vrAIn: A Deep Learning Approach Tailoring Computing and Radio Resources in Virtualized RANs

Jose A. Ayala-Romero
NEC Laboratories Europe &
Technical University of Cartagena

Andres Garcia-Saavedra*
andres.garcia.saavedra@neclab.eu
NEC Laboratories Europe

Marco Gramaglia
Universidad Carlos III de Madrid

Xavier Costa-Perez
NEC Laboratories Europe

Albert Banchs
Universidad Carlos III de Madrid &
IMDEA Networks Institute

Juan J. Alcaraz
Technical University of Cartagena

ABSTRACT

The virtualization of radio access networks (vRAN) is the last milestone in the NFV revolution. However, the complex dependencies between computing and radio resources make vRAN resource control particularly daunting. We present `vrAIn`, a dynamic resource controller for vRANs based on deep reinforcement learning. First, we use an autoencoder to project high-dimensional context data (traffic and signal quality patterns) into a latent representation. Then, we use a deep deterministic policy gradient (DDPG) algorithm based on an actor-critic neural network structure and a classifier to map (encoded) contexts into resource control decisions.

We have implemented `vrAIn` using an open-source LTE stack over different platforms. Our results show that `vrAIn` successfully derives appropriate compute and radio control actions irrespective of the platform and context: (*i*) it provides savings in computational capacity of up to 30% over CPU-unaware methods; (*ii*) it improves the probability of meeting QoS targets by 25% over static allocation policies using similar CPU resources in average; (*iii*) upon CPU capacity shortage, it improves throughput performance by 25% over state-of-the-art schemes; and (*iv*) it performs close to optimal policies resulting from an offline oracle. To the best of our knowledge, this is the first work that thoroughly studies the computational behavior of vRANs, and the first approach to a model-free solution that does not need to assume any particular vRAN platform or system conditions.

*Contact author email: andres.garcia.saavedra@neclab.eu

MobiCom '19, October 21–25, 2019, Los Cabos, Mexico

ACM ISBN 978-1-4503-6169-9/19/10…$15.00
https://doi.org/10.1145/3300061.3345431

CCS CONCEPTS

• **Networks** → *Network algorithms*; **Mobile networks**; • **Computing methodologies** → *Machine learning*.

KEYWORDS

RAN virtualization; resource management; machine learning

ACM Reference Format:
Jose A. Ayala-Romero, Andres Garcia-Saavedra, Marco Gramaglia, Xavier Costa-Perez, Albert Banchs, and Juan J. Alcaraz. 2019. vrAIn: A Deep Learning Approach Tailoring Computing and Radio Resources in Virtualized RANs. In *The 25th Annual International Conference on Mobile Computing and Networking (MobiCom '19), October 21–25, 2019, Los Cabos, Mexico.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3300061.3345431

1 INTRODUCTION

Radio Access Network virtualization (vRAN) is well-recognized as a key technology to accommodate the ever-increasing demand for mobile services at an affordable cost for mobile operators [4]. vRAN centralizes *softwarized* radio access point (RAP)[1] stacks into computing infrastructure in a cloud location—typically at the edge, where CPU resources may be scarce. Fig. 1 illustrates this with a set of vRAPs sharing a common pool of CPUs to perform radio processing tasks such as signal modulation and encoding (red arrows). This provides several advantages, such as resource pooling (via centralization), simpler update roll-ups (via softwarization) and cheaper management and control (via commoditization), leading to savings of 10-15% in capital expenditure per square kilometer and 22% in CPU usage [2, 34].

It is thus not surprising that vRAN has attracted the attention of academia *and* industry. OpenRAN[2], O-RAN[3] or Rakuten's vRAN[4]—led by key operators (such as AT&T, Verizon or China Mobile), manufacturers (such as Intel, Cisco or

[1]The literature uses different names to refer to different radio stacks, such as base station (BS), eNodeB (eNB), new radio (NR) gNodeB (gNB), access point (AP), etc. We will use RAP consistently to generalize the concept.
[2]https://telecominfraproject.com/openran/
[3]https://www.o-ran.org/
[4]https://global.rakuten.com/corp/news/press/2019/0605_01.html

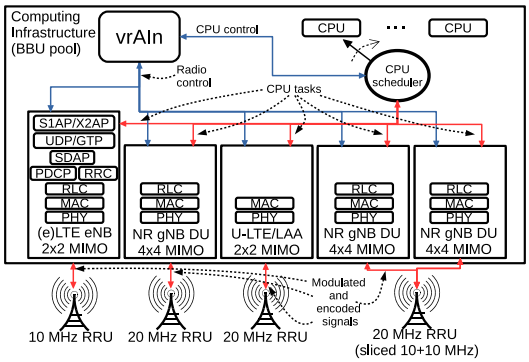

Figure 1: `vrAIn`: A vRAN resource controller

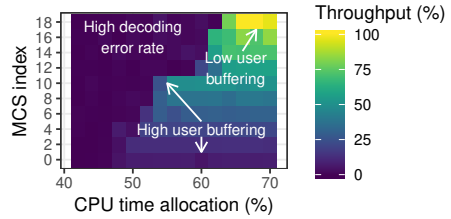

Figure 2: A SISO 10-MHz LTE vRAP with maximum uplink traffic load and high SNR. High CPU and MCS allocations yield low data buffering (100% throughput). Low MCS allocation causes high user data buffering (<100% throughput). Low CPU time allocation renders high decoding error rate (≪100% throughput).

NEC) and research leaders (such as Standford University)—are examples of publicly disseminated initiatives towards fully programmable, virtualized and open RAN solutions based on general-purpose processing platforms and decoupled base band units (BBUs) and remote radio units (RRUs).

Despite the above, the gains attainable today by vRAN are far from optimal, and this hinders its deployment at scale. In particular, computing resources are inefficiently pooled since most implementations over-dimension computational capacity to cope with peak demands in real-time workloads [1, 28]. Conversely, *substantial cost savings can be expected by dynamically adapting the allocation of resources to the temporal variations of the demand across vRAPs* [2, 10]. There is nonetheless limited hands-on understanding on the computational behavior of vRAPs and the relationship between radio *and* computing resource dynamics. ***Such an understanding is required to design a practical vRAN resource management system—indeed the goal of this paper.***

Towards a cost-efficient resource pooling. Dynamic resource allocation in vRAN is an inherently hard problem:

(*i*) The computational behavior of vRAPs depends on many factors, including the radio channel conditions or users' load demand, that may not be controllable. More specifically, there is a strong dependency with the *context* (such as data bit-rate load and signal-to-noise-ratio (SNR) patterns), the RAP configuration (e.g., bandwidth, MIMO setting, etc.) and on the infrastructure pooling computing resources;

(*ii*) Upon shortage of computing capacity (e.g., with nodes temporarily overloaded due to orchestration decisions) CPU control decisions *and* radio control decisions (such as scheduling and modulation and coding scheme (MCS) selection) are coupled; certainly, it is well known that scheduling users with higher MCS incur in higher instantaneous computational load [1].

Let us introduce up front some toy experiments to illustrate this. Note that we deliberately omit the details of our experimental platform (properly introduced in §4) to keep our motivation simple. We set up an off-the-shelf LTE user equipment (UE) and a vRAN system comprising `srsLTE`, an open-source LTE stack, over an i7-5600U CPU core @ 2.60GHz as BBU and a software-defined radio (SDR) USRP as RRU radio front-end. We let the UE transmit uplink UDP data at maximum nominal load with high SNR channel conditions and show in Fig. 2 the ratio of bits successfully decoded (throughput) when selecting different MCS indexes (y axis) and relative CPU time shares (x axis). The results yield an intuitive observation: higher modulation levels achieve higher performance, which in turn require larger allocations of computing resources. ***This dependency motivates us to (i) devise novel algorithms to adjust the allocation of computing resources to the needs of a vRAN; and (ii) upon shortage of computing resources, explore strategies that make compute/radio control decisions jointly.***

Model-free learning. The aforementioned issues have been identified in some related research [1, 29, 30] (a proper literature review is presented in §6). Nevertheless, these works rely on models that need pre-calibration for specific scenarios and they do not consider the effect that different bit-rate patterns and load regimes have on computing resource utilization. *In reality, however, the relationship that system performance has with compute and radio scheduling policies is far from trivial and highly depends on the context (data arrival patterns, SNR patterns)* and *on the software implementation and hardware platform hosting the pool of BBUs.*

To emphasize the above point, we repeat the previous experiment for different SNR regimes (high, medium and low) and different mean bit-rate load regimes (10%, 30%, 50% and 70% of the maximum nominal capacity) for two different compute cores, the i7-5600U CPU core @ 2.60GHz used before and an i7-8650U CPU core @ 1.90GHz, and show in Fig. 3 (maximum load, variable SNR) and Fig. 4 (high SNR, variable load) the *relative throughput* with respect to the load demand (where 100% denotes that all the demand is served). The results make it evident that the system behavior shown in Fig. 2 substantially varies with the context (SNR, load) and the platform pooling computing resources, ***More importantly, the underlying model capturing this behavior is highly non-linear and far from trivial.***

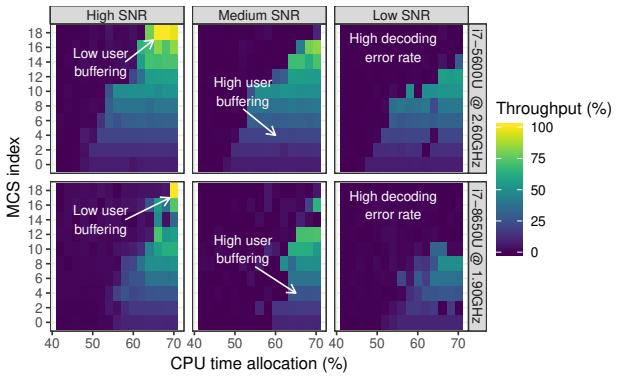

Figure 3: vRAP with maximum uplink traffic load. Different computing platforms and SNR conditions yield different performance models.

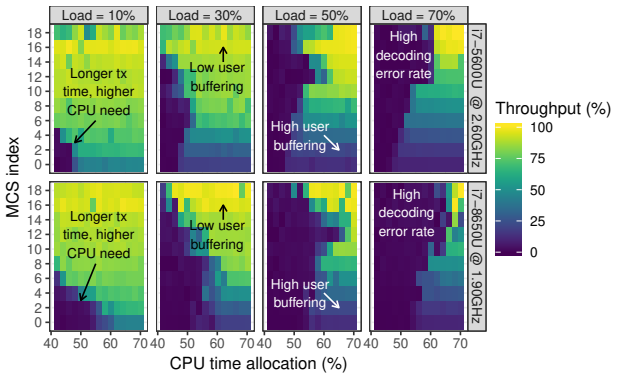

Figure 4: vRAP with high SNR. Performance model is highly complex and non-linear. Light/dark areas (good/bad performance) follow irregular patterns.

All the above render tractable models in the literature (e.g., [1, 29, 30]) inefficient for practical resource control. Indeed, mechanisms based on such models are not able to accurately capture the complex behavior evidenced by our early experiments and hence perform poorly. We demonstrate this empirically in our performance evaluation in §5.

In contrast, we resort to model-free reinforcement learning methods that adapt to the actual contexts and platforms. We present `vrAIn`, an artificial intelligence-powered (AI) vRAN controller that governs the allocation of computing and radio resources (blue arrows in Fig. 1). The main *novel* contributions of this paper are as follows:

- We design a *deep autoencoder* that captures context information about vRAP load, signal quality and UE diversity time dynamics in a low-dimensional representation;
- We cast our resource control problem as a *contextual bandit* problem and solve it with a novel approach: (*i*) we decouple radio and computing control decisions to efficiently manage the multi-dimensional action space; and (*ii*) we design a deep deterministic policy gradient (DDPG) algorithm for our contextual bandit setting to handle the real-valued nature of the control actions in our system;
- We implement a *proof-of-concept* of `vrAIn` using SDR boards attached to commodity computing nodes hosting software-based LTE eNB stacks, and assess its performance in a variety of scenarios and against different benchmarks.

To the best of our knowledge, this is the first paper in the literature that thoroughly explores empirically the computational behavior of a vRAN by means of an experimental setup. *Our results do not only shed light on the computational behavior of this technology across different contexts (radio and data traffic patterns), but also show that substantial gains can be achieved by developing autonomous learning algorithms that adapt to the actual platform and radio channel.*

In the sequel, §2 provides background information; §3 introduces the `vrAIn` design; and §4 and §5 show our experimental proof-of-concept and its performance, respectively. Finally, §6 revises related work and §7 concludes the paper.

2 BACKGROUND

Prior to presenting the design of `vrAIn` (see §3), we introduce relevant information and notation used in the paper.

2.1 Radio Access Point

A Radio Access Point (RAP) implements the necessary processing stack to transfer data to/from UEs. These stacks may be heterogeneous, e.g., (from left to right in Fig. 1) 4G LTE, 5G NR, unlicensed LTE, RAPs sharing a radio front-end (via *network slicing* [22, 23, 31]), and/or implement different functional splits [7–9], but they all share common fundamentals, such as OFDMA modulation schema and channel coding techniques at the physical layer (PHY) that make `vrAIn` general across these vRAPs. Despite this heterogeneity, RAPs are typically dissected into three layers (L1, L2, L3).

L1 (PHY). We focus on sub-6GHz; specifically, on the uplink of 4G LTE and 5G NR since it is the more complex case as we have to rely on periodic feedback from users (while our implementation focuses on uplink, our design applies to both uplink and downlink; the extension to downlink is straightforward as user buffers are local). L1 is implemented through a set of OFDMA-modulated channels, using a Resource Block (RB) filling across ten 1-ms subframes forming a frame. The channels used for data heavy lifting are the Physical Uplink Shared Channel (PUSCH) and the Physical Downlink Shared Channel (PDSCH); usually modulated with QAM constellations of different orders (up to 256 in 5G) and MIMO settings, and encoded with a turbo decoder (4G) or LDCP code (5G). There are some differences between 4G and 5G PHYs, such as 5G's scalable numerology, but these are not relevant to `vrAIn`, which simply *learns* their computational behavior in a model-free manner. In brief, RBs assigned to UEs by the MAC layer are modulated and encoded with a MCS that depends on the user's Channel Quality Indicator (CQI), a measure of SNR that is locally available in the uplink and is reported periodically by UEs in the downlink. The scheme reported in [16] to map CQI values into MCSs is the most common approach and is blind to CPU availability.

L2 (MAC, RLC, PDCP). The MAC sublayer is responsible for (de)multiplexing data from/to different radio bearers to/from PHY transport blocks (TBs) and perform error correction through hybrid ARQ (HARQ). In the uplink, demultiplexing is carried out by the MAC scheduler by assigning RBs to UEs at every transmission time interval (TTI, usually equal to 1ms). Once this is decided, the RAP feeds the scheduling information to the UEs through a scheduling grant. 3GPP leaves the scheduler design open for vendor implementation. Moreover, the MAC layer also provides a common reference point towards different PHY carriers when using carrier aggregation. The higher sublayers (RLC, PDCP) carry out tasks such as data reordering, segmentation, error correction and cyphering; and provide a common reference point towards different PHY/MAC instances (e.g., from different vRAPs). Another L2 aspect relevant for the design of `vrAIn` are the Buffer State Reports (BSRs), which provide feedback to the RAPs about the amount of data each UE has pending to transmit. This information will be used by `vrAIn` to design a system state signal used for feedback on resource allocation decisions.

L3 (RRC, GTP). The Radio Resource Control (RRC) and GTP-U sublayers manage access information, QoS reporting and tunneling data between RAPs and the mobile core.

Notably, PHY (de)modulation/(de)coding operations consume most of the CPU cycles of the stack [46], which explains the dependency between CPU and MCS shown in §1. PDCP's (de)ciphering tasks consume most of the CPU cycles in L2 [36], albeit L2 is substantially less compute demanding than L1 [46] and, furthermore, PDCP will be decoupled from the distributed unit (DU) in 5G (see NR gNB in Fig. 1).

2.2 Notation

We let $\mathbb{R}$ and $\mathbb{Z}$ denote the set of real and integer numbers, and $\mathbb{R}_+$ and $\mathbb{R}^n$ represent the sets of non-negative real numbers and n-dimensional real vectors, respectively. Vectors are in column form and written in bold font. Subscripts represent an element in a vector and superscripts elements in a sequence. For instance, $\langle \mathbf{x}^{(t)} \rangle$ is a sequence of vectors with $\mathbf{x}^{(t)} = (x_1^{(t)}, \ldots, x_n^{(t)})^T$ being a vector from $\mathbb{R}^n$ and $x_i^{(t)}$ being the i'th component of the t'th vector in the sequence.

3 VRAIN DESIGN

In the sequel we present the design of `vrAIn`, schematically depicted in Fig. 5. As shown by the figure, `vrAIn` is divided into two blocks operating at different timescales:

- In the first block, **CPU schedulers** (which assign tasks to CPUs, e.g., subframes for decoding) and **radio schedulers** (which assign radio resources to UEs, e.g., selecting MCSs and allocating RBs) operate at sub-millisecond timescales. `vrAIn` relies on simple computing and radio policies, which we introduce in §3.1, to influence their behavior.

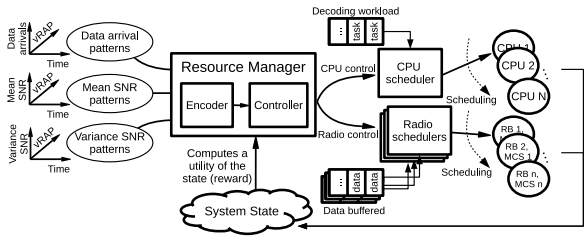

Figure 5: `vrAIn` system design.

- The second block is the **resource manager**, the main contribution of this paper, a sequential decision-making entity that configures the above schedulers using respective compute and radio policies over larger timescales (seconds).

Such an approach based on multiple timescales, with a resource manager or orchestrator governing the operation of low-level agents, is common when dealing with cloud-based radio solutions (see, e.g., [27]) and enables the implementation of smart resource control policies in a simple manner.

To overcome the issues mentioned in §1, we design a feedback control loop in the resource manager where:

(*i*) *Contextual* information (SNR and data load patterns) is collected and encoded;
(*ii*) A learned *policy* maps contexts into computing and radio control decisions; and
(*iii*) A *reward* signal assesses the decisions taken and fine-tunes the policy accordingly.

This falls naturally into the realm of **reinforcement learning** (RL) [35], an area of machine learning applied in human-level control (mastering games such as Go [32] or StarCraft II [41]), health-care [25] or finances [6]. Full-blown RL problems are usually modeled with Markov decision processes and use some model-free policy learning method (e.g., Q-learning) to estimate an action-value function [44]. However, *the impact that instantaneous actions have on future contexts, which RL usually captures with the recursive Bellman equation, is very limited in our case because of the different timescales between the schedulers and the resource manager.* Thus, we can resort to a **contextual bandit** (CB) model, a type of RL applied in health [38], advertisement [37] or robot [18] control systems that can learn context-action mapping policies in a much simpler setup (without recursive action-value functions). We still face several challenges, formally addressed in §3.2, to solve this problem effectively; among others, we have continuous and high-dimensional context/action spaces.

3.1 CPU and radio schedulers

CPU scheduling implies assigning tasks such as subframes to decode to an available CPU. In turn, radio resource scheduling involves deciding upon the number of RBs assigned to UEs, their location in frequency and time, their MCS and their transmission power. A plethora of computing and radio scheduling mechanisms [3, 40] have been proposed.

When orchestrating CPU and radio resources, our goal is both to provide *good performance*—minimizing data delivery delay—and make an *efficient resource usage*—minimizing CPU usage while avoiding decoding errors due to a deficit of computing capacity. To achieve these goals, when there is sufficient computing capacity, we can decode all frames with the maximum MCS allowed by the SNR conditions while provisioning the necessary CPU resources to this end. However, whenever there is deficit of computing capacity, *we need to constraint the set of selected MCSs*, as otherwise we would incur into decoding errors that would harm the resulting efficiency. In this case, our approach is to limit the maximum eligible MCSs within each RAP when required, which has several advantages: (*i*) it is simple, as we only need to determine a single MCS bound for each RAP; and (*ii*) it provides fairness across UEs, reducing the performance of the UEs that are better off and preserving the less favorable ones. Thus, to implement the control of CPU and radio resources, `vrAIn` relies on the following control actions at each vRAP i:

- A *maximum fraction of time* $c_i \in C := [0, 1] \subset \mathbb{R}$ allotted to a CPU (our **computing control decisions**); and
- A *maximum eligible MCS* $m_i \in \mathcal{M}$, where $\mathcal{M}$ is a discrete set of MCSs (our **radio control decisions**).

These control settings are configured by the resource manager and can be easily implemented in any scheduler. These are upper bounds, CPU/radio schedulers still have the freedom to optimize the use of resources within these bounds.

Our job is hence to design a resource manager that *learns* the behavior of *any* radio/CPU scheduler and maximize performance using such interfaces, as we introduce next.

3.2 Resource manager

We hence formulate our resource control problem as a contextual bandit (CB) problem, a sequential decision-making problem where, at every time stage $n \in \mathbb{N}$, an agent observes a *context* or *feature* vector drawn from an arbitrary feature space $\mathbf{x}^{(n)} \in \mathcal{X}$, chooses an action $\mathbf{a}^{(n)} \in \mathcal{A}$ and receives a reward signal $r(\mathbf{x}^{(n)}, \mathbf{a}^{(n)})$ as feedback. The context $\mathbf{x}$ need not be stationary, as network conditions may change over time, and the sequence of context arrivals $\langle \mathbf{x}^{(n)} \rangle_{n \in \mathbb{N}}$ and the distribution E over context-reward pairs $(\mathbf{x}, r)$ are fixed and unknown *a priori*. Furthermore, we let $\pi(\mathbf{x}) : \mathcal{X} \to \mathcal{A}$ denote a deterministic policy that maps contexts into actions, and

$$R_\pi := \mathbb{E}_{(\mathbf{x},\mathbf{r})\sim E}\Big[r\left(\mathbf{x}, \pi(\mathbf{x})\right) \Big] \tag{1}$$

denote the expected instantaneous reward of a policy π. The goal is hence to learn an optimal policy $\pi^* := \arg\max_{\pi \in \Pi} R_\pi$ that maximizes instantaneous rewards subject to $\sum_{i \in \mathcal{P}} c_i \leq 1$ to respect the system capacity, Π being the space of policies.

Context space. As shown by our early experiments in §1, SNR and traffic load are the contextual features that have most impact on the performance of a vRAP. Hence, we divide the time between stage $n-1$ and n into $t := \{1, 2, \ldots, T\}$ *monitoring slots* and collect, at the end of each slot t, the total amount of *new* bits pending to be transmitted, $\delta_{i,n}^{(t)}$, mean $\bar{\sigma}_{i,n}^{(t)}$ and variance $\tilde{\sigma}_{i,n}^{(t)}$ SNR samples between monitoring slot $t-1$ and t across all UEs attached to vRAP $i \in \mathcal{P}$, with $|\mathcal{P}| = P$. This provides information about the *time dynamics* of the various variables of interest, namely (*i*) aggregate traffic load, (*ii*) the quality of the signals each vRAP has to process and (*iii*) the variability of the signal quality, which captures the impact of having multiple (heterogeneous) UEs in the vRAP in addition to their mobility. The time interval between monitoring slots can be decided based upon the reception of BSRs from UEs, for instance. Then, at the beginning of each stage n, we gather all samples into sequences of mean-variance SNR pairs and a sequence of traffic load samples and construct a *context sample* $x_i^{(n)} := \left\{ \langle \bar{\sigma}_{i,n}^{(t)} \rangle, \langle \tilde{\sigma}_{i,n}^{(t)} \rangle, \langle \delta_{i,n}^{(t)} \rangle \right\}_{t=\{1,\ldots,T\}}$ for vRAP i. Consequently, a *context vector* aggregates all context samples for all vRAPs, i.e., $\mathbf{x}^{(n)} = (x_i)_{\forall i \in \mathcal{P}} \in \mathcal{X} \subset \mathbb{R}^{3TP}$.

Action space. Our action space comprises all pairs of compute and radio control actions introduced in §3.1. In this way, $c_i^{(n)} \in C$ and $m_i^{(n)} \in \mathcal{M}$ denote, respectively, the *maximum computing time share* (compute control action) and the *maximum MCS* (radio control action) allowed to vRAP i in stage n. We also let $c_0^{(n)}$ denote the amount of computing resource left unallocated (to save costs). Thus, a resource allocation action on vRAP i consists of a pair $a_i := \{c_i, m_i\}$ and a system action $\mathbf{a} = (a_i)_{\forall i \in \mathcal{P}} \in \mathcal{A} := \{(c_i \in C, m_i \in \mathcal{M})\}_{\forall i \in \mathcal{P}}$.

Reward function. The objective in the design of `vrAIn` is twofold: (*i*) *when the CPU capacity is sufficient*, the goal is to minimize the operation cost (in terms of CPU usage) as long as vRAPs meet the desired performance; (*ii*) *when there is deficit of computing capacity* to meet such performance target, the aim is to avoid decoding errors that lead to resource wastage, thereby maximizing throughput and minimizing delay. To meet this objective, we design the reward function as follows. Let q_{i,x_i,a_i} be the (random) variable capturing the aggregate buffer occupancy across all users of vRAP i given context x_i and action a_i at any given slot. As a quality-of-service (QoS) criterion, we set a target buffer size Q_i for each vRAP. Note that this criterion is closely related to the latency experienced by end-users (low buffer occupancy yields small latency) and throughput (a high thropughput keeps buffer occupancy low). Thus, by setting Q_i, a mobile operator can choose the desired QoS in terms of latency/throughput. This can be used, for instance, to provide QoS differentiation among vRAPs serving different network slices. We let $J_i(x_i, a_i) := \mathbb{P}\left[q_{i,x_i,a_i} < Q_i\right]$ be the probability that q_{i,x_i,a_i} is below the target per vRAP i and define the reward as:

$$r(\mathbf{x}, \mathbf{a}) := \sum_{i \in \mathcal{P}} J_i\left(x_i, a_i\right) - M\varepsilon_i - \lambda c_i \tag{2}$$

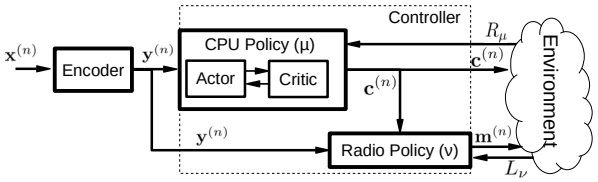

Figure 6: Resource Manager.

where ε_i is the *decoding error probability* of vRAP i (which can be measured locally), and M and λ are constant parameters that determine the weight of decoding errors and the trade-off between computing resources and performance, respectively. We set M to a large value to avoid decoding errors due to overly low CPU allocations (and thus ensure that we do not waste resources) and λ to a small value to ensure that QoS requirements are met (while minimizing the allocation of compute resources).

Design challenges. vrAIn's resource manager, illustrated in Figs. 5 and 6, is specifically designed to solve the above CB problem tackling the following two challenges:

(*i*) *The first challenge* is to manage the high number of dimensions of our contextual snapshots. We address this by implementing an **encoder** e that projects each context vector $\mathbf{x}$ into a latent representation $\mathbf{y} = e(\mathbf{x})$ retaining as much information as possible into a lower-dimensional space. The design of our encoder is introduced in §3.2.1.

(*ii*) *The second challenge* is the continuous action space. Recall that an action $\mathbf{a} \in \mathcal{A}$ comprises a (real-valued) compute control vector $\mathbf{c} \in C^P$ and a (discrete) radio control vector $\mathbf{m} \in \mathcal{M}^P$. We design a **controller** that decouples policy $\pi(\mathbf{x}) : \mathcal{X} \to \mathcal{A}$ into two policies applied sequentially:

- A radio control policy $\nu(\mathbf{y}, \mathbf{c}) = \mathbf{m}$, described in §3.2.2, which we design as a deep classifier that maps an (encoded) context $e(\mathbf{x})$ into a radio control vector $\mathbf{m}$ that guarantees near-zero decoding error probability *given compute allocation* $\mathbf{c}$; and
- A compute control policy $\mu(\mathbf{y}) = \mathbf{c}$, described in §3.2.3, more challenging due to the continuous nature of C, which we address with a deep deterministic policy gradient (DDPG) algorithm [21] that considers deterministic policy ν as part of the environment to maximize reward.

While the above design decouples radio and compute policies, this does not affect the optimality of the solution. Indeed, as our radio policy consists in a deterministic classifier that selects the most appropriate maximum MCS *for the allocation chosen by the CPU policy*, when optimizing the CPU policy (allocation of compute resources), *we also optimize implicitly the radio policy* (maximum MCS).

We next detail the design of the resource manager's encoder (§3.2.1), radio policy ν (§3.2.2) and CPU policy μ (§3.2.3).

3.2.1 Encoder. Evidently, such a high-dimensional contextual space makes our CB problem difficult to handle. To address this, we encode each context vector $\mathbf{x}^{(n)} \in \mathcal{X}$ into a lower-dimensional representation $\mathbf{y}^{(n)} \in \mathbb{R}^D$ with $D \ll \dim(\mathcal{X})$ implementing *encoding* function $e(\mathbf{x}^{(n)})$ in the first functional block of the system described in Figs. 5 and 6.

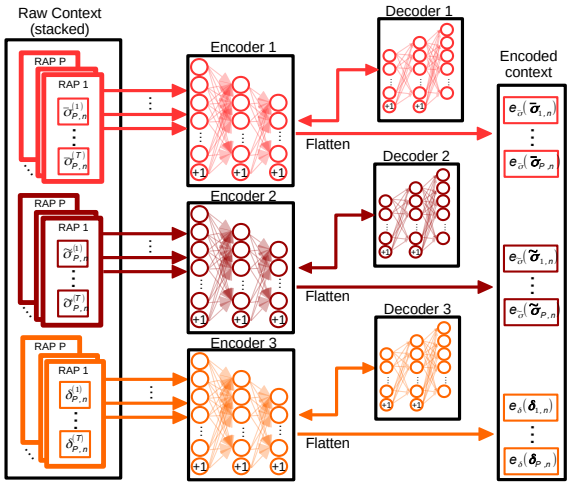

Figure 7: Encoder design.

Note that our contextual data consists in highly complex signals (in time and space) as they concern human behavior (communication and/or user mobility patterns) and so, identifying handcrafted features that are useful yet low-dimensional is inherently hard. Moreover, useful representations may substantially differ from one scenario to another. For instance, the average function may be a good-enough encoder of the SNR sequences in low-mobility scenarios, a linear regression model may be useful in high-mobility scenarios, and the variance function may be needed in crowded areas. Similarly, the average data bit-rate may be sufficient when handling a large number of stationary flows whereas variance may be important for real-time flows. Therefore, there is no guarantee that such hand-picked context representations are useful for the problem at hand.

Conversely, we resort to unsupervised representation learning algorithms. In particular, we focus on a particular construct of neural network called Sparse Autoencoder (SAE), which is commonly used for such cases [12, Ch.14]. A SAE consists of two feed-forward neural networks: an encoder e_ξ (with an output layer of size D) and a decoder d_ψ (with an output layer of size $\dim(\mathcal{X})$) characterized by weights ξ and ψ, respectively. They are trained together so that the reconstructed output of the decoder is as *similar* as possible to the input of the encoder $\mathbf{x}$, i.e., $d(\mathbf{y}) = d(e(\mathbf{x})) \approx \mathbf{x}$.

A linear autoencoder, with linear activation functions in the hidden layers, will learn the principal variance directions (eigenvectors) of our contextual data (like classic principal component analysis (PCA) does [5]). However, our goal is to discover more complex, multi-modal structures than the one obtained with PCA, and so we (*i*) use rectified linear units

(ReLUs), and (*ii*) impose a sparsity constraint in the bottleneck layer (limiting the number of hidden units that can be activated by each input pattern) by adding the Kullback-Leibler (KL) divergence term to the loss function. In this way, we solve the following optimization problem during training:

$$\arg\min_{\xi,\psi} \sum_{i=1}^{PT} \frac{\|x_i - d(x_i)\|^2}{2PT} + \omega \|\xi,\psi\| + \Omega \sum_{j=1}^{D} KL(\rho \| \rho_j) \quad (3)$$

where $KL(\rho\|\rho_j) := \rho \log \frac{\rho}{\rho_j} + (1-\rho) \log \frac{1-\rho}{1-\rho_j}$, with ρ being a sparsity parameter indicating the desired frequency of activation of the hidden nodes (typically small) and ρ_j being the average threshold activation of hidden node j over all training samples. Moreover ω and Ω are hyper-parameters that determine the relative importance given to the weight decay regularization term and the sparseness term in the loss function. The above function and parameters build on well-know machine learning techniques to let our encoder learn a code dictionary that minimizes reconstruction error *with minimal number of code words*, thus providing an accurate and efficient encoding.

Recall that $\mathbf{x}^{(n)} = \left(\langle \bar{\sigma}_{i,n}^{(t)} \rangle, \langle \tilde{\sigma}_{i,n}^{(t)} \rangle, \langle \delta_{i,n}^{(t)} \rangle\right)_{t=\{1,\ldots,T\}, \forall i \in \mathcal{P}}$ consists of 3 different sequences. To avoid losing the temporal correlations within the sequences, we encode each of the three sequences independently, proceeding as follows:

(*i*) First, we train three different SAEs, one for each sequence comprising the triple $\left\{\langle \bar{\sigma}_{i,n}^{(t)} \rangle, \langle \tilde{\sigma}_{i,n}^{(t)} \rangle, \langle \delta_{i,n}^{(t)} \rangle\right\}$;
(*ii*) Second, we encode sequences corresponding to each individual vRAP i independently, i.e., $y_i = \{e_{\xi_k}(x_i)\}_{k=\{\bar{\sigma},\tilde{\sigma},\delta\}}$;
(*iii*) Finally, we append all encoded sequences into a single vector $\mathbf{y} = (y_i)_{\forall i \in \mathcal{P}}$.

This approach, depicted in Fig. 7, avoids that the SAEs attempt to find correlations across vRAPs or sequences of different nature (SNR vs traffic load sequences) when optimizing the autoencoder parameters.

As a result, our controller receives an encoded representation of the context $\mathbf{y}^{(n)} \in e(\mathcal{X})$ as input. To accommodate this in our formulation, we let $\hat{\pi} : \mathbb{R}^{(D_{\bar{\sigma}}+D_{\tilde{\sigma}}+D_{\delta})P} \to \mathcal{A}$ be the corresponding function mapping $\mathbf{y}^{(n)} = e(\mathbf{x}^{(n)})$ into an action in $\mathcal{A}$, with $D_{\bar{\sigma}}$, $D_{\tilde{\sigma}}$ and D_{δ} being the output layer of each of our encoders, and redefine $\hat{\Pi} = \{\hat{\pi} : \mathcal{X} \to \mathcal{A}, \pi(\mathbf{x}) = \hat{\pi}(e(\mathbf{x}))\}$.

3.2.2 Radio Policy (ν). In case there are no sufficient CPU capacity to decode all the frames at the highest MCS allowed by the wireless conditions, we may need to impose radio constraints to some vRAP. To this end, our radio policy consists in imposing an upper bound $\mathbf{m}$ to the set of MCSs eligible by the radio schedulers such that the computational load does not exceed capacity. Note that our radio policy will provide the highest possible $\mathbf{m}$ when there are no CPU constraints.

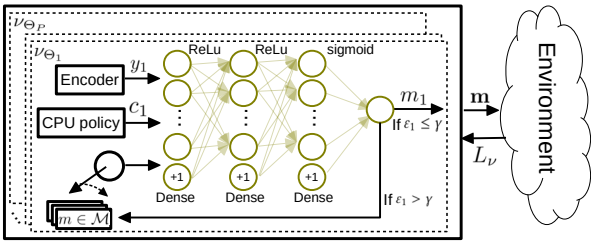

Figure 8: Radio policy ν design.

Following the above, we design a policy ν that receives an encoded context $\mathbf{y}$ and a compute allocation $\mathbf{c}$ as input, and outputs a suitable radio control decision $\mathbf{m}$. Our design consists in a simple neural network ν_{Θ_i} per vRAP i characterized by weights Θ_i with an input layer receiving (y_i, c_i, m_i), a single-neuron output layer activated by a sigmoid function and hidden layers activated by a ReLu function. We define the parameter γ as the threshold corresponding to the maximum acceptable decoding rate, which we set to a small value. Then, we proceed as follows to find the largest MCS satisfying this threshold. We train each ν_{Θ_i} as a classifier that indicates whether an upper bound MCS equal to m_i satisfies $\varepsilon_i \leq \gamma$ (in such a case m_i is an eligible bound for vRAP i as it ensures low decoding error rate given compute allocation c_i and context y_i) or $\varepsilon_i > \gamma$ (it is not). We use a standard loss function L_ν to train the classifiers with measurements of ε_i obtained at each stage n. In order to implement our policy $\nu_\Theta = \{\nu_{\Theta_i}\}_{i \in \mathcal{P}}$, we iterate, for each vRAP i, over the set of MCSs in descending order and break in the first m_i flagged by the classifier as appropriate ($\varepsilon_i \leq \gamma$), as shown in Fig. 8.

In this way, we decouple the radio control actions $\mathbf{m}$ from our action space and rely on the following CPU policy to maximize the reward function defined in §3.2.

3.2.3 CPU Policy (μ). In the following, we design a policy μ that determines the allocation of computing resources in order to maximize the reward function R provided in eq. (2). Note that R depends on both compute control decisions, $\mathbf{c}$, and radio control decisions $\mathbf{m}$ (determined by policy ν). We remark that our MCS selection policy ν is deterministic given a compute allocation vector $\mathbf{c}$. As a result, when deriving the optimal CPU policy we can focus on an algorithm that learns the optimal $\mathbf{c}$ while treating ν as part of the environment. We hence redefine our reward function as:

$$R_\mu := \mathbb{E}_{(\mathbf{y},r)\sim E}\Big[r\left(\mathbf{y}, \mu(\mathbf{y})\right) \Big], \text{ with} \quad (4)$$

$$r(\mathbf{y}, \mathbf{c}) = \sum_{i \in \mathcal{P}} J_i(y_i, c_i) - M\varepsilon_i - \lambda c_i \quad (5)$$

and $J_i(y_i, c_i) := \mathbb{P}\left[q_{i,y_i,a_i} < Q_i\right]$. Our goal is hence to learn an optimal compute policy $\mu^* := \arg\max_\mu R_\mu$ subject to $\sum_{i=0}^{P} c_i = 1$ to respect the system capacity (note that c_0 denotes unallocated CPU time).

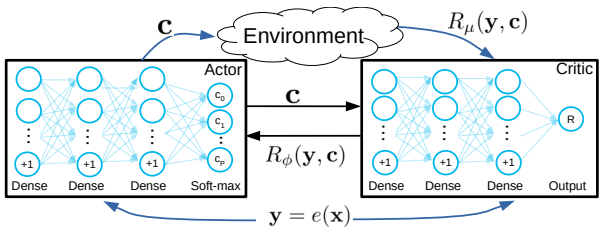

Figure 9: CPU policy μ design.

Since the above expectation depends only on the environment and a deterministic MCS selection policy, we can learn R_μ off-policy, using transitions generated by a different stochastic exploration method. Q learning [44] is an example of a popular off-policy method. Indeed, the combination of Q learning and deep learning (namely DQNs [26]), which use deep neural network function approximators to learn an action-value function (usually represented by the recursive Bellman equation), has shown impressive results in decision-making problems with high-dimensional contextual spaces like is our case. However, DQNs are restricted to discrete and low-dimensional action spaces. Their extension to continuous domains like ours is not trivial, and obvious methods such as quantization of the action space result inefficient and suffer from the *curse of dimensionality*.

Instead, we resort to a deep deterministic policy gradient (DDPG) algorithm [21] using a model-free *actor-critic* approach, which is a reinforcement learning method successfully adopted in continuous control environments such as robotics [13] or autonomous navigation [42]. Our approach is illustrated in Fig. 9. We use a neural network μ_θ (the actor) parametrized with weights θ to approximate our deterministic compute allocation policy $\mu_\theta(\mathbf{y}) = \mathbf{c}$, and another neural network $R_\phi(\mathbf{y},\mathbf{c})$ (the critic) parametrized with weights ϕ to approximate the action-value function R, which assesses the current policy μ_θ and stabilizes the learning process. As depicted in the figure, the output of μ_θ (the actor) is a soft-max layer to ensure that $\sum_{i=0}^{P} c_i = 1$. Although they both run in parallel, they are optimized separately. The critic network needs to approximate the action-value function $R_\phi(\mathbf{y},\mathbf{c}) \approx r(\mathbf{y},\mu(\mathbf{y}))$ and to this end we can use standard approaches such as the following update:

$$\Delta\phi = \beta\left(r(\mathbf{y},\mu(\mathbf{y})) - R_\phi(\mathbf{y},\mathbf{c})\right)\nabla_\phi R_\phi(\mathbf{y},\mathbf{c}) \tag{6}$$

with learning rate $\beta > 0$. Regarding the actor, it is sufficient to implement an stochastic gradient ascent algorithm:

$$\nabla_\theta R_\mu \approx \mathbb{E}\left[\nabla_\theta \mu_\theta(\mathbf{y}) \nabla_\mathbf{c} R_\phi(\mathbf{y},\mathbf{c})\right] \tag{7}$$

Silver *et al.* [33] proved that this is the *policy gradient*. In this way, the actor updates its weights θ as follows:

$$\Delta\theta = \alpha \nabla_\theta \mu_\theta(\mathbf{y}) \nabla_\mathbf{c} R_\phi(\mathbf{y},\mathbf{c}) \tag{8}$$

with learning rate $\alpha > 0$.

Algorithm 1: `vrAIn` algorithm

1 Initialize autoencoders $\{e_{\xi_k}, d_{\psi_k}\}_{k=\{\bar{\sigma},\hat{\sigma},\delta\}}$ — Encoder
2 Set batch size B_1 and training period N_1
3 Initialize actor-critic networks μ_θ, R_ϕ — CPU policy
4 Set batch size B_2 and exploration rate ϵ
5 Initialize radio policy $v_\Theta = \{v_{\Theta_i}\}_{\forall i \in \mathcal{P}}$ — Radio policy
6 Set batch size B_3 and training period N_3
7 **for** $n = \langle 1, 2, \dots \rangle$ **do** *#Main Loop*
8 Measure reward $\tilde{r}^{(n-1)}$ and $\{\tilde{\varepsilon}_i^{(n-1)}\}_{i\in\mathcal{P}}$
9 Store $\left\{\mathbf{x}^{(n-1)}, \mathbf{y}^{(n-1)}, \mathbf{a}^{(n-1)}, \tilde{r}^{(n-1)}, \boldsymbol{\varepsilon}^{(n-1)}\right\}$
10 Observe context $\mathbf{x}^{(n)}$
11 **if** $\text{mod}(n, N_1) == 0$ **then** — Encoder
12 Update SAES $\{e_{\xi_k}, d_{\psi_k}\}_{k=\{\bar{\sigma},\hat{\sigma},\delta\}}$ using eq. (3) with B_1 samples
13 $\mathbf{y}^{(n)} \leftarrow e(\mathbf{x}^{(n)})$
14 Update critic R_ϕ using eq. (6) with B_2 samples — CPU policy
15 Update actor μ_θ using eq. (8) with B_2 samples
16 $\mathbf{c}^{(n)} \leftarrow \mu_\theta(\mathbf{y}^{(n)}) + \text{Bern}(\epsilon^{(n)}) \cdot \eta^{(n)}$
17 **if** $\text{mod}(n, N_3) == 0$ **then** — Radio policy
18 Update classifiers $\{v_{\Theta_i}\}$ using $L_v(\{\tilde{\varepsilon}_i\})$ with B_3 samples
19 $\mathbf{m}^{(n)} \leftarrow v_\Theta(\mathbf{y}^{(n)}, \mathbf{c}^{(n)})$
20 $\mathbf{a}^{(n)} \leftarrow (\mathbf{c}^{(n)}, \mathbf{m}^{(n)})$ *#enforce action*

3.3 vrAIn system

`vrAIn`'s workflow is summarized in Algorithm 1. All neural networks are initialized with random weights or pre-trained with a dataset collected in lab, as depicted in **steps (1)-(6)**.

At the beginning of each stage n, `vrAIn`:

(*i*) Measures the empirical reward and decoding error rate of the previous stage, respectively, as $\tilde{r}^{(n-1)} := \sum_{i\in\mathcal{P}} \tilde{J}_i^{(n-1)} - M\tilde{\varepsilon}_i^{(n)} - \lambda c_i^{(n-1)}$ and $\tilde{\varepsilon}_i^{(n-1)}$ (**step (8)**);
(*ii*) Stores $\{\mathbf{x}^{(n-1)}, \mathbf{y}^{(n-1)}, \mathbf{a}^{(n-1)}, \tilde{r}^{(n-1)}, \boldsymbol{\varepsilon}^{(n-1)}\}$ (**step (9)**);
(*iii*) Observes the current context $\mathbf{x}^{(n)}$ (**step (10)**).

Context $\mathbf{x}^{(n)}$ is first encoded into $\mathbf{y}^{(n)}$ in **step (13)**. Then, we use the actor network μ_θ to obtain $\mathbf{c}^{(n)}$ in **step (16)** and policy v to obtain $\mathbf{m}^{(n)}$ in **step (19)**. At last, `vrAIn` constructs action $\mathbf{a}^{(n)}$ for the current stage n in **step (20)**.

The encoders ($\{e_{\xi_k}, d_{\psi_k}\}_{k=\{\bar{\sigma},\hat{\sigma},\delta\}}$) and the radio classifiers ($\{v_{\Theta_i}\}_{\forall i\in\mathcal{P}}$) are trained every N_1 and N_3 stages with the last B_1 and B_3 samples, respectively (**steps (12)** and **(18)**). Conversely, policy μ's actor-critic networks (μ_θ, R_ϕ) are trained every n with the last B_2 samples (**steps (14)-(15)**). Last, we implement a standard exploration method that adds random noise $\eta^{(n)}$ to the actor's output with probability $\epsilon^{(n)}$, Bern(ϵ) being a Bernoulli-distributed variable with parameter ϵ.

It is worth highlighting that `vrAIn` consists of a set of simple feed-forward neural networks involving simple algebraic operations that require low computational effort.

4 VRAIN PLATFORM

Our vRAN system comprises one SDR USRP[5] per RAP as RRU radio front-end attached via USB3.0 to (*i*) a 2-core i7-5600U @ 2.60GHz compute node or (*ii*) a 4-core i7-8650U @ 1.90GHz compute node,[6] where we deploy our vRAP instances. Although there may be different approaches to implement a vRAP stack, it is reasonable to focus on open-source projects such as `OpenBTS`[7] (3G) and `OpenAirInterface`[8] or `srsLTE` [11] (4G LTE) to ensure reproducibility and *deployability*.

We build our experimental prototype around `srsLTE eNB`, but we note that the same design principles can be applied to any OFDMA-based vRAP, such as unlicensed LTE or the upcoming 5G NR. Similarly, we deploy an UE per RAP,[9] each using one USRP attached to an independent compute node where an `srsLTE UE` stack runs (UEs do not share resources). Finally, with no loss in generality, we configure the vRAPs with SISO and 10 MHz bandwidth. Let us summarize the design keys of `srsLTE eNB` in the sequel. The interested reader can revise a more detailed description in [11].

Fig. 10 depicts the different modules and threads implementing an LTE stack in `srsLTE eNB`. Red arrows indicate data paths whereas dark arrows indicate interaction between threads or modules. Every 1-ms subframe is assigned to an idle PHY DSP *worker*, which executes a pipeline that consumes most of the CPU budget of the whole stack [11], including tasks such as OFDM demodulation, PDCCH search, PUSCH/PUCCH encoding, PDSCH decoding, uplink signal generation and transmission to the digital converter. Having multiple DSPs allows processing multiple subframes in parallel. Since our compute infrastructure consists of 2 and 4-core processors, we set up a total number of 3 DPSs that is sufficient since the HARQ process imposes a latency deadline of 3 ms (3 pipeline stages). The remaining threads perform important operations that are less CPU demanding such as scheduling subframes to DSP workers (PHY RX) or procedures such as random access, uplink/downlink HARQ and scheduling data to physical resource blocks (MAC procedures), timer services (MAC timer), or pushing data from a buffer of uplink TBs to the upper layers (MAC UL reader).

In this way, a multi-thread process, which can be virtualized with virtual machines (like in [23]) or with Linux containers (LXCs), handles all the stack. `vrAIn` relies on the latter since it provides both resource isolation (through `namespaces`) and fine-grained control (through Linux control groups or `cgroups`) with minimal overhead. We next detail our platform's compute and radio control interfaces.

[5]USRP B210 from National Instruments/Ettus Research.
[6]Intel Turbo Boost and hyper-threading are deactivated.
[7]http://openbts.org/
[8]https://www.openairinterface.org/
[9]We use a single UE transmitting aggregated load (from several users)—note that `vrAIn` is scheduler-agnostic.

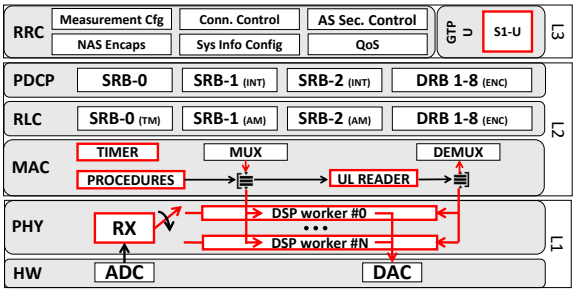

Figure 10: Threading architecture in srsLTE. Boxes with red borders are threads.

4.1 CPU control

When allocating CPU resources to vRAPs, we follow a typical NFV-based approach [24] providing CPU reservations, which ensures isolation across different vRAPs.[10] We rely on Docker[11] for BBU isolation and fine-grained control of computing resources. Docker is an open-source solution that extends LXCs with a rich API to enforce computing resource allocations. Docker uses control groups (`cgroups`), a Linux kernel feature that limits, accounts for, and isolates resource usage of Linux processes withing the group. Docker uses CFS (Completely Fair Scheduler) for CPU bandwidth control of `cgroups`. CFS provides *weight* based allocation of CPU bandwidth, enabling arbitrary slices of the aggregate resource. Hence, we implement a computing resource control action $c_i \in C$ as a CFS CPU quota, which effectively upper bounds the relative CPU time allowed to each vRAP i. In detail, CFS allows the `cgroup` associated with the vRAP container to `cpu.cfs_quota_us` units of CPU time within the period of `cpu.cfs_period_us` (equal to 100 ms by default) by implementing a hybrid global CPU pool approach. More details can be found in [40].

In order for `vrAIn` to exploit Docker's CFS resource controller, we need to set the default scheduling policy of the DSP threads in `srsLTE eNB`, real-time by default, to `SCHED_NORMAL`, which is the default scheduling policy in a Linux kernel. This can be easily done with a minor modification to the PHY header files of `srsLTE eNB`. Moreover, it is worth remarking that, although our platform uses, for simplicity, Docker containers over a single compute node for resource pooling, `vrAIn` can be integrated in a multi-node cloud using, e.g, Kubernetes or Docker Swarm. In such cases, a compute control action $c_i \in C$ requires Kubernetes or Docker Swarm to schedule vRAPs into compute nodes first, and then assign an appropriate CPU time share.

[10]It is widely accepted in NFV that Virtual Network Functions (VNFs) need to have the required CPU resources reserved to ensure the proper operation of the network as well as to isolate VNFs that may belong to different actors (such as, e.g., different tenants in a network slicing context [22, 31]).
[11]https://www.docker.com/

4.2 Radio control

As a proof of concept, we focus on srsLTE's uplink communication, which is the most challenging case as decoding is the most CPU-demanding task and we only receive feedback from UEs periodically. Specifically, srsLTE allocates scheduling grants to UEs in a round robin fashion and then computes their TB size (TBS) and MCS as follows. First, srsLTE maps the SNR into CQI using [16, Table 3]. Then, it maps the UE's CQI into *spectral efficiency* using 3GPP specification tables (TS 36.213, Table 7.2.3-1). Finally, it implements a simple loop across MCS indexes to find the MCS-TBS pair that approximates the calculated spectral efficiency the most. To this aim, srsLTE relies on an additional 3GPP specification table (TS 36.213, Table 7.1.7.1-1) to map an MCS index into a TBS.

A plethora of more elaborated scheduling methods have been proposed (proportional fair, max-weight, exp-rule, etc. [3]). However, as explained in §3.1, vrAIn can learn the behavior of *any* low-level scheduler and hence, in order to integrate our resource manager, we only need to write a handful of lines of code in srsLTE's MAC procedures (see Fig. 10) to (*i*) upper bound the eligible set of MCSs with $m_i \in \mathcal{M}$—which we do by modifying the aforementioned MCS-TBS loop, and (*ii*) expose an interface to the resource manager to modify $m_i \in \mathcal{M}$ online—which we do through a Linux socket.

4.3 Resource Manager

In our implementation, the time between two stages takes 20 seconds and each context sample consists in $T = 200$ samples of mean-variance SNR pairs and data arrivals per vRAP, i.e, $\mathbf{x}^{(n)} = (\langle\bar{\sigma}_{i,n}^{(t)}\rangle, \langle\tilde{\sigma}_{i,n}^{(t)}\rangle, \langle\delta_{i,n}^{(t)}\rangle)_{t=\{1,\ldots,200\}, \forall i \in \mathcal{P}}$. We implement all the neural networks with Python and Keras library.

CPU policy. We implement our compute control policy μ with two neural networks (actor and critic) comprised of 5 hidden layers with $\{20, 40, 80, 40, 10\}$ neurons activated by a ReLu function. The actor has an input layer size equal to $\dim(\mathcal{Y})$ and output layer size equal to $P + 1$ activated with a soft-max layer guaranteeing that $\sum c_i = 1$. In contrast, the critic has an input layer size equal to $\dim(\mathcal{Y}) + P + 1$ to accommodate the input context and a compute control policy, and an output layer size equal to 1 to approximate reward. Both neural networks are trained using Adam optimizer [17] with $\alpha = \beta = 0.001$ and a mean-squared error (MSE) loss function. Finally, unless otherwise stated, we set $M = 2$, $\lambda = 0.25$ and $\epsilon^{(n)} = 0.995^n$, which effectively reduces exploration as the learning procedure advances.

Radio policy. We implement our radio policy ν with a set of P neural networks (one per vRAP), each with 11 hidden layers of sizes $\{5, 8, 20, 30, 40, 40, 40, 40, 30, 20, 5\}$. We pre-train them using the dataset mentioned below with Adam optimizer and use a binary cross-entropy loss function L_ν, typical in classification problems. Then, online training is performed according to Algorithm 1.

Encoder. The encoder networks consist of 3 hidden layers of size $\{100, 20, 4\}$ (mirrored for the decoders), that is, each 200-sampled raw contextual sequence is encoded into a 4-dimensional real-valued vector and appended together as shown in Fig. 7. We have selected four encoded dimensions as this choice provides a good trade-off between low dimensionality and reconstruction error, according to the analysis in §5. We train our neural networks using Adam gradient descend algorithm to minimize eq. (3) using a training dataset introduced next. After pre-training, the autoencoder is periodically trained "in the wild" following Algorithm 1.

Training dataset.[12] To generate our pre-training set, we set up one vRAP and one UE transmitting traffic in different scenarios and repeat each experiment for both compute nodes (i7-5600U and i7-8650U) and a wide set of different control actions as shown in §1:

- *Scenario 1 (static).* The UE is located at a fixed distance from the vRAP and transmits Poisson-generated UDP traffic with fixed mean and fixed power for 60 seconds (i.e. three contextual snapshots). We repeat the experiment for different mean data rates such that the load relative to the maximum capacity of the vRAP is $\{1, 5, 10, 15, \ldots, 100\}$% and different transmission power values such that the mean SNR of each experiment is $\{10, 15, 20, \ldots, 40\}$ dB. Figs. 2, 3 and 4 visualize some results from this scenario.
- *Scenario 2 (dynamic).* We let the UE move at constant speed on a trajectory that departs from the vRAP location (maximum SNR), moves ∼25 meters away (minimum reachable SNR) and then goes back to the vRAP location. We repeat the experiment 12 times varying the speed such that the whole trajectory is done in $\{10, \ldots, 120\}$ seconds.
- *Scenario 3 (2 users).* We repeat Scenario 2 with two UEs moving in opposite directions, producing in this way patterns with different SNR variances.

5 PERFORMANCE EVALUATION

We next assess our design and prototype implementation, evaluating the ability of vrAIn to: (*i*) reduce the dimensionality of the input raw context sequences while preserving *expressiveness* (§5.1); (*ii*) achieve a good trade-off between cost (CPU usage) and QoS performance when there are sufficient CPU resources (§5.2); and (*iii*) maximize performance and distribute resources efficiently across vRAPs when CPU resources are limited (§5.3).

5.1 Encoder

The performance of vrAIn's context encoders is essential to derive appropriate CPU and radio policies. We thus begin our evaluation by validating the design of our autoencoder.

First, we evaluate different encoder dimensions (ranging from 2 to 128) for the different sequence types of our context

[12] Our dataset is available at https://github.com/agsaaved/vrain.

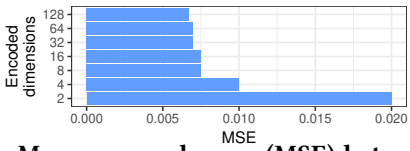

Figure 11: Mean squared error (MSE) between validation dataset and reconstructed sequences after encoding for a variable number of latent dimensions.

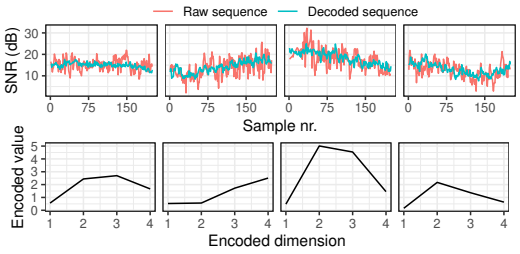

Figure 12: Examples of 200-dimensional raw *vs* reconstructed SNR sequences (top). 4-dimensional encoded representations used by `vrAIn`'s controller (bottom).

(mean SNR, SNR variance and data load patterns). To this aim, we train our autoencoder with 66% of our pre-training dataset, leaving the remaining 34% for validation. Fig. 11 depicts the resulting mean squared error (MSE) of the reconstructed sequences for one sequence type (the mean SNR). From the figure, we conclude that 4 dimensions provide a good trade-off between low dimensionality and reconstruction error, and hence we use this value hereafter.

Second, we visually analyze if the encoder with the above setting captures higher-order pattern information. Fig. 12 shows a few examples of mean SNR sequences $\langle \bar{\sigma}^{(t)} \rangle$ from our pre-training dataset (red, top subplots) encoded into 4-dimensional vectors (bottom subplots) and reconstructed using the decoders introduced in §3.2.1 (blue line, top plots). We observe that the decoder reconstructs the input raw sequences remarkably well. We have observed a similar behavior for the other sequence types in our dataset: SNR variance $\langle \tilde{\sigma}^{(t)} \rangle$ and data load $\langle \delta^{(t)} \rangle$ (results omitted for space reasons). We hence conclude that our design is effective in projecting high-dimensional contextual snapshots into manageable representations—input signals of our controller.

5.2 Unconstrained computational capacity

Next, we evaluate `vrAIn` with the synthetic context patterns shown in Fig. 13. These sequences are constructed to reflect extreme scenarios with high variability. Overall, an *epoch* (the cycling period of our contexts) consists of 54 stages. We first consider a single vRAP on both our compute nodes. This depicts a scenario where computational capacity is "unconstrained" since, as shown by Figs. 2-4, each of our vRAP prototypes requires one full CPU core at most.

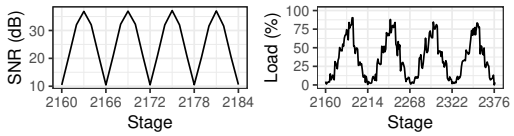

Figure 13: Synthetic context patterns. SNR $\{\langle \bar{\sigma}_n^{(t)} \rangle, \langle \tilde{\sigma}_n^{(t)} \rangle\}$ patterns are generated by changing the UE's tx power to emulate one ~120-s round-trip (Scenario 2 in §4.3) in 6 stages (left plot). Load $\langle \delta_n^{(t)} \rangle$ is sampled from a Poisson process with a mean that varies every 6 stages as $\bar{\delta} = \{5, 10, 30, 50, 70, 85, 50, 30, 10\}$% of the maximum capacity (right plot).

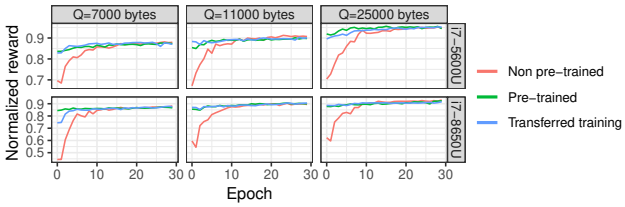

Figure 14: Convergence. Pre-training `vrAIn`, even on different platforms and using pre-default context patterns, expedites convergence ("Transferred training").

Convergence. We first study the convergence of `vrAIn`. We present in Fig. 14 the evolution over time of the *normalized reward*. This is computed as $\sum_{i \in \mathcal{P}} \tilde{J}_i^{(n)} - M\tilde{\varepsilon}_i^{(n)} - \lambda c_i^{(n)}$, where $\tilde{J}_i^{(n)}$ is the fraction of samples where the aggregate data queued by the vRAP is below a target Q_i and $\tilde{\varepsilon}_i^{(n)}$ corresponds to the fraction of unsuccessfully decoded subframes. For visualization purposes, we normalize it between 0 and 1, where 0 corresponds to 100% buffer occupancy violation, 100% decoding errors and 100% CPU usage and 1 corresponds to 0% violation, 0% decoding errors and 0% CPU usage.

We evaluate convergence for both computing nodes and different values of Q, considering three pre-training methods: (*i*) non pre-trained; (*ii*) pre-trained with the dataset introduced in §4.3; and (*iii*) pre-trained with the same dataset but collected for a different platform (i7-5600U node is pre-trained with "i7-8650U" dataset and *vice versa*), which we refer to as "Transferred training". As we can see from the figure, `vrAIn` requires between 10 and 20 epochs to converge for the highly dynamic contexts under evaluation *when it is not pre-trained*. As expected, when `vrAIn` is pre-trained with pre-defined patterns collected in lab, convergence becomes much faster. Furthermore, such pre-training does not necessarily have to be obtained from the same platform, as "Transferred training" allows much faster convergence too.

Performance. We now evaluate `vrAIn` once it has converged, focusing in "i7-5600U" only to reduce clutter, and plot in Fig. 15 (top) the temporal evolution of (*i*) QoS performance (J in eq. (2)), and (*ii*) compute control actions taken by `vrAIn`, for 4 epochs randomly chosen (after convergence)

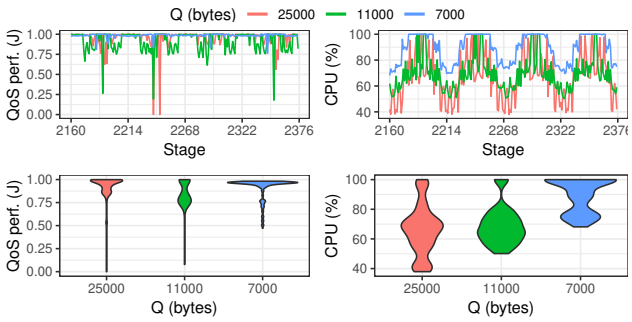

Figure 15: Evolution over time (top) and distribution (bottom) of QoS performance (left) and CPU policy (right). One vRAP deployed over i7-5600U node.

and the same Q values used before. Notice that vrAIn timely follows the context dynamic patterns shown in Fig. 13. In turn, Fig. 15 (bottom) presents the distribution across all epochs. We draw three conclusions from these experiments. The *first conclusion* is that, the lower the parameter Q, the higher the CPU allocation chosen by vrAIn; indeed, higher CPU allocations induce lower decoding delay and thus lower buffer occupancy. The *second conclusion* is that higher Q targets render higher QoS performance, which is intuitive as lower Q implies requirements that are harder to meet. The *third conclusion* is that vrAIn achieves zero decoding error rate when not exploring. This is shown in Fig. 16 (left top) along with two benchmarks that we introduce next. We further observe that vrAIn follows load and SNR dynamics; also, as the computing capacity is always sufficient, the radio policy does not bound the eligible MCSs (not shown for space reasons).

Cost savings. Let us now consider two benchmarks:

(*i*) C_{st}-R_{leg}: Static CPU policy assigning fixed allocations and legacy radio policy (CPU-unaware);

(*ii*) C_{st}-R_{vrAin}: Static CPU policy and vrAIn's radio policy ν, which is CPU-aware.

Note that C_{st}-R_{vrAin} goes beyond the related literature closest to our work, which we revise in §6, namely [1, 29], as we augment such approaches with the ability to adapt the radio allocations to *both* SNR and traffic load dynamics. We apply the above benchmarks in our platform for the same contexts used before and for a wide range of static CPU policies from $\{30, \ldots, 100\}$%. The results, shown in Fig. 16, depict the decoding error rate (left) and QoS performance (right) of both benchmarks as a function of vrAIn's CPU savings for all static policies (i.e., the mean saving/deficit that vrAIn has over the static polices for the chosen CPU allocation). The results make evident the following points:

– Static CPU policies that provide equal or less computing resources than vrAIn's average allocation (**x-axis** $\leq$ **0**) render substantial performance degradation. Specifically, C_{st}-R_{leg} yields high decoding error rate because it selects MCSs based on radio conditions only and does not take into account the availability of computing resources. Conversely, C_{st}-R_{vrAin} worsens QoS performance because its CPU policy fails to adapt to the context dynamics, e.g., data queues build up excessively during peak traffic;

– Static CPU polices that increase the allocation of computing resources above vrAIn's average allocation (**x-axis** > **0**) only match vrAIn's performance when the full pool of computing resources are allocated (with >20% more CPU usage over our approach).

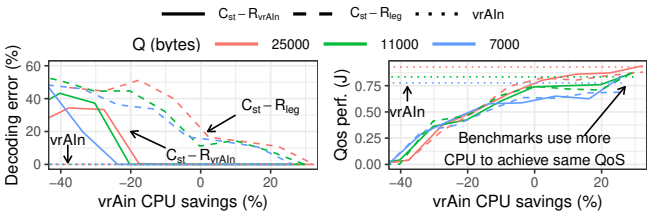

Figure 16: vrAIn *vs* two benchmarks: C_{st}-R_{leg} and C_{st}-R_{vrAin}. vrAIn renders a good trade-off between CPU allocations (cost) and QoS performance.

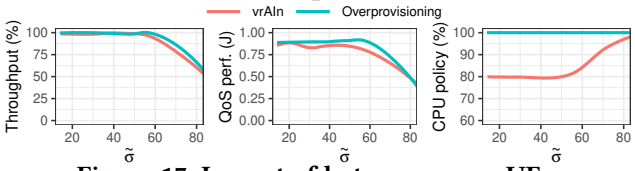

Figure 17: Impact of heterogeneous UEs.

As a result, *we conclude that* vrAIn *achieves a good balance between system cost and QoS performance.*

Heterogeneous UEs. By encoding SNR variance patterns $\tilde{\sigma}$ across all UEs in each vRAP, we enable vrAIn to adapt to contexts involving heterogeneous UEs. To analyze the behavior of vrAIn in such environments, we set up an experiment with two UEs (UE1 and UE2) attached to a vRAP. We fix the transmission power of UE1 such that its mean SNR is equal to 32 dB (high SNR) and vary the transmission power of UE2 to induce different values of SNR variance in the sequence of signals handled by the vRAP. To focus on the impact of the SNR variability, we fix the load of both UEs to 7.3 Mb/s and set $Q = 25000$ bytes. Fig. 17 depicts the resulting aggregate throughput (relative to the load), QoS performance (J) and CPU policy when SNR variance is $\tilde{\sigma} = \{15, \ldots, 80\}$, comparing vrAIn with a policy that allocates all CPU resources to the vRAP ("Overprovisioning"). We observe that throughput and J degrade as $\tilde{\sigma}$ increases, due to the lower signal quality of UE2. We conclude that vrAIn performs well under heterogeneous UEs, as it provides substantial savings over "Overprovisioning" while delivering a similar performance.

5.3 Constrained compute capacity

To complete our evaluation, we evaluate vrAIn under limited CPU capacity. To this end, we set up a second vRAP in our i7-5600U compute node and limit the node's compute capacity to a single CPU core, i.e., both vRAPs ("vRAP1" and "vRAP2")

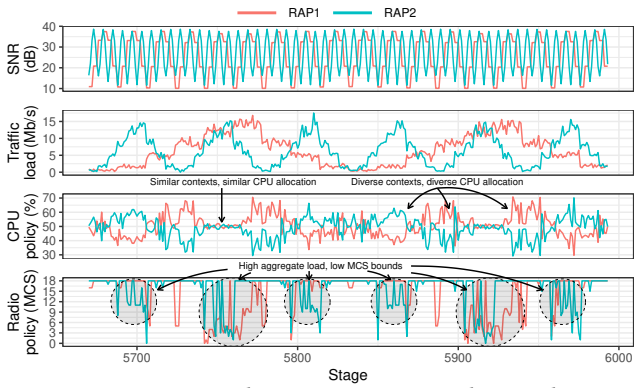

Figure 18: vRAN with 2 vRAPs. `vrAIn` shares the CPU and adapts the radio policy to minimize decoding errors (which are negligible and therefore not shown).

have to compete for these resources during peak periods. Moreover, we fix hereafter $Q_i = 7000\ \forall i = \{1, 2\}$.

Analysis of `vrAIn` dynamics. We first let the vRAPs experience the same dynamic context patterns used before but *3 times slower* for "RAP1", i.e., each epoch of "RAP1" corresponds to 3 epochs of "RAP2". This renders the SNR and load patterns shown in Fig. 18 (top) and allows us to study diverse aggregate load regimes. Note that such uncorrelated patterns may occur for short-term fluctuations even when long-term average loads at different RAPs are correlated.

Fig. 18 depicts the temporal evolution of `vrAIn`'s CPU policy (3rd plot) and radio policy (bottom plot). First, we can observe that `vrAIn` distributes the available CPU resources across both vRAPs following their context dynamics; equally between them when the contexts are similar. More importantly, we note that `vrAIn` reduces the MCS upper bound allowed when the aggregate demand is particularly high to ensure no decoding errors due to CPU capacity deficit.

Comparison against benchmark approaches. We now assess the performance of `vrAIn` against the following benchmarks in scenarios with heterogeneous vRAPs:

- (*i*) $\texttt{C}_{\texttt{vrAIn}}\texttt{-R}_{\texttt{leg}}$: `vrAIn`'s CPU policy and a legacy radio policy that is blind to the availability of CPU capacity.
- (*ii*) `R-Optimal`: An oracle approach that *knows the future contexts* and selects the CPU and radio policies that maximize reward by performing an exhaustive search over all possible settings. Although unfeasible in practice, this provides an upper bound on performance.
- (*iii*) `T-Optimal`: An oracle like `R-Optimal` that optimizes overall throughput instead of reward. Like `R-Optimal`, it is unfeasible in practice.
- (*vi*) `Heuristic`: A linear model between MCS and CPU load is obtained by fitting a standard linear regression to our dataset. Using this model, we derive the CPU load needed by each RAP for the largest MCS allowed with the current mean SNR. If the system capacity is sufficient to handle such CPU load, we apply the resulting CPU/MCS policy. Otherwise, we apply the algorithm of [14] to obtain a fair CPU policy and use our linear model to find the corresponding MCS policy.

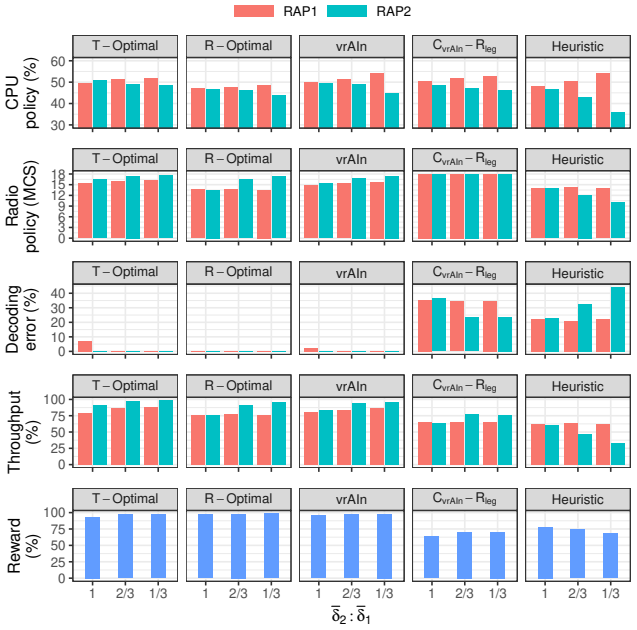

Figure 19: vRAN with 2 heterogeneous vRAPs *vs.* 4 benchmarks: a throughput-optimal oracle (`T-Optimal`), a reward-optimal oracle (`R-Optimal`), `vrAIn`'s CPU policy with a legacy radio policy blind to the CPU availability ($\texttt{C}_{\texttt{vrAIn}}\texttt{-R}_{\texttt{leg}}$), and an heuristic that leverages on a linear model fit with our dataset.

In order to evaluate these mechanisms, we use similar dynamic contexts to those of Fig. 18 but vary the average traffic load of "RAP2" $\bar{\delta}_2$ such that $\bar{\delta}_2 = k \cdot \bar{\delta}_1$ to illustrate the impact of heterogeneous conditions. Fig. 19 shows the performance for all approaches in terms of (*i*) CPU policy, (*ii*) radio policy, (*iii*) decoding error rate, (*iv*) throughput relative to the load, and (*v*) reward, for $k = \{\frac{1}{3}, \frac{2}{3}, 1\}$.

The main conclusion that we draw from the above results is that `vrAIn` performs very closely to the optimal benchmarks (`R-Optimal` and `T-Optimal`) and substantially outperforms the other ones ($\texttt{C}_{\texttt{vrAIn}}\texttt{-R}_{\texttt{leg}}$ and `Heuristic`). Indeed, `vrAIn` provides almost the same reward as `R-Optimal` (the difference is below 2%) and almost the same throughput as `R-Optimal` (the difference is also below 2%). Furthermore, it provides improvements over 25% as compared to $\texttt{C}_{\texttt{vrAIn}}\texttt{-R}_{\texttt{leg}}$ and `Heuristic` both in terms of reward and throughput.

Looking more closely at the results for `vrAIn`, we observe that, as expected, the allocation of computing resources of our CPU policy favors the RAP with higher load, i.e. "RAP1" for $k = \{\frac{1}{3}, \frac{2}{3}\}$, and provides very similar allocations for $\bar{\delta}_2 = \bar{\delta}_1$. In addition, we observe that `vrAIn` appropriately trades high MCS levels off for near-zero decoding error, selecting the highest possible MCS while avoiding decoding errors.

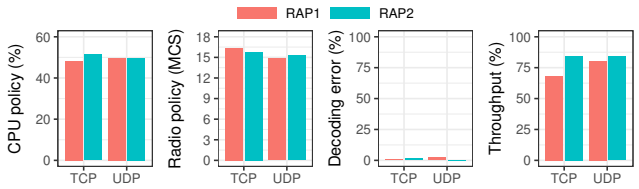

Figure 20: vrAIn impact on TCP

In contrast to vrAIn, C_{vrAIn}-R_{leg} and Heuristic fail to select appropriate policies. The former fails to decode a large number of frames: *as it is blind to the computing capacity*, it employs overly high MCSs under situations of CPU deficit, and thus sacrifices roughly 25% of throughput *w.r.t.* vrAIn. The latter does adapt its radio policy to the CPU capacity; however, it does so employing an oversimplified model that does not provide a sufficiently good approximation and yields poor choices: in some cases, it selects overly high MCS bounds, leading to decoding errors, while in other cases it chooses overly small MCSs, leading to poor efficiency. As a result, Heuristic also sacrifices substantial throughput *w.r.t.* vrAIn (losing as much as 30% in some cases).

TCP flows. Finally, we assess the performance of vrAIn in the presence of TCP traffic. Fig. 20 shows the performance of vrAIn using both TCP and UDP transport protocols for the same context dynamics used before when $\bar{\delta}_2 = \bar{\delta}_1$. The figure shows that both transport layer protocols obtain similar performance: vrAIn attains similar CPU savings for TCP and UDP (left plot of Fig. 20) without penalizing the overall throughput (right plot of Fig. 20). This shows that vrAIn works well under short-term traffic fluctuations such as the ones resulting from the adaptive rate algorithm of TCP.

6 RELATED WORK

There exists a large amount of literature on the management of wireless resources, i.e., scheduling and MCS selection mechanisms, with different scenarios and optimization criteria (e.g., [15, 19, 20]). The advent of virtualized RAN has fostered some research work to understand the relationship between computing and wireless resources, e.g., [1, 2].

Theoretical work. The works of [29] and [1] set a theoretical basis for CPU-aware radio resource control. However, both works rely on the same model relating computing requirements and channel quality conditions, which needs to be pre-calibrated for the platform and scenario they operate in and neglect variations on the traffic load (i.e. they assume persistent full buffers). While this issue is addressed in [43], this work also relies on a simplistic baseband processing model and lacks of experimental validation.

Experimental work. Despite [2] being the first study on the cost savings that can be obtained by cross-optimizing computing resources across vRAPs, its heuristic does not consider important factors such as load and SNR variations, which we have showed to have a great impact on the overall performance. Similar conclusions can be drawn from the work of [39]. Other efforts, such as PRAN [45], which proposes a resource demand prediction model and RT-OPEX [10], which implements a CPU scheduler tailored for vRAP workload, are complementary to our work.

vrAIn. In contrast to this prior work, we make an in-depth experimental study of the relationship between performance, radio and computing resources. We conclude that traditional system modeling approaches are highly ineffective due to the dependency between them, with the context (SNR, traffic load patterns) and with the actual platform. In light of this, our approach, vrAIn, exploits model-free learning methods to dynamically control the vRAN resources while it adapts to contextual changes and/or different platforms.

7 CONCLUSIONS

Virtualized radio access networks (vRANs) are the future of base stations design. In this paper, we have presented vrAIn, a vRAN solution that *dynamically learns* the optimal allocation of computing and radio resources. Given a specific QoS target, vrAIn determines the allocation of computing resources required to meet such target and, in case of limited capacity, it jointly optimizes radio configuration (MCS selection) and CPU allocation to maximize performance. To this end, vrAIn builds on deep reinforcement learning to adapt to the specific platform, vRAN stack, computing behavior and radio characteristics. Our results shed light on the behavior of vrAIn across different scenarios, showing that vrAIn is able to meet the desired performance targets while minimizing CPU usage, and gracefully adapts to shortages of computing resources. Moreover, performance is close to optimal and shows substantial improvements over static policies or simple heuristics. To the best of our knowledge, this is the first work that thoroughly studies the computational behavior of vRAN, and vrAIn is the first *practical* approach to the allocation of computing and radio resources to vRANs, adapting to any platform by *learning* its behavior on the fly.

ACKNOWLEDGMENTS

We would like to thank our shepherd Bo Chen and reviewers for their valuable comments and feedback. The work of NEC Laboratories Europe was supported by H2020 5G-TRANSFORMER project (grant agreement no. 761536) and 5GROWTH project (grant agreement no. 856709). The work of University Carlos III of Madrid was supported by H2020 5G-MoNArch project (grant agreement no. 761445) and H2020 5G-TOURS project (grant agreement no. 856950). The work of University of Cartagena was supported by Grant AEI/FEDER TEC2016-76465-C2-1-R (AIM) and Grant FPU14/03701.

REFERENCES

[1] D. Bega, A. Banchs, M. Gramaglia, X. Costa-Perez, and P. Rost. CARES: Computation-Aware Scheduling in Virtualized Radio Access Networks. *IEEE Transactions on Wireless Communications*, 17(12):7993–8006, Dec. 2018.

[2] S. Bhaumik, S. P. Chandrabose, M. K. Jataprolu, G. Kumar, A. Muralidhar, P. Polakos, V. Srinivasan, and T. Woo. CloudIQ: A Framework for Processing Base Stations in a Data Center. In *Proceedings of the 18th ACM International Conference on Mobile Computing and Networking (ACM MobiCom 2012)*, Istanbul, Turkey, Aug. 2012.

[3] F. Capozzi, G. Piro, L. A. Grieco, G. Boggia, and P. Camarda. Downlink Packet Scheduling in LTE Cellular Networks: Key Design Issues and a Survey. *IEEE Communications Surveys Tutorials*, 15(2):678–700, July 2013.

[4] A. Checko, H. L. Christiansen, Y. Yan, L. Scolari, G. Kardaras, M. S. Berger, and L. Dittmann. Cloud RAN for Mobile Networks—A Technology Overview. *IEEE Communications Surveys Tutorials*, 17(1):405–426, Sept. 2015.

[5] U. Demšar, P. Harris, C. Brunsdon, A. S. Fotheringham, and S. McLoone. Principal Component Analysis on Spatial Data: An Overview. *Routledge Annals of the Association of American Geographers*, 103(1):106–128, July 2012.

[6] Y. Deng, F. Bao, Y. Kong, Z. Ren, and Q. Dai. Deep Direct Reinforcement Learning for Financial Signal Representation and Trading. *IEEE Transactions on Neural Networks and Learning Systems*, 28(3):653–664, Mar. 2017.

[7] A. Garcia-Saavedra, X. Costa-Perez, D. J. Leith, and G. Iosifidis. Fluidran: Optimized vran/mec orchestration. In *Proceedings of the IEEE International Conference on Computer Communications (IEEE INFOCOM 2018)*, pages 2366–2374, Apr. 2018.

[8] A. Garcia-Saavedra, G. Iosifidis, X. Costa-Perez, and D. J. Leith. Joint optimization of edge computing architectures and radio access networks. *IEEE Journal on Selected Areas in Communications*, 36(11):2433–2443, Nov. 2018.

[9] A. Garcia-Saavedra, J. X. Salvat, X. Li, and X. Costa-Perez. WizHaul: On the Centralization Degree of Cloud RAN Next Generation Fronthaul. *IEEE Transactions on Mobile Computing*, 17(10):2452–2466, Oct. 2018.

[10] K. C. Garikipati, K. Fawaz, and K. G. Shin. RT-OPEX: Flexible Scheduling for Cloud-RAN Processing. In *Proceedings of the 12th ACM International on Conference on Emerging Networking EXperiments and Technologies (ACM CoNEXT 2016)*, Irvine, USA, Dec. 2016.

[11] I. Gomez-Miguelez, A. Garcia-Saavedra, P. D. Sutton, P. Serrano, C. Cano, and D. J. Leith. srsLTE: An Open-source Platform for LTE Evolution and Experimentation. In *Proceedings of the 10th ACM International Workshop on Wireless Network Testbeds, Experimental Evaluation, and Characterization (ACM WiNTECH 2016)*, New York City, USA, Oct. 2016.

[12] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[13] S. Gu, E. Holly, T. Lillicrap, and S. Levine. Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates. In *Proceedings of the 2017 IEEE International Conference on Robotics and Automation (IEEE ICRA 2017)*, Singapore, May 2017.

[14] C. Joe-Wong, S. Sen, T. Lan, and M. Chiang. Multiresource Allocation: Fairness-Efficiency Tradeoffs in a Unifying Framework. *IEEE/ACM Transactions on Networking*, 21(6):1785–1798, Dec. 2013.

[15] M. Kalil, A. Shami, and A. Al-Dweik. QoS-Aware Power-Efficient Scheduler for LTE Uplink. *IEEE Transactions on Mobile Computing*, 14(8):1672–1685, Aug. 2015.

[16] M. T. Kawser, N. I. B. Hamid, M. N. Hasan, M. S. Alam, and M. M. Rahman. Downlink SNR to CQI Mapping for Different Multiple Antenna Techniques in LTE. *International Journal of Information and Electronics Engineering*, 2(5):757–760, Sept. 2012.

[17] D. P. Kingma and J. Ba. Adam: A Method for Stochastic Optimization. *arXiv preprint arXiv:1412.6980*, Jan. 2017.

[18] J. Kober, J. A. Bagnell, and J. Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32(11):1238–1274, Aug. 2013.

[19] Y. Li, M. Sheng, X. Wang, Y. Zhang, and J. Wen. Max-Min Energy-Efficient Power Allocation in Interference-Limited Wireless Networks. *IEEE Transactions on Vehicular Technology*, 64(9):4321–4326, Sept. 2015.

[20] Z. Li, S. Guo, D. Zeng, A. Barnawi, and I. Stojmenovic. Joint Resource Allocation for Max-Min Throughput in Multicell Networks. *IEEE Transactions on Vehicular Technology*, 63(9):4546–4559, Nov. 2014.

[21] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. In *Proceedings of the 2016 International Conference on Learning Representations (ICLR 2016)*, San Juan, Puerto Rico, May 2016.

[22] C. Marquez, M. Gramaglia, M. Fiore, A. Banchs, and X. Costa-Perez. How should i slice my network?: A multi-service empirical evaluation of resource sharing efficiency. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking (ACM MobiCom 2018)*, New Delhi, India, Oct. 2018.

[23] J. Mendes, X. Jiao, A. Garcia-Saavedra, F. Huici, and I. Moerman. Cellular access multi-tenancy through small-cell virtualization and common RF front-end sharing. *Elsevier Computer Communications*, 133:59–66, Jan. 2019.

[24] R. Mijumbi, J. Serrat, J.-L. Gorricho, N. Bouten, F. De Turck, and R. Boutaba. Network Function Virtualization: State-of-the-Art and Research Challenges. *IEEE Communications Surveys Tutorials*, 18(1):236–262, Sept. 2015.

[25] R. Miotto, F. Wang, S. Wang, X. Jiang, and J. T. Dudley. Deep learning for healthcare: review, opportunities and challenges. *Briefings in Bioinformatics*, 19(6):1236–1246, Nov. 2018.

[26] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):518–529, Feb. 2015.

[27] B. Niu, Y. Zhou, H. Shah-Mansouri, and V. W. S. Wong. A Dynamic Resource Sharing Mechanism for Cloud Radio Access Networks. *IEEE Transactions on Wireless Communications*, 15(12):8325–8338, Dec. 2016.

[28] P. Rost, I. Berberana, A. Maeder, H. Paul, V. Suryaprakash, M. Valenti, D. Wübben, A. Dekorsy, and G. Fettweis. Benefits and challenges of virtualization in 5G radio access networks. *IEEE Communications Magazine*, 53(12):75–82, Dec. 2015.

[29] P. Rost, A. Maeder, M. C. Valenti, and S. Talarico. Computationally Aware Sum-Rate Optimal Scheduling for Centralized Radio Access Networks. In *Proceedings of 2015 IEEE Global Communications Conference (IEEE GLOBECOM 2015)*, San Diego, USA, Dec. 2015.

[30] P. Rost, S. Talarico, and M. C. Valenti. The Complexity-Rate Tradeoff of Centralized Radio Access Networks. *IEEE Transactions on Wireless Communications*, 14(11):6164–6176, Nov. 2015.

[31] J. X. Salvat, L. Zanzi, A. Garcia-Saavedra, V. Sciancalepore, and X. Costa-Perez. Overbooking Network Slices Through Yield-driven End-to-end Orchestration. In *Proceedings of the 14th ACM International Conference on Emerging Networking EXperiments and Technologies (ACM CoNEXT 2018)*, Heraklion, Greece, Dec. 2018.

[32] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, Jan. 2016.

[33] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller. Deterministic policy gradient algorithms. In *Proceedings of the 31st*

International Conference on Machine Learning (ICML 2014), Beijing, China, June 2014.

[34] V. Suryaprakash, P. Rost, and G. Fettweis. Are Heterogeneous Cloud-Based Radio Access Networks Cost Effective? *IEEE Journal on Selected Areas in Communications*, 33(10):2239–2251, Oct. 2015.

[35] R. S. Sutton, A. G. Barto, et al. *Introduction to reinforcement learning*, volume 135. MIT press Cambridge, 1998.

[36] D. Szczesny, A. Showk, S. Hessel, A. Bilgic, U. Hildebrand, and V. Frascolla. Performance analysis of LTE protocol processing on an ARM based mobile platform. In *Proceedings of 2009 International Symposium on System-on-Chip*, Oct. 2009.

[37] L. Tang, R. Rosales, A. Singh, and D. Agarwal. Automatic Ad Format Selection via Contextual Bandits. In *Proceedings of the 22nd ACM International Conference on Information & Knowledge Management (CIKM 2013)*, San Francisco, USA, Oct. 2013.

[38] A. Tewari and S. A. Murphy. *From ads to interventions: Contextual bandits in mobile health*. Springer Mobile Health: Sensors, Analytic Methods, and Applications, July 2017.

[39] T. X. Tran, A. Younis, and D. Pompili. Understanding the Computational Requirements of Virtualized Baseband Units Using a Programmable Cloud Radio Access Network Testbed. In *Proceedings of 2017 IEEE International Conference on Autonomic Computing (ICAC 2017)*, July 2017.

[40] P. Turner, B. B. Rao, and N. Rao. CPU bandwidth control for CFS. In *Proceedings of 2010 Ottawa Linux Symposium (OLS 2010)*, volume 10, 2010.

[41] O. Vinyals, I. Babuschkin, J. Chung, M. Mathieu, M. Jaderberg, W. M. Czarnecki, A. Dudzik, A. Huang, P. Georgiev, R. Powell, et al. AlphaStar: Mastering the Real-Time Strategy Game StarCraft II. Online: https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/, Jan. 2019.

[42] C. Wang, J. Wang, X. Zhang, and X. Zhang. Autonomous navigation of UAV in large-scale unknown complex environment with deep reinforcement learning. In *Proceedings of the 5th IEEE Global Conference on Signal and Information Processing (IEEE GlobalSIP 2017)*, Montreal, Canada, Nov. 2017.

[43] K. Wang, X. Yu, W. Lin, Z. Deng, and X. Liu. Computing aware scheduling in mobile edge computing system. *Springer Wireless Networks*, pages 1–17, Jan. 2019.

[44] C. J. Watkins and P. Dayan. Q-learning. *Springer Machine learning*, 8(3-4):279–292, May 1992.

[45] W. Wu, L. E. Li, A. Panda, and S. Shenker. PRAN: Programmable Radio Access Networks. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks (ACM HotNets 2014)*, Los Angeles, USA, Oct. 2014.

[46] C. Y. Yeoh, M. H. Mokhtar, A. A. A. Rahman, and A. K. Samingan. Performance study of LTE experimental testbed using OpenAirInterface. In *Proceedings of the 18th International Conference on Advanced Communication Technology (ICACT 2016)*, PyeongChang, Korea, Jan. 2016.