# Model-Based Reinforcement Learning with Kernels for Resource Allocation in RAN Slices

Juan J. Alcaraz, Fernando Losilla, Andrea Zanella *Senior Member, IEEE* and Michele Zorzi *Fellow, IEEE*

***Abstract*—Network slicing is a key feature of 5G and beyond networks, allowing the deployment of diverse logical networks, referred to as network slices, sharing a common underlying physical infrastructure. Each network slice is characterized by distinct descriptors and behaviors according to their type of service (e.g., enhanced mobile broadband, eMBB, or massive machine type communication, mMTC). The dynamic allocation of physical network resources among coexisting slices should address a challenging trade-off: it should be efficient in terms of resource usage, but also assign to each slice the necessary resources to guarantee its service level agreement (SLA). We address the allocation of time-frequency resources among network slices, a problem that has been addressed in previous works using model-free reinforcement learning (MFRL) algorithms. Nevertheless, our approach takes a new perspective: to design a control algorithm capable of learning over the operating network (online learning), while keeping the SLA violation rate under an acceptable level during the learning process. The main drawbacks of MFRL algorithms for this online learning application are their low sample efficiency, their extensive exploration of the policy space, and their inability to discriminate between the two conflicting objectives, causing inefficient use of the resources and/or frequent SLA violations during the learning process. To overcome these limitations, we propose a model-based RL (MBRL) approach built upon a novel modeling strategy that comprises a kernel-based classifier and a self-assessment mechanism. In numerical experiments, our proposal, referred to as kernel-based RL (KBRL), clearly outperforms state-of-the-art RL algorithms in terms of SLA fulfillment, resource efficiency, and computational overhead.**

## I. Introduction

Network slicing is a core feature of 5G networks that allows the deployment of several logical networks over a common physical infrastructure, each one providing a specific type of service, such as enhanced mobile broadband (eMBB) or massive machine type communications (mMTC). These logical networks, referred to as *network slices* can be managed and exploited by third party entities or *tenants*. Under this paradigm, the infrastructure provider has to assign the necessary spectrum, backhaul, and computational resources to each network slice to fulfill the service level agreement (SLA) established with the tenant of this slice. The SLA specifies a set of requirements on performance indicators, such as throughput or latency, that depend on the tenant's preferences and service type. The allocation of resources should guarantee that network slices are properly *isolated* from each other, but this allocation should also be *resource-efficient* and *elastic* under varying radio and network traffic conditions [1].

In the radio access network (RAN), the radio frequency (RF) resources of each base station should be distributed among the network slices of the users (UE) connected to this base station, based on the states of the slices and their SLAs. The state of a network slice is determined by its traffic descriptors, its current distribution of resources, and the radio channel conditions of its UEs, resulting in state observations that can potentially involve multiple variables. For example, in an eMBB slice, the observation could comprise the incoming data rate, the delivered data rate, the buffered data, and the resources occupied by each type of traffic, plus the channel quality indicators per user flow. Besides, the SLAs involve a diverse combination of requirements which can be defined in terms of aggregated metrics. The SLA of an eMBB slice could set an average delay objective for guaranteed bit rate (GBR) flows whenever the resources occupied by this type of traffic are below a predefined level. See [2] for other examples of SLA configurations. Determining the minimum share of RAN resources fulfilling a specific SLA at each observed state is a challenging task, and it is further complicated by the interactions with the mechanisms operating within each slice in a per-flow basis, such as scheduling algorithms or adaptive modulation and coding schemes.

Our objective is to develop a control algorithm capable of *learning* how to allocate RAN resources in an efficient way, maximizing the amount of free resources available to the infrastructure provider (which could be used to accommodate additional slices or for its own users), while guaranteeing the SLAs of the hosted tenants. A crucial feature of our proposal is its *plug-and-play* capability, enabling it to learn on an operating network (*online learning*), without any previous information about the system's response.

Reinforcement Learning (RL) has become the most widely used control technique for radio resource management [3] in general, and resource orchestration in network slices [4], [5] in particular. The main limitation of previous works is their use of a model-free RL (MFRL) approach, which can be very effective if the agents are trained *offline*, either with a simulator or with samples obtained from the real system, but is not especially suitable when the agents learn on a network in operation (*online learning*). MFRL algorithms generally require a large number of samples, which involves an extensive exploration of policies, including inefficient ones. In our scenario, this may lead to long training periods containing multiple episodes of SLA violations and/or excessive resource over-provisioning, which are detrimental to both the tenants and the infrastructure provider.

Manuscript received March X, 2021. The associate editor coordinating the review of this paper and approving it for publication was To Be Added. This work was partially developed during a visit of Juan J. Alcaraz to the University of Padova under *Salvador de Madariaga* grant PRX 19/00429. Juan J. Alcaraz and F. Losilla acknowledge Grant PID2020-116329GB-C22 funded by MCIN/AEI/10.13039/501100011033.

We propose a novel approach to the problem of dynamic allocation of RAN resources among network slices that involves the following contributions:

- We present a new approach to the problem, focusing on the importance of learning online (on the real system in operation), efficiently (with few samples) and safely (degrading as little as possible the QoS provided by the network slices).
- To this aim, we use a model-based RL (MBRL) approach, which overcomes much of the limitations of the usual MFRL approaches. MBRL allows for greater sample efficiency and gives us more control over the two objectives pursued: maximizing the efficiency in the use of resources and guaranteeing the SLA of the slices.
- We develop a novel modeling scheme comprising a nonlinear classifier and an estimator of the classifier's accuracy. The classifier is based on a kernel-based online learning scheme known as *Projectron* [6] that we enhance with a sample augmentation strategy that exploits the structure of our problem. As a result, we propose a novel mechanism referred to as kernel-based RL (KBRL) aimed at learning efficient resource allocation policies under SLA constraints.
- We compare our proposal with state-of-the-art RL algorithms by means of extensive simulations over diverse environments, providing one of the more complete empirical evaluations of RL in this application so far. Our results show that KBLR systematically outperforms the baselines in terms of resource usage, SLA fulfillment guarantees and computational overhead.

The rest of this paper is organized as follows. Section II discusses related work. The system under study is described in Section III and the addressed problem is formulated in Section IV. We provide the necessary preliminaries about online learning with kernels in Section V, and then we present our proposal in Section VI. The numerical results are provided in Section VII and finally we summarize our conclusions in Section VIII.

## II. Related Work

Network slicing has received considerable attention from the research community and the industry during the last five years. Survey papers like [1], [7], [8] provide an extensive coverage of recent contributions on different aspects of this network functionality. These papers also identify open problems and research challenges in network slicing, including the one addressed in our paper: to dynamically scale up/down the RAN resources assigned to network slices so that SLA requirements are met while the RF spectrum is efficiently utilized [1] [7]. In line with our proposal, [8] highlights the importance of the computational overhead of these algorithms, arguing that small computing times allow a more frequent update of the resource allocation, thus improving the elasticity of the system.

Network slices make use of resources in all sections of the underlying network, including RF spectrum, fronthaul, backhaul, and computational resources for virtualized network functions. Resource management in network slicing is therefore a broad topic, where we find related works focused on different network elements such as computational resources [9]–[11], radio resources [12], [13], or a combination of radio and computational resources [14], [15]. Two main resource management mechanisms have been studied: admission control for on-demand slice deployment [9], [15]–[17], and dynamic resource scaling for active slices [11], [13], [18], [19]. Our proposal falls in the latter category.

Although a variety of techniques have been used for resource management (e.g., queueing theory [17], Lagrange methods for optimization [12], [20], Thompson sampling [21], heuristic methods [22]), those based on artificial intelligence (AI) are currently the most widely used. Indeed, AI is considered a key architectural feature for providing resource elasticity in future networks [5]. Consequently, RL is becoming a standard approach for resource management in network slicing. The main issues addressed by RL in this domain have been: resource management for virtualized functions [9], [11], [23]–[26], admission control of new slices [9], [15], [16], handling of computational resources in mobile edge computing (MEC) [11], [27], [28], UE scheduling [29], [30], and radio resource allocation among network slices [13], [18], [19], [31], [32], which is the problem addressed in this work.

Radio resource allocation comprises two conflicting objectives: 1) to maximize efficiency in the use of resources (sepectrum efficiency, SE), and 2) to maximize the SLA satisfaction rate (SSR). However, conventional RL formulations are limited to a single objective function, and consequently these previous works need to aggregate both objectives into a single one by means of a weighted sum of both metrics (SE and SSR), or by multiplying them [31]. The problem with this approach is that it does not allow establishing a performance target on some objective, e.g. to guarantee that the SSR remains above a desired level. Moreover, the relative performances of SE and SSR vary from one scenario to another (as shown in the Section VII), and thus the proper adjustment of the weighting is unknown a priory and should be done by trial and error, limiting the feasibility of this approach for online learning, which is the challenge addressed by our proposal.

Algorithms like Q-learning, used in [19], or its deep learning version, DQN, used in [18], are conceived for discrete action spaces thus, as noted in [13], they become unfeasible as the number of network slices increases, since the number of actions grows almost exponentially with the number of slices. Note that [19] and [18] considered scenarios with relatively small action spaces (2 and 3 network slices respectively). To overcome this limitation, [13] proposes the use of normalized advantage functions (NAF), a technique allowing the use of DQN strategies for continuous action spaces. In fact, transforming the discrete action space into a continuous one is a standard strategy for applying RL in this problem, and is the one that we adopted to evaluate the RL baselines in our experiments.

But the most distinctive feature of all these previous works is the use model-free RL methods, therefore requiring reletively long training periods. Using a simulator for training MFRL agents can be extremely costly, and does offer sufficient performance guarantees when agents are deployed in production, since it is practically impossible to replicate all

the relevant aspects of a real network in a simulator. If the agents are trained on a real operating network (online learning), the system will experience poor performance during the learning periods, because MFRL agents need to explore multiple policies until converging to an efficient one. For example, the distributional RL approach of [32] required between 5000 and 15000 steps to converge, and [31] trained its DQN-based proposal during $2 \times 10^6$ steps before performing the evaluation experiments. In contrast, we follow a model-based approach, aiming at increasing the sample efficiency of the learning process so that the control algorithm is suitable for online learning. There are no precedents of this approach for dynamic resource allocation among network slices. Previous online learning proposals were focused on different network functions (e.g., interference coordination and energy saving [33]–[35]) and used specific ad hoc mechanisms based on multi-armed bandits [33], sequential likelihood ratio tests [34], or bayesian models [35].

MBRL is known to be more sample efficient than MFRL [36], [37] but also more demanding in terms of computation. However, we propose a novel modeling strategy involving an online learning classifier that reduces the computational overhead dramatically, and helps our proposal to outperform MFRL algorithms in this metric. In Section V we review the principles of kernel-based online learning, and provide references to related works on this topic.

The use of kernel-based online learning for the definition of the model in an MBRL algorithm is a novel approach. Nevertheless it should be emphasized that our proposal can be complementary to previous ones. For example, our method prescribes the amount of physical radio resources to be assigned to each slice, but does not arrange them within the time-frequency frame structure of the RF interface. For this task, the heuristic scheme proposed in [22] can be used. Our proposal can operate concurrently with an admission control scheme for on-demand incoming slices, such as those developed in [15], [16], and with a mechanism for the allocation of computational resources among slices [9], [26]. The promising results shown in Section VII suggest that our proposal could be extended to the control of additional resources (computation, storage, backhaul, fronthaul) requiring elasticity and efficiency.

## III. System Description

We consider a typical cellular network system, similar to the scenarios described in [13], [19], [31], [32], consisting of a base station providing access to multiple UEs belonging to $K$ network slices. We consider downlink transmission on an hexagonal cell. Figure 1 shows a schematic overview of the system for $K = 3$ slices. The radio interface between the BS and the UEs is structured into frames, and each frame is partitioned in time and frequency: in the time dimension, frames are divided into transmission time intervals (TTIs), also referred to as subframes, and in the frequency dimension, the bandwidth is divided into subcarriers. The physical layer of 5G RANs provides high flexibility in the use of waveforms and time-frequency frame structures, allowing diverse configurations of the TTI duration and sub-carrier spacing, known

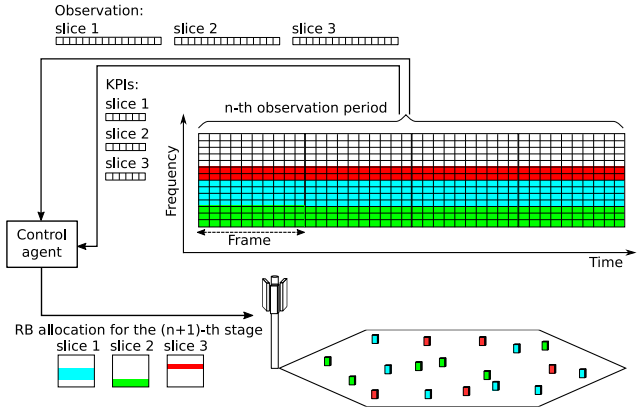


Fig. 1. Diagram of controlled system. One base station transmitting donwlink in a hexagonal sector cell covering UEs belonging three different RAN slices. Each RAN slice is granted the exclusive use of a predefined subset of RBs in each radio frame. During an observation period, spanning several frames, the control system monitors the performance, and updates the RB allocation for the next period according to the observed variables of the slices and the SLA fulfillment indicators

as numerologies. The selected numerology depends on the deployed frequency band and on the desired transfer service capabilities for the slice [2]. In our case, we will assume a TTI duration of 1 ms and a sub-carrier spacing of 15 MHz, but different numerologies can coexist in the same radio interface.

Each network slice is assigned a subset of the time-frequency resources of the radio interface. The smallest time-frequency allocation unit is referred to as resource block (RB), and consists of 1 TTI and 12 sub-carriers (1 ms $\times$180 MHz, in our setting). We consider, as [13], [18], [19], [31], that the RBs assigned to a slice are used exclusively by this slice, thus assuring slice *isolation*. Consequently, each slice runs its own scheduler for allocating its RBs among the users of this slice in a per-TTI basis. As shown in Figure 1, the time-frequency resources allocated to each slice consist of a group of RBs within each radio frame, sometimes referred to as a *tile*, where a specific numerology can be adopted.

The SLA defined for each slice depends on the service requirements and the preferences of its tenant, and comprises a set of configuration descriptors and key performance indicators KPIs that can be very diverse. We will consider descriptors of the authorized capacity for each slice (see [2]). For eMBB slices, these descriptors can set specific limits to the average number of RBs consumed by each type of traffic (non-GBR and GBR) within the slice. In mMTC it is usual to define a limit on the maximum number of simultaneous active devices (UE contexts). Key performance indicators can specify QoS objectives such as maximum average delay or maximum buffer length. Section VII contains the SLA specifications for eMBB and mMTC slices used in our numerical simulations.

The number of RBs allocated to each slice can be scaled up or down in periodic time instants referred to as decision stages or steps, which are typically spaced by several frames. At each stage, the control agent selects a control *action* that specifies the number of RBs that will be available exclusively to each

**eMBB RAN Slice observed variables**

| variable | units |
|---|---|
| GBR incoming traffic | bit/s |
| GBR delivered traffic | bit/s |
| GBR average occupied resources | RBs/subframe |
| GBR average queue per user | bits/subframe |
| GBR average SINR per user | dB/subframe |
| non-GBR incoming traffic | bit/s |
| non-GBR delivered traffic | bit/s |
| non-GBR occupied resources | RBs/subframe |
| non-GBR average queue per user | bits/subframe |
| non-GBR average SINR per user | dB/subframe |

**mMTC RAN Slice observed variables**

| variable | units |
|---|---|
| Simultaneous UE contexts | UEs |
| Remaining repetitions per UE | repetitions/UE |
| Average delay per UE | seconds/UE |

TABLE I
EXAMPLE OF SLICE OBSERVATIONS

slice until the next decision stage. Between consecutive stages, the agent collects per-slice measurements regarding user data traffic, channel quality conditions and SLA compliance parameters. These observations are used by the control agent to make the next decision and to learn about the response of the system. Table I shows a set of measurements that can be gathered from an eMBB or an mMTC slice. In the case of eMMB, the observation comprises a differentiated subset of variables for each type of traffic (GBR and non-GBR), since each type is associated to a specific QoS requirement in the SLA. These variables provide aggregated information at the system level: incoming traffic rate, delivered traffic rate, average resource occupation, average queue length, and average signal to interference ratio (SINR). In mMTC, each device is associated to a constant number of packet repetitions related to its estimated pathloss [38]. Therefore, the observed variables include the number of simultaneous UE contexts, the average delay per UE, and the average number of remaining packet repetitions per UE by the end of the previous stage.

Let us summarize the control sequence: at each decision stage $n$, the control agent receives the observation vector of each slice (containing the variables of Table I), and the KPIs of each slice (allowing the agent to assess if the slice's SLA has been fulfilled during the last decision period). Based on these observations, the agent selects a resource allocation with which the system will operate until the next decision stage. The control objective is to allocate the RBs as efficiently as possible while assuring that the SLAs are fulfilled with high probabilty.

## IV. Problem Definition

### A. Observations and Actions

Let $\mathcal{K}$ denote the set of $K$ active network slices coexisting in the RF interface, and let $C$ denote the total number of RBs to be allocated among the slices. At each decision stage $n = 1, 2, \ldots$, the control agent receives the per-slice observations gathered between stages $n-1$ and $n$, denoted by $s_{n-1}^{(i)}$, for each network slice $i \in \mathcal{K}$. The combination of the $K$ slice observations at stage $n$, denoted by $\mathbf{s}_n = (s_n^{(1)}, \ldots s_n^{(K)})$, is the system observation. Note that $s_n^{(i)}$ is a vector containing samples of the variables defined in Table I, which are random variables, since they are obtained by aggregating and/or averaging the realizations of multiple stochastic processes during the TTIs elapsed between stages $n-1$ and $n$. These processes are, for example, the arrival and departure of UEs to the cell, the GBR or non-GBR traffic generated by each UE, the data buffered at each UE, or the SINR measurements on each channel. Therefore, we will use the uppercase notation $S_n^{(i)}$ to refer to the slice observation as a random (multi-dimensional) variable, and the lowercase notation $s_n^{(i)}$ to denote a particular sample of the random variable $S_n^{(i)}$. Note also that $\mathbf{s}_n$ is not the state of the system, which is extremely complex to define since it involves multiple UEs, protocol layers, traffic flows, propagation conditions, and so on. Instead, it is a *partial* observation comprising system-level measurements that can be sufficient to make decisions on the allocation of bandwidth resources to network slices.

At decision stage $n$, the control agent selects the number of RBs $a_n^{(i)}$ to be allocated to each slice $i \in \mathcal{K}$. The combination of all the assignments $\mathbf{a}_n = (a_n^{(1)}, \ldots a_n^{(K)})$ denotes the control action at $n$. Since the decisions of the controller are determined by random variables, $S_n^{(i)}$ for $i \in \mathcal{K}$, the per-slice allocations are also random variables, $A_n^{(i)}$, while $a_n^{(i)}$ refers to a specific value.

For each slice $i$, we define an indicator function $I^{(i)}$ that informs the controller about the SLA fulfillment on a per-stage basis: $I^{(i)}(s_n^{(i)}, a_n^{(i)}) = 1$ if the SLA of slice $i$ has been violated between decision stages $n$ and $n+1$, and $I^{(i)}(s_n^{(i)}, a_n^{(i)}) = 0$ otherwise. Note that the function $I^{(i)}$ condenses all the key performance indicators (KPI) that have been defined in the SLA. For instance, in an eMBB slice, if the system has not been able to meet the GBR QoS level or the non-GBR QoS level, or any of them, the indicator function for this slice will return 1. It will return 0 only if all the specified QoS levels have been satisfied during the observation period.

### B. Control Policy

The decisions of the control agent are determined by its policy $\pi$, defined as a function that receives the system observation $\mathbf{s}_{n-1}$, and provides the control action $\mathbf{a}_n$ with the resource allocation per slice. Given the random system dynamics, and an initial distribution of the slice observations $S_0^{(1)}, \ldots S_0^{(K)}$, the policy $\pi$ determines a random sequence of observation-action pairs $S_0^{(i)}, A_1^{(i)}, S_1^{(i)}, A_2^{(i)}, S_2^{(i)}, \ldots$ for $i \in \mathcal{K}$. These $K$ sequences constitute a *trajectory* of the system. In our setting, a policy is *admissible* if it prescribes only actions that comply with $\sum_{i \in \mathcal{K}} a_n^{(i)} \leq C$ and $a_n^{(i)} \geq 0$ for $i \in \mathcal{K}$ (admissible actions). The set of admissible policies is denoted by $\Pi$.

### C. Constrained Markov Decision Process Formulation

Using the above definitions, the problem of finding an efficient policy for the control agent can be formulated as a constrained Markov decision process (CMDP) [39]. The objective of our CMDP is to find a policy $\pi \in \Pi$ that

minimizes the average amount of allocated resources, while the average number of SLA violations per slice is kept under a desired bound $\delta$:

$$\begin{aligned} &\min_{\pi\in\Pi} \lim_{N\to\infty} \tfrac{1}{N}\mathbb{E}_\pi \left[\textstyle\sum_{n=1}^{N}\sum_{i=1}^{K} A_n^{(i)}\right] \\ \text{s.t.} \quad &\lim_{N\to\infty} \tfrac{1}{N}\mathbb{E}_\pi \left[\textstyle\sum_{n=1}^{n} I^{(i)}(S_n^{(i)}, A_n^{(i)})\right] \leq \delta, \text{ for all } i \in \mathcal{K} \end{aligned} \tag{1}$$

where $\mathbb{E}_\pi$ denotes the expected value with respect to the distribution of the trajectories under policy $\pi$. Note that the capacity constraint is implicitly included by considering only admissible policies $\pi \in \Pi$, and all the QoS objectives of each slice are condensed in its indicator function $I^{(i)}$. This CMDP cannot be directly addressed because the system dynamics is unknown and the state of the system is not directly observable. Even without these limitations, the large dimension of the state and control spaces would render any numerical approach unfeasible.

### D. Markov Decision Process Formulation

In order to apply RL methods, it is necessary to reformulate (1) as an MDP by removing the SLA constraints and including them into the objective. The usual approach [13], [18], [29], [32] is to add the SLA violation to the objective function as a penalty term, multiplied by a weight factor $\lambda$. The resulting MDP is:

$$\min_{\pi\in\Pi} \lim_{N\to\infty} \frac{1}{N}\mathbb{E}_\pi \left[\sum_{n=1}^{N}\sum_{i=1}^{K}\left[A_n^{(i)} + \lambda I^{(i)}(S_n^{(i)}, A_n^{(i)})\right]\right]. \tag{2}$$

The term $\sum_{i=1}^{K}(a_n^{(i)} + \lambda I^{(i)}(s_n^{(i)}, a_n^{(i)}))$ is interpreted as the cost incurred by the system at stage $n$. This cost term can be also denoted by $-r_n(\mathbf{s}_n, \mathbf{a}_n)$, expressing the total (negative) reward of the observation-action pair $\mathbf{s}_n, \mathbf{a}_n$. The problem (2) is an *average* cost MDP because it aims at minimizing the average cost per stage (or maximizing the average reward per stage). However, most RL methods address discounted MDPs. We can reformulate (2) as a discounted MDP as follows:

$$\max_{\pi\in\Pi} \lim_{N\to\infty} \mathbb{E}_\pi \left[\sum_{n=1}^{N} \gamma^n r_n(\mathbf{S}_n, \mathbf{A}_n)\right]; \tag{3}$$

where $\gamma$ is a discount factor, and $\mathbf{S}_n, \mathbf{A}_n$ denote the (random) observation-action pair visited by the system trajectory at stage $n$. The discounted MDP (3) aims at maximizing the expected *return*, defined as the sum of the discounted rewards along the system trajectory.

### E. Solution Strategies

*1) Model-Free RL:* Model-free RL methods assume that the transition dynamics of the system is unknown. Their main idea is to estimate the expected return in (3) by taking samples of the trajectory. In order to do this, these methods build a parametric estimator of the expected return using, for example, a deep neural network. Policy gradient algorithms include a parametric policy, and estimate the gradient of the expected return with respect to the policy parameters. This makes it possible to gradually adjust these parameters by making gradient descent updates (see [40] for a coverage of RL techniques). The last 5 years have been particularly productive in terms of novel algorithms of this type. In Section VII we briefly review the state-of-the-art MFRL algorithms that we use as baselines in our performance evaluation experiments.

*2) Model-Based RL:* Model-based RL typically relies on learning the transition dynamics of the system instead of the optimal state values and/or policies [36], [37]. The main task of the learning process is to fit an approximation of the true transition function, given the states and the actions observed from the real system. Once a model is learned, the agent can use it to predict the expected return of each action at each observed state. Consequently, at each decision stage, the agent can evaluate multiple candidate action sequences, and select the optimal action sequence to use.

Our approach, instead of learning the transition dynamics, learns the effects of these dynamics on the SLA violation indicator functions $I^{(i)}$ for $i \in \mathcal{K}$. In particular, we build, for each slice $i$, a model $h_n^{(i)}$ that predicts if a given assignment $a_n^{(i)}$ will fulfill the SLA, given the observation $s_{n-1}^{(i)}$ received at the end of previous stage. Note that this strategy does not generate multi-step trajectories, i.e., we do not predict $S_n^{(i)}$, $S_{n+1}^{(i)} \ldots$, and therefore the agent will only be able to plan over a one-stage horizon. Although this strategy generally leads to suboptimal solutions, it allows us to address the original problem (1) instead of the modified one (2), resulting in better empirical results compared to model-free RL approaches (which are farsighted), as shown in Section VII.

Given $h_n^{(i)}$, we could approximately address the CMDP problem (1) as a one-step lookahead control problem, obtaining a model predictive controller (MPC) [40] in which the observation-action pairs must satisfy the SLA of each slice according to $h_n^{(i)}$ for $i \in \mathcal{K}$. However, this approach does not take the reliability factor $\delta$ into account. An insufficiently accurate predictor could cause excessive SLA violations.

We need to define the error function $e^{(i)}$ as the probability that the prediction given by $h_n^{(i)}$, on a given pair $(s_n^{(i)}, a_n^{(i)})$, is a false negative:

$$\begin{aligned} &e^{(i)}(h_n^{(i)}, s_{n-1}^{(i)}, a_n^{(i)}) = \\ &\mathbb{P}\left(I^{(i)}(S_n^{(i)}, a_n^{(i)}) = 1 \middle| h_n^{(i)}(s_{n-1}^{(i)}, a_n^{(i)}) = 0\right). \end{aligned} \tag{4}$$

Note that, by convention, we are associating the null hypothesis to the absence of any SLA violation in stage $n$, i.e. $I^{(i)}(S_n^{(i)}, a_n^{(i)}) = 0$. Therefore, $e^{(i)}$ denotes the probability of a type II error.

We have transformed the problem into a one-step lookahead control problem, in which each control action $\mathbf{a}_n$ should be admissible, and each element $a_n^{(i)}$ in $\mathbf{a}_n$ should be SLA compliant according to $h_n^{(i)}$ with an error probability bounded by the reliability factor $\delta$:

$$\begin{aligned} &\min_{a_n^{(1)} \dots a_n^{(i)}} \textstyle\sum_{i=1}^{K} a_n^{(i)}; \\ \text{s.t.} \quad & e^{(i)}(h_n^{(i)}, s_{n-1}^{(i)}, a_n^{(i)}) \leq \delta \quad \text{for } i \in \mathcal{K}, \\ & a_n^{(i)} \geq 0 \quad \text{for } i \in \mathcal{K}, \\ & \textstyle\sum_{i \in \mathcal{K}} a_n^{(i)} \leq C. \end{aligned} \tag{5}$$

It is straightforward to decompose the above problem into $K$ sub-problems, facilitating its online operation. As we will describe in the following section, the error functions $e^{(i)}$ will be also learned online, and thus the controller will use the learned functions $\hat{e}_n^{(i)}$ instead.

## V. Preliminaries

This section starts by describing the fundamentals of online learning for classification, and then explains how kernel-based online learning can be used for nonlinear classification, focusing on the algorithm that learns $h_n^{(i)}$ in our proposal.

### A. Online Learning

An online learning algorithm aims at learning a mapping $h : \mathcal{X} \to \mathbb{R}$ from a sequence of examples $(\mathbf{x}_n, y_n)$, $n = 1, \dots, N$, where $\mathbf{x}_n \in \mathcal{X}$ is called an *instance* and $y_n \in \mathbb{R}$ is called a *label*. In a linear binary classification task, the goal is to learn a linear classifier $h : \mathcal{X} \to \{-1, +1\}$ such that $h(\mathbf{x}_n, \theta) = \text{sgn}(\langle \theta, \mathbf{x}_n \rangle)$, $\mathcal{X}$ is typically a $d$-dimensional vector space, $\theta \in \mathbb{R}^d$ is a weight vector to be learned, $\langle \cdot, \cdot \rangle$ denotes the dot product, and $\text{sgn}(z)$ is an indicator function that outputs $+1$ when $z > 0$ and $-1$ otherwise. The function $h$ is called the hypothesis (function) or the (prediction) model, and is denoted by $h_n$ at stage $n$.

The main feature of online learning is that learning takes place in rounds or stages. At each stage $n = 1, 2, \dots$, an instance $\mathbf{x}_n$ is presented to the algorithm, which predicts a label $\hat{y}_n \in \{-1, +1\}$ using the current hypothesis function: $\hat{y}_n = h_n(\mathbf{x}_n)$. Then, the correct label $y_n$ is revealed, and the learner can measure the suffered loss, which in online binary classification can be given by the hinge-loss $\ell((\mathbf{x}_n, y_n); h_n) = \max(0, 1 - y_n h_n(\mathbf{x}_n))$. Whenever the loss is nonzero, the learner updates the prediction model $h_n$ according to an algorithm-specific strategy. The classic goal of online learning is to minimize the regret of the learner's predictions against the best fixed model in hindsight. The regret is defined as follows:

$$R_N = \sum_{n=1}^{N} \ell((\mathbf{x}_n, y_n); h_n) - \inf_{h \in \mathcal{H}} \sum_{n=1}^{N} \ell((\mathbf{x}_n, y_n); h) \tag{6}$$

where $\mathcal{H}$ denotes the model space. For example, in linear classification, $\mathcal{H}$ is the set of functions of the form $h(\mathbf{x}, \theta) = \text{sgn}(\langle \theta, \mathbf{x} \rangle)$ for $\theta \in \mathbb{R}^d$. Note that the second term in (6) is the loss suffered by the optimal model $h^* \in \mathcal{H}$ that can only be known in hindsight after seeing all the examples. Regret minimization formalizes the concept of sample efficiency.

### B. Online Learning with Kernels

If the online algorithm needs to learn a nonlinear model $h$, one way to introduce nonlinearity is by the use of kernels [41], [42]. In this case $\mathcal{H}$ is known as a Reproducing Kernel Hilbert Space (RKHS) and is defined by a kernel function $\kappa : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$. The kernel $\kappa(\mathbf{x}, \mathbf{x}')$ expresses the similarity between $\mathbf{x}$ and $\mathbf{x}'$, among other required properties [42], and allows us to write the hypothesis function $h_n$ as a kernel expansion as follows:

$$h_n(\mathbf{x}) = \sum_{\mathbf{x}_n \in \mathcal{X}_n} \alpha_n \kappa(\mathbf{x}_n, \mathbf{x}) \tag{7}$$

where $\alpha_n$ are coefficients (typically $\alpha_n = y_n$), and $\mathcal{X}_n$ is defined as the set of instances for which a prediction error occurred (and thus $h_n$ was updated), that is $\mathcal{X}_n = \{\mathbf{x}_t, 0 \leq t \leq n | \hat{y}_t \neq y_t\}$. The set $\mathcal{X}_n$ is called the *support set*.

The usual update step (see, e.g., the Kernelized Perceptron in [41]), involves adding the new instance $\mathbf{x}_{n+1}$ to the support set $\mathcal{X}_{n+1} = \mathcal{X}_n \cup \{\mathbf{x}_{n+1}\}$ and updating the hypothesis function as $h_{n+1} = h_n + y_{n+1}\kappa(\mathbf{x}_{n+1}, \cdot)$. This functional update expresses the addition of a new term $y_{n+1}\kappa(\mathbf{x}_{n+1}, \mathbf{x})$ to the summation in (7). One critical issue of this strategy is the unbounded growth of the support set $\mathcal{X}_n$, which increases the computational and space complexity over time. To address this drawback, one possible strategy is to set an upper bound (budget) on the cardinality $|\mathcal{X}_n|$ of the support set, and include a budget maintenance strategy to select which instance to remove when $|\mathcal{X}_n|$ reaches the budget [43]. An alternative, and more effective strategy is the use of *projected hypothesis*.

The hypothesis projection technique was introduced with the Projectron algorithm [6] and works as follows. Before adding an instance $\mathbf{x}_{n+1}$ to the support set, a *temporary hypothesis* is constructed as $h'_{n+1} = h_n + y_{n+1}\kappa(\mathbf{x}_{n+1}, \cdot)$. Additionally, a *projected hypothesis* $h''_{n+1}$ is obtained by computing the values for the coefficients $\alpha_n$ in expression (7) that best approximate $h'_{n+1}$ using the existing instances in the RKHS $\mathcal{X}_n$. We say that $h''_{n+1}$ is the *projection* of $h'_{n+1}$ onto $\mathcal{X}_n$. If the distance between $h''_{n+1}$ and $h'_{n+1}$ is below some threshold $\eta$, the next hypothesis will be $h''_{n+1}$, and the support set will remain unchanged. Otherwise, the next hypothesis will be $h'_{n+1}$ and the support set will incorporate $\mathbf{x}_{n+1}$. Algorithm 1 summarizes the Projectron algorithm.

## VI. Kernel-Based Online Learning for MBRL

### A. Elements of the Proposal

Our proposal is based on the following operating principles:

- *One-step lookahead planning:* the agent selects, at each decision stage, a control action addressing the problem formulated in (5) which is a one-step version of the CMDP (1).
- *Model-based RL*: in order to address (5), the agent learns a model of the system, instead of a policy or a value function. The model is intended to predict if the SLAs will be fulfilled or violated in each slice $i$, for any given state-action pair. Our modeling strategy includes a self-assessment procedure that computes the error function estimators $\hat{e}_n^{(i)}$, so that the constraint in (5) can be taken into account.

**Algorithm 1** Projectron

Initialize $\mathcal{X}_n \leftarrow \emptyset$, $h_0 \leftarrow \mathbf{0}$
**for** $n = 1, 2, \ldots$ **do**
  Receive new instance $\mathbf{x}_n$
  Predict $\hat{y}_n \leftarrow \text{sign}(h_{n-1}(\mathbf{x}_n))$
  Receive label $y_n$
  **if** $y_n \neq \hat{y}_n$ **then**
    $h'_n \leftarrow h_{n-1} + y_n \kappa(\mathbf{x}_n, \cdot)$
    $h''_n \leftarrow$ projection of $h'_n$ onto the space $\mathcal{X}_n$
    $\gamma_n \leftarrow h''_n - h'_n$
    **if** $\|\gamma_n\| \leq \eta$ **then**
      $h_n \leftarrow h''_n$
      $\mathcal{X}_n \leftarrow \mathcal{X}_{n-1}$
    **else**
      $h_n \leftarrow h'_n$
      $\mathcal{X}_n \leftarrow \mathcal{X}_{n-1} \cup \{\mathbf{x}_n\}$
    **end if**
  **else**
    $h_n \leftarrow h_n$
    $\mathcal{X}_n \leftarrow \mathcal{X}_{n-1}$
  **end if**
**end for**

- *Online learning for classification*: the model consists of a set of $K$ classifiers, one per slice, that are learned online. From any state-action pair, the observation variables that correspond to a particular slice constitute the input for the classifier associated with this slice.

Let $\mathcal{H}_n = \{h_n^{(1)}, \ldots, h_n^{(i)}\}$ and $\mathcal{E}_n = \{\hat{e}_n^{(1)}, \ldots, \hat{e}_n^{(i)}\}$ denote the set of hypothesis functions and the set of error function estimators in step $n$, respectively. The input variables of the agent are the amount of radio resources $C$, the reliability factor $\delta$ and the maximum number of resources per slice, denoted by $\mathbf{a}_{max} = (a_{max}^{(1)}, \ldots, a_{max}^{(i)})$.

As shown in Figure 2 and Algorithm 2, our proposal, kernel-based RL (KBRL), is structured into three modules:

1) A *controller* that generates, at each stage $n$, a control action vector $\mathbf{a}_n = (a_n^{(1)}, \ldots, a_n^{(i)})$ based on the observed state at the end of previous stage $\mathbf{s}_{n-1} = (s_{n-1}^{(1)}, \ldots, s_{n-1}^{(i)})$, and on the system model determined by $\mathcal{H}_n$ and $\mathcal{E}_n$. The controller provides two additional vectors: $\hat{\mathbf{y}}_n = (\hat{y}_n^{(1)}, \ldots, \hat{y}_n^{(K)})$, with the SLA violation predictions for all slices in stage $n$, and $\mathbf{m}_n = (m_n^{(1)}, \ldots, m_n^{(K)})$ containing the security margins for each slice. These vectors are necessary for updating $\mathcal{E}_n$ as explained later in subsection VI-C.
2) The *H-learner*, which runs $K$ online learning algorithms for classification, in parallel, one for each of the $K$ hypothesis functions in $\mathcal{H}_n$. These functions are updated at every stage, based on $\mathbf{s}_{n-1}$, $\mathbf{a}_n$, and the array of observed labels $\mathbf{y}_n$.
3) The *E-learner*, which updates the $K$ estimators in $\mathcal{E}_n$, taking $\hat{\mathbf{y}}_n$, $\mathbf{y}_n$, $\mathbf{m}_n$ as inputs, and providing $\mathcal{E}_{n+1}$ as output. This module monitors the accuracy of the classifiers learned by the H-learner module.

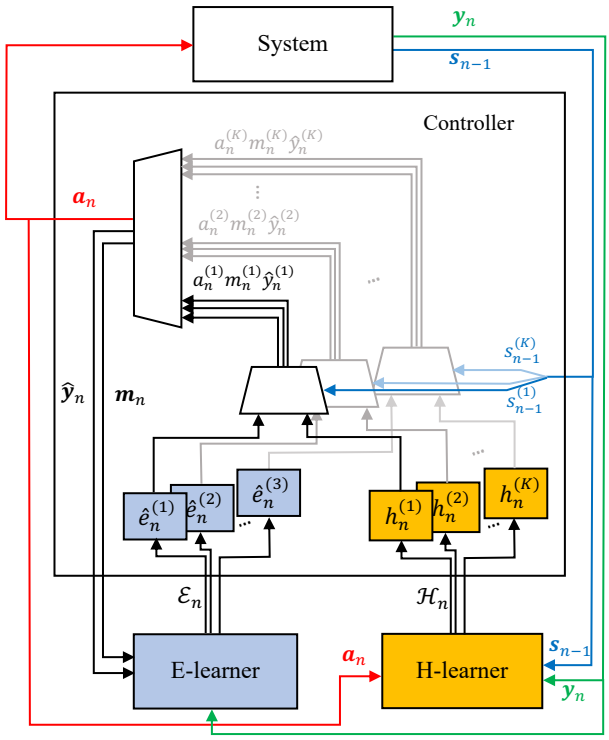


Fig. 2. Diagram of the proposed control system.

**Algorithm 2** KBRL

**Inputs:** $C$, $\delta$, $\mathbf{a}_{max} = (a_{max}^{(1)}, \ldots, a_{max}^{(i)})$
Initialize $\mathcal{H}_1 = \{h_1^{(1)}, \ldots, h_1^{(i)}\}$, $\mathcal{E}_1 = \{\hat{e}_1^{(1)}, \ldots, \hat{e}_1^{(i)}\}$
Observe $\mathbf{s}_0$
**for** $n = 1, 2, \ldots$ **do**
  $(\mathbf{a}_n, \mathbf{m}_n, \hat{\mathbf{y}}_n) \leftarrow$ Controller$(\mathcal{H}_n, \mathcal{E}_n, \mathbf{s}_{n-1})$
  Apply $\mathbf{a}_n$ and observe $\mathbf{y}_n$ and $\mathbf{s}_n$
  $\mathcal{H}_{n+1} \leftarrow$ H-learner$(\mathcal{H}_n, \mathbf{y}_n, \mathbf{s}_{n-1}, \mathbf{a}_n)$
  $\mathcal{E}_{n+1} \leftarrow$ E-learner$(\mathcal{E}_n, \hat{\mathbf{y}}_n, \mathbf{y}_n, \mathbf{m}_n)$
**end for**

### *B. H-Learner: Online Learning for Classification*

To increase the learning efficiency of $h_n^{(i)}$, we introduce a sample augmentation strategy, for which we use the following assumption.

**Assumption 1**. Given an initial set of conditions for a slice $i \in \mathcal{K}$, summarized in $s_{n-1}^{(i)}$, if the SLA of slice $i$ is fulfilled in stage $n$ with $a$ resources assigned to this slice, then the SLA is also fulfilled with $a' > a$ resources. Conversely, if the SLA is not fulfilled with $a$ resources, then it is neither fulfilled with $a'' < a$ resources.

This assumption simply reflects a desirable feature of the underlying schedulers that allocate the resources among the UEs within each slice: more available resources can only improve the quality of service provided to the UEs.

As shown in Algorithm 3, the H-learner receives, for each slice $i \in \mathcal{K}$, the $s_{n-1}^{(i)}$, $a_n^{(i)}$ pair, and the observed label $y_n^{(i)}$. When $y_n^{(i)} = 0$, meaning that no SLA violation happened in slice $i$ during stage $n$, Assumption 1 implies that any action $a > a_n^{(i)}$ would have provided the same label. Therefore, we

can augment the number of samples by generating additional instances $\mathbf{x}_n = (s_{n-1}^{(i)}, a)$ for $a = a_n^{(i)}, \ldots, a_{max}^{(i)}$ associated to label $y_n^{(i)} = 0$. Similarly, when the observed label is $y_n^{(i)} = 1$, we insert new instances with $a = 0, \ldots, a_n^{(i)}$.

**Algorithm 3** H-learner

**Parameters:** $\mathbf{a}_{max} = (a_{max}^{(1)}, \ldots, a_{max}^{(i)})$
**Inputs:** $\mathcal{H}_n = \{h_n^{(1)}, \ldots, h_n^{(i)}\}$, $\mathbf{y}_n = (y_n^{(1)}, \ldots, y_n^{(i)})$, $\mathbf{s}_{n-1} = (s_{n-1}^{(1)}, \ldots, s_{n-1}^{(i)})$, $\mathbf{a}_n = (a_n^{(1)}, \ldots, a_n^{(i)})$
**Outputs:** $\mathcal{H}_{n+1} = \{h_{n+1}^{(1)}, \ldots, h_{n+1}^{(i)}\}$
**for** $i = 1, \ldots, K$ **do**
  **if** $y_n^{(i)} = 0$ **then**
    **for** $a = a_n^{(i)}, \ldots, a_{max}^{(i)}$ **do**
      $\mathbf{x}_n \leftarrow (s_{n-1}^{(i)}, a)$
      $h_{n+1}^{(i)} \leftarrow$ Projectron update with $\mathbf{x}_n$ and $y_n^{(i)}$
    **end for**
  **else**
    **for** $a = 0, \ldots, a_n^{(i)}$ **do**
      $\mathbf{x}_n \leftarrow (s_{n-1}^{(i)}, a)$
      $h_{n+1}^{(i)} \leftarrow$ Projectron update with $\mathbf{x}_n$ and $y_n^{(i)}$
    **end for**
  **end if**
**end for**

### C. E-Learner: Estimating the Classification Error Probability

In order to estimate the prediction accuracy for each action, we use the following structural result that is a direct consequence of Assumption 1.

**Proposition 1**. Given an initial set of conditions for a slice $i \in \mathcal{K}$, summarized in $s_{n-1}^{(i)}$, if two actions $a < a'$ are predicted to fulfill the SLA: $h_n^{(i)}(s_{n-1}^{(i)}, a) = h_n^{(i)}(s_{n-1}^{(i)}, a') = 0$, then $e^{(i)}(h_n^{(i)}, s_{n-1}^{(i)}, a') \leq e^{(i)}(h_n^{(i)}, s_{n-1}^{(i)}, a)$.

*Proof.* For any $s_{n-1}^{(i)}$, $a$, and $a'$, with $a < a'$, if $I^{(i)}(S_n^{(i)}, a) = 1$, $I^{(i)}(S_n^{(i)}, a')$ can be either 1 or 0. But, from Assumption 1 if $I^{(i)}(S_n^{(i)}, a) = 0$, then $I^{(i)}(S_n^{(i)}, a') = 0$, thus $\mathbb{E}\left[I^{(i)}(S_n^{(i)}, a) | s_{n-1}^{(i)}, a\right] \geq \mathbb{E}\left[I^{(i)}(S_n^{(i)}, a') | s_{n-1}^{(i)}, a'\right]$. If $s_{n-1}^{(i)}$, $a$, and $a'$, are such that $h_n^{(i)}(s_{n-1}^{(i)}, a) = h_n^{(i)}(s_{n-1}^{(i)}, a') = 0$ then $\mathbb{E}\left[I^{(i)}(S_n^{(i)}, a) | h_n^{(i)}(s_{n-1}^{(i)}, a) = 0\right] \geq \mathbb{E}\left[I^{(i)}(S_n^{(i)}, a') | h_n^{(i)}(s_{n-1}^{(i)}, a') = 0\right]$, which by definition is equivalent to $e^{(i)}(h_n^{(i)}, s_{n-1}^{(i)}, a) \geq e^{(i)}(h_n^{(i)}, s_{n-1}^{(i)}, a')$. □

Given the conditions of Proposition 1, if $a$ is the smallest action predicted to fulfill the SLA of slice $i$, we say that $a'$ has a security margin of $a' - a$. The security margin $m_n^{(i)}$ for a given action $a_n^{(i)}$ is defined as follows:

$$m_n^{(i)} = a_n^{(i)} - \min\{a : h_n^{(i)}(s_{n-1}^{(i)}, a) = 0\}. \tag{8}$$

By Proposition 1, the larger the security margin, the smaller the classification error probability $e^{(i)}$. Therefore, to obtain an estimator of $e^{(i)}$, we compute the average prediction error associated to all (positive) security margins $m = 0, 1, \ldots, a_{max}$, for each slice. For a given $m$, the error probability estimator, denoted by $\hat{e}_n^{(i)}(m)$, approximates $e^{(i)}(h_n^{(i)}, s_{n-1}^{(i)}, a')$, where $a'$ has a security margin of $m$.

As shown in Algorithm 4, the E-learner receives, for each slice $i$, the prediction $\hat{y}_n^{(i)}$, the true label $y_n^{(i)}$, and the security margin used in stage $n$, $m_n^{(i)}$.Since $e^{(i)}$ corresponds to a Type-II error, $\hat{e}^{(i)}$ is only updated when $\hat{y}_n^{(i)} = 0$. Therefore, the prediction error at stage $n$ is equal to $y_n^{(i)}$ and we can update the estimator $\hat{e}^{(i)}$ for a given $m$ as

$$e_{n+1}^{(i)}(m) = (1 - \beta)\hat{e}_n^{(i)}(m) + \beta y_n^{(i)} \tag{9}$$

where $\beta$ is the learning rate. The E-learner uses a sample augmentation strategy similar to the one discussed in previous subsection. If $y_n^{(i)} = 0$ (meaning that the SLA is fulfilled) for a given $m_n^{(i)}$, by Assumption 1, we can update $\hat{e}_n^{(i)}$ with $y_n^{(i)} = 0$ for $m = m_n^{(i)}, \ldots, a_{max}^{(i)}$. Similarly, if $y_n^{(i)} = 0$ we can update $\hat{e}_n^{(i)}$ with $y_n^{(i)} = 1$ for $m = m_n^{(i)}, \ldots, a_{max}^{(i)}$.

**Algorithm 4** E-learner

**Parameters:** $\beta$, $\mathbf{a}_{max} = (a_{max}^{(1)}, \ldots, a_{max}^{(i)})$
**Inputs:** $\mathcal{E}_n = \{\hat{e}_n^{(1)}, \ldots, \hat{e}_n^{(i)}\}$, $\hat{\mathbf{y}}_n = (\hat{y}_n^{(1)}, \ldots, \hat{y}_n^{(i)})$, $\mathbf{y}_n = (y_n^{(1)}, \ldots, y_n^{(i)})$, $\mathbf{m}_n = (m_n^{(1)}, \ldots, m_n^{(i)})$
**Outputs:** $\mathcal{E}_{n+1} = \{\hat{e}_{n+1}^{(1)}, \ldots, \hat{e}_{n+1}^{(i)}\}$
**for** $i = 1, \ldots, K$ **do**
  **if** $\hat{y}_n^{(i)} = 0$ **then**
    **if** $y_n^{(i)} = 0$ **then**
      **for** $m = m_n^{(i)}, \ldots, a_{max}^{(i)}$ **do**
        $\hat{e}_{n+1}^{(i)}(m) \leftarrow (1 - \beta)\hat{e}_n^{(i)}(m)$
      **end for**
    **else**
      **for** $m = 0, \ldots, m_n^{(i)}$ **do**
        $\hat{e}_{n+1}^{(i)}(m) \leftarrow (1 - \beta)\hat{e}_n^{(i)}(m) + \beta$
      **end for**
    **end if**
  **end if**
**end for**

### D. Controller

Algorithm 5 shows how the controller uses the classifiers in $\mathcal{H}_n$ and the error estimators in $\mathcal{E}_n$ to select the action at stage $n$. For each slice $i \in \mathcal{K}$, the controller selects the minimum security margin $m_n^{(i)}$ that, according to $\hat{e}_n^{(i)}$, attains an error probability below $\delta$. Then, it looks for the minimum action for which the classifier $\hat{e}_n^{(i)}$ estimates no SLA violation, and increments this action by $m_n^{(i)}$. In case $a_n^{(i)}$ surpasses $a_{max}^{(i)}$, $m_n^{(i)}$ is downsized so that $a_n^{(i)} = a_{max}^{(i)}$.

When the solution to the $K$ sub-problems does not fit the global resource constraint, $\sum_{i \in \mathcal{K}} a_n^{(i)} > C$, we project the solution onto the space of admissible actions. This is done by computing the set of actions $\bar{a}^{(1)}, \ldots, \bar{a}^{(i)}$ such that $\sum_{i \in \mathcal{K}} \bar{a}_n^{(i)} = C$, and $\frac{\bar{a}^{(i)}}{C} = \frac{a^{(i)}}{\sum_{i' \in \mathcal{K}} a^{(i')}}$.

### E. Complexity

The most time-consuming part of the proposed method is the Projectron update (Algorithm 1), whose time complexity, analyzed in [6], is $\mathcal{O}(|\mathcal{X}_n|^2)$ (recall that $|\mathcal{X}_n|$ denotes the cardinality of the support set). $|\mathcal{X}_n|$ tends to increase over time,

**Algorithm 5** Controller

**Parameters:** $C$, $\delta$, $\mathbf{a}_{max} = (a_{max}^{(1)}, \ldots, a_{max}^{(i)})$
**Inputs:** $\mathcal{H}_n = \{h_n^{(1)}, \ldots, h_n^{(i)}\}$, $\mathcal{E}_n = \{\hat{e}_n^{(1)}, \ldots, \hat{e}_n^{(i)}\}$, $\mathbf{s}_{n-1} = (s_{n-1}^{(1)}, \ldots, s_{n-1}^{(i)})$
**Outputs:** $\mathbf{a}_n = (a_n^{(1)}, \ldots, a_n^{(i)})$, $\mathbf{m}_n = (m_n^{(1)}, \ldots, m_n^{(i)})$, $\hat{\mathbf{y}}_n = (\hat{y}_n^{(1)}, \ldots, \hat{y}_n^{(i)})$
**for** $i = 1, \ldots, K$ **do**
    $m_n^{(i)} \leftarrow \min\{m : \hat{e}_n^{(i)}(m) \leq \delta\}$
    $a_n^{(i)} \leftarrow -1$ and $\hat{y}_n^{(i)} \leftarrow 1$
    **while** $\hat{y}_n^{(i)} = 1$ and $a_n^{(i)} < a_{max}^{(i)}$ **do**
        $a_n^{(i)} \leftarrow a_n^{(i)} + 1$
        $\hat{y}_n^{(i)} \leftarrow h_n^{(i)}(s_{n-1}^{(i)}, a_n^{(i)})$
        **if** $\hat{y}_n^{(i)} = 0$ **then**
            $m_n^{(i)} \leftarrow \min(m_n^{(i)}, a_{max}^{(i)} - a_n^{(i)})$
            $a_n^{(i)} \leftarrow a_n^{(i)} + m_n^{(i)}$
        **end if**
    **end while**
**end for**
**if** $\sum_i a_n^{(i)} > C$ **then**
    **for** $i = 1, \ldots, K$ **do**
        $\bar{a}^{(i)} \leftarrow \lfloor \frac{C a^{(i)}}{\sum_{i' \in \mathcal{K}} a^{(i')}} \rfloor$
        $m_n^{(i)} \leftarrow m_n^{(i)} - (a_n^{(i)} - \bar{a}^{(i)})$
        $a_n^{(i)} \leftarrow \bar{a}^{(i)}$
        $\hat{y}_n^{(i)} \leftarrow h_n^{(i)}(s_{n-1}^{(i)}, a_n^{(i)})$
    **end for**
**end if**

but in our experiments it remained at relatively low values (accumulating fewer than 60 elements after 40000 learning steps). The H-learner (Algorithm 3) performs this update $K$ times, resulting in a time-complexity of $\mathcal{O}(K|\mathcal{X}_n|^2)$. The E-learner (Algorithm 4) performs, at most, $C$ updates per slice. Similarly, the controller (Algorithm 5) assesses, at most, $C$ values of $a_n^{(i)}$ per slice. The resulting time-complexity of KBRL is thus $\mathcal{O}(K|\mathcal{X}_n|^2 + 2KC)$ which is smaller, in practice, than the time-complexity of a backpropagation step in deep RL algorithms, $\mathcal{O}(n_{layers} n_{neurons}^3)$, where $n_{layers}$ is the number of layers and $n_{neurons}$ is the number of neurons per layer. In Section VII we show a detailed empirical evaluation of the per-stage execution time of each algorithm.

## VII. Numerical Evaluation

We have conducted extensive simulation experiments to evaluate our proposal and compare it with state-of-the-art RL baselines. The simulation environment was developed in Python[1] to emulate the allocation of RBs among several RAN slices. Each slice is devoted to one type of communication, either eMBB or mMTC, characterized by its traffic model and its SLA. The traffic generator of a eMBB slice simulates the arrival and departure of GBR and non-GBR UEs, characterized by constant bit rate and variable bit rate traffic flows respectively, according to the parameters shown in Table II. The traffic source of a mMTC silce simulates 1000 devices, each one characterized by a transmission period and a number of packet repetitions. For each device, these two parameters are randomly selected from the sets shown in Table II. The nominal received power at the UE uses the macro cell propagation model for urban areas described in Section 4.5 of TR 36.942 [44]. Frequency-selective fading is generated by drawing samples from datasets containing fading traces. Our simulator uses the datasets of the ns-3 simulator [45] corresponding to usual fading/mobility models (pedestrian A, Typical Urban, Vehicular A, as defined in Annex B.2 of TS 36.104 [46]). For each new arrival, the simulator randomly selects one of the datasets, and generates a random integer as the first index from which to draw samples from the selected dataset. Within each slice, a proportional fair scheduler allocates the RBs among the UEs of this slice, according to their buffer state reports and their SNR estimations. Given the SNR estimated by the UE, the transmitter selects a Modulation and Coding Scheme (MCS) aiming to a block error rate (BLER) below 0.1. The spectral efficiency of the selected MCS, with the number of allocated RBs, determines the amount of bits transmitted (transport block size) from the UE queue. It should be highlighted that we conducted the experiments using a different strategy for the generation of SNR samples based on a the dataset obtained in [47], and the results were very similar, which suggests that the channel modeling details are not of crucial importance in the evaluation and comparison of the algorithms. Note that the slice resource allocation operates on a larger time scale and at a higher system level than the scheduling algorithms, and consequently its performance is relatively decoupled from the channel models.

In the simulated scenarios, the allocation of radio resources among the RAN slices is updated every 50 radio subframes, and the duration of each subframe is 1 ms. Therefore, the

| **eMBB GBR traffic model** | |
|---|---|
| UE arrivals | Poisson process with arrival rate = 2 users/min |
| UE connection time | Exponentially distributed with mean = 30 seconds |
| Bit rate | 0.5 Mb/s |
| **eMBB non GBR traffic model** | |
| UE arrivals | Poisson process with arrival rate = 5 users/min |
| UE connection time | Exponentially distributed with mean = 30 seconds |
| Burst arrivals | Poisson process with arrival rate = 1 burst/second |
| Burst length | Exponentially distributed with mean = 500 packets |
| Burst bit rate | 1 Mb/s |
| **mMTC traffic model** | |
| mMTC devices | 1000 |
| Transmission periods | $\{10, 50, 100, 150, 200, 250, 500, 1000\}$ seconds |
| Packet repetitions | $\{2, 4, 8, 16, 32, 64, 128\}$ |
| Packet size | 1000 bits |
| **Radio channel parameters** | |
| Transmitted power | 30 dBm |
| Base station (BS) antenna gain | 15 dBi |
| BS antenna pattern | Section 4.2.1, TS 36.942 [44] |
| Cell range | 2 Km |
| Noise figure | 9 dB |
| Interference plus noise per RB | $-110$ dBm |
| Carrier frequency | 2 GHz |
| Propagation model | Macro cell urban (Section 4.5.1, TS 36.942 [44]) |
| Fading models | Pedestrian A, Typical Urban, Vehicular A (Annex B.2, TS 36.104 [46]) |
| **KBLR parameters** | |
| E-learner $\beta$ parameter | 0.01 |
| Projectron $\eta$ parameter | 0.1 |
| Projectron Kernel | Gaussian with $\sigma = 1$ |

TABLE II
Simulation parameters

[1]The code of the simulator, the proposed algorithm, and the scripts for replicating the experiments can be downloaded from https://github.com/jjalcaraz-upct/network-slicing

| eMBB RAN Slice SLA | |
|---|---|
| GBR authorized capacity | 20 RBs/subframe |
| non-GBR QoS compliant capacity | 30 RBs/subframe |
| Maximum average queue per GBR user | 100 Kbit/UE |
| Maximum average queue per non-GBR user | 150 Kbit/UE |
| **mMTC RAN Slice SLA** | |
| Maximum number of UE contexts | 1000 |
| Maximum per user delay | 300 ms |

TABLE III
SERVICE LEVEL AGREEMENTS (SLAS)

elapsed time between consecutive decision stages is 50 ms. Three main scenarios have been considered for the experimental evaluation:

- *Scenario 1*: 5 eMBB slices sharing 200 RBs per subframe.
- *Scenario 2*: 3 eMBB and 2 mMTC RAN slices. 150 RBs per subframe.
- *Scenario 3*: 1 eMBB and 4 mMTC RAN slices. 100 RBs per subframe.

Besides, an additional scenario has been defined for experiments requiring a smaller action space:

- *Scenario 4*: 1 eMBB and 1 mMTC RAN slice. 70 RBs per subframe.

The SLAs for the eMBB and mMTC slices of all scenarios are summarized in Table III. The number of observed variables, which is determined by Table I in Section III, is 50, 36, 22 and 13 for scenarios 1, 2, 3 and 4 respectively.

### A. RL Baselines

We compare our proposed KBRL controller against the following algorithms, considered state-of-the-art baselines in RL:

- **Deep Q-Networks (DQN)** [48] Is the deep learning version of Q-learning, a classical model-free off-policy RL algorithm. As discussed in Section II, Q-learning and DQN have been used for the allocation of RBs among network slices by [19] and [18], respectively. Nevertheless, its application is limited to scenarios with a small action space, i.e. with only 2 or 3 slices. Consequently, the comparison of KBRL with DQN was conducted only on scenario 4.
- **Trust region policy optimization (TRPO)** [49] is a model-free deep policy gradient algorithm that updates policies while satisfying a constraint on how different the new and old policies are allowed to be. This difference is expressed in terms of Kullback-Liebler Divergence.
- **Proximal policy optimization (PPO)** [50] is a variant of the TRPO idea, that uses a simpler technique to estimate the difference between policies. Two implementations have been used: PPO1 and PPO2, described in [51].
- **Twin delayed DDPG (TD3)** [52] is an off-policy deep actor-critic algorithm. It is an improvement over deep deterministic policy gradient (DDPG) [53].
- **Soft actor critic (SAC)** [54] is an off-policy deep actor-critic algorithm that incorporates the idea of entropy regularization, and reports better empirical performance than DDPG.
- **Synchronous advantage actor critic (A2C)** [55] is an on-policy deep actor-critic algorithm. It can execute multiple instances of the algorithm in parallel, although this feature is not applicable in online learning, where only one instance of the environment is available.
- **Normalized Advantage Function (NAF)** is a technique for extending deep Q networks (DQN) to continuous actions spaces, proposed in [56], and used in [13] for RB allocation among network slices.

In our experiments, we have used the implementations of the RL algorithms provided by *Stable Baselines* [51], which is an improved version of the OpenAI Baselines [57]. For NAF, which is not included in these libraries, we have used the Keras-RL implementation [58].

### B. Online Performance

To evaluate and compare our proposal with all the RL baselines, the simulation experiments were designed as follows: In each scenario we executed $30$ simulation runs for each algorithm. Each run comprises two consecutive phases: the learning phase and the inference phase. During the learning phase, which lasts $40000$ decision stages (equivalent to $33.3$ minutes of simulated time), the algorithms learn from the interaction with the system, starting without any prior knowledge of the system's response. This situation is equivalent to the establishment of new network slices, or an update of the SLAs of existing slices. The evaluation of this phase is critical since the objective of our proposal is to learn while the network is operating, minimizing the negative effects of this learning on the service offered. During the inference phase, lasting $10000$ steps ($8.3$ minutes of simulated time), the RL algorithms are not learning anymore and they simply use the policies obtained in the learning phase to make RB allocation decisions at each step. This phase models a situation where the network slices remain stable beyond the $40000$ steps of the learning phase, and its objective is to compare our proposal to the baselines once they have been previously trained. This evaluation has been done for the sake of completeness, even though its interest is restricted to scenarios where changes in the network slices are infrequent. The performance of the algorithms is characterized by the following measurements:

1) The number of *SLA violations* per stage. This measurement ranges from $0$ to $5$ (the number of network slices).
2) The *cumulative number of SLA violations*, obtained by aggregating the values of previous measurements, is equivalent to the *regret*, allowing us to compare the learning rate of the algorithms.
3) The amount of *allocated resources*, ranging from $0$ to the maximum RBs per subframe in each scenario. This measurement allows us to compare the spectral efficiency of the algorithms.

The proposed algorithm has been evaluated for two values of the reliability factor: $\delta = 0.97$ and $\delta = 0.99$, to assess its impact on the performance. For the baselines, two values of the penalty factor were considered, $\lambda = 100$ and $\lambda = 1000$,

and we selected the second one for being more effective in avoiding SLA violations during the learning phase, which is crucial in an online learning setting. Besides, we also evaluated the baselines under normalized rewards (between $-1$ and $+1$), resulting in better performance for TD3 and NAF, whose results use this configuration.

Figure 3 shows the performance of the algorithms in scenario 1 during the first $20000$ steps of the learning phase. For each measurement, we plot the average value and the confidence interval for a $90\%$ confidence degree. The results for scenarios 2 and 3 are presented in Figures 4 and 5 respectively. It is evident that KBRL largely outperforms even the best baselines in terms of SLA fulfillment, and also shows greater efficiency in the use of resources. These results show the effectiveness of our model-based approach for learning efficient policies with much fewer samples than MBRL algorithms[2]. Moreover, the model used by KBRL is aimed at minimizing exploration, which prevents the selection of over-provisioning actions as well as under-provisioning actions from the early stages of the learning process. Another consequence of this design is that the performance of KBRL remains consistent across the different scenarios, while the performance of the RL baselines shows notable variations from one scenario to another.

Let us see how the algorithms compare during the inference phase. For this phase, we evaluated the two conflicting objectives of the problem: the average SLA violations per stage, and the average resource occupation (average number of used RBs divided by the total number of RBs) and then, to facilitate the visualization of the performance, we represented both metrics in an euclidean axis for each scenario. Figure 6 shows the results, where the average performance of each algorithm corresponds to a point and centered on the point is the confidence interval of each metric in its respective dimension.

The findings of the training phase are observed also in the inference phase. The relative performance of the baselines varies notably from one scenario to another. For example, we see that PPO1 is the best baseline in scenario 1 in terms of SLA fulfillment but it is outperformed by TD3 with regard to resource efficiency. In scenario 3, PPO1 is the baseline using fewer resources but we see that four baselines (TD3, TRPO, PPO2 and SAC) attain lower SLA violation rate. In sharp contrast, KBRL performs consistently across the three scenarios: its SLA violation rate is almost negligible, clearly outperforming all the baselines in every scenario, while using fewer resources than all or most of the baselines. Not surprisingly, KBRL with $\delta = 0.99$ uses more resources than KBRL with $\delta = 0.97$, since the higher the reliability factor, the larger the security margin ($m_n^{(i)}$ defined in Section VI), and thus more resources are assigned to each slice.

To better understand KBRL operation, it is illustrative to evaluate two additional measurements specific to this algorithm. The first one is the rate at which KBRL generates solutions exceeding the available resources $C$, thus requiring to be adjusted as detailed in Algorithm 5. We use the term *adjustments* to refer to these events. Figure 7 shows the average number of adjustments per decision stage on each scenario, including the confidence intervals with a $90\%$ confidence degree. As we can see, the adjustment rate is generally below $0.1$. For a given scenario, if we used a smaller $C$, we would obtain a higher adjustment rate, but we consider that operating at a relatively low adjustment rate is indicative of a proper dimensioning of the system, i.e. the existing resources are enough to accommodate all the slices, fulfilling the required SLA, and leaving room to absorb occasional traffic peaks. Note that, thanks to the security margins, the SLA violation rates (shown in Figure 6) are clearly smaller than the adjustment rates.

The second measurement of interest for KBRL evaluation is the accuracy of the online classifiers. Figure 8 depicts the estimated accuracy of the online classifier in KBRL. For each stage, the plots show the average number of classifier hits with its confidence interval. Consistently with previous results, the accuracy tends to increase during the episode, and the accuracy values for $\delta = 0.99$ are generally above the values for $\delta = 0.97$.

To conclude our evaluation of KBRL, we show how it compares to an optimal performance, which was estimated using an *oracle* mechanism. The oracle operates as follows: at each decision stage $n$, it conducts an exhaustive search over all the possible allocations. For each allocation $\mathbf{a}_n$, the system is simulated up to decision stage $n+1$, to assess the SLA fulfillment for all the slices. For all the allocations evaluated, the simulation starts from the same system state. Once the best allocation $\mathbf{a}_n^*$ is found (i.e. the one fulfilling the SLAs using the fewest resources), the system advances up to the next stage $n+1$, and the searching process for $\mathbf{a}_{n+1}^*$ starts. Note that the oracle control is unfeasible for implementation in a real system, since it requires prescient knowledge of all the stochastic processes involved, it is computationally cumbersome, and not scalable even for simulation, due to the exponential growth of the action space with the number of slices. Consequently, KBRL and the oracle were compared only in scenario 4, which comprises only 2 network slices, resulting in a relatively small action space ($556$ actions). This feature makes this scenario useful for the evaluation of DQN, which was used by previous works [19] [18] for RB allocation among network slices. As a reference, we also include the results of NAF, which was used in [13] to overcome the limitations of DQN in this problem. For the RL baselines, we consider a learning phase lasting $20000$ steps, and an inference phase of $4000$ steps.

Figure 9 compares the performance of the RL algorithms (the baselines and KBRL) during the training phase. We see that DQN outperforms NAF in this scenario. Note that NAF needs to approximate the discrete action space as a continuous one, which may impact its performance. As in previous scenarios, KBRL is capable of learning with a negligible amount of SLA violations, outperforming both baselines in this metric.

Figure 10 shows the performance of the oracle policy in comparison to KBRL, NAF and DQN. In the experiments

[2]We believe that our proposal could be applicable to other resource allocation problems beyond network slicing. This issue is open for future research.

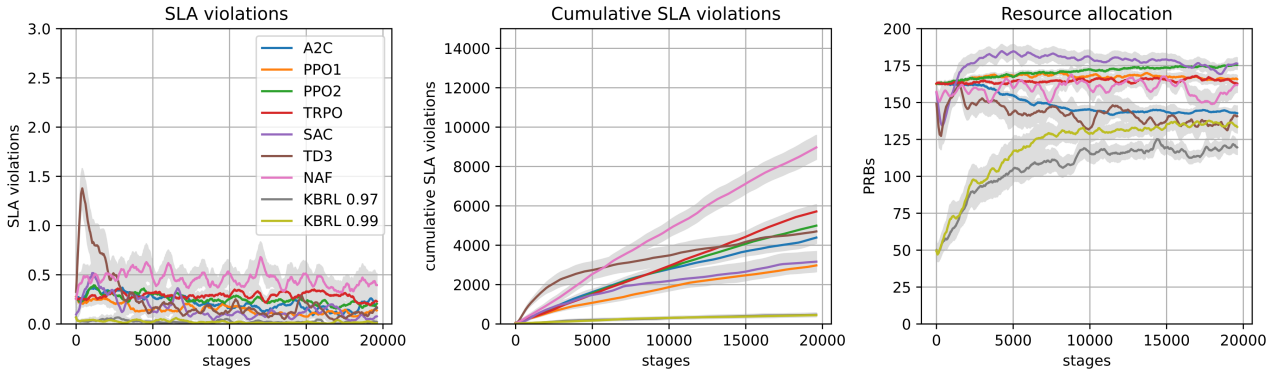


Fig. 3. Results for scenario 1. From left to right: SLA violations per stage, cumulative SLA violations, and total RBs allocated. Each plot shows the per-stage average of 30 simulation runs and its confidence interval with a 90% confidence degree.

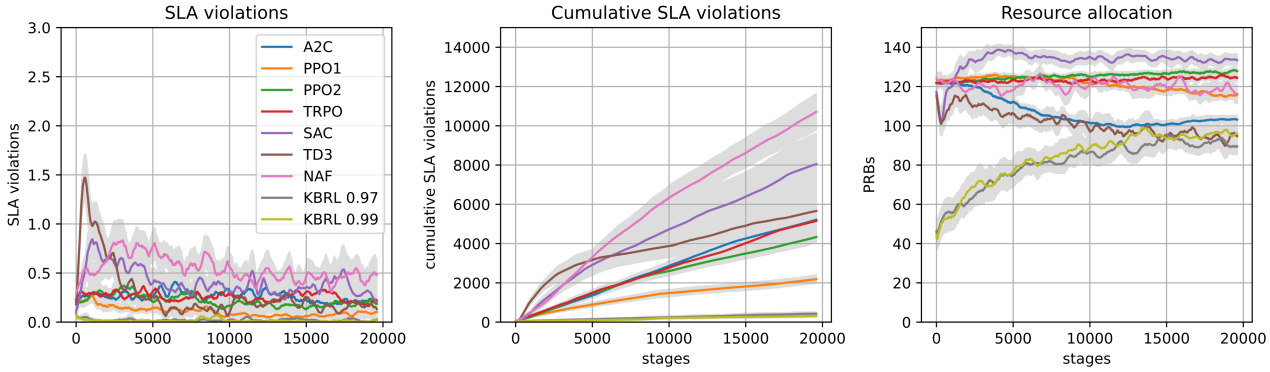


Fig. 4. Results for scenario 2. From left to right: SLA violations per stage, cumulative SLA violations, and total RBs allocated. Each plot shows the per-stage average of 30 simulation runs and its confidence interval with a 90% confidence degree.

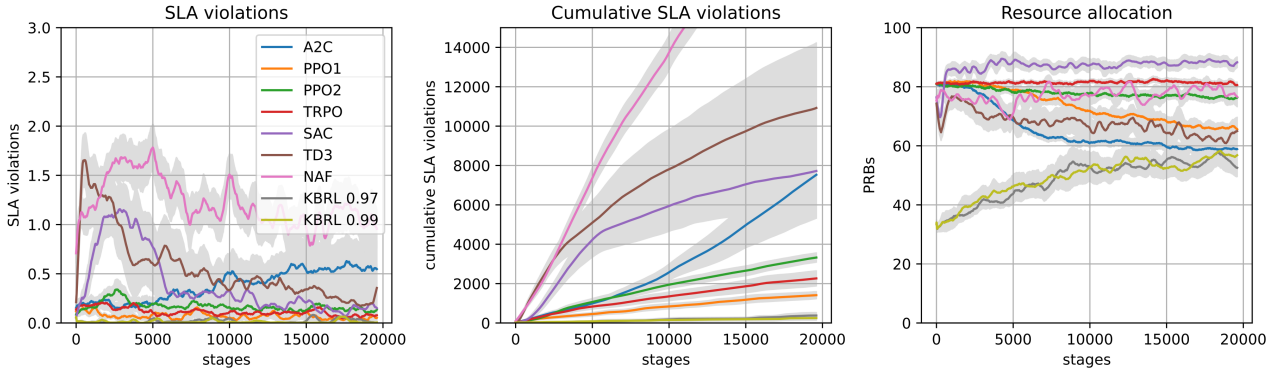


Fig. 5. Results for scenario 3. From left to right: SLA violations per stage, cumulative SLA violations, and total RBs allocated. Each plot shows the per-stage average of 30 simulation runs and its confidence interval with a 90% confidence degree.

summarized in this figure, NAF and DQN are already in the inference phase (i.e. they have been trained previously for 20000 steps), while KBRL is in learning phase. We see that the oracle policy perfectly fulfills the SLAs using roughly 50% of the resources used by the RL agents. This illustrates the inherent difficulty of the problem due to its stochastic nature. We also observe that, although the baselines improve their performance in the inference phase, KBRL still outperforms them in SLA fulfillment even without previous training.

### C. Computational Overhead

A usual concern regarding the deployment of reinforcement learning algorithms is the computational overhead that these algorithms may introduce in the system. This is especially relevant in our online learning setting, in which RL agents need to learn from scratch on the operating network, updating their policies/models between decision stages. In this subsection we show an empirical evaluation of the per-stage computation time of our proposal and the RL baselines, both in the learning phase and in the inference phase. For each scenario and each phase, we measured the execution times of the algorithms in

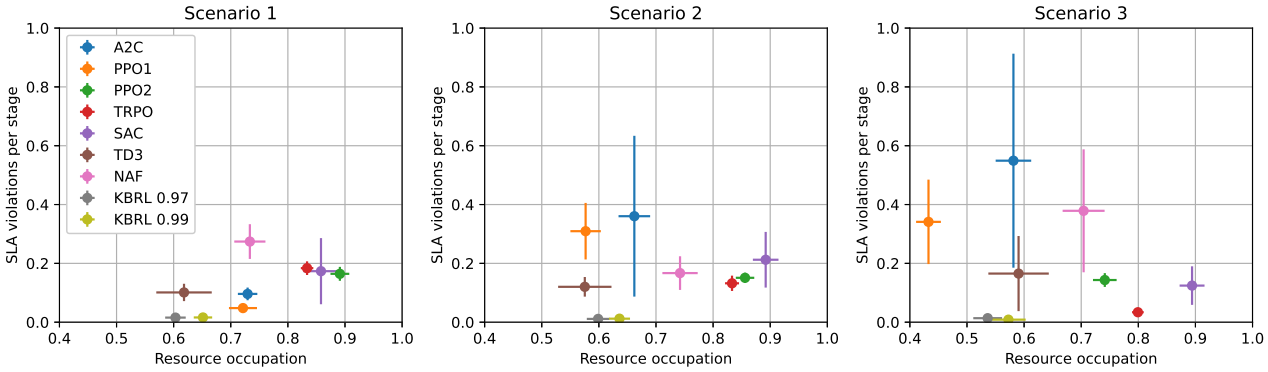


Fig. 6. Performance during the inference phase. Confidence intervals are depicted for each metric with a 95% confidence degree.

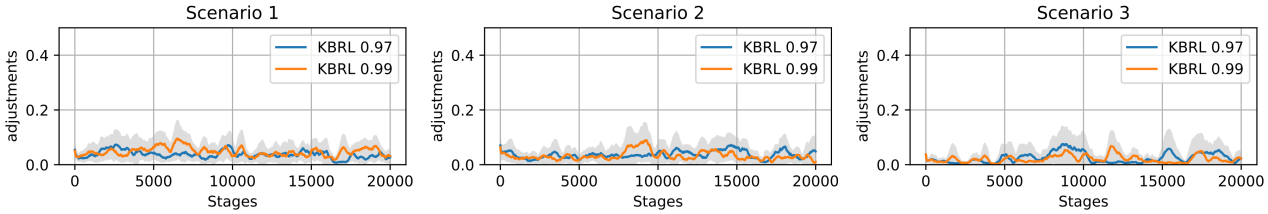


Fig. 7. Average number of adjustments per stage in each scenario.

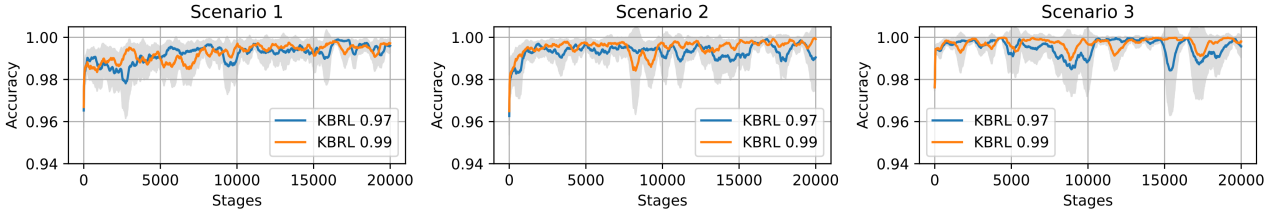


Fig. 8. Empirical prediction accuracy of the KBRL classifiers.

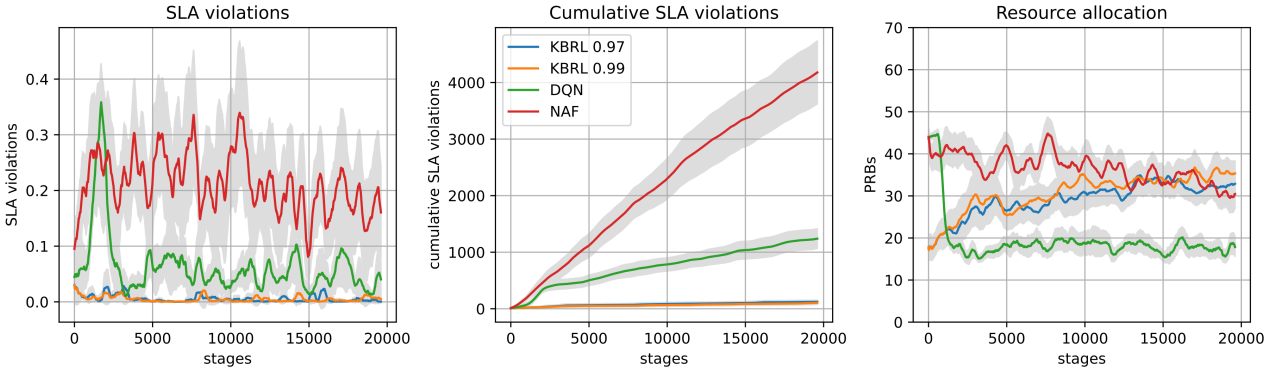


Fig. 9. Results for scenario 4 in the learning phase.

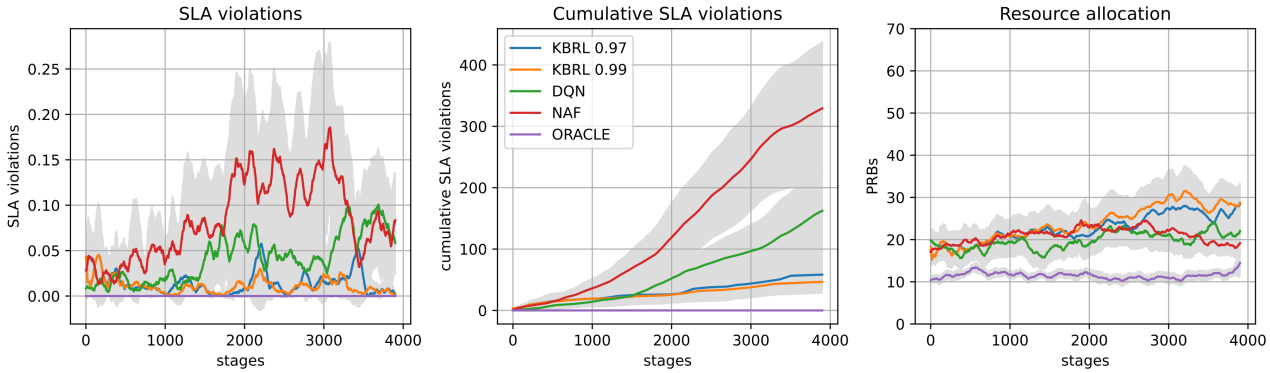


Fig. 10. Comparison with the estimated optima solution in scenario 4. DQN and NAF operate in the inference phase.

30 learning episodes. The experiments were conducted on an Intel Xeon E5-2650V3 CPU.

Figure 11 shows the average execution time per stage consumed by each algorithm on each scenario during the learning phase. As expected, when scenarios are associated to observations of larger dimension, the execution time is longer. This effect is particularly noticeable in the baseline algorithms, where the execution time doubles from scenario 3 to 2, and from 2 to 1. It is also evident that our proposal introduces much less computational overhead than the baselines during the learning phase. It should be noted that these algorithms, including our proposal, are implemented for experimentation purposes, and are not optimized for production in terms of execution time. What these results show is that computational overhead is not a major obstacle for the deployment of our algorithm, and even less so if a production-optimized implementation is used.

Figure 12 shows the execution time per decision stage during the inference phase. As expected, the baselines notably reduce their computation time in this phase, since the actions are simply obtained by forward propagation on the policy's neural network. The time required by KBRL is in the same order of magnitude of the baselines. The outlier results of NAF are probably due to implementation differences (recall that NAF uses the Keras-RL implementation instead of Stable Baselines).

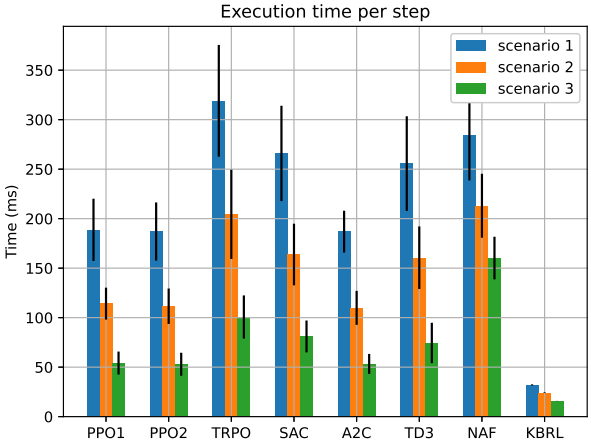


Fig. 11. Execution time per decision stage during the learning phase. Confidence intervals are depicted on the top of each bar for a 90% confidence degree.

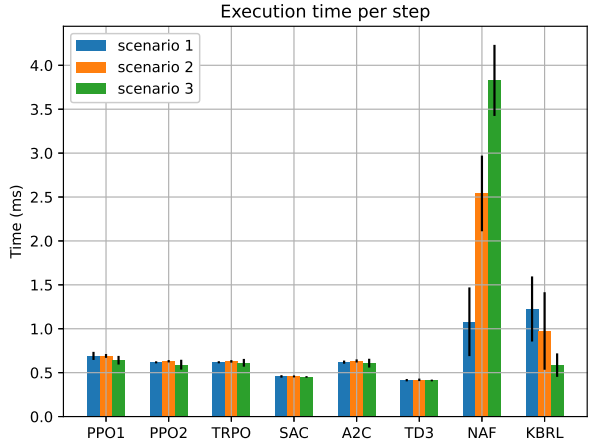


Fig. 12. Execution time per decision stage during the inference phase. Confidence intervals are depicted on the top of each bar for a 90% confidence degree.

## VIII. Conclusions

This work has shown that a model-based RL approach can efficiently manage the allocation of RAN resources among network slices, and is especially well suited for online operation. Our proposal, KBRL, combines a one-step lookahead model predictive control with a model that comprises two elements, a classifier and an accuracy estimator for the classifier, both of which are learned from scratch while the network is in operation. This structure for an MBRL agent is novel and presents several advantages: i) it benefits from the high sample efficiency of existing online learning algorithms. In our case, we use a kernel-based algorithm known as Projectron. ii) It enables a sample-augmentation strategy that further enhances the sample efficiency of the learning process, and iii) it manages all the system objectives in parallel (resource efficiency and SLA fulfillment for each slice). These advantages largely outweigh the potential sub-optimality associated with

the use of a one-step horizon by the control agent, as shown by our numerical results. In our experiments, we compared KBRL with state-of-the-art RL algorithms, all of which are farsighted, in different scenarios. KBRL outperformed all the baselines in terms of resource efficiency, SLA fulfillment, and computational overhead, during online learning episodes.

## REFERENCES


[1] I. Afolabi, T. Taleb *et al.*, "Network slicing and softwarization: A survey on principles, enabling technologies, and solutions," *IEEE Communications Surveys & Tutorials*, vol. 20, pp. 2429–2453, Mar. 2018.

[2] R. Ferrus, O. Sallent, J. Perez-Romero, and R. Agusti, "On 5G radio access network slicing: Radio interface protocol features and configuration," *IEEE Communications Magazine*, vol. 56, pp. 184–192, May 2018.

[3] N. C. Luong, D. T. Hoang *et al.*, "Applications of deep reinforcement learning in communications and networking: A survey," *IEEE Communications Surveys & Tutorials*, vol. 21, pp. 3133–3174, May 2019.

[4] Z. Xiong, Y. Zhang *et al.*, "Deep reinforcement learning for mobile 5G and beyond: Fundamentals, applications, and challenges," *IEEE Vehicular Technology Magazine*, vol. 14, pp. 44–52, Jun. 2019.

[5] D. M. Gutierrez-Estevez, M. Gramaglia *et al.*, "Artificial Intelligence for Elastic Management and Orchestration of 5G Networks," *IEEE Wireless Communications*, vol. 26, pp. 134–141, Oct. 2019.

[6] F. Orabona, J. Keshet, and B. Caputo, "Bounded kernel-based online learning," *Journal of Machine Learning Research*, vol. 10, pp. 2643–2666, Nov. 2009.

[7] X. Foukas, G. Patounas, A. Elmokashfi, and M. K. Marina, "Network slicing in 5G: Survey and challenges," *IEEE Communications Magazine*, vol. 55, pp. 94–100, May 2017.

[8] R. Su, D. Zhang *et al.*, "Resource allocation for network slicing in 5G telecommunication networks: A survey of principles and models," *IEEE Network*, vol. 33, pp. 172–179, Nov. 2019.

[9] M. R. Raza, C. Natalino *et al.*, "Reinforcement Learning for Slicing in a 5G Flexible RAN," *IEEE/OSA Journal of Lightwave Technology*, vol. 37, pp. 5161–5169, Oct. 2019.

[10] J. Prados-Garzon, A. Laghrissi *et al.*, "A complete LTE mathematical framework for the network slice planning of the EPC," *IEEE Transactions on Mobile Computing*, vol. 19, pp. 1–14, Jan. 2019.

[11] J. S. P. Roig, D. M. Gutierrez-Estevez, and D. Gündüz, "Management and orchestration of virtual network functions via deep reinforcement learning," *IEEE Journal on Selected Areas in Communications*, vol. 38, pp. 304–317, Feb. 2019.

[12] H. Zhang, N. Liu *et al.*, "Network slicing based 5G and future mobile networks: mobility, resource management, and challenges," *IEEE Communications Magazine*, vol. 55, pp. 138–145, Aug. 2017.

[13] C. Qi, Y. Hua *et al.*, "Deep reinforcement learning with discrete normalized advantage functions for resource management in network slicing," *IEEE Communications Letters*, vol. 23, pp. 1337–1341, Aug. 2019.

[14] Y. L. Lee, J. Loo, T. C. Chuah, and L.-C. Wang, "Dynamic network slicing for multitenant heterogeneous cloud radio access networks," *IEEE Transactions on Wireless Communications*, vol. 17, pp. 2146–2161, Apr. 2018.

[15] N. Van Huynh, D. T. Hoang, D. N. Nguyen, and E. Dutkiewicz, "Optimal and fast real-time resource slicing with deep dueling neural networks," *IEEE Journal on Selected Areas in Communications*, vol. 37, pp. 1455–1470, Jun. 2019.

[16] V. Sciancalepore, X. Costa-Perez, and A. Banchs, "RL-NSB: Reinforcement Learning-Based 5G Network Slice Broker," *IEEE/ACM Transactions on Networking*, vol. 27, pp. 1543–1557, Aug. 2019.

[17] B. Han, V. Sciancalepore *et al.*, "Multiservice-based network slicing orchestration with impatient tenants," *IEEE Transactions on Wireless Communications*, vol. 19, pp. 5010–5024, Jul. 2020.

[18] R. Li, Z. Zhao *et al.*, "Deep reinforcement learning for resource management in network slicing," *IEEE Access*, vol. 6, pp. 74 429–74 441, Nov. 2018.

[19] H. D. R. Albonda and J. Pérez-Romero, "An efficient RAN slicing strategy for a heterogeneous network with eMBB and V2X services," *IEEE Access*, vol. 7, pp. 44 771–44 782, Mar. 2019.

[20] V. N. Ha and L. B. Le, "End-to-end network slicing in virtualized OFDMA-based cloud radio access networks," *IEEE Access*, vol. 5, pp. 18 675–18 691, Sep. 2017.

[21] A. D. Shoaei, M. Derakhshani, and T. Le-Ngoc, "Reconfigurable and Traffic-Aware MAC Design for Virtualized Wireless Networks via Reinforcement Learning," *IEEE Transactions on Communications*, vol. 67, pp. 5490–5505, Aug. 2019.

[22] C.-Y. Chang and N. Nikaein, "RAN runtime slicing system for flexible and dynamic service execution environment," *IEEE Access*, vol. 6, pp. 34 018–34 042, Jun. 2018.

[23] F. Wei, G. Feng *et al.*, "Network slice reconfiguration by exploiting deep reinforcement learning with large action space," *IEEE Transactions on Network and Service Management*, vol. 17, pp. 2197–2211, Dec. 2020.

[24] H. Wang, Y. Wu *et al.*, "Data-driven dynamic resource scheduling for network slicing: A deep reinforcement learning approach," *Information Sciences*, vol. 498, pp. 106–116, 2019.

[25] T. Li, X. Zhu, and X. Liu, "An End-to-End network slicing algorithm based on deep Q-Learning for 5G network," *IEEE Access*, vol. 8, pp. 122 229–122 240, Jul. 2020.

[26] J. A. Ayala-Romero, A. Garcia-Saavedra *et al.*, "vrAIn: Deep Learning based Orchestration for Computing and Radio Resources in vRANs," *IEEE Transactions on Mobile Computing, Early Access, doi: 10.1109/TMC.2020.3043100*, 2020.

[27] Y. Kim and H. Lim, "Multi-agent reinforcement learning-based resource management for end-to-end network slicing," *IEEE Access*, vol. 9, pp. 56 178–56 190, Apr. 2021.

[28] X. Chen, Z. Zhao *et al.*, "Multi-tenant cross-slice resource orchestration: A deep reinforcement learning approach," *IEEE Journal on Selected Areas in Communications*, vol. 37, pp. 2377–2392, Oct. 2019.

[29] J. Mei, X. Wang *et al.*, "Intelligent radio access network slicing for service provisioning in 6g: A hierarchical deep reinforcement learning approach," *IEEE Transactions on Communications*, vol. 69, pp. 6063–6078, Sept. 2021.

[30] Y. Xu, Z. Zhao *et al.*, "Constrained reinforcement learning for resource allocation in network slicing," *IEEE Communications Letters*, vol. 25, pp. 1554–1558, May 2021.

[31] Y. Abiko, T. Saito *et al.*, "Flexible resource block allocation to multiple slices for radio access network slicing using deep reinforcement learning," *IEEE Access*, vol. 8, pp. 68 183–68 198, 2020.

[32] Y. Hua, R. Li *et al.*, "Gan-powered deep distributional reinforcement learning for resource management in network slicing," *IEEE Journal on Selected Areas in Communications*, vol. 38, pp. 334–349, Feb. 2020.

[33] J. A. Ayala-Romero, J. J. Alcaraz, J. Vales-Alonso, and E. Egea-Lopez, "Online optimization of interference coordination parameters in small cell networks," *IEEE Transactions on Wireless Communications*, vol. 16, pp. 6635–6647, Oct. 2017.

[34] J. J. Alcaraz, J. A. Ayala-Romero, J. Vales-Alonso, and F. Losilla-López, "Online reinforcement learning for adaptive interference coordination," *Transactions on Emerging Telecommunications Technologies*, vol. 31, pp. 1–24, Aug. 2020.

[35] J. A. Ayala-Romero, J. J. Alcaraz, A. Zanella, and M. Zorzi, "Online Learning for Energy Saving and Interference Coordination in HetNets," *IEEE Journal on Selected Areas in Communications*, vol. 37, pp. 1374–1388, Jun. 2019.

[36] K. Chua, R. Calandra, R. McAllister, and S. Levine, "Deep reinforcement learning in a handful of trials using probabilistic dynamics models," in *Proceedings of the 32nd International Conference on Neural Information Processing Systems, NeurIPS*, Dec. 2018, pp. 4759–4770.

[37] L. Kaiser, M. Babaeizadeh *et al.*, "Model-based reinforcement learning for Atari," *arXiv preprint arXiv:1903.00374*, Feb. 2020.

[38] S. Ahmadi, *5G NR: Architecture, Technology, Implementation, and Operation of 3GPP New Radio Standards*. Academic Press, 2019.

[39] E. Altman, *Constrained Markov decision processes*. CRC Press, 1999, vol. 7.

[40] D. P. Bertsekas, *Reinforcement learning and optimal control*. Athena Scientific Belmont, MA, 2019.

[41] Y. Freund and R. E. Schapire, "Large margin classification using the perceptron algorithm," *Machine Learning*, vol. 37, pp. 277–296, Dec. 1999.

[42] T. Hofmann, B. Schölkopf, and A. J. Smola, "Kernel methods in machine learning," *The annals of statistics*, vol. 36, pp. 1171–1220, Jun. 2008.

[43] K. Crammer and Y. Singer, "Ultraconservative online algorithms for multiclass problems," *Journal of Machine Learning Research*, vol. 3, pp. 951–991, Jan. 2003.

[44] 3GPP, "Evolved Universal Terrestrial Radio Access (E-UTRA); Radio Frequency (RF) system scenarios," 3rd Generation Partnership Project (3GPP), Technical Report (TR) 36.942, 06 2018, version 15.0.0.

[45] "NS-3 simulator, LTE module," https://www.nsnam.org/docs/models/html/lte.html, accessed: 2022-05-20.

[46] 3GPP, "E-UTRA Base Station (BS) radio transmission and reception," 3rd Generation Partnership Project (3GPP), Technical Specification (TS) 36.104, 04 2022, version 15.15.0.

[47] J. A. Ayala-Romero, A. Garcia-Saavedra *et al.*, "vrAIn: A deep learning approach tailoring computing and radio resources in virtualized RANs," in *The 25th Annual International Conference on Mobile Computing and Networking*, Oct. 2019, pp. 1–16.

[48] V. Mnih, K. Kavukcuoglu *et al.*, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[49] J. Schulman, S. Levine *et al.*, "Trust region policy optimization," in *International Conference on Machine Learning*. PMLR, Jul. 2015, pp. 1889–1897.

[50] J. Schulman, F. Wolski *et al.*, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, Aug. 2017.

[51] A. Hill, A. Raffin *et al.*, "Stable baselines," https://github.com/hill-a/stable-baselines, 2018.

[52] S. Fujimoto, H. Hoof, and D. Meger, "Addressing function approximation error in actor-critic methods," in *International Conference on Machine Learning*. PMLR, Jul. 2018, pp. 1587–1596.

[53] T. P. Lillicrap, J. J. Hunt *et al.*, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, Jul. 2019.

[54] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," in *International Conference on Machine Learning*. PMLR, Jul. 2018, pp. 1861–1870.

[55] V. Mnih, A. P. Badia *et al.*, "Asynchronous methods for deep reinforcement learning," in *International Conference on Machine Learning*. PMLR, Jun. 2016, pp. 1928–1937.

[56] S. Gu, T. Lillicrap, I. Sutskever, and S. Levine, "Continuous deep q-learning with model-based acceleration," in *International conference on machine learning*. PMLR, 2016, pp. 2829–2838.

[57] P. Dhariwal, C. Hesse *et al.*, "OpenAI Baselines," https://github.com/openai/baselines, 2017.

[58] M. Plappert, "keras-rl," https://github.com/keras-rl/keras-rl, 2016.