Tabular Reinforcement Learning Methods for Artificial Intelligence Tasks Offloading in Smart Eye-Wears

ABEDNEGO WAMUHINDO KAMBALE, HAMTA SEDGHANI, FEDERICA FILIPPINI, GIACOMO VERTICALE, and DANILO ARDAGNA, Politecnico di Milano, Milan, Italy

Virtual and Extended Reality technologies are increasingly adopted in fields such as healthcare, entertainment, and education. These applications heavily rely on Smart Eye-Wears (SEWs) and AI to provide users with new ways to perceive their environment. However, SEWs face limitations in computational power, memory, and battery life. Offloading computations to external servers is a prominent example of edge computation. However, this also presents considerable challenges due to delays caused by varying network conditions and server workloads. This article proposes self-adaptive techniques based on tabular reinforcement learning (RL) to optimize the offloading of Deep Neural Network tasks between the SEW, the user's smartphone, and cloud servers. The goal is to maintain a high-quality user experience while minimizing energy consumption and 5G connection costs. We evaluated our framework under varying 5G and WiFi bandwidths and cloud latency. The results show that Q-learning, SARSA, and Expected SARSA achieve near-optimal policies, with Q-learning demonstrating superior performance in reducing execution time violations (approximately at 10%) and improving agent stability. Additionally, our approach offers a more favorable tradeoff between energy efficiency and execution time violations compared to two baseline methods. Real-system experiments reveal that the proposed solution can double SEW battery life with respect to local computation while maintaining a good quality of service, with only 11% execution time violations. These findings highlight the effectiveness of our approach in managing resources and enhancing the overall user experience in SEW AI applications.

CCS Concepts: • **Computing methodologies** → **Intelligent agents**; • **Information systems** → *Computing platforms*;

Additional Key Words and Phrases: Smart glasses, Smart Eye-Wear, Reinforcement Learning, Edge Computing, Task offloading

ACM Reference format:
Abednego Wamuhindo Kambale, Hamta Sedghani, Federica Filippini, Giacomo Verticale, and Danilo Ardagna. 2026. Tabular Reinforcement Learning Methods for Artificial Intelligence Tasks Offloading in Smart Eye-Wears. *ACM Trans. Autonom. Adapt. Syst.* 21, 1, Article 3 (March 2026), 38 pages.
https://doi.org/10.1145/3771092

Authors' Contact Information: Abednego Wamuhindo Kambale (corresponding author), Politecnico di Milano, Milan, Italy; e-mail: abednegowamuhindo.kambale@polimi.it; Hamta Sedghani, Politecnico di Milano, Milan, Italy; e-mail: hamta.sedghani@polimi.it; Federica Filippini, Politecnico di Milano, Milan, Italy; e-mail: federica.filippini@polimi.it; Giacomo Verticale, Politecnico di Milano, Milan, Italy; e-mail: giacomo.verticale@polimi.it; Danilo Ardagna, Politecnico di Milano, Milan, Italy; e-mail: danilo.ardagna@polimi.it.

This work is licensed under Creative Commons Attribution-NonCommercial-NoDerivatives International 4.0.

© 2026 Copyright held by the owner/author(s).
ACM 1556-4703/2026/3-ART3
https://doi.org/10.1145/3771092

1 Introduction

AI recently became pervasive, finding application in various domains such as healthcare, agriculture, and entertainment [10, 18, 33, 40]. In all these fields, AI has yielded promising outcomes: for example, in healthcare, AI algorithms play a pivotal role in expediting diagnoses and improving patient well-being [10]. Meanwhile, in agriculture, AI aids farmers in making informed decisions to optimize crop performance while minimizing resource wastage [33]. In the realm of entertainment, AI applications have elevated user satisfaction levels and delivered more efficient and captivating experiences [18, 40]. In an effort to offer advanced features, many smart device manufacturers are integrating AI models into various user devices (e.g., smartphones) as well as into smaller, resource-constrained devices (e.g., smart glasses) [49]. These AI applications predominantly rely on complex image-based processes and frequently involve the utilization of **Deep Neural Networks (DNNs)**. Due to their large number of parameters, these DNNs necessitate considerable computational power and memory, particularly in applications where real-time processing is required.

Although significant advancements have been made in enhancing the memory and computational capabilities of IoT and edge devices, challenges related to limited battery capacity and computational power persist when executing complex AI models [4]. Depending on the circumstances, these constraints may make it impractical to execute all the DNN layers on a single device. Moreover, the limitation posed by battery capacity detrimentally affects the user experience.

Several studies [23, 24, 35] proposed DNN partitioning as a solution to address these challenges. DNN partitioning involves splitting the DNN into different parts, each suitable for running on different devices across a computing continuum. This partitioning offers various configurations, allowing the system to switch between them to optimize performance. An emerging application for this technology is found in smart glasses (**Smart Eye-Ear [SEW]**) and virtual/augmented reality scenarios. SEW devices collect data from their surroundings, process a portion of the DNN locally, and delegate the remaining computation to a mobile phone or the cloud exploiting all the resources available in the computing continuum. However, communication between the SEW and the mobile phone, as well as between the mobile phone and the cloud, may experience fluctuations, potentially impacting application performance. For instance, in latency-sensitive applications with an execution time constraint of 500 ms, such fluctuations can cause the end-to-end execution time to increase to several seconds, resulting in violations of the constraint. In the remainder of this article, we refer to a *violation* as any instance where the end-to-end execution time exceeds the predefined execution time constraint of the application. To maintain a satisfactory user experience under varying network bandwidth and back-end cloud latency, it is essential to dynamically determine the most appropriate system configuration at each time interval.

This article introduces a self-adaptive runtime optimization framework, leveraging **Reinforcement Learning (RL)** to distribute the computation of AI applications over various computing units. The goal is to reduce energy and 5G networking costs from the user's viewpoint while maintaining end-to-end latency at levels that preserve a satisfactory user experience.

The main contributions of this article are the following:

- We design a three-layer computing continuum architecture to model the different components when managing the execution of SEW AI applications. We formulate the general problem as a **Markov Decision Process (MDP)** that can be solved using different approaches. We consider image-processing inference applications where frames are provided as input to the DNN at a given rate.
- We propose an RL-based runtime management solution in which an agent selects the optimal configuration to minimize **Energy Consumption (EC)** and 5G network costs from the user's perspective, while ensuring that end-to-end latency requirements are met. We implement our

proposed solution by comparing different tabular methods including Q-learning [48], Double Q-learning [17], Weighted Q-learning [9], SARSA [42], and Expected SARSA [43] to evaluate their performance and identify the ones yielding the most promising results.

—We evaluate the proposed solution by accounting for network bandwidth and cloud latency variability, as well as by adjusting execution time constraints and the distribution of importance weights for the agent's objectives. The results identify Q-learning as the most effective method, with execution time violations around 10%, suggesting that the strategy is promising for efficient resource allocation and user satisfaction.

—We compared our solution with two baselines: *Neurosurgeon*, a seminal work presented in [23], and the static solution of the underlying optimization problem, referred to as *Opt**, which considers the average WiFi and 5G bandwidths. Our approach demonstrated superior performance compared to both baselines, reducing execution time violations by 42% compared to Opt* and achieving up to 41% and 19% energy savings compared to *Neurosurgeon* and *Opt**, respectively, while maintaining a comparable level of execution time violations to *Neurosurgeon*.

—Finally, we validate the performance of the trained agent on a real system prototype comprising the Microsoft HoloLens 2 [32], focusing on EC and execution time constraint violations. Our agent achieves a 47% reduction in HoloLens EC compared to running the entire DNN locally, effectively doubling the device's battery life while incurring only 11% of execution time violations. These results demonstrate the practical utility of our approach.

The remainder of this article is organized as follows. Section 2 details the scenario considered in this work. Section 3 presents the system model and formally defines the optimization problem, while Section 4 presents the RL-based formulation. In Section 5, we present the experimental setup and discuss the results of different analyses, including the validation on the real system prototype. Section 6 provides an overview of the related literature proposals. Finally, Section 7 concludes the article and presents future perspectives.

2 Reference Scenario

This section presents the reference scenario foundational to our research. The SEW device runs an AI application encompassing a DNN capable of executing various AI functionalities related to image processing, for example, object classification, detection, or tracking.

The primary objective is to ensure a positive user experience by setting time thresholds on the AI application execution. For example, in the case of an object-tracking application, it is crucial to maintain a minimum application frame rate of 30 **frames per second (fps)**, which translates to an end-to-end execution time of less than 33 ms [28]. For low-frame rate videos, a rate of 1–5 fps (i.e., 200–1,000 ms) is acceptable [2]. For Simultaneous Localization and Mapping applications, an acceptable frame rate falls within the range of 15–30 fps, implying that the end-to-end execution time should range in about 30–66 ms [15, 25].

Figure 1 depicts the three primary elements in our system. In the remainder of this article, we assume a forward-looking scenario where the reference hardware utilizes next-generation **System-on-Chips (SoCs)** as projected in the Snapdragon series [38]. The SEW device (based on, e.g., a Snapdragon AR2 chip [38]) will establish a connection with the mobile phone through WiFi7. Additionally, we assume that it incorporates components, including CPU and an AI neural engine, to execute specific computations. Analogously, the mobile phone (integrating, e.g., a Snapdragon 8 Gen 2 chip [39]) features high-end computing components, such as a GPU.

The SEW collects data from its surroundings and transmits them to the mobile phone. Depending on the current system state (e.g., battery level or network quality), an RL agent running on the

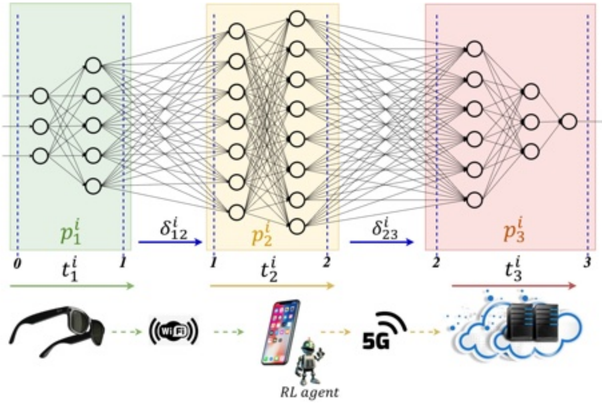
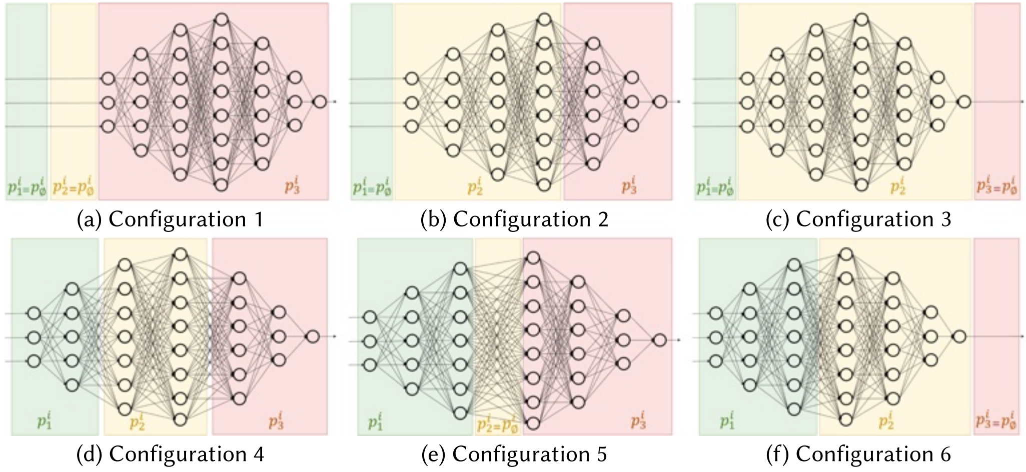


Fig. 1. Reference scenario for performing AI tasks.

phone determines whether the acquired data are sent directly to the phone itself or undergo local DNN processing before transmission. As already mentioned, a DNN-based AI application can be characterized by different partitioning points (see Figure 2), which enable the dynamic computation offloading [7, 23]. The phone will run the DNN from the layer where the SEW terminated the execution, either completing the processing, if it has sufficient resources, or offloading part of it to the cloud according to the RL agent decision. In such cases, the cloud handles the remaining computations and returns the final results. In other terms, the RL agent is responsible for determining the appropriate DNN configuration to run. Note that, as in shown in [23], only specific partitioning points are relevant for optimizing EC and data transfers among different system components and therefore are considered as viable candidates. When the RL agent selects a configuration, it directly informs which partition should run on the SEW, on the mobile phone, or in the cloud. The RL agent takes one action every few seconds or whenever a given number of consecutive violations (e.g., five consecutive frames processed at the application level) are observed prior to the conclusion of the control interval. The main goal is to choose a setup that reduces energy usage and communication costs while ensuring adherence to execution time constraints, thus maintaining a satisfactory user experience.

The various components of the system communicate through distinct network domains. The phone uses a 5G network to transmit data to the cloud server. This results in a significant reduction in transfer delay between the connected entities [3, 6] since 5G mmWave systems offer substantial bandwidth, albeit with a notable degree of variability [34]. At the same time, WiFi interference can become a concern when multiple devices operate at the same frequency, resulting in disruptions and fluctuations in network throughput. This interference, which occurs, for example, when two SEW end-users come into close proximity, can result from overlapping signals and transmissions, reducing the quality and reliability of the wireless connection. Consequently, the network throughput may exhibit fluctuations, impacting the overall performance of WiFi-based systems and applications [36]. Furthermore, cloud servers, which function as the computational backbone of the computing continuum, exhibit inherent performance variability due to fluctuating workloads. This variability impacts server queuing times, which in turn influence the overall end-to-end latency of AI applications, potentially affecting their performance and responsiveness [1]. While our management system is designed to make local decisions to manage such variability, it is important to note that cloud resource allocation remains beyond our control.



(a) Configuration 1 (b) Configuration 2 (c) Configuration 3
(d) Configuration 4 (e) Configuration 5 (f) Configuration 6

Fig. 2. Candidate configurations of the DNN.

The total application processing time includes several components: the local execution time on the SEW, the transmission latency from the SEW to the smartphone, the processing time on the smartphone, the data-transfer latency from the smartphone to the cloud, and, finally, the cloud processing time. In this study, we will neglect the time required to return results, as it is negligible when compared to the overall execution time and data transfer delays. For example, in the context of a classification task, the returned results include only the labels of the classified objects. On the other hand, the results of an object-detection task encompass objects labels, their bounding boxes, and their class probabilities; while the size is surely larger than in a classification task, it is still significantly smaller than the intermediate tensors transmitted during the processing.

3 System Model and Problem Definition

In this section, we present the system model and formulate the runtime optimization problem of AI applications partitioning in the SEW computing continuum. In particular, Section 3.1 presents the DNN partition model. Section 3.2 describes the communication model we consider to determine the data transfer rates between system components for different network domains and the characterization of the 5G connection cost. Section 3.3 introduces how the latency is computed at the different system levels, Section 3.4 presents the EC model, and Section 3.5 provides the complete problem formulation. For the reader's convenience, all the problem parameters and variables detailed in the next sections are summarized in Appendix A, Table A1.

3.1 DNN Partitions Model

In this work, we consider image-based AI applications that process requests supplied in the form of image frames with a frequency Λ (expressed in fps).

The decision made by the agent dictates whether the application should be run entirely on the SEW itself or if some DNN layers should be offloaded to either the smartphone or the cloud server [23] (in this case, Λ is also the rate at which the SEW interacts with the phone). This choice corresponds to the selection of a specific DNN configuration, and it is motivated by the fact that the SEW might have limited computational power, not enough to ensure that the whole AI application will be executed within the required time frame. Moreover, its battery may have limited residual capacity. Depending on the system state, the choice of a specific configuration is guided by the

minimization of incurred costs. At runtime, in case the offloading is needed, the first partition of the selected DNN configuration will be executed on the SEW, the second one on the phone, and the last one on the cloud server. We denote by $\mathscr{D} = \{1, 2, 3\}$ the set of the computing component indices, so that $d = 1$ denotes the SEW, $d = 2$ the mobile phone, and $d = 3$ the cloud server. Moreover, we define the set $\mathscr{K}$ of all candidate configurations. Each element $\kappa^i \in \mathscr{K}$ represents a system configuration including three DNN partitions, defined as $\kappa^i = \{p_d^i\}_{d \in \mathscr{D}}$.

Figure 2 illustrates representative candidate configurations for a DNN. To simplify the modeling and the identification of the partition that is executed on a specific device, we introduce a partition $p_\emptyset^i$ that does not include any DNN layer, meaning that no computation is executed. This allows us to establish a one-to-one correspondence between partitions and devices, and therefore to characterize *a priori* where each partition is deployed. Indeed, for every configuration κ^i, the first partition p_1^i is always executed on the SEW, the second partition p_2^i on the mobile phone, and the third partition p_3^i on the cloud server. Figure 2(a) and (c) shows cases where the whole DNN is executed on the cloud server or the mobile phone, respectively; accordingly, the remaining partitions correspond to $p_\emptyset^i$. Figure 2(b), (e), and (f) represents cases where a single DNN partitioning point is chosen: two partitions run on the phone and the cloud, the SEW and the phone, or the SEW and the cloud, respectively, while the third computational component remains idle (i.e., it is assigned $p_\emptyset^i$). Finally, Figure 2(d) illustrates the case of three partitions, executed by all the three computational components.

Throughout the application execution, the agent might switch from one configuration to another; for instance, Figure 2(a) and (b) might represent the solutions adopted when the SEW battery level is low. We assume that all the partitions fit in the device's memory, since those violating this constraint can be filtered *a priori*.

We denote the control time period as τ. All the problem parameters we introduce in the following are measured in each of these time windows; unless differently specified, their value, which may vary from one interval to the other, will be considered as constant within τ. This allows us to drop the explicit time-dependency from the parameter's definition and, instead, add an apex τ to the parameter name only when we want to mark its relationship with a specific time slot.

We define x^i as a binary variable that equals 1 if configuration κ^i is selected, and we denote by $|\mathscr{K}|$ the total number of configurations. Note that one single configuration must be chosen in each time slot τ, which can be expressed by the following constraint:

$$\sum_{i=1}^{|\mathscr{K}|} x^i = 1. \tag{1}$$

When a computing component does not run the entire DNN or its last layer, it sends the intermediate tensor to the next component. Specifically, the SEW sends the data to the smartphone and this to the cloud. We denote by δ_{dh}^i the amount of per-request data related to configuration κ^i that is sent by partition p_d^i to partition p_h^i. Thus, δ_{12}^i is the amount of per-request data that are sent from the SEW to the smartphone and δ_{23}^i from the smartphone to the cloud. When everything is executed on the SEW, we have $\delta_{12}^i = \delta_{23}^i = 0$, while in case everything is offloaded to the cloud $\delta_{12}^i = \delta_{23}^i = \delta_0$, where δ_0 is the size of the input tensor.

The parameter μ_d^i, measured in floating point operations, denotes the per-request computational workload of running the partition p_d^i. This information can be easily gathered by development frameworks such as PyTorch, TensorFlow, and so on. This quantity, as first approximation [51], determines the EC of the computing resources of the device d when running a partition p_d^i. Moreover, to each partition p_d^i, we associate a partition latency t_d^i, which defines the processing time of partition p_d^i when running on the associated device. This latency can be computed from a

partition latency model that predicts the DNN latency on the specific device [30] or by performing a runtime profiling of the DNN [11, 26] (which is the approach we follow in this work).

3.2 Network Latency and Cost Model

As in [45], we estimate networks latency through the access time (T), bandwidth (ϕ) and $data_size$ to transfer:

$$l_{network} = \left(T + \frac{data_size}{\phi}\right), \tag{2}$$

where T and ϕ depend on the network domain we consider (WiFi7 or 5G). In particular, $T_{5\text{G}}$ depends on the geographic location of the network node that the user wants to access. For simplicity and as discussed in [34], we consider it as invariant with respect to the time, even though it may sometimes fluctuate due to the solicitation of the network (for instance, if many users are concurrently accessing the network, they may experience an access time greater than the one they would observe when the network is not saturated).

Since information about the network access time and bandwidth are not easily accessible in practical applications, we focus on the network throughput that characterizes each domain, denoted as r_{WiFi7} and $r_{5\text{G}}$, respectively. The network throughput can be influenced by many factors, among which the network access time, network congestion, and packet loss, and it intervenes in the computation of both the data-transfer time and the EC associated with it. Therefore, we can use it to express Equation (2) as:

$$l_{WiFi7} = \frac{data_size}{r_{WiFi7}}, \quad l_{5G} = \frac{data_size}{r_{5G}}. \tag{3}$$

Note that the WiFi7 throughput may be reduced when two users approach each other, due to the interference between their two WiFi hotspots. As already mentioned, the network throughput is measured in each time interval τ, thus we should denote it by r^{τ}_{WiFi7} and $r^{\tau}_{5\text{G}}$ for the two network domains. Since we consider it to be constant within the time interval τ, we drop the apex in the rest of the formulation.

We assume that users enable the WiFi7 hotspot to connect the SEW to the phone, while they leverage 5G to connect from the phone to the cloud. Therefore, the data-transfer latencies l_{sp} between the SEW device and the smartphone and l_{pc} between the smartphone and the cloud are:

$$l_{sp} = \frac{\sum_{i=1}^{|\mathcal{K}|} \delta^i_{12}\, x^i}{r_{\text{WiFi7}}}, \quad l_{pc} = \frac{\sum_{i=1}^{|\mathcal{K}|} \delta^i_{23}\, x^i}{r_{5\text{G}}}. \tag{4}$$

Note that, by Constraint (1), only one x^i is equal to 1.

The user is interested in minimizing the cost c_{5G} associated with the 5G connection used to offload the computation from the smartphone to the cloud. This depends on the size δ^i_{23} of the data sent from the smartphone to the cloud, and on the cost g paid to transfer one byte of data, that is:

$$c_{5\text{G}} = \sum_{i=1}^{|\mathcal{K}|} \Lambda\, g\, \delta^i_{23}\, x^i. \tag{5}$$

On the other hand, the WiFi7 hotspot between the SEW and the phone does not require any internet connection, therefore we do not consider WiFi costs in our model.

3.3 Application Latency Model

We recall that the end-to-end latency comprises different entity execution times, and the data-transfer latency between those entities. Let l_{SEW} and l_{phone} be the SEW and phone execution times,

respectively, which can be expressed as:

$$l_{SEW} = \sum_{i=1}^{|\mathcal{K}|} t_1^i \, x^i, \quad l_{phone} = \sum_{i=1}^{|\mathcal{K}|} t_2^i \, x^i. \tag{6}$$

Similarly, we consider a simple cloud model in which the latency and its variance are measured in the control time slots τ. We denote the total latency on the cloud system by l_{cloud}^{τ} and define it as $l_{cloud}^{\tau} = f(\lambda^{\tau}, \sum_i t_3^i \, x^i)$, where λ^{τ} is the (unknown) workload injected in the cloud in the time slot τ.

Exploiting the information on the data-transfer latencies l_{sp} and l_{pc} introduced in the previous section, the end-to-end latency is given by:

$$l_{total} = l_{SEW} + l_{phone} + l_{sp} + l_{pc} + l_{cloud}. \tag{7}$$

3.4 EC Model

The power consumption of the devices includes four components, namely:

- The power consumed by the SEW and phone when running a partition locally, denoted as P_{exeSEW} and $P_{exePhone}$, respectively, which consider the CPU and AI neural engine/GPU EC models;
- The power consumption of the **wireless network interface (WNI)** and **5G interface (5GI)** when uploading the intermediate tensor to the smartphone and cloud, denoted as P_{wnu} and P_{5gu}, respectively;
- The power consumed by the WNI and 5GI when retrieving the remote execution results, denoted as P_{wnd} and P_{5gd}, respectively;
- The power consumed by the WNI and 5GI when the SEW and phone are in *idle* state, denoted as P_{wnidle} and P_{5gidle}, respectively. The idle state is enabled when the SEW and phone are waiting for the results from the execution of the remote partitions.

The power terms P_{exeSEW} and $P_{exePhone}$ can be computed as in [51] and are given by:

$$\begin{aligned} P_{exeSEW} &= \sum_{i=1}^{|\mathcal{K}|} \Lambda \, z_{SEW} \, \mu_1^i \, x^i \\ P_{exePhone} &= \sum_{i=1}^{|\mathcal{K}|} \Lambda \, z_{phone} \, \mu_2^i \, x^i, \end{aligned} \tag{8}$$

where z_{SEW} and z_{phone} are the EC for performing one floating point operation on SEW and phone, respectively.

When the devices offload all or some parts of the computation, they consume the power P_{wnu} and P_{5gu} to send the intermediate tensor. Let θ_{SEW} and θ_{phone} denote the power consumption of the SEW WNI and phone 5GI when transmitting data, respectively. The value of P_{wnu} and P_{5gu} are computed as proposed in [41]:

$$\begin{aligned} P_{wnu} &= \Lambda \, \theta_{SEW} \, l_{sp} = \frac{\sum_{i=1}^{|\mathcal{K}|} \Lambda \, \theta_{SEW} \, \delta_{12}^i \, x^i}{r_{\text{WiFi7}}} \\ P_{5gu} &= \Lambda \, \theta_{phone} \, l_{pc} = \frac{\sum_{i=1}^{|\mathcal{K}|} \Lambda \, \theta_{phone} \, \delta_{23}^i \, x^i}{r_{\text{5G}}}. \end{aligned} \tag{9}$$

In this work, we neglect both the power consumption of downloading back the result (P_{wnd} and P_{5gd}) and the power consumption in an idle state (P_{wnidle} and P_{5gidle}), which are usually significantly

lower than the other terms. Therefore, we define the total power consumption P_{SEW} and P_{phone} as:

$$P_{SEW} = P_{exeSEW} + P_{wnu}, \quad P_{phone} = P_{exePhone} + P_{5gu}. \tag{10}$$

Note that, since all the terms in Equations (8) and (9) are powers, the EC in a specific time slot with duration τ is computed as follows:

$$E^{\tau}_{SEW} = P_{SEW}\tau, \quad E^{\tau}_{phone} = P_{phone}\tau. \tag{11}$$

3.5 Problem Definition

As we mentioned in Section 3.1, the agent must choose an appropriate DNN configuration depending on the system state. We formulate this decision problem as the minimization of the total processing cost as follows:

$$\min_{x^i} \left(\alpha(P_{SEW} + P_{phone}) + c_{5G}\right)\tau \tag{P1a}$$

subject to Constraint (1) and:

$$l_{total} < L_{max} \tag{P1b}$$

$$x^i \in \{0, 1\}, \forall\, i \in \{1, 2, \ldots, |\mathscr{K}|\}. \tag{P1c}$$

The goal is to minimize the EC defined in the previous section and the 5G connection cost. In the objective function above, α denotes the energy unit price (measured in \$/J); moreover, L_{max} is the maximum time latency that guarantees a satisfactory user experience. This work does not deal with the cost of the cloud service since we are trying to optimize the system configuration from the user's perspective (while cloud costs are incurred by the application provider). The optimization of the overall system (which includes the decision, e.g., on the number of virtual machines to run in the cloud back-end) will be considered as part of our future work. Note that, in a specific time slot τ, the optimization problem can be solved by inspection, by considering all the possible system configurations, with a complexity of $O(|\mathscr{K}|)$ (as done by Neurosurgeon [23]). However, due to the variability in system performance between time slots, leveraging RL offers the advantage of learning system dynamics and making proactive decisions, enabling the achievement of better energy-performance tradeoffs (as it will be shown in Section 5).

4 RL Approach

In this section, we model the runtime management of AI applications as an MDP (see Section 4.1). The easiest way to solve MDPs is to use dynamic programming, but in some cases, as the one we consider, this is not possible due to the state space size or the unknown system dynamic. Therefore, we present an RL-based approach to tackle the MDP in Section 4.2.

4.1 Problem Formulation as an MDP

We formulate the SEW offloading problem as a discrete-time infinite-horizon MDP [44]. This can be defined by a 5-tuple $\langle \mathbf{S}, \mathbf{A}, P, R, \gamma \rangle$, where $\mathbf{S}$ represents the infinite set of all the possible states; $\mathbf{A}(s)$ denotes the finite set of all possible actions in state s; $P(s'|s, a)$ is the transition probability from a given state s to a state s' given an action $a \in \mathbf{A}(s)$; $R(s, a, s')$ is the immediate reward received when an action a is executed in state s and the system transits to state s'; and $\gamma \in [0, 1]$ is a discount factor that adjusts the importance of future reward (more details in the next section). In modeling the system as an MDP, we make two key assumptions: the *Markov property* and *stationarity*. The Markov property implies that the system next state depends only on its current state and action, and not on the sequence of events that preceded it. The assumption of *stationarity* means that the rules governing the transition dynamics of the environment do not change over time; that

is, the probability of transitioning from one state to another given an action remains constant. These assumptions are standard in RL formulations and are critical for ensuring the stability and tractability of the learning process [44].

We define the agent state as:

$$s = (r_{\text{WiFi7}},\ r_{\text{5G}},\ l_{SEW},\ l_{phone},\ l_{cloud}). \tag{12}$$

Among these, the data rates r_{WiFi7} and r_{5G}, and the cloud latency l_{cloud} are exogenous parameters that are not under the agent control. Therefore, their variability is independent on the chosen actions and is simply observed from the environment. Moreover, we assume that the phone works in light-load conditions, that is, there are no other running tasks that compete for its resources. On the other hand, l_{SEW} and l_{phone} change according to the actions, but their value is known *a priori*: indeed, we can estimate the execution times t_1^i and t_2^i of the partitions executed on the SEW and the smartphone for all the configurations κ^i (see Section 6.1). If we discretize all the continuous variables with 10 values each, the dimension of the state space **S** is in the order of 10^5.

An action consists in selecting a configuration κ^i, thus the number of actions equals the number of configurations $|\mathscr{K}|$. Note that, in some states, the agent may choose not to change the configuration selected in the previous time window; we model this scenario by introducing an action η that represents the *do nothing* choice. The set of all the possible actions $\mathbf{A}(s)$ is hence given by:

$$\mathbf{A}\,(s) = \left\{a^1, a^2, \ldots, a^{|\mathscr{K}|}\right\} \cup \{\eta\}, \tag{13}$$

where each a^i represents the action of raising to 1 the variable x^i, thus selecting configuration κ^i.

We associate a cost $c(s, a, s')$ to each triplet state–action–next state; $c(s, a, s')$ embeds the energy costs $c_{e_{SEW}}\,(s, a) = \alpha e_{SEW}$ and $c_{e_{phone}}\,(s, a) = \alpha e_{phone}$, the 5G connection cost $c_{5G}\,(s, a)$, and the penalty $c_{lat}\,(s, a, s')$ we pay when violating the maximum latency requirement (Constraint (P1b)). Recall that we defined *violation* as any instance where the end-to-end execution time exceeds the predefined execution time constraint of the application. Moreover, if the agent chooses an action different from η in a given time slot, the system must be reconfigured, which implies some overhead; therefore, we introduce a reconfiguration penalty c_{rcfg}, defined as $c_{rcfg} = \mathbb{1}_{\{a \neq \eta\}}$, meaning that we pay a penalty equal to 1 every time the chosen action is different from η, 0 otherwise. Similarly, the cost $c_{lat}\,(s, a, s')$ incurred when the system violates the latency constraint can be simply defined as $c_{lat}\,(s, a, s') = \mathbb{1}_{\{l_{total} > L_{max}\}}$. We combine the different costs using a simple additive weighting approach [50], writing:

$$c\,(s, a, s') = \omega_{e_{SEW}} \frac{c_{e_{SEW}}}{c_{e_{SEW},max}} + \omega_{e_{phone}} \frac{c_{e_{phone}}}{c_{e_{phone},max}} + \omega_{5G} \frac{c_{5G}}{c_{5G,max}} + \omega_{lat} c_{lat} + \omega_{cfg}\, c_{rcfg}, \tag{14}$$

where $\omega_{e_{SEW}}$, $\omega_{e_{phone}}$, ω_{5G}, ω_{lat}, and ω_{rcfg} are non-negative weights that sum up to 1. $c_{e_{SEW},max}$, $c_{e_{phone},max}$, and $c_{5G,max}$ are the normalization parameters for the energy cost on SEW and phone and the cost of the 5G connection, respectively. The immediate reward $R(s, a, s')$ in the MDP formulation is defined as the negative of the cost $c(s, a, s')$. As a result, maximizing the cumulative reward is equivalent to minimizing the cumulative cost.

When we have a high battery level, we may want to promote the local computation on the SEW, while, when the battery level drops, we may foster the computation offloading to the mobile phone or the cloud. Therefore, we might vary the weights according to the actual battery level.

The system dynamic is stochastic; hence, the transition probability matrix cannot be defined. As a consequence, it is impossible to solve this MDP using traditional dynamic programming. The next section will present the model-free RL-based approaches we propose to tackle this problem.

4.2 RL Approaches

RL methods aim to learn an optimal policy by interacting directly with the system and leveraging the collected experience [44]. Specifically, the objective of an RL agent is to learn a policy that maximizes the expected cumulative reward over time. In each control time interval τ, the agent observes a state s_τ, selects an action a_τ, receives a reward $R_{\tau+1}$, and transitions to a state $s_{\tau+1}$. The long-term return, denoted by G_τ, is defined as the discounted sum of future rewards $G_\tau = \sum_{i=0}^{\infty} \gamma^i R_{\tau+i+1}$, where $\gamma \in [0, 1)$ is the *discount factor* that determines the present value of future rewards. A smaller γ makes the agent prioritize short-term rewards, while a value closer to 1 encourages long-term planning. The agent goal is to find a policy that maximizes the expected return $\mathbb{E}[G_\tau \mid s_\tau]$ from each state. RL approaches assume that the system complies with the Markov property and the stationary assumptions defined in Section 4.1. While these properties might not hold in real systems, some works have successfully used RL techniques in non-Markovian scenarios [14, 29].

In this work, we consider several Q-tabular methods, that is, algorithms that use a matrix (called the Q-table) to store estimates of the state–action value function $Q(s, a)$. Each entry $Q(s, a)$ represents the estimated expected cumulative return obtained by taking action a in state s. The agent uses these estimates to decide which action to take at the next step, and then updates $Q(s, a)$ based on the observed reward, thus improving its policy. The Q-tabular methods considered in this work are model-free, meaning they do not require prior knowledge of the system dynamics. We consider a simple ε-greedy action selection approach that, at each step τ, chooses the greedy action (i.e., $a_\tau = \text{argmin}_{a \in A(s_\tau)} Q(s_\tau, a)$) with probability $1 - \varepsilon$, *exploiting* its knowledge of the system, whereas it picks a random action with probability ε to *explore* the space, enhancing its knowledge of the application. The ε-greedy policy guarantees the exploration of sub-optimal actions, albeit with low probability, while choosing the optimal action for a given state most of the times.

The choice of preferring Q-tabular methods over, for example, **deep reinforcement learning (DRL)** is based on several considerations aligned with our problem characteristics and deployment constraints. First, although the state space may seem large in theory, it is effectively constrained and discretized through quantized metrics such as DNN layer indices, hardware availability, and a limited set of throughput and energy levels. This makes the state space tractable without requiring function approximation. Second, Q-tabular methods offer significant computational efficiency, which is critical for real-time applications on resource-limited devices such as embedded or wearable platforms. Unlike DRL methods that involve costly forward passes and backpropagation, tabular Q-learning updates a single table entry per step, enabling fast and lightweight learning. Third, in our setting with a relatively small and structured state–action space, Q-tabular methods achieve high data efficiency and convergence speed, avoiding the complexity and overhead associated with training deep models. While DRL is powerful for high-dimensional or continuous domains, the tabular approach offers a more practical and effective solution for our targeted use case.

Since most RL methods are highly problem-dependent, we conduct a comprehensive comparison among different tabular methods, namely Q-learning [48], SARSA [42], Expected SARSA [43], double Q-learning [17], and Weighted Q-learning [9], to investigate the effects on our problem of different levels of exploration and of the tradeoff between sample efficiency and computational time. Moreover, we first train our agents in a simulated environment, and later move to a real system where they can further update the developed policies.

Algorithm 1 summarizes the approach we followed for offline training while Algorithm 2 describes the RL agent behavior when deployed on a real system prototype. Algorithm 1 receives as inputs the state and action spaces, the cost function, the number of steps T, and some RL hyper-parameters as the learning rate lr, γ, and ε (line 1). Then, the Q-table of state–action values is randomly initialized (lines 3), and a state is randomly sampled from the environment (line 4).

Algorithm 1: ε-Greedy Tabular Methods

1: **Input:** $S, A, C, T, lr \in [0,1], \gamma \in [0,1], \varepsilon \in [0,1]$, set of weights **W**, set of L_{max}, *method*
2: **Initialization:**
3: $Q : S \times A \rightarrow \mathbb{R}$ initialize arbitrarily
4: Start in state $s \in S$
5: **for** $1, .., T$ **do**
6: Set **W** or L_{max} according to the step
7: $a \leftarrow \varepsilon$-greedyPolicy(s, A, ε)
8: Take action a, observe new state s' and get reward R considering **W**
9: Update Q table(s) based on *method*
10: $s \leftarrow s'$
11: **end for**
12: **return** Q

Algorithm 2: Algorithm Used to Update the RL Agent on Real System Prototype

1: **Input:** $S, A, C, T, lr \in [0,1], \gamma \in [0,1], \varepsilon \in [0,1]$, set of weights **W**, set of L_{max}, control time-period τ, $Max_consecutive_violations$, *method*
2: **Initialization:**
3: $Q : S \times A \rightarrow \mathbb{R}$ initialize with learned Q table
4: Initialize *violations* and *elapsed_time*
5: Start in state $s \in S$
6: **for** $1, .., T$ **do**
7: Set **W** or L_{max} according to battery level or user preference
8: $a \leftarrow \varepsilon$-greedyPolicy(s, A, ε)
9: **while** $elapsed_time < \tau$ **and** $violations < Max_consecutive_violations$ **do**
10: Take action a, observe state s' and get reward R considering **W**
11: Store (s, a, R, s') in transition buffer T_b
12: **if** $latency > L_{max}$ **then**
13: $violations \leftarrow violations + 1$
14: **else**
15: $violations \leftarrow 0$
16: **end if**
17: **end while**
18: Update Q table(s) based on T_b and *method*
19: Reset *violations*, flush the buffer T_b and reset *elapsed_time*
20: **end for**
21: **return** Q

The general outline of all tabular methods is presented in lines 5–11: in each step, the importance weights **W** or the threshold L_{max} are set according to the user preference to account for energy efficiency (more details in Section 5); then, the agent chooses an action following the ε-greedy policy, observes the new state s', and receives the reward (lines 6–8). The Q-table is updated depending on the selected method, and the environment moves to the next state (lines 9–10). The line 9 in 1 can be detailed for the different methods as follows.

— *Q-learning* learns the optimal policy directly by maximizing rewards through a greedy action-selection strategy, eliminating the possibility of the agent taking an exploration step from the second step in the update function; accordingly, it uses the action a' that maximizes the

Q-value (i.e., minimizes the cost) to update the Q-table:

$$Q(s,a) \leftarrow (1-lr)Q(s,a) + lr\Big(R + \gamma \max_{a'} Q(s',a')\Big).$$

— *Double Q-learning* exploits two Q-tables (i.e., two value estimates, which makes the algorithm more stable) to reduce bias and avoid the overestimation of action values often incurred by Q-learning. To perform the update, it randomly chooses one of the Q-tables (Q^A and Q^B), selects the action a' that maximizes its value, and updates it based on the other Q-value. For instance, if Q^A is selected, we set:

$$a' = \underset{a}{\operatorname{argmax}}\, Q^A(s',a)$$
$$Q^A(s,a) \leftarrow (1-lr)Q^A(s,a) + lr(R + \gamma Q^B(s',a')).$$

— *Weighted Q-learning* aims to address a Double Q-Learning drawback, that is, the fact that the relative performance of the two estimators is highly problem-dependent. To increase robustness, it estimates the maximum expected value relying on a weighted average of sample means; weights are computed using Gaussian approximations for the sample means distributions. For each action $a_i \in A$, it defines w_i as the number of times a_i minimizes the samples, divided by the total number of samples. Then, the weighted Q is computed as $W = w^T Q(s', \mathbf{a})$, where $\mathbf{a}$ is the vector of all actions. Finally, Q is updated as:

$$Q(s,a) \leftarrow (1-lr)Q(s,a) + lr(R + \gamma W).$$

— *SARSA* incorporates an exploration step by consistently following the ε-greedy strategy to choose action a', and it updates the Q-table as follows:

$$Q(s,a) \leftarrow (1-lr)Q(s,a) + lr(R + \gamma Q(s',a')).$$

Directly learning from the exploration policy (on-policy) makes SARSA more conservative and stable than Q-learning, even though the latter tends to converge more quickly to an optimal policy.

— *Expected SARSA* updates the Q-table based on a weighted average of the Q-values for all the possible actions in the next state, taking into account the likelihood of each action under the current policy, instead of stochastically sampling state–action values. Accordingly:

$$Q(s,a) \leftarrow (1-lr)Q(s,a) + lr\left(R + \gamma \sum_{a'} \pi(a'|s')Q(s',a')\right),$$

where $\pi(a'|s')$ is the probability of selecting action a' in state s', according to the current policy, that is updated as follows:

$$\pi(a'|s') = \begin{cases} \frac{1-\epsilon}{|\operatorname{argmax}_{a''} Q(s',a'')|} + \frac{\epsilon}{|A|} & \text{if } Q(s',a') = \max_{a''} Q(s',a'') \\ \frac{\epsilon}{|A|} & \text{otherwise.} \end{cases}$$

When running Algorithm 2 in a real system, the Q-table is initialized with the Q-table learned during offline training (line 3). Also, all the other variables needed by the algorithm are initialized in line 4. The importance weights $\mathbf{W}$ and L_{max} remain constant or are updated based on user preferences or battery level (line 7). The agent follows an ε-greedy policy, executing actions and storing transitions (s, a, R, s') in a buffer T_b during the control time (10 seconds). No Q-table updates occur during this period, but constraints violations are recorded (lines 8–14). Once the control time elapses, the agent updates its Q-table using transitions from T_b and requests a new action (lines 18–20). If the number of consecutive violations exceeds the acceptable threshold, the agent requests

a new action even before the control time ends (line 9). After each cycle, we reset the violation count and elapsed time, and clear the buffer T_b (line 19).

5 Experimental Results

This section evaluates the performance of the RL agent under different scenarios. Section 5.1 presents the experimental setup, with a focus on the network bandwidth and cloud performance variability. Section 5.2 introduces the alternative methods we considered as baseline for the evaluation. Sections 5.3 and 5.4 present the results of the analyses varying the importance weights and the maximum time latency, respectively. These variations allow us to assess the agent's ability to maintain effective system control under different operational preferences and timing constraints, although full adaptability to real-time stochastic preference changes remains an open challenge in this work. In Section 5.5, we compare our approach against the baselines, focusing on energy efficiency and execution-time violations. Finally, in Section 5.6, we validate the best-performing agent (based on Q-learning) by considering a prototype application run in a real edge system. The instances we considered for the numerical analyses reported in this section are available on Zenodo.[1]

5.1 Experimental Setup

We considered a reference AI application for object tracking. These can be used in an indoor scenario, where the user wearing the SEW may want to identify various objects in her/his environment, or outdoor for, for example, people tracking. A rate of 1–5 fps, which can be associated to an execution time threshold between 200 and 1,000 ms, is considered acceptable for low-frame rate videos [2].

According to [31], DNN-based object-tracking frameworks incorporate CNNs for object detection. These frameworks include post-processing operations such as bounding boxes filtering and drawing, as well as object assignment in different frames. In our experiments, we focused on the object detection phase, and we adopted the widely used YOLOv5 model.[2] YOLOv5 utilizes a CNN as its backbone and incorporates additional operations in its neck and head to enable object detection. A profiling campaign was carried out to measure the execution time of different partitions, considering HoloLens 2 as a representative of the computing power that will be available in the next generation SEW, a Samsung Galaxy S23[3] as the second device on the second layer, and a Dell Precision 5480 PC as the cloud server. The HoloLens 2 comprises a Qualcomm Snapdragon 850 SoC with an Octa-core, 2840 MHz, Kryo 385, 64-bit ARM processor, 4 GB of RAM, and Adreno 630 GPU. The Samsung S23 includes a Snapdragon 8 Gen 2 SoC, 8 GB RAM, 128 GB storage, an Octa-core processor with a maximum frequency of 3.36 GHz, and an Adreno 740 GPU, which is compliant with the announced Qualcomm SEW-phone AR ecosystem. The PC is characterized by 64 GB of RAM, a 13th Gen Intel Core i7-13800Hx20 processor, and a Nvidia 8 GB GPU. All three devices were connected to the same WiFi hotspot provided by the Samsung S23. The HoloLens 2 and the Samsung S23 support WiFi5 and WiFi6E, respectively. The DNN time and energy profiling were performed using a C# application on HoloLens 2 (implemented using Microsoft Visual Studio and Unity), an Android server on the mobile phone, and the PC running a docker container for the third partition of the DNN model. The details of YOLOv5 model partitioning and configuration parameters are presented in Appendix B.2.

We set the parameter ε of the agents ε-greedy policy to 0.05 in all the experiments. The learning rate is initialized to 1 and exponentially decays with the number of training steps; it is reset to 1

[1] https://zenodo.org/records/15827810.
[2] https://github.com/ultralytics/yolov5.
[3] https://www.phonearena.com/phones/Samsung-Galaxy-S23_id11999.

at the beginning of every interval when the weights or the latency threshold are changed, to let the agent learn the new scenario. It is important to note that, since the environment changes are triggered by the users (e.g., depending on the battery level of users' devices), the learning rate can be updated accordingly. The discount factor γ in all experiments is 0.99

As explained in Appendix B.2, YOLOv5 offers 104 feasible configurations. To enhance the experiment robustness, we generated 10 random applications with a random number of configurations $|\mathscr{K}| \in [80, 104]$. Moreover, for each application (with a fixed configuration), we performed 10 runs of the RL agent considering 10 different random seeds. In the following, we will report the average results among all 10 applications and their 10 random runs.

For each configuration κ^i, the execution times t_1^i, t_2^i and t_3^i (see Section 3.1) are randomly sampled from a range having as mean the profiled value, and bounded by ±20%. This work considers both WiFi and 5G base traces as the foundation for generating additional traces across all application instances. The WiFi base trace, consisting of 3,000 rows, was obtained from profiling conducted as described above. On the other hand, the 5G base trace [34], derived from real-time measurements, comprised 11,024 data samples, with the maximum uplink throughput reaching 230.75 Mbps. The base traces were replayed to reach the total runtime of each experiment. Various operations were applied during replay to eliminate periodicity, including random shifts, insertion of random noise (±10% of the value at timestep t), and inversion of the trace in its middle. Different 5G and WiFi traces for each application are considered across the 10 runs. Additionally, for each partition p_3^i executed on the cloud, a random cloud latency is sampled using an exponential distribution with $\lambda = 1/t_3^i$, as in [45]. Examples of 5G, WiFi, and cloud latency traces are presented in Appendix B.1.

The agent control time period τ is set to 10 s but the next action is also triggered if 5 consecutive frames (sampled every L_{max}) incur in a latency threshold violation.

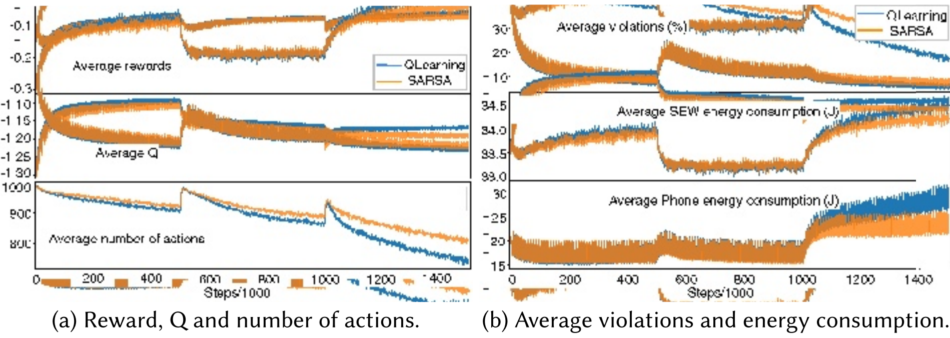
The action space of the agent for each application includes the number of corresponding configurations plus η (*do nothing*). We considered an MDP with an infinite horizon; however, for each experiment, we run the agent for a total of 1.5×10^6 steps, which we collectively refer to as an episode. We will call the periods $[0, 5] \times 10^5$, $[5, 10] \times 10^5$, and $[1, 1.5] \times 10^6$ the first, second, and third interval, respectively. In all experiments, we compute the reward as the negative of the cost.

5.2 Alternative Methods for Evaluation

To evaluate the effectiveness of our approach, we consider two baselines: the first one, denoted by *Opt**, is a static setup that represents the optimal solution of Problem (P1), when the average WiFi and 5G throughput values derived from the traces are considered. This setup enables us to evaluate whether dynamic runtime offloading provides any benefits over a static pre-computed solution.

The second baseline we consider is *Neurosurgeon* [23], a state-of-the-art algorithm that selects the optimal partition point based on the previous sample of the network's throughput. *Neurosurgeon* leverages prediction models that estimate the latency and power consumption of a DNN layer based on its specific type and configuration, and it evaluates each potential partition point to finally select the one that minimizes either total latency or EC, but not both simultaneously. In contrast, our approach incorporates both latency and EC into the agent decision-making process, weighting each factor according to its importance in the objective function.

Since *Neurosurgeon* uses a single partition point, executing part of the DNN on the mobile device and offloading the rest to either an edge or a cloud server, we extended it to support two partition points. This change enables handling three computational layers, facilitating a fair comparison with our approach.



(a) Reward, Q and number of actions. (b) Average violations and energy consumption.

Fig. 3. Experimental results when varying the importance weights (average across 10 applications and 10 instances each).

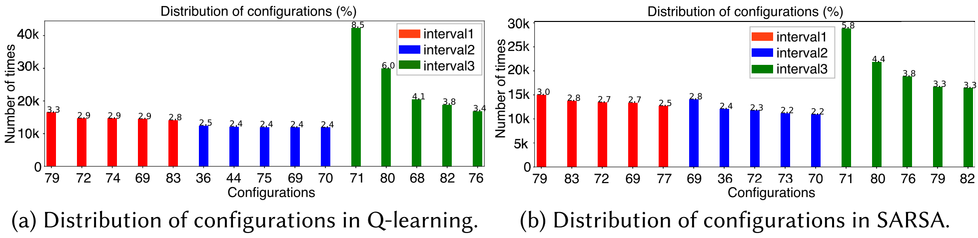
5.3 *Varying Importance Weights* Scenario

This section presents the results of a scenario where the latency constraint threshold L_{max} is set to 500 ms for all steps, while the cost-function importance weights (see Equation (14)) change in the three-episode intervals. These changes can be triggered by the operating system of the mobile phone when the available battery capacity drops below some specific thresholds or manually by the user to set his priority depending on the current battery state. This section evaluates whether the agent can learn considering these changes and maintain effective system control. In particular, we set $\omega_{lat} = 0.93$, $\omega_{e_{SEW}} = 0.03$, $\omega_{e_{phone}} = 0.02$, $\omega_{5G} = 0$, and $\omega_{rcfg} = 0.02$ in the first interval to put more emphasis on the execution time when the SEW is fully charged; in the second interval, since the battery capacity has decreased, we consider $\omega_{lat} = 0.4$, $\omega_{e_{SEW}} = 0.5$, $\omega_{e_{phone}} = 0.05$, $\omega_{5G} = 0.03$, and $\omega_{rcfg} = 0.02$ to slightly relax the latency constraint and shift the focus on SEW EC. Finally, in the third interval, we assume the SEW has been partially recharged; therefore we have $\omega_{lat} = 0.86$, $\omega_{e_{SEW}} = 0.07$, $\omega_{e_{phone}} = 0.02$, $\omega_{5G} = 0.03$, and $\omega_{rcfg} = 0.02$.

Figure 3 illustrates the comparison between Q-learning and SARSA obtained after training the agent over 1.5×10^6 steps, considering several significant metrics including the agent reward over time, the Q function value, the percentage number of violations, the number of reconfigurations, and the EC on SEW and phone. All the metrics are computed as a moving average over a window of 1,000 steps (recall that average values are computed across 10 applications and 10 random runs for each application as well). Note that Q-learning and SARSA provided better results compared with other methods in terms of violation; therefore, we only reported their results to enhance the figure clarity. A full comparison among the methods is shown in Appendix B.3.

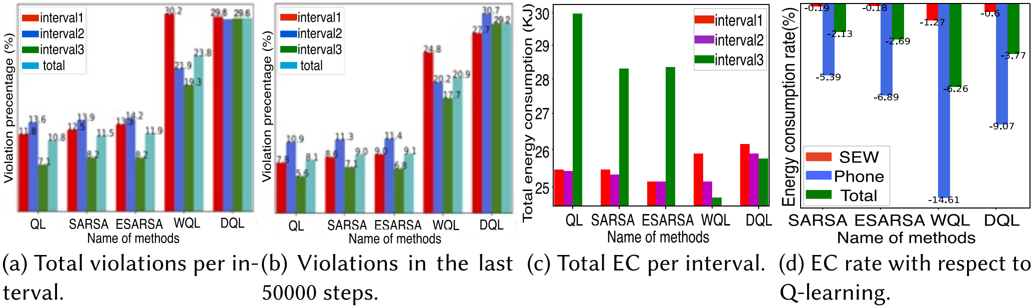
In the initial steps of the first interval, the lack of knowledge about system dynamics leads to increased exploration and a higher rate of violations (see Figure 3(b), first line), resulting in greater EC (see Figure 3(b), second line) and more frequent reconfigurations (see Figure 3(a), third line). This phase is also characterized by lower Q-values and rewards, as observed in Figure 3(a) (first and second line). However, after approximately 10^5 steps, the agent progressively learns to adapt, leading to a reduction in the violation rate to around 7%. This adaptation results in lower EC and fewer reconfigurations, ultimately yielding higher rewards.

At the beginning of the second interval, as the weight associated with latency constraint violations decreases and the weight for SEW EC increases, the agent adapts by reducing EC on the SEW while increasing EC on the phone, which is assigned a lower weight. Consequently, the reward experiences a slight decline, and the percentage of violations increases, reaching 27% in the initial steps ($[5, 5.5] \times 10^5$ in all plots of Figure 3). However, the agent gradually learns to select configurations that reduce



(a) Distribution of configurations in Q-learning. (b) Distribution of configurations in SARSA.

Fig. 4. Distribution of configurations while changing importance weights (single instance).



(a) Total violations per interval. (b) Violations in the last 50000 steps. (c) Total EC per interval. (d) EC rate with respect to Q-learning.

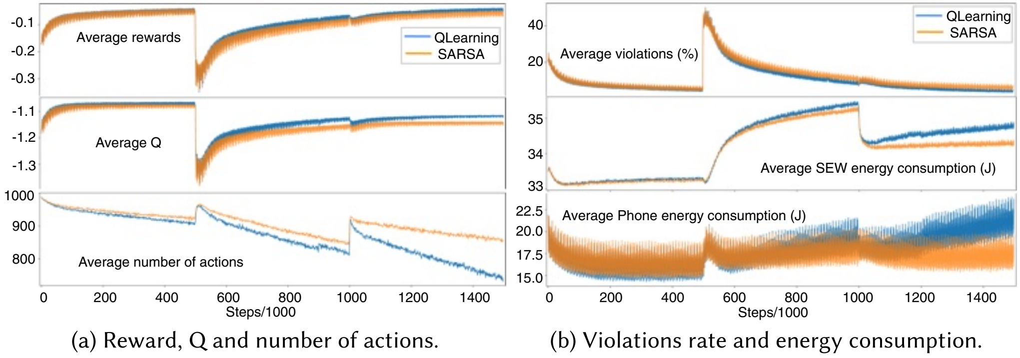
Fig. 5. Violation rates and EC at different intervals when changing the importance weights (average across 10 applications and 10 instances each).

both the number of violations and EC also on the phone (see steps after 5.5×10^5 in all plots of Figure 3). Finally, when the weight associated with violations increases again in the last interval, the agent compensates by increasing EC on both the SEW and the phone to optimize the reward.

Figure 4 provides a detailed view of the most popular actions by showing the five most frequently selected configurations for each interval. Using the first instance of the first application as a representative example (characterized by 84 candidate configurations and closely aligned with the average), the analysis reveals distinct patterns in agent behavior across the three intervals. In the first and third intervals, where ω_{lat} is dominant and emphasis is placed on minimizing execution time violations, configurations 68 to 83 are most frequently chosen by both Q-learning and SARSA agents. This preference reflects the agent strategy to execute the DNN as locally as possible to minimize transmission delays. As an example, configuration 71 emerges as the most popular choice in the third interval, running the DNN entirely on the SEW and phone to avoid 5G delays associated with cloud transmission. Conversely, in the second interval where energy efficiency is prioritized and the SEW EC weight $\omega_{e_{SEW}}$ becomes dominant, the agents adapt their selection strategy, favoring configurations like 36, 44. Configuration 36 exemplifies this shift, running only a small part of the DNN on SEW and phone while offloading the substantial portion of the DNN to prevent unnecessary EC on the SEW device.

Figure 5(a) and (b) shows the percentage number of violations in the whole episode and in the last 5×10^4 steps of each interval, respectively, for all methods. They also report the average over the three intervals, referred to as *total* in the legend.

According to Figures 3 and 5(b), we conclude that, as it can be expected due to the lack of any prior experience, the agent needs more steps to converge in the first interval, and it reaches a lower number of violations by consuming more energy on the SEW and phone. Indeed, despite the weight for violations is lower in the third interval than in the first interval, we experience more violations in the last 5×10^4 steps of the first interval than in the third one, while the constraint threshold



(a) Reward, Q and number of actions.

(b) Violations rate and energy consumption.

Fig. 6. Experimental results while changing execution time constraint (average across 10 applications and 10 instances each).

is the same. Figure 5(c) shows that the methods slightly reduce EC in the second interval. This occurs because the violation and EC weights are lower and higher, respectively, compared to the first interval, and the agent prioritizes minimizing EC to maximizing the overall reward. However, in the third interval, the methods significantly increase EC due to the updated weights. This last aspect is particularly true for Q-learning, due to its nature of determining the optimal action in each state.

Figure 5(a) shows that all methods experience the lowest percentage number of violations in the last interval, and that Q-learning is the method achieving the lowest values in all intervals. Figure 5(b) highlights that the agent reduces the violations between 21% and 25% in Q-learning, SARSA and Expected SARSA after 4.5×10^5 steps. Since Q-learning yields the lowest percentage number of violations but a similar reward value compared with SARSA and Expected SARSA, we computed the EC rate of other methods with respect to Q-learning (see Figure 5(d)) to have a more comprehensive comparison:

$$EC_rate = \frac{EC_{method} - EC_{Qlearning}}{EC_{Qlearning}} \times 100. \tag{15}$$

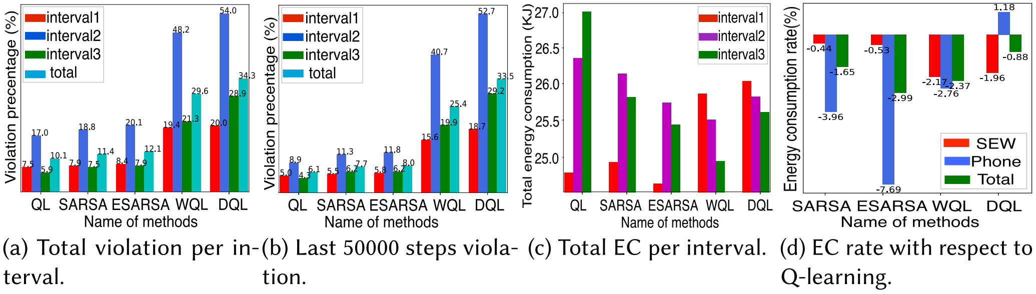
Comparing Figure 5(a) and Figure 5(d), we observe that, with respect to SARSA and Expected SARSA, Q-learning reduces the violations by 6.5% and 10%, respectively, at the expense of consuming 2.13% and 2.69% more energy.

5.4 *Varying Latency Constraint* Scenario

To assess how the agent adapts to different latency constraints, we change the threshold value L_{max} at the beginning of each intervals as: $L_{max} = 600, 400, 500$ ms. The importance weights are fixed during the whole episode, and set to $\omega_{lat} = 0.93$, $\omega_{e_{SEW}} = 0.03$, $\omega_{e_{phone}} = 0.02$, $\omega_{5G} = 0$, and $\omega_{rcfg} = 0.02$ (the same weights as the first interval analysed in the previous section).

Figure 6 illustrates the comparison between Q-learning and SARSA in terms of the significant metrics, averaged among all the 10 applications and 10 runs as in Section 5.3. The full comparison among the methods is shown in Appendix B.3. In the first interval, similarly to the previous scenario, the agent explores more during the early steps, but it converges faster and experiences less violations and EC because of a more relaxed latency constraint ($L_{max} = 600$) than in the first interval of Section 5.3.

At the beginning of the second interval, since the latency constraint becomes very strict, the reward experiences a sudden decline and the violation rate increases up to 50%. The agent tries

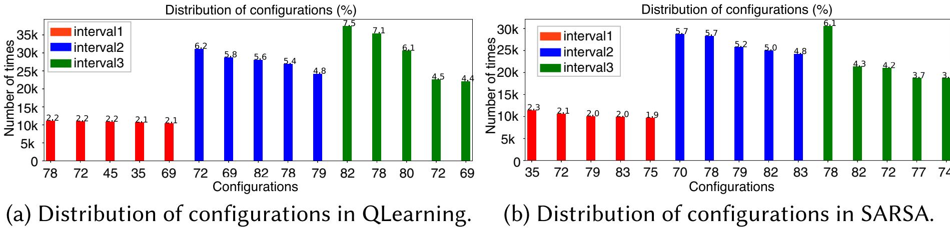


(a) Total violation per interval. (b) Last 50000 steps violation. (c) Total EC per interval. (d) EC rate with respect to Q-learning.

Fig. 7. Violation rates and EC at different intervals while changing L_{max} (average across 10 applications and 10 instances each).

to balance the reward by exploring more actions and increasing the EC: it first explores actions that consume more energy on the phone rather than the SEW (according to the values of the importance weights), but, after some steps (see steps $[5, 5.5] \times 10^5$ in Figure 6(b)), it realizes that the violations can be compensated only by choosing configurations that process the DNN mostly on SEW, avoiding the communication delay (and its variability) between SEW, phone, and cloud. Note that, in the first interval we observe a similar behaviour for SARSA and Q-learning because of the large weight on violations, while in the second interval we observe different behaviors because SARSA uses the current policy to select the next action to update Q-values while Q-learning selects the action that maximizes the Q-value in the next state. This leads Q-learning to select configurations that are mostly run on SEW and phone, increasing the EC and decreasing violations, while SARSA tends to be less extreme or conservative compared to Q-learning due to its on-policy nature. Finally, when the constraint is relaxed at the beginning of the last interval, the agent can significantly decrease the EC in both SEW and phone, while only slightly decreasing the reward. After some steps, the Q-learning agent learns that, if it runs the DNN mostly on the phone, it can cope with the constraint, consequently consuming more energy on the phone and less on SEW compared with the second interval. SARSA does not necessarily follow the maximum Q-value policy, which leads to a lower EC and to more violations.

Figure 7 compares all the tabular methods in terms of average violations and EC. In particular, Figure 7(a) highlights that all methods but Weighted Q-learning and double Q-learning observe the lowest percentage number of violations in the last interval, even though L_{max} is lower than in the first interval. This shows that the agent is trained well particularly in the second interval because of the strict constraint. Additionally, it illustrates that Q-learning has the lowest violation rate, both per-interval and overall, among all methods. Figure 7(b) shows the violation percentage during the last 5×10^5 steps of each interval, and the average of the three intervals, referred to as *total* in the legend. This figure exhibits that the agent manages to reduce the violation percentage between 32% and 40% in Q-learning, SARSA and Expected SARSA after 4.5×10^5 learning steps per interval, which is an acceptable improvement. In Figure 7(c), we analyze the EC of all methods across each interval. For SARSA and Expected SARSA, EC is higher in the second interval due to the stricter constraints imposed during that phase. In contrast, for Q-learning, the maximum EC occurs in the third interval, driven by the policy selection of the maximum Q-value action. Comparing the violation rates in Figure 7(a) and the EC rate of all methods with respect to Q-learning (Equation (15)) in Figure 7(d), we observe that Q-learning reduces the violations by 1.3% and 2% while consuming 1.65% and 2.99% more energy compared with SARSA and Expected SARSA, respectively. According to all results, Weighted Q-learning and Double Q-learning did not manage to achieve acceptable performance and they need much more steps to converge to a near-to-optimal policy.



(a) Distribution of configurations in QLearning. (b) Distribution of configurations in SARSA.

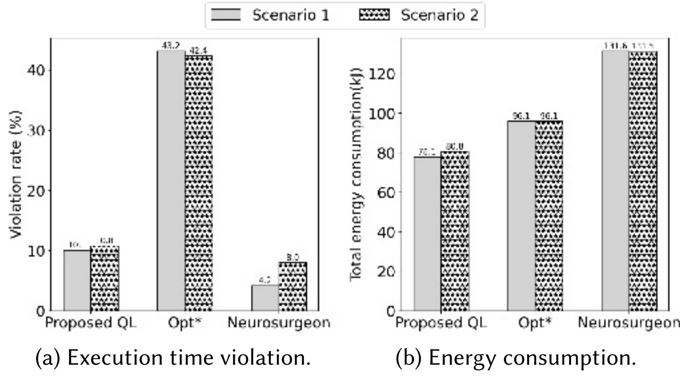
Fig. 8. Distribution of configurations, with changing L_{max} (single instance).

Figure 8 shows the five most popular configurations per interval related to the first instance of the first application. In the first interval, where the execution time constraint is relaxed and most configurations can satisfy it, the agent exhibits broad exploration with relatively even distribution across many actions (see Figure 8, *interval 1*). The most popular configuration is selected only 3.3% of the time and the fifth most popular 2.8%, indicating that a large fraction of the state–action space is visited. However, as constraints tighten in the second interval, the agent behavior becomes more focused and strategic, bringing more attention onto configurations 69–84, which correspond to those running the DNN mostly on the SEW device. For Q-learning, the top-five configurations account for 28% of all choices in this interval. Configuration 72 emerges as the most popular: it runs mostly on the SEW, nothing on the phone, and only a tiny part on the cloud to meet the strict timing requirements (see Figure 8(a), *interval 2*). In the third interval, when constraints are relaxed again, the range of visited state–action pairs expands to configurations 45–84, although the distribution remains more focused than in the first interval, with the top-five configurations representing 30% of all choices for Q-learning. The most popular configurations in this final interval are 82 and 78 (see Figure 8, *interval 3*) for both methods. Configuration 82 runs the DNN entirely on SEW and phone (nothing on the cloud), while configuration 78 runs mostly on SEW with two tiny parts on phone and cloud. These patterns demonstrate that the agent learns to strategically populate the Q-table, visiting a broader fraction of state–action pairs when constraints permit flexibility, but concentrating on the most effective subset when constraints become restrictive.

5.5 Comparison with Opt* and Neurosurgeon

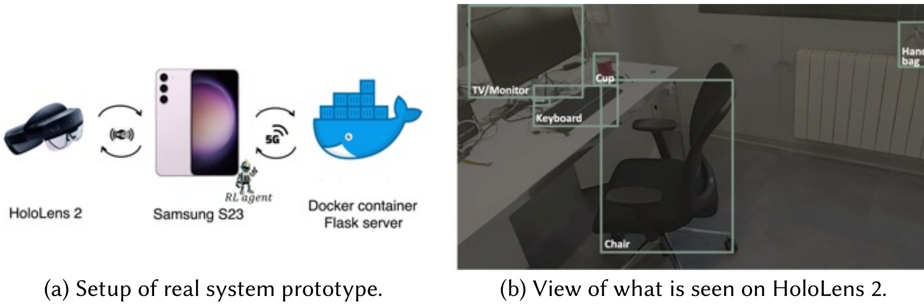
In this section, we compare our best-performing method, referred to as *Proposed QL*, with the *Opt** and *Neurosurgeon* [23] baselines described in Section 5.2. Figure 9 presents comparison results of the three approaches considering two scenarios. Scenario 1 is latency-oriented and involves varying execution time thresholds as described in Section 5.4 (i.e., taking values in [600, 400, 500] ms), while setting the importance weights to $\omega_{lat} = 0.93$, $\omega_{e_{SEW}} = 0.03$, $\omega_{e_{phone}} = 0.02$, $\omega_{5G} = 0$, and $\omega_{rcfg} = 0.02$. Scenario 2 is an energy-aware scenario in which the execution time threshold is fixed at 500 ms, while *Proposed QL* adjusts the importance weights as detailed in Section 5.3. *Neurosurgeon* objective is latency minimization in Scenario 1, while it is the minimization of total EC in Scenario 2. For *Opt**, as the configuration remains static, we compute the execution time violation rate and EC in both scenarios. Recall that both the execution time violations and EC were computed using the same WiFi and 5G traces, as described in Section 5.1.

For Scenario 1, Figure 9(a) shows that *Opt** performs the worst in terms of execution time violations, with a violation rate of 43.2%. In contrast, both *Proposed QL* and *Neurosurgeon* demonstrate significantly better performance, with violation rates of 10.1% and 4.2%, respectively. Regarding energy efficiency, Figure 9(b) reveals that *Proposed QL* is the most efficient, consuming 41% and 19% less energy compared to Neurosurgeon and *Opt**, respectively. In Scenario 2, considering both EC and execution time violation rate, *Proposed QL* proves to be the best-performing method. It



(a) Execution time violation. (b) Energy consumption.

Fig. 9. Comparison of Proposed QL, Neurosurgeon, and Opt* considering execution time violation rate and EC for latency-oriented scenario (Scenario 1) and energy-aware scenario (Scenario 2).



(a) Setup of real system prototype. (b) View of what is seen on HoloLens 2.

Fig. 10. Setup of real system prototype and example of the HoloLens view.

achieves a violation rate similar to *Neurosurgeon* (10.8% versus 8%), while maintaining a significantly lower violation rate than *Opt** (see Figure 9(a)). Furthermore, *Proposed QL* ensures substantial energy savings, consuming 39% and 16% less energy than *Neurosurgeon* and *Opt**, respectively (see Figure 9(b)). In conclusion, *Proposed QL* demonstrates superior performance across both metrics, offering the best tradeoff between execution time violations and EC. It effectively captures system dynamics, outperforming the other methods. While *Neurosurgeon* is the most energy-intensive, *Opt** results inadequate for this problem, as it fails to account for system dynamics, inducing higher violation rates.

5.6 Experiments on Prototype Environment

This section presents the results obtained by executing the trained agent on a real system, to evaluate its performance and ascertain whether it provides the anticipated benefits as described in the previous section. It is important to note that only limited studies have addressed this aspect [12], which is crucial to provide a clear understanding of how the simulation results can be applied to actual systems and how the agent can be utilized in production. We focused on the Q-learning method, identified in Section 5.3 as the optimal approach.

The actual system configuration, illustrated in Figure 10(a), precisely aligns with the description provided in Section 5.1. The YOLOv5 object-detection task follows an execution sequence that begins with image capture by the HoloLens 2 camera and ends with the visualization of bounding boxes on its display as seen in Figure 10(b).

We used the Samsung S23 WiFi hotspot to establish a wireless connection between different entities. This phone has WiFi6E capability to connect different devices using WiFi6E over the 5 GHz band, with a maximum throughput reaching 1,000 Mbps. To ensure compatibility between Samsung S23 and the third device regarding 5G network throughput, we clipped the connection throughput according to the maximum 5G throughput reported in the real trace presented in the Appendix. Specifically, the 5G throughput value is computed by multiplying the peak value observed in the 5G base trace by the ratio of the actual WiFi6 throughput to its maximum value. Consequently, during data transmission from Samsung S23 to the PC, if the real WiFi6 throughput exceeds the computed value, the server on the PC introduces a delay with a predetermined duration to accommodate the disparity between the real WiFi6 throughput and the computed value. This approach ensures that the experiment accurately reflects the compliance with 5G network specifications in the connection between the second and third computational layers.

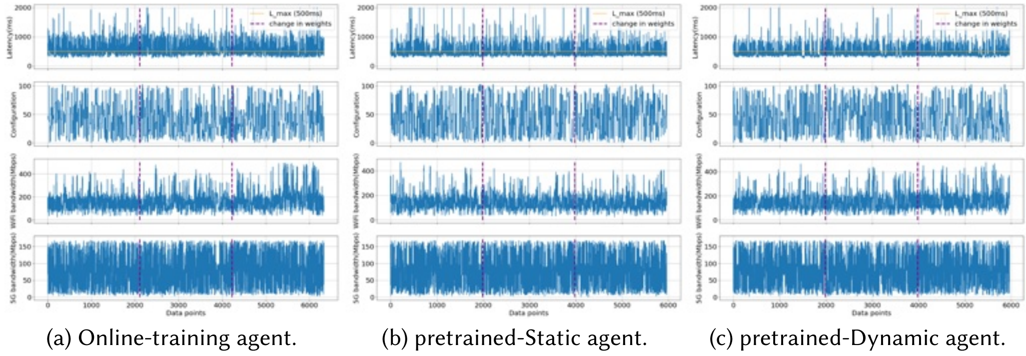
We considered an execution time threshold L_{max} = 500 ms and adjusted the importance weights to accommodate various setups, following the procedures implemented during the training phase (Section 5.3). As already mentioned, these adjustments account for when there are changes in the battery capacity of the SEW or the phone. To effectively adapt the agent to the new scenario, we profiled the DNN on the real system to evaluate the determinism of execution time across different devices, thereby improving training efficiency. The profiling results revealed that the coefficient of variation in execution time across all partitions consistently remained below 20%. Consequently, we trained the agent using Q-learning, considering all 104 possible configurations of YOLOv5n (YOLOv5 *nano*), while utilizing the WiFi trace from the profiling and incorporating the 5G trace from [34]. The training process consisted of 1.5 million steps, considering $\varepsilon = 0.05$. At the end of the offline training, the total violation achieved is 13.2%.

We evaluated the trained agent in terms of latency constraint violation and EC. The evaluation encompasses four main scenarios: the first one involves a randomly initialized agent, referred to as the *Online-training agent*, which begins the learning process from scratch. The second scenario leverages an agent whose Q-table is initialized with the values from the training process, but which is not updated thereafter (*pretrained-Static agent*). The third scenario follows the same initialization process as the second one, but the agent continues to learn during the experiment, leading to updates in its Q-table (*pretrained-Dynamic agent*). The fourth scenario involves no DNN partitioning, while the entire DNN is executed locally on the HoloLens 2 (*All local*).

For each scenario, the experiment comprises three intervals, each lasting 20 minutes. At the beginning of each interval, the importance weights change as described in Section 5.3. The agent control time (τ) is set to 10 seconds, ensuring that a new action is requested either upon the lapse of the control time or when there are five consecutive violations within the control period.

The following sections are organized as follows: Sections 5.6.1 and 5.6.2 focus on latency constraint violations and EC analyses, respectively. They underscore the *pretrained-Dynamic* and *pretrained-Static* agents' capabilities in minimizing latency violations and saving energy, surpassing both the *Online-training* agent and the *All local* configuration. Section 5.6.3 explores how reducing the number of alternative configurations, which also decreases the number of actions and the agent memory requirements, affects the agent's performance. This emphasizes the critical importance of configuration optimization in real system deployment.

5.6.1 Assessment of Latency Constraint Violations. Figure 11 provides insights onto the latency constraint violations and the configurations chosen during the experiment for each scenario. Additionally, it includes the WiFi and 5G throughputs. From Figure 11(a), we observe that the *Online-training* agent exhibits more violations compared to the others due to the fact that it starts



(a) Online-training agent. (b) pretrained-Static agent. (c) pretrained-Dynamic agent.

Fig. 11. Latency, reconfigurations, 5G, and WiFi throughputs for *Online-training*, *pretrained-Static*, and *pretrained-dynamic* agents on real system with 104 configurations in prototype environment. Purple dashed lines indicate the end of an interval, where importance weights are changed.

Table 1. Total and Per-Interval Execution Time Violation and Energy Consumption for *Random*, *Static*, *Dynamic*, *All Local* Agents with 104 Configurations in Prototype Environment

	Execution time violation (%)				**SEW energy consumption (kJ)**				**Phone energy consumption (kJ)**			
Agent	Violation per interval			Total	Energy per interval			Total	Energy per interval			Total
	First	Second	Third		First	Second	Third		First	Second	Third	
Random	45.1	45.8	41.5	44.2	7.2	6.9	6.6	20.7	2.1	2.2	2.1	6.4
Static	22.5	23.7	22.9	23.0	5.0	4.7	6.7	16.4	1.5	**1.6**	2.5	5.6
Dynamic	17.8	23.7	20.5	20.6	**4.8**	**4.4**	**4.6**	**13.8**	**1.4**	**1.6**	**1.5**	**4.5**
All local	**0.0**	**0.0**	**0.0**	**0.0**	9.2	9.2	9.2	27.6	-	-	-	-

The bold values are the best value among the other in the same column.

from a random initialization and explores actions that may not be beneficial. The *pretrained-Static* agent, on the other hand, follows the policy learned from the profiled data. As shown in Figure 11(b), it experiences fewer violations than the *Online-training* agent since it already possesses some knowledge and selects actions according to the previously learned policy. Finally, Figure 11(c) demonstrates that the *pretrained-Dynamic* agent exhibits fewer violations compared to the *Online-training* agent and *pretrained-Static* agent.

These observations are supported by the results presented in Table 1, where the *pretrained-Dynamic agent* consistently outperforms the *Online-training* and *pretrained-Static agent* in all three intervals. Overall, the *pretrained-Dynamic agent* incurred a violation rate of 20.6%, compared to 22.9% for the *pretrained-Static agent* and 44.2% for the *Online-training agent*. This indicates that the *pretrained-Dynamic agent* surpasses the *pretrained-Static agent* as it continues to learn from interactions with the real system and can adjust its policy online. In contrast, the violations remain relatively constant across the three intervals for the *pretrained-Static agent* since the policy is fixed and no real system information is learned. *All local* achieves 0% violation rate, as the execution time averaged 450 ms and remains consistently below 500 ms. However, this comes at the cost of consuming twice the energy compared to other scenarios, as it will be detailed in Section 5.6.2.

5.6.2 Assessment of EC. We assessed the performance of the trained agent on the real system analyzing the EC of both the Hololens 2 (representing the SEW) and the mobile phone. As in the previous section, we analyzed the performance of the *Online-training*, *pretrained-Static*, *pretrained-Dynamic* agents, and the *All local* configuration.

As discussed in Section 3.4, the EC on a device consists of the energy consumed for computations on the device itself and the energy consumed when transmitting data through the network interface. In this study, we rely on energy profiling on the real system to estimate the device parameters, namely the energy consumed per FLOP (z_{SEW} and z_{phone}) and the power consumed by the WNI (θ_{SEW} and θ_{phone}). The profiling involved determining the energy consumed when executing the partitions of YOLOv5n on the SEW (the phone) without forwarding any data to the subsequent device. From the profiling results, the values of z_{SEW} and z_{phone} are 1085.1 pJ/FLOP and 112.5 pJ/FLOP, respectively. Furthermore, the profiling aimed to estimate EC when no operations were performed on the SEW (the phone), but intermediate tensors were transmitted to the subsequent device, facilitating the evaluation of data transfer power consumption. Based on the profiling results, θ_{SEW} and θ_{phone} were estimated to be 7.9 W and 4.5 W, respectively. More details about the profiling are provided in Appendix B.2.

Table 1 presents the EC data (in kilojoules (kJ)) for SEW and phone across different time intervals and scenarios. Notably, we focus on comparing the EC while changing the importance weights and across different scenarios.

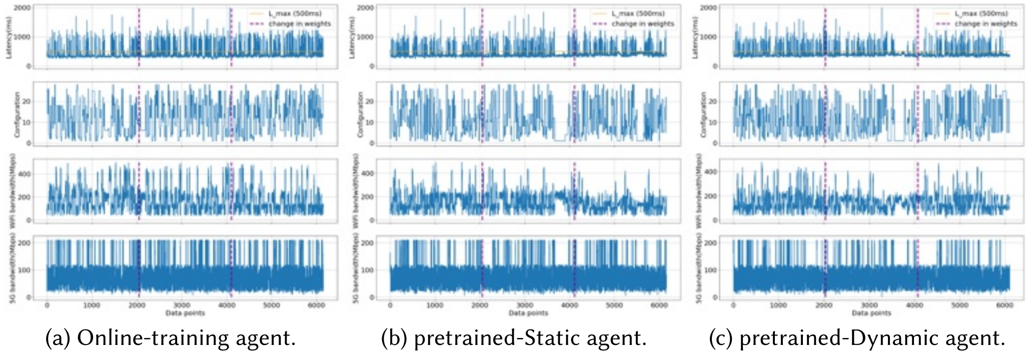
We observe that the SEW EC decreases in the second interval, due to a significant increase in the corresponding weight (from $\omega_{e_{SEW}} = 0.03$ in the first interval to $\omega_{e_{SEW}} = 0.5$). In the third interval, the EC varies among the agents: it increases for the *pretrained-Static* and *pretrained-Dynamic* agents but decreases for the *Online-training* agent.

Comparing the other agents with the *All local* scenario, Table 1 demonstrates that partitioning is significantly more energy-efficient than executing the entire DNN locally. The *pretrained-Dynamic* agent exhibits the highest energy savings on the SEW, reducing EC by 50% compared to the *All local* configuration, effectively doubling the battery life of the Hololens 2. The *Online-training* and *pretrained-Static* agents achieve savings of 25% and 40.6%, respectively, relative to the *All local* scenario. While the *Online-training* agent outperforms the *All local* configuration in energy savings, its exploratory nature results in more execution time violations, making it less advantageous compared to the other agents.

For the mobile phone, the *pretrained-Dynamic* agent proves to be the most energy-efficient, consuming 4.5 kJ of energy, compared to 5.6 kJ and 6.4 kJ for the *pretrained-Static* and *Online-training* agents, respectively. Overall, considering the combined EC of the SEW and the mobile phone, the *pretrained-Dynamic* agent emerges as the most energy-efficient option, with total consumption of 18.3 kJ, compared to 22 kJ, 27.1 kJ, and 27.6 kJ for the *pretrained-Static*, *online-training*, and *all local* scenarios, respectively.

5.6.3 Experiment with Reduced Number of Configurations. While conducting experiments on the real system, we observed that the agents face significant challenges in recovering from violations due to the large number of candidate configurations. To mitigate this issue, we implemented a strategy to reduce the action space by including only those actions that were selected at least 80% of the time during the initial training. This resulted in a reduced action space of 30 candidate configurations (out of the original 104), which not only minimizes violations but also decreases the memory requirements on the mobile phone where the agent runs, as the memory size of the Q-table decreased from 110 MB to 9.2 MB. A scalability analysis with respect to the number of configurations is provided in Appendix B.4.

Subsequently, the agent was retrained using this reduced action space, utilizing the same traces and conditions as in the initial training. As a result, the average latency constraint violations decreased to 8.7% after the offline training. The experiment was then repeated on the prototype system, and the corresponding plots illustrating the violations, chosen configurations, the 5G and WiFi throughputs are reported in Figure 12.



(a) Online-training agent. (b) pretrained-Static agent. (c) pretrained-Dynamic agent.

Fig. 12. Latency, configurations, 5G and WiFi throughput of Online-training, pretrained-Static and pretrained-Dynamic agents on real system with 30 configurations in prototype environment.

Table 2. Total and Per-Interval Execution Time Violation and Energy Consumption for *Random*, *Static*, *Dynamic*, *All Local* Agents with 30 Configurations in Prototype Environment

Agent	**Execution time violation (%)**				**SEW energy consumption (kJ)**				**Phone energy consumption (kJ)**			
	Violation per interval			Total	Energy per interval			Total	Energy per interval			Total
	First	Second	Third		First	Second	Third		First	Second	Third	
Random	18.4	18.1	20.66	19.0	**4.6**	**4.4**	**4.8**	**13.8**	**1.3**	1.5	1.6	4.4
Static	11.9	10.5	12.9	11.8	4.7	4.7	5.3	14.7	1.4	**1.4**	1.5	**4.3**
Dynamic	13.3	9.5	11.8	11.5	5.1	4.9	**4.8**	14.8	1.5	**1.4**	**1.4**	**4.3**
All local	**0.0**	**0.0**	**0.0**	**0.0**	9.2	9.2	9.2	27.6	-	-	-	-

The bold values are the best value among the other in the same column.

Comparing the performance of the *Online-training agent*, the *pretrained-Static*, and *pretrained-Dynamic* in Figure 12 to the one on the 104 configurations scenario, we observe that the average latency constraint violations decrease for all three agents. Specifically, the *Online-training*, *pretrained-Static*, and *pretrained-Dynamic* agents achieve total violations of 19.0%, 11.8%, and 11.5%, respectively, confirming the superior performance of the *pretrained-Dynamic agent* in this regard.

Moreover, the *pretrained-Dynamic* and *pretrained-Static* agents exhibit similar EC on the SEW, at 14.8 kJ and 14.7 kJ, respectively, while the *Online-training* agent consumes slightly less energy at 13.8 kJ, as shown in Table 2. In comparison to the *All local* scenario, all three alternative scenarios consistently demonstrate superior energy-saving capabilities, aligning with the results observed across 104 configurations. Specifically, the *pretrained-Dynamic* agent achieves a 46.4% reduction in EC compared to *All local*, while the *pretrained-Static* and *Online-training* agents save 46.7% and 50%, respectively. On the mobile device, EC is nearly identical across the three agents.

Overall, when considering both devices, the *pretrained-Dynamic* and *pretrained-Static* agents consume 8.6 kJ less energy than the *All local* scenario. Notably, the 30-configuration scenario offers the best balance between EC and execution time violations, providing an optimal tradeoff compared to the 104-configuration scenario. This underscores the importance of configuration optimization in reducing the violations.

5.6.4 Discussion. To summarize, the findings in this section demonstrate the practical feasibility of employing the trained agent in practical systems. It is also advantageous to start from a pre-learned policy as it diminishes both latency constraint violations and EC. Additionally, utilizing a

pretrained-Dynamic agent offers several benefits as it continues to learn and update its knowledge based on the real system state. Furthermore, we noted that reducing the number of configurations by selecting those that are more likely to be chosen effectively reduces violations, since the agent can more promptly recover from actions that result in higher violations.

6 Related Work

Smart devices have enabled the widespread integration of AI applications in various facets of daily life. However, challenges arise when implementing these applications on systems with constrained computational and energy resources. In response to this challenge, researchers have explored techniques such as DNN partitioning and offloading resource-intensive computations to either edge devices or cloud resources. This section briefly reviews the state of the art concerning DNN partitioning (Section 6.1), and resource management and task offloading (Section 6.2, with a particular focus on RL in Section 6.3).

6.1 DNN Partitioning

Authors in [23] present Neurosurgeon, a layer partitioning model based on edge-cloud computing collaboration. This approach involves the deployment of neural networks on both mobile phones and cloud servers, while dynamically adjusting the model partitioning based on the network bandwidth or EC, depending on the specific objective. A similar work in [27] proposes an algorithm to identify the most efficient partitioning point within a multi-device and multi-edge server configuration. This algorithm takes into account the minimum cost associated with each edge server for every mobile device. Authors used a partitioning point retain algorithm to reduce the search space and run a procedure to find the optimal partitioning point considering the different capabilities of the mobile device and edge server. Differently, authors in [5] considered sparse neural networks and suggested a hypergraph model for partitioning matrices to enhance the scalability and efficiency of parallel sparse matrix-vector multiplications computations. Their approach reduces communication costs and promotes a balanced distribution of computational load across processors which they showed as an advantage for stable training and inference. All the previous studies have predominantly adopted a two-tier partitioning model, where computational tasks are solely distributed between the edge device and an edge/cloud server, and for most of them, the validation is often based on simulation. However, these approaches neglect the potential benefits of a three-tier architecture, where tasks can be dynamically orchestrated across edge devices, edge/fog servers, and a remote cloud server depending on network conditions.

6.2 Resource Management and Task Offloading

Numerous studies tackle the resource allocation and task offloading problems in edge-cloud computing, aiming to achieve equitable distribution among various entities while minimizing expenses, to guarantee acceptable performance levels [8, 13, 21, 47]. The work in [47] introduces a reference system involving multiple users and servers, and proposes an adaptive multi-objective algorithm designed to minimize power consumption, response times, and costs. This is accomplished by implementing a two-tier matching game, which effectively addresses resource optimization.

Authors in [8] propose an alternative solution to solve a similar two-tier matching game. The first tier solves the association problem between the users and edge servers to maximize social welfare, and the second tier focuses on collaboration among the edge servers to minimize the resource cost and transfer delay. Although this study considers the collaboration between different edge servers, a task can only be served by one VM on a single edge server, limiting the problem to a two-tier binary offloading scheme. Authors in [21] optimize computation configuration and bandwidth allocation in **Multi-Access Edge Computing (MEC)** for enhanced user experience, particularly in

energy-constrained environments. It presents a joint offloading and resource allocation framework, dynamically allocating resources to mobile devices based on task demands and energy constraints.

Although the mentioned studies yield promising results, the authors consider only resource management for generic tasks with no specific solutions for AI applications. On the other hand, Authors in [13] delve into the exploration of DNN model partitioning and offloading strategies for multiple users and propose an evolutionary pricing strategy and a slot-based model to optimize the efficient distribution of DNN subtasks. Although this work combines DNN partitioning and task offloading and minimizes the cost on mobile devices, it considers only a two-tier architecture and overlooks the variability of edge server's workload.

6.3 Task Offloading RL-Based Methods

Recently, RL has gained extensive popularity in the realm of task offloading and runtime resource management. For instance, authors in [19] introduce the DROO framework to optimize **wireless devices (WD)** task offloading decisions and time allocation among WDs in wireless powered MEC networks. The authors use DRL to maximize the weighted sum of computation rates for all WDs, improving network performance by leveraging past offloading experiences to refine action generation. In [46], a model-free distributed algorithm leveraging DRL is presented to handle dynamic load variations at edge nodes, enabling mobile devices to independently decide on offloading tasks. Focusing on non-divisible, time-sensitive tasks, the aim is to minimize long-term costs, factoring in task delays and penalties for task drops. However, both [19] and [46] consider a binary offloading strategy and overlook the EC concerns.

Conversely, authors in [16] developed a task-offloading algorithm that relies on DRL and takes into account both logical and data dependencies among subtasks within user applications. Although designed for scalability and capable of handling complex tasks in multi-user and edge computing environments, this approach overlooks the dynamic nature of server workloads, assuming that a single edge server supports multiple devices without accommodating server workload variations. Furthermore, the binary offloading mechanism restricts task execution to either exclusively local or remote, limiting flexibility and potentially underutilizing available resources. Authors in [20] tackle the fog-cloud task offloading problem using various optimization metrics. They formulated this problem as an MDP to enhance the average resource utility. The considered metrics encompass factors such as task execution time, total service latency, transmission electric power, and task priority. The authors used a DRL-based algorithm and performed an offline training of a multi-agent model on the cloud data center, while the real-time offloading decisions are performed online on the fog. Nonetheless, similar to most of the studies presented in this subsection, the validation results are based on simulation. In [37], authors proposed a DRL-based algorithm to tackle the challenges of computation offloading and channel allocation in MEC environments. Unlike previous methods, this approach utilized a collaborative distributed training scheme and N-steps learning, resulting in faster convergence, stable training, and reduced exploration costs. Their method optimizes long-term performance across diverse applications and is validated through experiments on a real testbed. However, it does not address the partitioning of AI applications and is limited to binary offloading. Table 3 summarizes the key features of the various works presented in this section, emphasizing the objective metrics considered (latency, EC), system factors (network and cloud variability), the adopted approaches (RL, DNN partitioning), and the evaluation setup (real system or simulation).

In this work, we proposed a novel approach for runtime offloading relying on a DNN partitioning model that operates at the layer level. Our strategy harnesses the capabilities of several tabular RL methods. The proposed approach considers the fluctuating conditions of 5G and WiFi bandwidth, as well as cloud latency. The key innovation of our work lies on addressing the real-time management of

Table 3. Summary of the Comparison of Our Approach with Related Work

Work	DNN partitioning	3-tier partitioning	Based on RL	Network variability	Cloud variability	Energy-aware	Latency-aware	Real system validation
Demirci et al. [5]	✓						✓	
Liao et al. [27]	✓			✓		✓	✓	
Du et al. [8]				✓		✓	✓	
Wang et al. [47]				✓		✓	✓	
Jiang et al. [21]				✓		✓	✓	
Qiu et al. [37]				✓		✓	✓	✓
Gao et al. [13]	✓			✓	✓	✓		
Huang et al. [19]			✓	✓			✓	
Tang et al. [46]			✓		✓		✓	
Gong et al. [16]			✓			✓	✓	
Jain et al. [20]			✓			✓	✓	
Kang et al. [23]	✓			✓	✓	✓	✓	✓
Our approach	✓	✓	✓	✓	✓	✓	✓	✓

AI tasks, focusing on enhancing the battery life of SEW devices and minimizing 5G-communication cost, all while maintaining an acceptable end-to-end latency to ensure a satisfactory user experience. In particular, this work extends our previous research [22], which focused solely on the EC of the SEW under varying importance weights using Q-learning. In this study, we additionally consider the EC of the phone and conduct extensive comparisons of multiple tabular methods. The best-performing method is further evaluated against two baselines. We also analyze the agent behavior in scenarios with varying maximum latency constraints to assess how it can adapt to user application constraints. Moreover, the evaluation of the trained RL agent on a real-system prototype utilizing Microsoft HoloLens 2 offers valuable insights into its potential for practical deployment. To the best of our knowledge, and as evidenced by Table 3, this is the first comprehensive study to address these challenges.

7 Conclusion

In this study, we tackled the runtime management of AI applications running on next-generation SEWs using various tabular RL methods. Q-learning emerged as the standout performer, exhibiting fewer execution time violations (approximately 10.0% on average across all simulations) and greater stability compared to other algorithms. Additionally, our approach offers a more favorable tradeoff between energy efficiency and execution time violations compared to the two baseline methods considered. We assessed the performance of the Q-learning agent on a real-system prototype, comparing outcomes from running the entire DNN locally with those of random, static, and dynamic agents. Results indicate that the static and dynamic agents are more energy-efficient than the all-local scenario. Experiments on the real-system prototype demonstrated our approach ability to double battery life while maintaining a satisfactory quality of experience, achieving an execution time violation rate of 11%, even under transient conditions (changes in the thresholds or weights) and the significant variability in network traces.

For future work, we aim to explore the potential of DRL algorithms, leveraging their generalization capabilities to potentially yield superior results.

Acknowledgment

This study was carried out in the EssilorLuxottica Smart Eyewear Lab, a Joint Research Center between EssilorLuxottica and Politecnico di Milano.

References

[1] Danilo Ardagna, Giuliano Casale, Michele Ciavotta, Juan F. Pérez, and Weikun Wang. 2014. Quality-of-service in cloud computing: Modeling techniques and their applications. *J. Internet Serv. Appl* 5, 1, Article 11 (2014), 1–17. DOI: https://doi.org/10.1186/s13174-014-0011-3

[2] Allysa Kate Brillantes, Edwin Sybingco, Argel Bandala, Robert Kerwin Billones, Alexis Fillone, and Elmer Dadios. 2022. Vehicle tracking in low frame rate scenes using instance segmentation. In *Proceedings of the IEEE 14th International Conference on Humanoid, Nanotechnology, Information Technology, Communication and Control, Environment, and Management (HNICEM)*, 1–5. DOI: https://doi.org/10.1109/HNICEM57413.2022.10109390

[3] Ramraj Dangi, Praveen Lalwani, Gaurav Choudhary, Ilsun You, and Giovanni Pau. 2022. Study and investigation on 5G technology: A systematic review. *Sensors (Basel, Switzerland)* 22, 1 (2022), 26. DOI: https://doi.org/10.3390/S22010026

[4] Olivier Debauche, Saïd Mahmoudi, and Adriano Guttadauria. 2022. A new edge computing architecture for IoT and multimedia data management. *Information* 13, 2 (2022), 89.

[5] Gunduz Vehbi Demirci and Hakan Ferhatosmanoglu. 2021. Partitioning sparse deep neural networks for scalable training and inference. In *Proceedings of the 35th ACM International Conference on Supercomputing (Virtual Event, USA) (ICS '21)*. ACM, New York, NY, 254–265. DOI: https://doi.org/10.1145/3447818.3460372

[6] Cailian Deng, Xuming Fang, Xiao Han, Xianbin Wang, Li Yan, Rong He, Yan Long, and Yuchen Guo. 2020. IEEE 802.11be-Wi-Fi 7: New challenges and opportunities. *IEEE Communications Surveys and Tutorials* 22, 4 (2020), 2136–2166. DOI: https://doi.org/10.48550/arxiv.2007.13401

[7] X. Dong, B. De Salvo, M. Li, C. Liu, Z. Qu, H. Kung, and Z. Li. 2022. SplitNets: Designing neural architectures for efficient distributed computing on Head-Mounted systems. In *Proceedings of the 2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE Computer Society, 12549–12559. DOI: https://doi.org/10.1109/CVPR52688.2022.01223

[8] Yu Du, Jun Li, Long Shi, Tingting Liu, Feng Shu, and Zhu Han. 2022. Two-Tier matching game in small cell networks for mobile edge computing. *IEEE Transactions on Services Computing* 15, 1 (2022), 254–265. DOI: https://doi.org/10.1109/TSC.2019.2937777

[9] C. D'Eramo, M. Restelli, and A. Nuara. 2016. Estimating maximum expected value through gaussian approximation. In *Proceedings of the 33rd International Conference on Machine Learning*, Vol. 48, PMLR, 1032–1040. Retrieved from https://proceedings.mlr.press/v48/deramo16.html

[10] Ersin Elbasi, Nour Mostafa, Zakwan AlArnaout, Aymen I. Zreikat, Elda Cina, Greeshma Varghese, Ahmed Shdefat, Ahmet E. Topcu, Wiem Abdelbaki, and Shinu Mathew. 2023. Artificial intelligence technology in the agricultural sector: A systematic literature review. *IEEE Access* 11 (2023), 171–202. DOI: https://doi.org/10.1109/ACCESS.2022.3232485

[11] Amir Erfan Eshratifar, Mohammad Saeed Abrishami, and Massoud Pedram. 2021. JointDNN: An efficient training and inference engine for intelligent mobile cloud computing services. *IEEE Transactions on Mobile Computing* 20, 2 (2021), 565–576. DOI: https://doi.org/10.1109/TMC.2019.2947893

[12] Enrico Galimberti, Bruno Guindani, Federica Filippini, Hamta Sedghani, Danilo Ardagna, Moltó Germán, and Caballer Miguel. 2023. OSCAR-P and aMLLibrary: Performance profiling and prediction of computing continua applications. In *Proceedings of the Companion of the 2023 ACM/SPEC International Conference on Performance Engineering (Coimbra, Portugal) (ICPE '23 Companion)*. ACM, New York, NY, 139–146. DOI: https://doi.org/10.1145/3578245.3584941.

[13] Mingjin Gao, Rujing Shen, Long Shi, Wen Qi, Jun Li, and Yonghui Li. 2023. Task partitioning and offloading in DNN-Task enabled mobile edge computing networks. *IEEE Transactions on Mobile Computing* 22, 4 (2023), 2435–2445. DOI: https://doi.org/10.1109/TMC.2021.3114193

[14] Maor Gaon and Ronen I. Brafman. 2020. Reinforcement learning with Non-Markovian rewards. *Proceedings of the AAAI Conference on Artificial Intelligence* 34, 4 (2020), 3980–3987. DOI: https://doi.org/10.1609/aaai.v34i04.5814

[15] Ruben Gomez-Ojeda, Francisco Angel Moreno, David Zuñiga-Noël, Davide Scaramuzza, and Javier Gonzalez-Jimenez. 2019. PL-SLAM: A stereo SLAM system through the combination of points and line segments. *IEEE Transactions on Robotics* 35, 3 (2019), 734–746. DOI: https://doi.org/10.1109/TRO.2019.2899783

[16] Bencan Gong and Xiaowei Jiang. 2023. Dependent task-offloading strategy based on deep reinforcement learning in mobile edge computing. *Wireless Communications and Mobile Computing* (2023), 1–12. DOI: https://doi.org/10.1155/2023/4665067

[17] H. V. Hasselt. 2010. Double Q-learning. In *Proceedings of the 24th International Conference on Neural Information Processing Systems*, Vol. 2. Curran Associates Inc., 2613–2621.

[18] Teresa Hirzle, Florian Müller, Fiona Draxler, Martin Schmitz, Pascal Knierim, and Kasper Hornbæk. 2023. When XR and AI Meet—A scoping review on extended reality and artificial intelligence. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems (CHI '23)*. ACM, New York, NY, Article 730, 45 pages. DOI: https://doi.org/10.1145/3544548.3581072

[19] Liang Huang, Suzhi Bi, and Ying-Jun Angela Zhang. 2020. Deep reinforcement learning for online computation offloading in wireless powered Mobile-Edge computing networks. *IEEE Transactions on Mobile Computing* 19, 11 (2020), 2581–2593. DOI: https://doi.org/10.1109/TMC.2019.2928811

[20] Vibha Jain and Bijendra Kumar. 2023. QoS-aware task offloading in fog environment using multi-agent deep reinforcement learning. *Journal of Network and Systems Management* 31, 1 (2023), 7.

[21] Hongbo Jiang, Xingxia Dai, Zhu Xiao, and Arun Iyengar. 2023. Joint task offloading and resource allocation for energy-constrained mobile edge computing. *IEEE Transactions on Mobile Computing* 22, 7 (2023), 4000–4015. DOI: https://doi.org/10.1109/TMC.2022.3150432

[22] A. W. Kambale, H. Sedghani, F. Filippini, G. Verticale, and D. Ardagna. 2023. Runtime management of artificial intelligence applications for smart eyewears. In *Proceedings of the IEEE/ACM 16th International Conference on Utility and Cloud Computing (UCC '23)*, Article 31, 1–8. DOI: https://doi.org/10.1145/3603166.3632562

[23] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. 2017. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. *ACM SIGPLAN Notices* 52, 4 (2017), 615–629. DOI: https://doi.org/10.1145/3037697.3037698

[24] Jyotirmoy Karjee, S. Praveen Naik, Kartik Anand, and Vanamala N. Bhargav. 2022. Split computing: DNN inference partition with load balancing in IoT-edge platform for beyond 5G. *Measurement: Sensors* 23 (2022), 100409. DOI: https://doi.org/10.1016/J.MEASEN.2022.100409

[25] Christian Kerl, Jurgen Sturm, and Daniel Cremers. 2013. Robust odometry estimation for RGB-D cameras. In *Proceedings of IEEE International Conference on Robotics and Automation*, 3748–3754. DOI: https://doi.org/10.1109/ICRA.2013.6631104

[26] Hengyi Li, Zhichen Wang, Xuebin Yue, Wenwen Wang, Hiroyuki Tomiyama, and Lin Meng. 2022. An architecture-level analysis on deep learning models for low-impact computations. *Artificial Intelligence Review*, 2022, 1–40. DOI: https://doi.org/10.1007/S10462-022-10221-5/FIGURES/25

[27] Zhuofan Liao, Weibo Hu, Jiawei Huang, and Jianxin Wang. 2023. Joint multi-user DNN partitioning and task offloading in mobile edge computing. *Ad Hoc Networks* 144 (2023), 103156. DOI: https://doi.org/10.1016/j.adhoc.2023.103156

[28] Wenhan Luo, Junliang Xing, Anton Milan, Xiaoqin Zhang, Wei Liu, and Tae-Kyun Kim. 2021. Multiple object tracking: A literature review. *Artificial Intelligence* 293 (2021), 103448. DOI: https://doi.org/10.1016/j.artint.2020.103448

[29] Sultan Javed Majeed and Marcus Hutter. 2018. On Q-learning convergence for non-Markov decision processes. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, 2546–2552. DOI: https://doi.org/10.24963/IJCAI.2018/353

[30] Alexandre Maros, Fabricio Murai, Ana Paula Couto da Silva, Jussara M. Almeida, Marco Lattuada, Eugenio Gianniti, Marjan Hosseini, and Danilo Ardagna. 2019. Machine learning for performance prediction of Spark cloud applications. In *Proceedings of the IEEE 12th International Conference on Cloud Computing*, 99–106. DOI: https://doi.org/10.1109/CLOUD.2019.00028

[31] Simone Mentasti, Abednego Wamuhindo Kambale, and Matteo Matteucci. 2022. Event-based object detection and tracking—A Traffic monitoring use case. In P. L. Mazzeo, E. Frontoni, S. Sclaroff, and C. Distante (Eds.). *Image Analysis and Processing (ICIAP '22)*. Lecture Notes in Computer Science, Vol. 13374, Springer-Verlag, Berlin, Heidelberg, 95–106. DOI: https://doi.org/10.1007/978-3-031-13324-4_9

[32] Microsoft. 2024. HoloLens 2. Retrieved June 01, from https://www.microsoft.com/en-au/hololens/hardware#document-experiences

[33] Zhou Nan. 2021. The application of artificial intelligence technology in the communication engineering industry. In *Proceedings of the Networking, Communications and Information Technology (NetCIT)*, 252–254. DOI: https://doi.org/10.1109/NetCIT54147.2021.00057

[34] Arvind Narayanan, Xumiao Zhang, Ruiyang Zhu, Ahmad Hassan, Shuowei Jin, Xiao Zhu, Xiaoxuan Zhang, Denis Rybkin, Zhengxuan Yang, Zhuoqing Morley Mao, et al. 2021. A variegated look at 5G in the wild: Performance, power, and QoE implications. In *Proceedings of 2021 ACM SIGCOMM Conference*, 610–625. DOI: https://doi.org/10.1145/3452296.3472923

[35] Arjun Parthasarathy and Bhaskar Krishnamachari. 2022. Partitioning and placement of deep neural networks on distributed edge devices to maximize inference throughput. In *Proceedings of the 32nd International Telecommunication Networks and Applications Conference*. IEEE, 239–246.

[36] Teemu Pulkkinen, Jukka K. Nurminen, and Petteri Nurmi. 2020. Understanding WiFi cross-technology interference detection in the real world. In *Proceedings of the International Conference on Distributed Computing Systems*, 954–964. DOI: https://doi.org/10.1109/ICDCS47774.2020.00061

[37] Xiaoyu Qiu, Weikun Zhang, Wuhui Chen, and Zibin Zheng. 2021. Distributed and collective deep reinforcement learning for computation offloading: A practical perspective. *IEEE Transactions on Parallel and Distributed Systems* 32, 5 (2021), 1085–1101. DOI: https://doi.org/10.1109/TPDS.2020.3042599

[38] Qualcomm. 2022. *Snapdragon Summit 2022—Snapdragon Tech Event*. Retrieved March 28, 2024 from https://www.qualcomm.com/company/events/snapdragon-summit

[39] Qualcomm. 2024. *Snapdragon AR2 Gen 1 Platform*. Retrieved March 28, https://www.qualcomm.com/products/application/xr-vr-ar/snapdragon-ar2-gen-1-platform

[40] Starlin Daniel RajKarthiban. 2022. Applications of artificial intelligence in healthcare. In *Proceedings of the International Conference on Computer Communications and Informatics*, 1–2. DOI: https://doi.org/10.1109/ICCCI54379.2022.9741057.

[41] Peter Rost and Gerhard Fettweis. 2010. On the transmission-computation-energy tradeoff in wireless and fixed networks. *IEEE Globecom Workshops*, 1394–1399. DOI: https://doi.org/10.1109/GLOCOMW.2010.5700167

[42] G. Rummery and M. Niranjan. 1994. *On-Line Q-Learning Using Connectionist Systems*. Technical Report CUED/F-INFENG/TR 166.

[43] H. V. Seijen, H. V. Hasselt, S. Whiteson, and M. Wiering. 2009. A theoretical and empirical analysis of expected Sarsa. In *Proceedings of the Adaptive Dynamic Programming and Reinforcement Learning*, 177–184. DOI: https://doi.org/10.1109/ADPRL.2009.4927542

[44] Richard S. Sutton and Andrew G. Barto. 2018. *Reinforcement Learning: An Introduction*. MIT Press.

[45] Uma Tadakamalla and Daniel, A. Menascé. 2022. Autonomic resource management for fog computing. *IEEE Transactions on Cloud Computing* 10, 4 (2022), 2334–2350. DOI: https://doi.org/10.1109/TCC.2021.3064629

[46] Ming Tang and Vincent W. S. Wong. 2022. Deep reinforcement learning for task offloading in mobile edge computing systems. *IEEE Transactions on Mobile Computing* 21, 6 (2022), 1985–1997. DOI: https://doi.org/10.1109/TMC.2020.3036871

[47] Peng Wang, Kenli Li, Senior Member, Bin Xiao, and Keqin Li. 2022. Multiobjective optimization for joint task offloading, power assignment, and resource allocation in mobile edge computing. *IEEE Internet of Things Journal* 9, 14 (2022), 11737–11748. DOI: https://doi.org/10.1109/JIOT.2021.3132080

[48] C. Watkins and P. Dayan. 1992. Autonomic resource management for fog computing. *Machine Learning* 8, (1992), 279–292. DOI: https://doi.org/10.1007/BF00992698

[49] Wenbo Yao. 2019. The application of artificial intelligence in the internet of things. In *Proceedings of the International Conference on Information Technology and Computer Application*, 141–144. DOI: https://doi.org/10.1109/ITCA49981.2019.00038

[50] K. Paul Yoon and Ching-Lai Hwang. 1995. *Multiple Attribute Decision Making: An Introduction*. Sage Publications.

[51] Ahmad Zendebudi and Salimur Choudhury. 2022. Designing a deep Q-learning model with edge-level training for multi-level task offloading in edge computing networks. *Applied Sciences* 12, 20 (2022), 10664. DOI: https://doi.org/10.3390/APP122010664

Appendices

A Parameters and Decision Variables

Table A1 summarizes the problem parameters and decision variables adopted in Section 3.

Table A1. Parameters and Variables

Parameters	
Λ	Input workload [frames/second]
$\mathscr{D}$	Set of computing component indices; $\mathscr{D} = \{1, 2, 3\}$
$\mathscr{K}$	Set of all the candidate DNN configurations κ^i
$p^i_\emptyset$	Fictitious partition meaning that nothing is executed
δ^i_{dk}	Per-request data sent by partition p^i_d to p^i_k, $\forall d, k = (d+1) \in \mathscr{D}$
δ_0	Size of the input tensor
μ^i_d	Per-request computational workload for running p^i_d [flops]
t^i_d	Latency for running partition p^i_d on device d
F	Set of the network domains
T_f	Access time of network domain f
ϕ_f	Channel bandwidth of network domain f
r_f	Throughput of network domain f
g	Cost paid to transfer one byte of data over the 5G network
c_{5G}	Time unit cost paid to transfer the data over the 5G network
L_{max}	Maximum time latency allowed for the AI task
θ_{SEW}	Transmission power of the SEW
θ_{phone}	Transmission power of the phone
α	per unit cost of the energy consumption
z_{SEW}	Energy consumption in joules for performing one floating point operation on the SEW
z_{phone}	Energy consumption in joules for performing one floating point operation on the phone
Decision Variables	
x^i	Binary variable specifying that the configuration c^i is selected
P_{exeSEW}	Power consumption of the SEW when running a partition locally
$P_{exePhone}$	Power consumption of the phone when running a partition locally
P_{wnu}	Power consumption of the SEW WNI when uploading data to the smartphone
P_{5gu}	Power consumption of the phone 5GI when uploading data to the cloud
P_{SEW}, P_{phone}	Total power consumption of the SEW and phone
R	Reward (this is the negative of the cost)
RL State Variables	
r_{WiFi7}	Data transmission rate between the SEW and the smartphone
r_{5G}	Data transmission rate between the smartphone and the cloud server
l_{SEW}	Latency associated with the execution on the SEW
l_{phone}	Latency associated with the execution on the smartphone
l_{cloud}	Latency associated with the execution on the cloud server
RL Parameters	
τ	Control time period
γ	Discount factor
lr	Learning rate
ε	Probability of selecting the next action at random

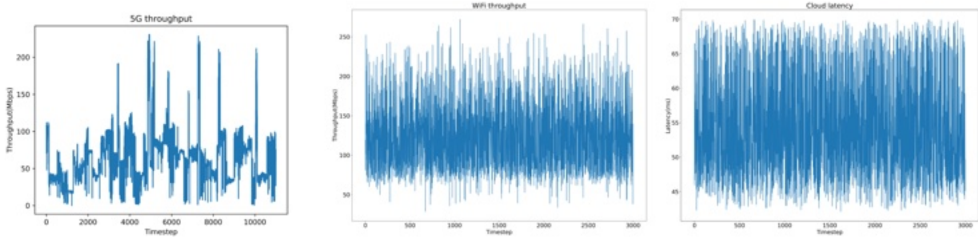
B Traces, Configurations, and Additional Training Details

This section provides additional details regarding the experiments outlined in Section 5. In Section B.1, we explain how we generated the 5G, WiFi, and cloud latency traces, while we provide information on the applications candidate configurations in Section B.2. In Section B.3, we present the comparison among all the methods, both when varying the importance weights and the latency constraint threshold. Finally, we provide a scalability analysis with respect to the number of configurations in Section B.4.

B.1 5G, WiFi, and Cloud Latency Traces

In real-world scenarios, the 5G network is an important source of variability. Hence, we used a 5G throughput trace taken from real-time measurements [34] to make the offline training close to the real environment conditions. The dataset contains 11,024 rows, and the uplink throughput values range between 0 and 230.75 Mbps. As an episode takes 1.5 Million steps to complete, we replayed the trace after 11,024 steps, and, to avoid the periodicity in the trace, we performed some operations as random shifts, insertion of random noise (± 10% of the value at time step t) and trace inversion in its middle. Figure B1(a) presents a snapshot of the 5G trace for the first 11,000 time steps.

On the other hand, when two or more people are using the SEWs in the same environment, the problem of interference may arise, and the WiFi5 network throughput may change. Therefore, we considered the WiFi5 network throughput as another source of variability. The WiFi5 trace was obtained by collecting the throughput during the DNN profiling (the HoloLens was connected to the mobile phone through a WiFi6E hotspot). Figure B1(b) provides a snapshot of the WiFi trace for 3,000 steps. As for the 5G trace, after 3,000 steps, the WiFi trace is replayed considering some random shifts and the insertion of random noise.



(a) 5G trace in steps [0, 11000. (b) WiFi5 trace in steps [0, 3000]. (c) Cloud latency in steps [0, 3000].

Fig. B1. The snapshot of traces used for the experiments as the sources of variability in the system.

Finally, cloud latency fluctuation is another important source of variability, particularly when the cloud is solicited by different entities, which can impact the execution time of the inference task coming from the SEW application. To emulate the behavior of real-world applications, we treated the cloud server as an M/M/1 queue, where requests follow a Poisson distribution, as it is widely assumed in cloud and edge literature [45]. The cloud latency trace was obtained by drawing the cloud execution time from exponential distributions whose mean values depend on the chosen DNN configuration according to the values reported in Table B1.

Table B1. Ranges of DNN Configuration Parameters

Configuration parameters							
δ (MB)	π (MB)	Partition 1		Partition 2		Partition 3	
		Latency (ms)	μ (MFLOPs)	Latency (ms)	μ (MFLOPs)	Latency (ms)	μ (MFLOPs)
(0, 6.25)	(0, 6.25)	(0, 108)	(0, 1,680)	(0, 222)	(0, 5,427)	(0, 78)	(0, 5,427)

A snapshot of the cloud latency samples corresponding to the configuration which runs DNN totally on cloud for the first 3,000 time steps is given in Figure B1(c).

B.2 YOLOv5 Partitioning and Configuration Parameters

We considered YOLOv5n model in ONNX format to avoid framework dependency, and relied on ONNX Runtime (version 1.13.1) to run different partitions on both the HoloLens 2 and the mobile phone. The goal of the profiling was to determine the execution time of each DNN partition on the three devices we consider. Figure B2 illustrates the different execution times for the DNN partitions considering the partitioning points between the HoloLens (i.e., SEW), the mobile phone, and between the phone and the edge (cloud) server. The end-to-end execution time ranges between 220 ms and 620 ms across the majority of the configurations.

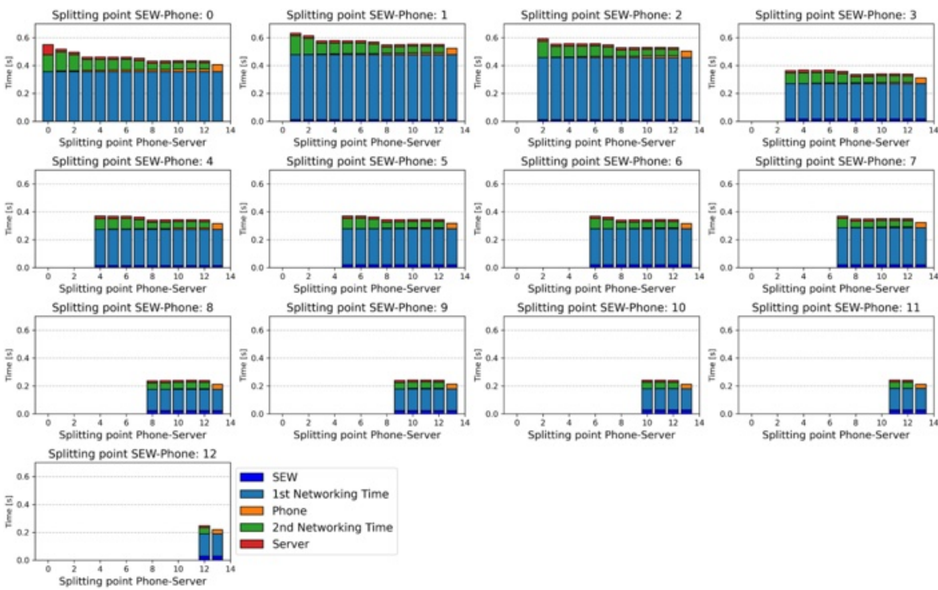
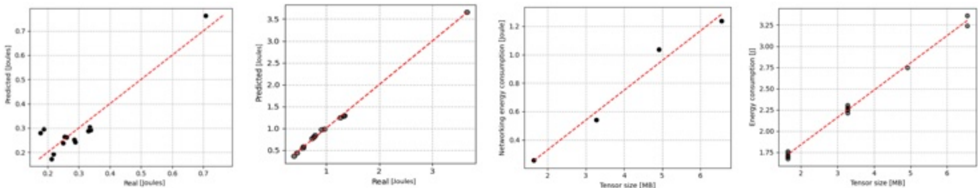


Fig. B2. Profiling of the YOLOv5n on three devices: the SEW (glasses), the phone, and the cloud server.



(a) Real versus predicted energy consumption of YOLOv5n partitions on Samsung S23

(b) Real versus predicted energy consumption of YOLOv5n partitions on HoloLens

(c) Energy consumption for YOLOv5n's intermediate tensors transfer from Samsung S23 to next server.

(d) Energy consumption for YOLOv5n's intermediate tensors transfer from HoloLens 2 to next server.

Fig. B3. Energy consumption on Samsung S23 and HoloLens 2 for executing YOLOv5n partitions and transferring intermediate tensors.

Note that, after executing the partitioner, the YOLOv5 model exhibited 12 feasible partitioning points. To account for scenarios where nothing is run on a device or the entire DNN model is run on the device, we added two additional partitioning points, denoted as 0 and 13, respectively. The candidate configurations are then generated as follows:

— Two configurations account for the fact that the whole DNN can be executed on the phone or in the cloud.
— Twelve candidate configurations can be constructed by choosing one of the possible partitioning points and deploying the first subset of layers on the SEW and the second one on the phone.
— Similarly, 12 candidate configurations consider one of the partitioning points and divide the computation between SEW and cloud.
— A third group of 12 candidate configurations selects a possible partitioning point and places the initial DNN layers on the phone and the others in the cloud.
— Finally, 66 candidate configurations can be generated by selecting two partitioning points among the 12 available, so that the computation is split among SEW, phone, and cloud.

The total number of configurations can be determined by $\frac{n!}{s!\,(n-s)!} + n$, where n represents the total number of possible partitioning points (in our case, 14), and s is equal to the number of computation layers (devices) minus one. Overall we have a total of 105 candidate configurations.

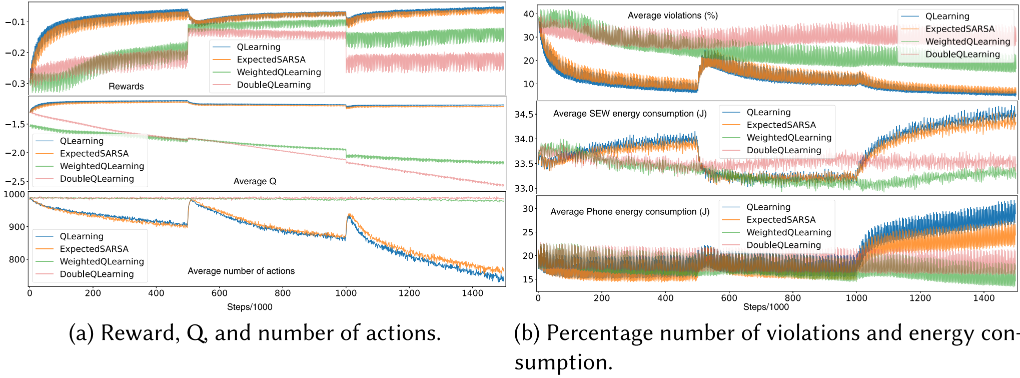
Figure B2 reports all the candidate configurations and the corresponding profiling data, listed as detailed in the following: each sub-plot identifies a specific partitioning point between SEW and phone, while its x-axis lists the possible partitioning points between phone and cloud. Therefore, the first sub-plot (corresponding to the synthetic partitioning point 0, which identifies *no splitting*) lists the 14 candidate configurations for which no computation runs on the SEW; the first bar denotes the configuration running the whole DNN on the cloud (also the second partitioning point is 0, see as an example Figure 2(a)), and thereafter we progressively shift the partitioning point up to 13 (the last bar), which corresponds to the whole DNN running on the phone (see Figure 2(b)). Similarly, the second sub-plot characterizes the candidate configurations generated by selecting the first candidate partitioning point between SEW and phone, and then considering all the possible partitioning points (from 1 to 13) to further divide the computation between phone and cloud. We proceed as follows in all the remaining sub-plots, up to the last one, which selects the twelfth candidate partitioning point between SEW and phone.

Table B1 presents the range of configuration parameters obtained from the profiling. It gives the range of tensor sizes δ and π (in MB of data) that are transferred from the SEW to the phone, and from the phone to the cloud, respectively. It also provides the range for the latency (in ms) and the workload (in MFLOPs) of all partitions when running on specific devices. Recall that Partition 1 runs on the SEW, Partition 2 on the phone, and Partition 3 on the cloud.

Crucial characteristics of the SEW (phone) in the experiments are the electrical energy z_{SEW} (z_{phone}) it needs to perform one FLOP and the electrical power θ_{SEW} (θ_{phone}) required by the network interface while sending data. We assessed these parameters using energy profiling on Hololens and Samsung S23, executing YOLOv5 model partitions either locally or by transmitting intermediate tensors to a server.

To determine z_{SEW}, we charged the HoloLens to 100%, then ran a YOLOv5n partition until the battery reached 95%. This operation was performed for all the 14 possible partitions running on the SEW. The following step involved computing the energy injected in the HoloLens while recharging it from 95% to 100%. We measured the energy during this process using a TC66[4] device. The energy

[4]https://gzhls.at/blob/ldb/2/5/2/8/0e6f880a0a23ad0ef7a16e5c990f2c5b0216.pdf.



(a) Reward, Q, and number of actions.

(b) Percentage number of violations and energy consumption.

Fig. B4. Comparison of methods in varying importance weights scenario (average across 10 applications and 10 instances).

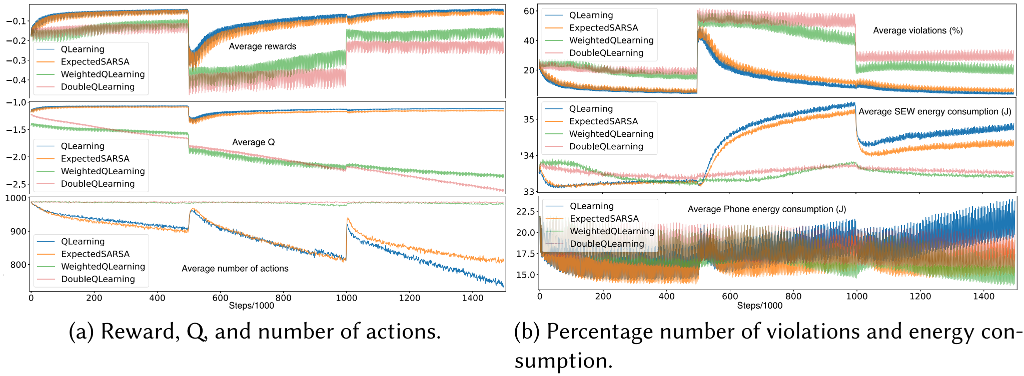
consumption of a partition is obtained as $\frac{e_m}{n_{iter}}$, where e_m represents the energy needed to charge the HoloLens from 95% to 100%, and n_{iter} is the number of partition executions (iterations) during the battery discharge from 100% to 95%. This yielded a value of $z_{SEW} \approx$ 1,085.1 pJ/FLOP through linear regression. Figure B3(a) and (b) provides views of the real value versus the predicted value of the energy consumption of YOLOv5n on Samsung S23 and HoloLens 2, respectively.

Additionally, to determine θ_{SEW}, no partitions were executed on the HoloLens. Instead, intermediate tensors matching the size of those in YOLOv5 partitions were transmitted to the subsequent device, enabling the evaluation of data-transfer power consumption. Employing a similar methodology as for determining z_{SEW}, the power consumed during data transfer can be computed as $\frac{e_m}{n_{iter} \cdot t}$, where e_m represents the energy needed to charge the phone from 95% to 100%, n_{iter} denotes the iterations performed while transmitting the tensor during the HoloLens discharge from 100% to 95%, and t is the time taken for this discharge. (c) and (d) report the energy consumption while considering the different intermediate tensor sizes on Samsung S23 and HoloLens 2, respectively. Considering the time taken to send the results and the energy consumption, the profiling results revealed a constant power consumption of $\theta_{SEW} \approx 7.9$ W during data transmission.

On Samsung S23, we performed exactly the same energy profiling as described earlier on Hololens 2 to determine z_{phone} and θ_{phone}. We obtained the values of 112.5 pJ/FLOP and 4.5 W for z_{phone} and θ_{phone}, respectively.

B.3 Comparison of All Methods

Figures B4 and B5 compare all the methods across all the metrics we consider in the *Varying importance weights* (see Section 5.3) and *Varying latency constraint* (see Section 5.4) scenarios, respectively. Since the corresponding comparison between Q-learning and SARSA is already shown in Figures 3 and 6, we omit SARSA in Figures B4 and B5 to enhance the figures readability. As mentioned in Section 5, Q-learning proved to be the most effective method for solving our problem. Both SARSA and Expected SARSA demonstrated behavior comparable to Q-learning, whereas the Weighted Q-learning and Double Q-learning agents were unable to adapt to our specific problem domain, even after hyperparameter exploration, and resulted in a significant number of execution time violations. While Double Q-learning effectively reduces overestimation bias in stationary environments by decoupling action selection and evaluation, its advantages diminish in highly dynamic scenarios such as mobile edge computing. In settings with fluctuating bandwidth and latency, delayed updates between estimators reduce responsiveness, making the algorithm less



(a) Reward, Q, and number of actions.

(b) Percentage number of violations and energy consumption.

Fig. B5. Comparison of methods in varying latency constraint scenario (average across 10 applications and 10 instances).

Table B2. Scalability Analysis of Tabular Q-Learning with Increasing Number of Configurations

#configurations	#steps to converge (violation rate < 11%)	Q-table size (MB)
30	3,000	9.2
50	54,000	26
105	410,000	110
210	>500,000	448

suitable for real-time applications like smart eyewear offloading. Additionally, the approach doubles memory usage—an impractical overhead given its limited performance gains in our tabular RL setting, so that the training time of one episode for Double Q-learning exceeded that of other methods by more than 20 times.

B.4 Scalability Analysis with Respect to the Number of Configurations

To evaluate the scalability of our Q-learning-based approach for AI task offloading in SEW, we conducted a set of experiments designed to systematically assess how performance and resource usage evolve as the size of the action space increases. Specifically, we varied the number of configurations (i.e., action space size) across four scales: 30, 50, 105, and 210 configurations. For each setting, we generated 10 random problem instances (based on parameter setting in Section 5.6.3), considering 500,000 training steps with $L_{max} = 500$ and weight set $\omega_{lat} = 0.4$, $\omega_{e_{SEW}} = 0.5$, $\omega_{e_{phone}} = 0.05$, $\omega_{5G} = 0.03$ and $\omega_{rcfg} = 0.02$, and measured:

- The average number of training steps required for the agent to converge (defined as maintaining a moving average violation rate below 11% over the last 1,000 steps).
- The size of the Q-table in megabytes across these 10 runs.

Results, shown in Table B2, reveal two key insights:

(1) *Scalability Limitations (Acknowledged)*: As the number of configurations increases, both the Q-table size and the required number of steps to converge grow significantly. With 210 configurations, convergence was not reached within 500,000 steps, and the Q-table exceeded 400 MB. These findings highlight a known limitation of tabular Q-learning in large action spaces.

(2) *Feasibility for Small to Medium Scales*: While tabular Q-learning does face scalability challenges in larger configuration spaces, our results demonstrate that it remains practical and efficient for small to medium-scale SEW applications (e.g., $\leq$ 105 configurations), which are typical in many edge AI deployment scenarios due to real-time and resource constraints.

Received 18 October 2024; revised 29 September 2025; accepted 30 September 2025