upper bound to the practically achievable performance.

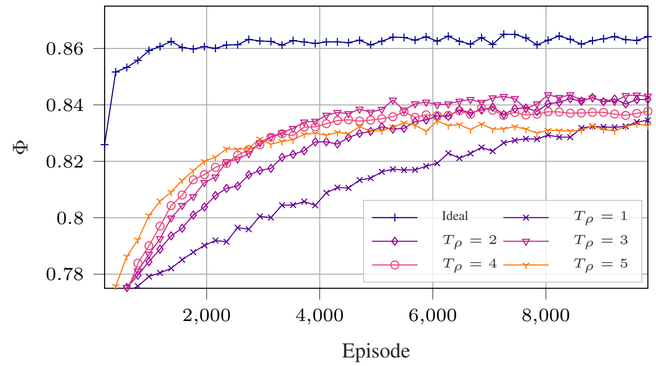

Fig. 1: Mean performance over time with fixed T_ρ.

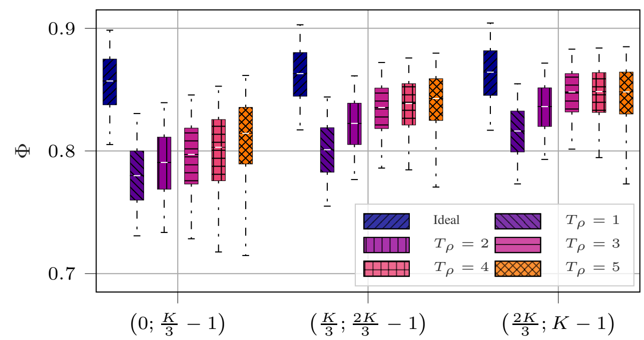

Fig. 2: Boxplots of the performance with fixed T_ρ.

We first consider the scheme with a constant update duration from Sec. II-A. In Fig. 1 we represent the average performance during the training of the different strategies, obtained aggregating the data of 100 independent simulations. It is easy to see that larger values of T_ρ lead to a quicker convergence toward the optimal policy and, in particular, training is rather slow if $T_\rho = 1$. On the other hand, the configurations with a higher T_ρ improve faster but have lower performance after convergence, since they continue to devote a fixed amount of resources to the agent training.

We can have a better insight into the different strategies by looking at Fig. 2, which uses boxplots to represent the performance statistics during the beginning, intermediate and last episodes of the training. The whiskers represent the 5th and 95th percentiles of the performance distribution, while the box goes from the 25th to the 75th. In the first third of the coherence period, it is convenient to select a high value for T_ρ to learn the optimal policy faster. After a sufficient number of episodes, the configurations with a lower T_ρ also converge and waste fewer system resources on further training, but the configuration with $T_\rho = 1$ is always outperformed by the others, since it needs more than K episodes to reach convergence.

We can now look at the adaptive system from Sec. II-B. We use a simple heuristic strategy to infer convergence: if the average reward over a rolling window of K_{avg} episodes stops increasing, the agent estimates that the optimal policy has been found and sets T_ρ to zero, assigning the whole link capacity to the users. This allows the agent to use the first

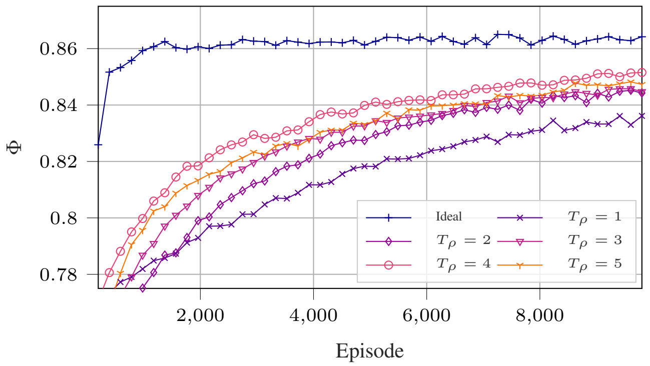

Fig. 3: Mean performance over time with adaptive T_ρ.

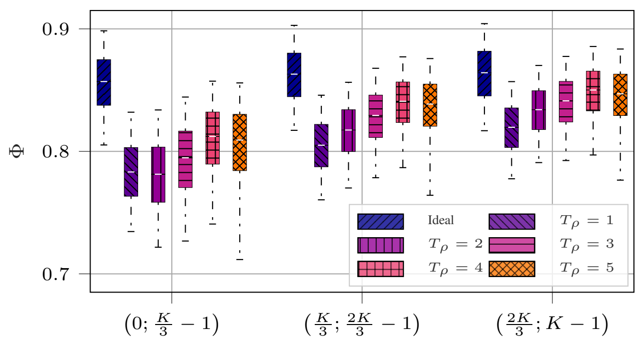

Fig. 4: Boxplots of the performance with adaptive T_ρ.

portion of the coherence period to learn the optimal policy, taking advantage of the acquired knowledge in the remaining episodes. The choice of T_ρ will then determine the time needed to switch to the full exploitation phase.

Fig. 3 shows the performance over time with this approach, considering $K_{\text{avg}} = 4000$. We observe that the performance of the strategies with $T_\rho \leq 3$ is almost identical to that of the previous scenario. In fact, the adaptive method does not reach convergence before the end of the coherence period with such a small T_ρ. On the other hand, setting $T_\rho = 5$ seems to be too aggressive, and the heuristic stops the learning process too soon, leading to suboptimal results.

The approach with $T_\rho = 4$ outperforms all the other strategies during the entire coherence period, striking a balance between convergence speed and cost of learning. This is confirmed by Fig. 4, which shows that $T_\rho = 4$ achieves better performance, both considering the lower and the higher percentiles of the distribution.

V. Conclusions

In this work, we analyzed the cost of exploiting DRL solutions for MEC network optimization. Specifically, we designed a novel cost of learning framework to optimize the amount of resources that a learning agent allocates to its own improvement, so as to balance the speed of convergence of the policy with the system performance during the training. We consider a test case based on a 5G system in which a DRL agent has to allocate the bandwidth of a backhaul link among multiple slices while consuming part of the network resources to transmit its own training updates. Our results show that there is a significant trade-off between the convergence speed and the communication overhead due to the training. Inferring the agent convergence is also a critical problem, especially in fast-varying scenarios which require the use of continual learning. Future work on the subject may involve more general solutions, based on hierarchical DRL or other meta-learning tools, that can apply to different problems in which cost of learning is a real concern.

References

[1] J. Navarro-Ortiz, P. Romero-Diaz, S. Sendra, P. Ameigeiras, J. J. Ramos-Munoz, and J. M. Lopez-Soler, “A survey on 5G usage scenarios and traffic models,” *IEEE Communications Surveys & Tutorials*, vol. 22, no. 2, pp. 905–929, February 2020.

[2] M. E. Morocho-Cayamcela, H. Lee, and W. Lim, “Machine learning for 5G/B5G mobile and wireless communications: Potential, limitations, and future directions,” *IEEE Access*, vol. 7, pp. 137 184–137 206, Sep. 2019.

[3] V. Mnih, K. Kavukcuoglu, D. Silver *et al.*, “Human-level control through deep reinforcement learning,” *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015.

[4] Z. Xiong, Y. Zhang, D. Niyato, R. Deng, P. Wang, and L.-C. Wang, “Deep reinforcement learning for mobile 5G and beyond: Fundamentals, applications, and challenges,” *IEEE Vehicular Technology Magazine*, vol. 14, no. 2, pp. 44–52, Apr. 2019.

[5] O. Gómez-Carmona, D. Casado-Mansilla, F. A. Kraemer, D. López-de Ipiña, and J. García-Zubia, “Exploring the computational cost of machine learning at the edge for human-centric Internet of Things,” *Future Generation Computer Systems*, vol. 112, pp. 670–683, Nov. 2020.

[6] F. Zenke, B. Poole, and S. Ganguli, “Continual learning through synaptic intelligence,” in *International Conference on Machine Learning*. PMLR, Jul. 2017, pp. 3987–3995.

[7] R. S. Villaça and R. Stadler, “Online learning under resource constraints,” in *International Symposium on Integrated Network Management (IM)*. IFIP/IEEE, May 2021, pp. 134–142.

[8] N. C. Thompson, K. Greenewald, K. Lee, and G. F. Manso, “The computational limits of deep learning,” *arXiv preprint arXiv:2007.05558*, Jul. 2020.

[9] D. Liu, H. Kong, X. Luo, W. Liu, and R. Subramaniam, “Bringing AI to edge: From deep learning’s perspective,” *arXiv preprint arXiv:2011.14808*, Nov. 2020.

[10] M. Chan, D. Scarafoni, R. Duarte, J. Thornton, and L. Skelly, “Learning network architectures of deep CNNs under resource constraints,” in *Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*. IEEE, Jun. 2018, pp. 1703–1710.

[11] L. U. Khan, S. R. Pandey, N. H. Tran, W. Saad, Z. Han, M. N. Nguyen, and C. S. Hong, “Federated learning for edge networks: Resource optimization and incentive mechanism,” *IEEE Communications Magazine*, vol. 58, no. 10, pp. 88–93, Nov. 2020.

[12] J. Chen and X. Ran, “Deep learning with edge computing: A review.” *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1655–1674, Aug. 2019.

[13] B. Luo, X. Li, S. Wang, J. Huang, and L. Tassiulas, “Cost-effective federated learning design,” in *Conference on Computer Communications (INFOCOM)*. IEEE, May 2021.

[14] I. Jang, H. Kim, D. Lee, Y.-S. Son, and S. Kim, “Knowledge transfer for on-device deep reinforcement learning in resource constrained edge computing systems,” *IEEE Access*, vol. 8, pp. 146 588–146 597, Aug. 2020.

[15] C. Tan, F. Sun, T. Kong, W. Zhang, C. Yang, and C. Liu, “A survey on deep transfer learning,” in *27th International Conference on Artificial Neural Networks (ICANN)*. Springer, Oct. 2018, pp. 270–279.

[16] T. Everitt, M. Hutter, R. Kumar, and V. Krakovna, “Reward tampering problems and solutions in reinforcement learning: A causal influence diagram perspective,” *Synthese*, pp. 1–33, May 2021.

[17] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT Press, Nov. 2018.