Deep Reinforcement Learning for the Computation Offloading in MIMO-based Edge Computing

Deep Reinforcement Learning for the Computation Offloading in MIMO-based Edge Computing

Abdeladim Sadiki*, Jamal Bentahar*, Rachida Dssouli*, Abdeslam En-Nouaary†
*Concordia Institute for Information Systems Engineering, Concordia University, Montreal, Canada
†Institut National des Postes et Télécommunications, STRS Lab, Rabat, Morocco
abdeladim.sadiki@mail.concordia.ca, bentahar@ciise.concordia.ca, rachida.dssouli@concordia.ca, abdeslam@inpt.ac.ma

***Abstract*—Multi-access Edge Computing (MEC) has recently emerged as a potential technology to serve the needs of mobile devices (MDs) in 5G and 6G cellular networks. By offloading tasks to high-performance servers installed at the edge of the wireless networks, resource-limited MDs can cope with the proliferation of the recent computationally-intensive applications. In this paper, we study the computation offloading problem in a massive multiple-input multiple-output (MIMO)-based MEC system where the base stations are equipped with a large number of antennas. Our objective is to minimize the power consumption and offloading delay at the MDs under the stochastic system environment. To this end, we formulate the problem as a Markov Decision Process (MDP) and propose two Deep Reinforcement Learning (DRL) strategies to learn the optimal offloading policy without any prior knowledge of the environment dynamics. First, a Deep Q-Network (DQN) strategy to solve the curse of the state space explosion is analyzed. Then, a more general Proximal Policy Optimization (PPO) strategy to solve the problem of discrete action space is introduced. Simulation results show that the proposed DRL-based strategies outperform the baseline ad state-of-the-art algorithms. Moreover, our PPO algorithm exhibits stable performance and efficient offloading results compared to the benchmark DQN strategy.**

***Index Terms*—Multi-access Edge Computing (MEC), Massive Multiple-Input Multiple-Output (MIMO), Deep Reinforcement Learning (DRL), Computation Offloading.**

I. Introduction

OVER the last years, with the great progress in the development of smart Mobile Devices (MDs) and wireless communication systems, there has been an explosive growth in the number of innovative applications that need a large amount of computing resources, such as ultra high definition media streaming, Virtual-Reality/Augmented-Reality (VR/AR) applications, real-time online 3D gaming, image processing, face recognition [1], some of which may also have a delay-sensitive characteristic [2]. However, even with their high performing modern hardware, MDs are usually limited in terms of computation capabilities, battery duration, and storage capacities. As a result, the Quality-of-Service (QoS) and Quality-of-Experience (QoE) of these applications are significantly impacted when tasks are executed on MDs [3].

To overcome this limitation, a viable solution was to leverage the powerful resources of specialized remote cloud servers to carry out the computation-intensive tasks. This approach is known as Mobile Cloud Computing (MCC) [4]. Although MCC could improve the battery life of MDs as well as the application experience [5], its major drawback is the long distance between the MDs and the cloud server, which results in a significant network congestion, a substantial latency of service and performance degradation [2][6].

To address the above issues, a recent promising approach, known as Multi-access Edge Computing (MEC), has been proposed. The key idea underlying MEC is to push cloud-like computing capabilities to the network edge (e.g., base station) providing high performance services in close proximity to MDs [4][7]. Thereby, users can offload their computationally intensive and time-critical tasks to the nearby MEC server (to which they are connected through wireless connections) for processing rather than a centralized remote cloud server [8]. As a result, the potential congestion can be decreased, the service delay can be reduced and the energy usage of MDs can be significantly enhanced [9].

Currently, the 6th generation (6G) wireless communication network is getting great attention from the research community. Along with MEC, which will be one of the major elements of the 6G era [10], massive Multiple-Input Multiple-Output (MIMO) is a key enabler technology to deliver the needs of the next generation networks [11]. Massive MIMO involves deploying a large number of antennas at the base stations, which leads to a significant improvement in the system's efficiency and throughput [12]. As a result, using massive MIMO is significant as it substantially assists computation offloading in MEC. In fact, by providing high spectral and energy efficiencies, massive MIMO will benefit the computation offloading with high transmission rates and lower energy consumption [13].

Although MEC has enormous benefits, there remain several challenges. To begin with, real-time applications are highly sensitive to the latency requirement, while service latency is affected by various elements in offloading, such as transmission and power allocation [8]. Moreover, the computation tasks are generated dynamically in the MDs, so deciding what part of the given task should be offloaded to the edge is a challenge in the offloading decision making process [6]. MDs are also limited in terms of energy consumption that needs to be considered. Furthermore, the MEC environment keeps changing over time and enjoys a stochastic characteristic at three levels: wireless channel, physical location of MDs and user mobility [6][8]. As a result, an efficient offloading framework under the stochastic MEC conditions has become

a major challenge. A number of offloading schemes have been proposed in the literature [1] [5] [14]. The objective functions were designed as reducing the energy consumption, satisfying the latency requirement or finding a trade-off between the energy and latency [15]. The first studies were relying on classical optimization algorithms to solve these objective functions. However, they do not consider the challenging dynamic aspect of the MEC environment as they focus solely on performance of a quasi-static system [15]. Accordingly, emerging Reinforcement Learning (RL) has been recognized as an effective approach to overcome this issue [15]. Some of the existing proposals use Q-learning [4][15] to develop dynamic computation offloading methods without any prior knowledge of the MEC dynamics. Yet, due to the curse of state space explosion [16], this approach is quickly outdated in favor of Deep Reinforcement Learning (DRL), especially the Deep Q-Network (DQN) algorithm which is being used in the majority of the most recent solutions [1][17]. However, the common limitation in these works is they are always considering discrete action space-based policies, which makes them powerless to control continuous quantities such as the energy or the latency. Recently, [13] considered studying MEC computation offloading under the next generation massive MIMO technology.

In this paper, we study the computation offloading problem in a MEC server powered by massive MIMO technology. We consider the stochastic wireless channel variations and the dynamically generated tasks at each mobile user, and we propose DRL-based strategies that can learn a dynamic offloading policy under the varying MEC conditions. In addition to the standard DQN-based strategy, we propose a novel offloading strategy based on the Proximal Policy Optimization (PPO) technique to support continuous action space. Therefore, the main contributions of this work are summarized as follows:

- We formulate the computation offloading in a MEC with MIMO network under stochastic wireless conditions and task arrivals as a joint minimization problem between power consumption and offloading delay at each MD.
- To address the formulated problem, we design A Markov Decision Process (MDP) where the state space, action space and reward function are defined.
- A DQN strategy is implemented based on our formulated MDP to learn a computation offloading policy that can minimize the system cost.
- A PPO strategy is introduced to derive better offloading policy over the continuous power allocation action space. To the best of our knowledge, this is the first work to use the PPO algorithm in the MEC computation offloading problem.
- A series of simulations were conducted to compare the performance of the proposed DRL-strategies. The results show that our PPO strategy gives better performance over the standard DQN strategy that is also used as benchmark, and both DRL-based strategies outperform the baseline and state-of-the art schemes.

The rest of this paper is organized as follows. Section II presents a review of the related work. Section III describes the system model and problem formulation. Section IV proposes the DRL-based offloading model. Specifically the MDP formulation, the DQN and PPO-based strategies. Section V provides experiments and simulation results. Finally, Section VI concludes the paper.

II. Related Work

MEC is emerging as a potential technology to cope with the limitations of the MDs with regard to the rapid progress of the next generation of mobile applications. Offloading execution to the edge improves the user experience by enhancing the device performance and lowering the energy consumption. Computation offloading techniques for MEC have lately been extensively studied. In [18], the authors proposed a partial offloading scheme as a joint minimization problem between the energy consumption of the smart mobile device and the latency of application execution using the dynamic voltage scaling technology. A novel framework was developed in [19] for offloading computation tasks from a MD to the edge using the Radio Network Information Service (RNIS) application programming interface (API) to drive the user offloading decision. Another approach for the computation offloading problem was given in [14]. In this work, the authors demonstrated that obtaining an optimal solution for the computation offloading problem is NP-hard, so they used a game theoretic approach to design an efficient offloading algorithm that can achieve the Nash equilibrium. Similarly, to reduce the delay and save the battery life of the user's devices, the authors in [20] proposed an efficient offloading algorithm by transforming the formulated NP-hard mixed integer non-linear problem into two solvable sub-problems, namely task placement and resource allocation. In [21], the authors studied the offloading problem over the 5th generation mobile network (5G). They created an intelligent offloading model based on a metric that can satisfy the latency requirements of user applications and achieve maximum energy saving.

Most of the previous mentioned proposals focus on the basic case in which both the users and the base station are equipped with a single antenna. This fails to leverage the benefits brought by the massive MIMO technology to MEC in terms of offloading performance [22]. For this reason, researchers begin to investigate MEC assisted with the MIMO technology. In [13], the authors studied the application of massive MIMO on MEC. They showed that MIMO boosts the performance of offloading in MEC by providing huge gains in spectral and energy efficiencies. Their results revealed that using more antennas reduce the energy consumption and the system delay. The work in [23] investigated the MEC computation offloading problem in massive MIMO enabled heterogeneous networks (HetNets). The formulated problem was defined as minimizing the energy consumption under a maximum latency requirement. An iterative low-complexity algorithm based on alternating optimization was proposed to solve this non-convex problem. Furthermore, the authors in [24] addressed the computation offloading problem in MEC with multi-user MIMO communication as a joint minimization of the energy consumption and time delay of MDs. The

formulated mixed-integer non-linear programming problem is solved by developing offloading decisions based on semi-definite relaxation and rounding methods. Similar to [13], the simulation results revealed that the use of MIMO communication in MEC reduces sufficiently the energy consumption and time delay during computation offloading.

Nevertheless, the MEC environment dynamics is usually complex and very challenging due to several reasons such as the stochastic network channel conditions, tasks arrival distribution, and power constraints of the MDs [16]. This makes the aforementioned classical optimization techniques very limited as they are mainly modeled based on a network snapshot and they must be reformulated when the dynamics changes over time. Besides, most of them need a high number of iterations and may provide a local rather than global optimum [8]. In fact, they mainly use heuristics to yield feasible solutions, which makes them subject to two major limitations [25] [26] [27]:

1) The divergence likelihood of the heuristic algorithms increases with the problem's input size.
2) The heuristic-based solutions do not allow the model to account for the dynamic changes of the environment and learn from previous experiences.

Fortunately, the emerging artificial intelligence technology has shown potential efficacy to tackle those limitations. Especially, RL is being identified as a suitable framework to deal with the stochastic MEC network environment. In [4], the authors used the Q-learning algorithm to jointly optimize the offloading decision with resource allocation while minimizing energy consumption on the MDs under the latency constraint. The same approach was adopted in [15]. The authors used the Q-learning algorithm to find the optimal policy for resource allocation and computation offloading in MEC. Their results show that the proposed solution leads to a significant decrease in energy consumption of the user's devices compared to baseline methods. However, classical RL algorithms cannot scale when the state space and action space become huge [8], which is mostly the case in a MEC environment. DRL is heavily used in recent proposals to address this issue. In [16], a DQN-based algorithm was introduced to obtain the optimal computation offloading policy. The objective was to minimize the long-term cost based on the channel characteristics between the mobile user and the base station, the energy queue and the task queue states. The same problem is solved in [3] using the Double DQN (DDQN) algorithm, which gave significant improvement in computation offloading performance. The authors in [1] used the DQN algorithm to design a joint task offloading and bandwidth allocation decision in order to minimize the overall offloading cost in terms of energy, computation and delay. The authors in [17] applied DRL to intelligent Internet of Things (IoT) inside a MEC with massive MIMO network. A DQN algorithm was proposed to learn offloading decisions in order to improve the system performance and reduce the latency and energy consumption. However, the cost function and the MDP formulation are different compared to our work. In fact, to minimize their cost function, the authors optimized the bandwidth allocation based on some deterministic predefined criteria before optimizing the offloading strategy. In addition, they didn't consider the varying wireless channel conditions in their MDP formulation. Moreover their DQN neural network architecture has not been fully specified. To overcome these limitations, in our work we reformulate the problem without predefined criteria while taking into consideration the stochastic wireless channel under the MIMO technology. To use this work as a benchmark, we followed the authors' approach to apply the DQN algorithm with a clear presentation of our neural network architecture. Furthermore, we introduced a new PPO algorithm for the computation offloading in order to solve the limitation of the discrete action space in the DQN algorithm. The simulation results (see Section V) show that our PPO algorithm has better performance.

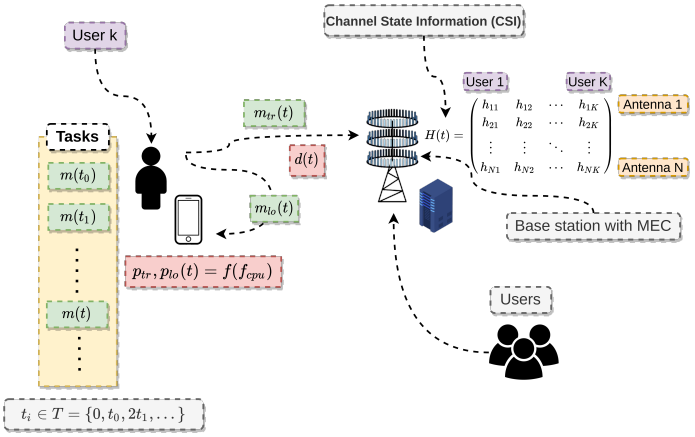

Fig. 1: Problem description

III. System Model and Problem Formulation

We consider a massive MIMO based MEC system, where a set of K mobile single-antenna users $\mathcal{U} = \{u_1, \dots, u_K\}$ communicate with a MEC server through a Base Station (BS) equipped with N antennas ($N >> K$), as shown in Figure 1. For simplicity but without lack of generality, we consider one BS. Generalizing to multiple BSs is straightforward because they are independent and different base stations serve different users at each time moment. We also assume that the system time T is discrete, so it can be split into equal time slots of the same length t_0, $T = \{0, t_0, 2t_0, \dots\}$.

A. Wireless channel model

At each time $t \in T$, a user $u_k \in \mathcal{U}$ transmits a signal $x_k(t) \in \mathbb{C}$ to the BS, where $\mathbb{C}$ is the set of complex numbers. Let $X(t) = [x_1(t), x_2(t), \dots, x_K(t)]^\top \in \mathbb{C}^{K\times 1}$ be the complex transmitted signal vector from all the users to the BS. The $(N \times 1)$ received signal vector in the uplink at the BS $Y(t) \in \mathbb{C}^{N\times 1}$ is expressed by:

$$Y(t) = H(t).X(t) + B(t)$$

where $H(t) \in \mathbb{C}^{N\times K}$ is the Channel State Information (CSI) matrix of complex channel gains between all the users and the BS:

$$H(t) = \begin{bmatrix} h_{11}(t) & h_{12}(t) & \dots & h_{1K}(t) \\ h_{21}(t) & h_{22}(t) & \dots & h_{2K}(t) \\ \vdots & \vdots & \vdots & \vdots \\ h_{N1}(t) & h_{N2}(t) & \dots & h_{NK}(t) \end{bmatrix}$$

$h_{nk}(t) \in \mathbb{C}$ represents the channel response between the k^{th} user antenna and the n^{th} BS antenna, and $B(t) = [b_1(t), b_2(t), ..., b_N(t)]^\top \in \mathbb{C}^{N\times 1}$ is the additive white Gaussian noise (AWGN) which follows a complex normal distribution with zero mean and variance $\sigma^2 : B(t) \sim \mathcal{CN}(0, \sigma^2\mathbf{I}_N)$ ($\mathbf{I_N}$ is the $N \times N$ identity matrix). We assume perfect CSI, and the Zero-Forcing (ZF) technique is adopted at the BS to suppress the inter-user interference [22]. Hence, the ZF-detection matrix is given by:

$$V(t) = H(t)^H (H(t)H(t)^H)^{-1}$$

Let $H_k(t) = [h_{1k}(t), h_{2k}(t), \dots, h_{NK}(t)]^\top \in \mathbb{C}^{N\times 1}$ be the CSI between the user u_k and the BS and V_k the k-th row of V, the normalized effective channel gain for user u_k can be defined as:

$$\Gamma_k(t) = \frac{|V_k(t)H_k(t)|^2}{\sigma^2|V_k(t)|^2} \tag{1}$$

Accordingly, the achievable channel capacity at user u_k is given by:

$$C_k(t) = W.log_2(1 + p_k^{tr}(t)\Gamma_k(t)) \tag{2}$$

where W is the channel bandwidth in Hz and $p_k^{tr}(t) \in [0, P_k^{tr}]$ is the transmit power of user u_k with P_{tr}^k being its maximum transmission power constraint.

B. Computation model

At each time $t \in T$, each user $u_k \in U$ has a computationally intensive task of size $m_k(t)$ bits to be executed. We assume that the tasks are coming from an application that can be partitioned, and therefore the user can offload part of it to the MEC server to assist with the computation. Some examples of such applications include image compression applications and virus detection software [15]. The MEC servers are generally equipped with higher computational capability than the MDs and they can serve multiple users at the same time. Motivated by the work in [18], we assume that the MD of each user is equipped with the dynamic voltage and frequency scaling (DVFS) technology, where the CPU frequency can be adjusted depending on the computation needs.

Let $f_k(t) \in [0, F_k]$ be the CPU frequency allocated to execute a task at time t with F_k being the maximum frequency constraint of the MD. We denote by $m_k^{lo}(t)$ the amount of data bits that the user can execute locally, hence:

$$m_k^{lo}(t) = \frac{t_0 f_k(t)}{\omega_k} \tag{3}$$

where ω_k (cycles/bit) is the number of cycles required to execute one bit of data.

We model the power consumption of the CPU executing this amount of data as in [18] and [28] by:

$$p_k^{lo}(t) = \eta f_k(t)^3 \tag{4}$$

where η is a coefficient that depends on the hardware architecture of the MD.

Let $m_k^{tr}(t)$ be the amount of data bits of the offloaded part to the edge server. Assuming that the latency resulting from the MEC computation and the downlink transmission is negligible, the time delay to get the result from the edge server is given by:

$$d_k(t) = \frac{m_k^{tr}(t)}{C_k(t)} \tag{5}$$

C. System cost

To take into consideration both the energy consumption and the offloading latency, we modeled the problem cost as a weighed sum of the total power consumption and the time delay resulting from the offloading operation. As a result, we define the system cost as:

$$\Phi_k(t) = \alpha(p_k^{lo}(t) + p_k^{tr}(t)) + (1-\alpha)d_k(t) \tag{6}$$

where α is a positive number between 0 and 1 to determine the weight of the trade-off between the energy and the delay. Our objective is to minimize the long-term cost by dynamically allocating the CPU frequency $f_k(t)$ for local computation and the transmit power $p_k^{tr}(t)$ for the offloading computation. Considering the maximum frequency and power constraints, the corresponding problem can be formulated as:

$$\underset{f_k(t), p_{tr}^k(t)}{\arg\min}\ \Phi_k(t), \forall t \in T$$

s.t

$$f_k(t) \in [0, F_k] \quad \forall t \in T \tag{7}$$

$$p_k^{tr}(t) \in [0, P_k^{tr}] \quad \forall t \in T \tag{8}$$

IV. Deep Reinforcement learning Based strategies for cost minimization

Deep Reinforcement Learning (DRL) is an active research field of machine learning that combines the Reinforcement Learning (RL) framework with artificial neural networks to optimize an objective where the dynamics of the environment keeps changing. In this section, we present briefly the RL framework and introduce our two strategies to solve the optimization objective in our problem formulation.

A. Background

DRL is based on the standard RL setup which consists of an agent interacting with its environment and, upon observing the results of its actions through a received reward from the environment, can learn the optimized behaviour. The RL problem is formalized mathematically as a Markov Decision Process (MDP) which is a discrete time stochastic control process defined as a 5-tuple $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma)$. $\mathcal{S}$ is a set that represents all the possible states of the environment. $\mathcal{A}$ is the action space, i.e., the set of actions that the agent can take. $\mathcal{T} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \longrightarrow [0, 1]$ is the transition probability matrix that maps each state-action pair at time

t to a probability distribution over states at time $t+1$. In most real-life applications, $\mathcal{T}$ is not known due to the fact that it is hard to come up with a model of the environment, which is the case also in our stochastic offloading problem. This is where model-free RL algorithms [29] can be used. $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \longrightarrow \mathbb{R}$ is a reward function that outputs a scalar value $r(t)$ based on the current state at time t, $s(t)$, the action taken $a(t)$ and the next state $s(t+1)$. $\gamma \in [0, 1)$ is a discount factor which reflects how much the agent cares about future rewards depending on how γ is close to 1. In general, the agent chooses its actions based on a policy π which is a mapping from the state space to a probability distribution over actions. Following a policy π, the sequence of states, actions and rewards resulting from the interaction of the agent with the environment constitutes a trajectory (or episode) of that policy. The discounted cumulative reward resulting from the trajectory is called a return $G = \sum_{t=0}^{+\infty} \gamma^t r(t+1)$. The end goal of RL algorithms is to find an optimal policy π^* that maximizes the expected return [30]. In order to apply DRL on our problem, we need to formalize it as a MDP first. In the sequel, we provide the details of our formalism.

B. Markov Decision Process (MDP) formulation

In order to apply DRL techniques, we formulate our problem as a MDP, taking into consideration the stochastic wireless channel conditions and the variation of the tasks to be executed. In our formulation, we consider the problem from the user's perspective, where the agent (the user's mobile device) will try to find the optimum policy to minimize its cost function. The fundamental blocks of our MDP are defined as follows:

1) ***State space****:* The state space is a representation of the environment the agent tries to interact with. In our problem, the agent needs to observe, at each time slot t, the actual task $m_k(t)$ that needs to be executed, and the wireless channel state information $H_k(t)$ in order to optimally decide the offloading part to the MEC server. As a result, each user $u_k \in \mathcal{U}$ will have its own internal representation of the environment. We define $s_k(t)$ the representation of the environment from the u_k user's perspective at each timestamp $t \in T$ by the vector: $s_k(t) = (m_k(t), h_{1k}(t), h_{2k}(t), ..., h_{Nk}(t))$ where $h_{ik}(t), i \in \{1, 2, ..., N\}$ the CSI between the user u_k and the i-th BS antenna.

2) ***Action space****:* In our MEC system model, the agent needs to find the optimal offloading strategy by allocating the optimal local execution power and the optimal offloading power for remote MEC execution. Therefore, for a user u_k, the action at time t is defined by the tuple $a_k(t) = (f_k(t), p_k^{tr}(t))$ where $f_k(t) \in [0, F_k]$ is the allocated CPU frequency for the local execution and $p_k^{tr}(t) \in [0, P_k^{tr}]$ is the transmission power for the offloading execution. However, this action space is defined on the continuous domain and some RL algorithms such as DQN do not support continuous action space. As a workaround to this limitation, we adopt the same approach as in [31] and we devise the above continuous domains into $l \in \{1, 2, \dots\}$ levels. Thus, we define the set of available CPU frequencies as $\mathcal{F} = \{0, \frac{F_k}{l}, \frac{2F_k}{l}, ..., F_k\}$ and the set of available transmission powers as $\mathcal{P} = \{0, \frac{P_k^{tr}}{l}, \frac{2P_k^{tr}}{l}, \dots, P_k^{tr}\}$. Accordingly, we define the l-level action space in the discrete case as the Cartesian product of the two sets $\mathcal{F}$ and $\mathcal{P}$ as follows: $\{(f_k(t), p_k^{tr}(t)) | f_k(t) \in \mathcal{F} \text{ and } p_k^{tr}(t) \in \mathcal{P};\ \forall t \in \mathcal{T}\}$

3) ***Reward function****:* Based on the state representation and the chosen action, the agent receives a reward that guides the learning process. The agent goal is to choose the action that will give the highest reward. The reward function is generally associated with the objective function. In our formulation, the goal is to achieve the minimum overall cost of user u_k. Accordingly, the value of the reward must be negatively correlated to the value of the cost. We define the immediate reward of a user $u_k \in \mathcal{U}$ at time slot $t \in T$ as $r_k(t) = -\Phi_k(t)$, where $\Phi_k(t)$ is the total cost defined in Equation 6.

C. Deep Q Network (DQN) strategy

Q-learning is one of most popular RL algorithms [30]. It is based on the action-value function (or the Q-function) $Q^\pi(s, a)$ which measures the expected return from state s and taking action a following the policy π. The optimal action-action value function $Q^*(s, a)$ which gives the maximum expected return over all policies can be obtained using the following *Bellman optimally equation*:

$$Q^*(s(t), a(t)) = \mathbb{E}[r(t) + \gamma \max_{a(t+1)} Q^*(s(t+1), a(t+1))] \quad (9)$$

The basic idea behind Q-learning is to use Equation 9 as a simple iterative update:

$$Q(s(t), a(t)) \longleftarrow Q(s(t), a(t)) + \beta[r(t) + \gamma \max_{a(t+1)} Q(s(t+1), a(t+1)) - Q(s(t), a(t))] \quad (10)$$

where $\beta \in (0, 1]$ is the learning rate. Storing all Q-values in a table structure for each state-action pair, and using Equation 10 plus an exploration and exploitation trade-off to ensure all the action space is explored, the Q-learning algorithm will converge eventually to the optimal Q-function [32].

It is worth noting that the state and action spaces grow with the complexity of the problem. Thus, using a table in the memory to store each state-action pair is very expensive in terms of computation, especially when trying to update the Q-values for each cell on the table. As a result, applying standard Q-learning in this case is time and memory consuming, and may even diverge [33]. This is known as the curse of high dimensionality problem [16]. In fact, the state space of our problem grows with the number of antennas at the base station. The action space increases as well depending on the l-level action we need (see the MDP design, Section IV-B).

To overcome this limitation, function approximation can be used to learn the Q-values. Recent advances in Deep Learning (DL) have revolutionized the field of RL, leading to the creation of the excited field of DRL. The usage of Deep Neural Network (DNN) as function approximator gives the agent the ability to learn from complex environments [34]. Deep Q Network (DQN) [35] is one of the effective algorithms that combines DL with RL. As shown in Figure 2, our DQN strategy uses DNN with weights θ to approximate the Q

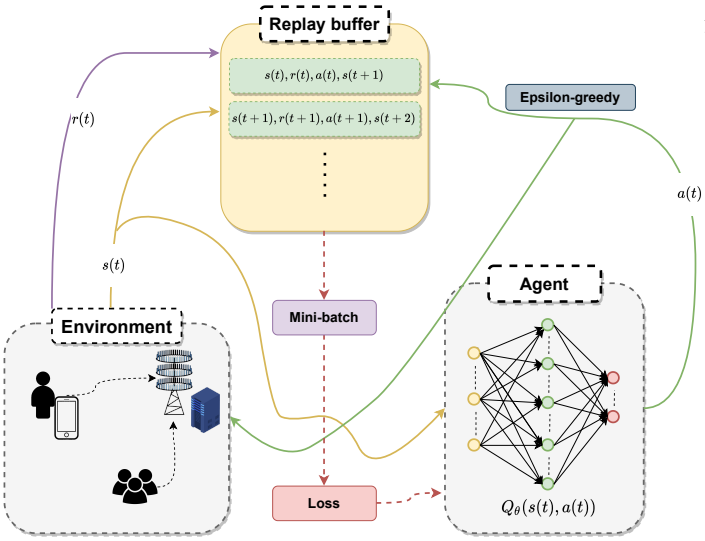

Fig. 2: The structure of our Deep Q-Network (DQN) strategy

Algorithm 1 DQN strategy for cost minimization

1: **Input:** reply buffer capacity C, discount factor γ, maximum number of iterations I, episode length L, mini-batch size B, l-level action l
2: Create a Deep Neural Network and initialize its weights θ with random values $Q(\theta)$
3: Initialize a queue 'Δ' as a reply buffer with capacity C;
4: **for** *iteration* $\leftarrow$ 1 ... I **do**
5: $s \leftarrow$ Get the state vector as defined in our MDP
6: **for** $i \leftarrow 1 \ldots L$ **do**
7: *# Following ϵ-greedy policy*
8: $\epsilon \leftarrow ExpD(\epsilon_0, \epsilon_f, \epsilon_d)$
9: **if** Random number $\xi \leq \epsilon$ **then**
10: $a \leftarrow$ Select randomly a frequency and a power transmission tuple from the discrete action space as defined in the MDP
11: **else**
12: $a \leftarrow arg\max Q(s;\theta)$
 # Get a predicted action by forwarding the state s through the neural network
13: **end if**
14: Execute the action a on the environment
15: Get the reward r and the next state s'
16: Push (s, a, r, s') into Δ
17: **if** length(Δ) $\geq$ B **then**
18: $\Delta' \leftarrow$ Sample a random mini-batch of size B from Δ
19: $Y \leftarrow$ Get the Q-values by forwarding the states $s \in \Delta'$ through the neural network
20: $X \leftarrow$ Get the next Q-values by forwarding next states $s' \in \Delta'$ through the neural network
21: $R \leftarrow$ All the rewards r from our mini-batch Δ'
22: $\hat{Y} \leftarrow R + \gamma\ X$ *# Element wise operations*
23: loss $\leftarrow MSE(Y, \hat{Y})$
24: Backpropagate the loss using gradient decent with the Adam optimizer
25: Dequeue the old element from Δ
26: $s \leftarrow s'$
27: **end if**
28: **end for**
29: **end for**

function. For every iteration, these weights get updated using gradient decent to converge towards the optimal Q values. Algorithm 1 provides a pseudo-code of our cost minimization strategy based on the DQN algorithm.

In Algorithm 1, we start by building a DNN used as a function approximator for the action-value function Q. The input of the model is a state vector, and the output is a layer containing as many neurons as the number of elements of our action space, which depends on the l-level action parameter l given as input (line 2). We then initialize a queue Δ as a reply buffer of size C given as input. It will be used to store the past experience of our agent, namely the state, action, reward and the next state observed by executing that action (line 3).

Inside the learning loop (line 4), which will be running for a maximum of I iterations (episodes) given as input, we start by resetting our environment to a random state s (line 5). Each episode is limited by L learning steps. In every step, we start by deciding on the action to take following the ϵ-greedy policy, which is an efficient method to achieve a trade-off between exploration and exploitation. Here ϵ is a number between zero and one. at the beginning, ϵ will be close to ϵ_0, which we set to one ($\epsilon_0 = 1$) to allow more exploration. With every iteration, ϵ will decay according to a decay factor $\epsilon_d = 5000$ until it reaches $\epsilon_f = 0.001$ as the agent should use more the knowledge learned through its experience with the environment. We use the exponential decay formula to decrease ϵ in every iteration, which is defined as $ExpD(\epsilon_0, \epsilon_f, \epsilon_d) = \epsilon_f + (\epsilon_0 - \epsilon_f) * \exp(-\frac{i+L*(iteration-1)}{\epsilon_d})$ (line 8). Then, we generate a random number ξ between zero and one. If $\xi \leq \epsilon$, the algorithm picks a random action a from the action space (lines 10), and in this case, we say that the agent is exploring. Otherwise, the algorithm picks the action with the maximum Q-value from the neural network (lines 12), and in this case we say that the agent is exploiting, i.e. using the results of the training process so far.

Executing the selected action on the environment, the agent gets the reward r and the next state s'. This transition of (state, action, reward, next state) will be pushed to the replay buffer Δ (lines 14-16). If Q has enough values, we sample a random mini-batch of size B from it, in the hope of making our data more independent and identically distributed (i.i.d) to provide better convergence performance when training the neural network (line 18). The replay buffer Δ will play the role of a data-set as if we are in a supervised learning context, we calculate the predicted Q-values using the states s stored in Δ (line 19), then we forward the next states s' through the network to get the Q-values of the next states (line 20). We can then calculate the expected Q-values using the Bellman equation (lines 22). In our implementation, we used the mean squared error to calculate the loss between the expected Q-values and the predicted Q-values as the following: $MSE(Y, \hat{Y}) = \frac{1}{|Y|}\sum_{i=1}^{|Y|}(Y_i - \hat{Y}_i)$ (line 23) and we used the Adam optimizer for the gradient decent algorithm to backpropagate this loss through the neural network (lines 24).

At the end of the iteration, we remove the old element from the replay buffer Δ to free the space to the next transition, keeping the agent on softly updating and learning as new data are being pushed to the buffer (line 25), and we assign the next state value to the actual state variable (line 26).

D. Proximal Policy Optimization (PPO) strategy

DQN has solved the problem of learning the Q-values for complex environments with high dimensional state space using Deep Neural Networks. However, it can only handle discrete and low-dimensional action spaces. Recently, Schulman et al. proposed the Proximal Policy Optimization (PPO) algorithm [36], a cutting-edge DRL method which can be applied to discrete as well as continuous state and action spaces. Moreover, the authors argue that it outperforms the other algorithms on their benchmark while providing a good balance between ease of tuning, efficient sampling and simple implementation.

PPO is based on the Actor-Critic approach. As shown in Figure 3, the Actor-Critic framework uses two separate neural networks: the Actor and the Critic. The former represents directly the policy π of the agent which controls the offloading scheme and decides on the amount of data that should be computed locally and the transmission power of the offloaded part to the MEC server. It receives the state containing the current task size and the wireless channel vector as input, and outputs a probability distribution over the actions. The agent chooses its next action by sampling from this distribution. The latter is used to approximate the value function $V^{\pi}(s) = \mathbb{E}_{\pi}\{\sum_{k=0}^{\infty}\gamma^{k}r(t+k+1)|s(t)=s\}$, which estimates how rewarding a given state can be for the agent. It receives the same state of the Actor as input and outputs the estimated value of $V^{\pi}(s)$. In other words, Actor-Critic is a combination of policy optimization and value optimization, the Actor decides which action to take and the Critic evaluates this action and tells the Actor how it should adjust.

During the learning process, PPO stores the interaction of the agent with the environment into a memory. This memory will be used to update the Actor-Critic networks. Unlike DQN's experience replay buffer, the PPO algorithm uses all the memory and not just a sample mini-batch to update the neural networks. Moreover, after each update, the entire memory is cleared and not just the oldest element. The updated networks are then used to collect new experience and refill the memory.

Policy optimization is based on the policy gradient methods, which use a stochastic gradient ascent algorithm to maximize an objective function. The most common objective function has the following form [36]:

$$L^{PG} = \mathbb{E}_t[\log \pi_\theta(a(t)|s(t))\hat{A}(t)] \tag{11}$$

and the gradient estimator is given by :

$$g = \mathbb{E}_t[\nabla_\theta \log \pi_\theta(a(t)|s(t))\hat{A}(t)] \tag{12}$$

where π_θ is the policy neural network with parameters denoted by θ and $\hat{A}(t)$ is an estimator of the advantage function defined by:

$$A(t) = Q(s(t), a(t)) - V(s(t)) \tag{13}$$

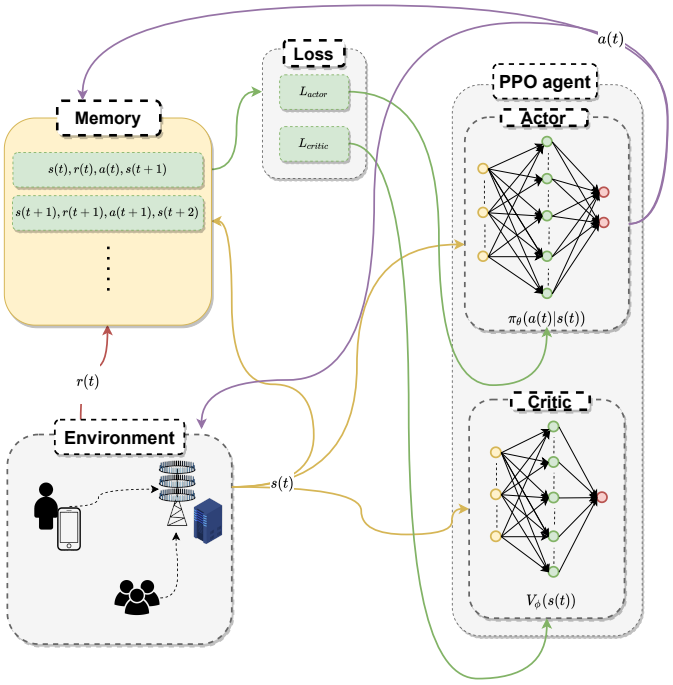

Fig. 3: The structure of our Actor-Critic PPO strategy

However, this approach can be unstable due to the large updates when performing multiple optimization steps on the policy [36]. To solve this issue, PPO provides another objective function in order to limit the new policies to get far from the old policies. Let $\rho_t(\theta)$ denote the probability ratio between the new policy π_θ and the old policy $\pi_{\theta_{old}}$:

$$\rho_t(\theta) = \frac{\pi_\theta(a(t)|s(t))}{\pi_{\theta_{old}}(a(t)|s(t))} \tag{14}$$

The new objective function is given by:

$$L = \mathbb{E}_t[\min(\rho_t(\theta)\hat{A}(t), clip(\rho_t(\theta), 1-\epsilon_c, 1+\epsilon_c)\hat{A}(t))] \tag{15}$$

where ϵ_c is a hyperparameter between zero and one, and $clip$ is a function that clips the probability ratio to keep it inside the interval $[1-\epsilon_c, 1+\epsilon_c]$. The intuition behind this objective function is to keep the old and new policies close to each other by taking the minimum between the unclipped term and the clipped term inside some small interval controlled by ϵ_c. As far as the advantage function, there are several function estimators [37], PPO uses a truncated version of the Generalized Advantage Estimator (GAE):

$$\hat{A}(t) = \delta(t) + (\gamma\lambda)\delta(t+1) + ... + (\gamma\lambda)^{T-t+1}\delta(T-1), \tag{16}$$

$$\text{where } \delta(t) = r(t) + \gamma V(s(t+1)) - V(s(t)) \tag{17}$$

with T is a given length and λ is a hyperparameter.

Our PPO-based strategy for cost minimization is illustrated in Algorithm 2. We start by creating two randomly initialized DNNs: one for the Actor (we denote its parameters by θ) and the other for the Critic (we denote its parameters by ϕ) (lines 2-3). The input of the Actor and the Critic networks is the state vector as defined in our MDP (Section IV-B). The output of the Actor is a layer containing two neurons

according to our action tuple. On the other hand, the output of the Critic is a layer that contains one neuron representing the estimation of the value function of the state given as input. In line 4 we create a memory list to store the experience of the agent. This will be used later to fit the Actor and Critic networks. We run the algorithm for E epochs given as input (line 5). Inside this learning loop, we start by gathering the experience by running the policy π_θ in the environment for T time-steps (line 6). First, the agent observes the actual state of the environment, which consists of the task that needs to be executed and the channel state information between the user and the edge server (line 7). Then, an action is sampled from the probability distribution based on the predictions of the policy network π_θ as in line 8. It consists of the CPU frequency allocated for the task and the transmission power of the offloaded part. This action is used then to compute the size of the task that can be executed locally based on the CPU frequency allocated, and the delay generated from the offloading operation according to the transmission energy (lines 9-10). In line 11, the system cost $\Phi(t)$ is calculated using Equation (6) and the aforementioned computed values, and the negative of this value is assigned to the reward as in line 12. We finish the iteration by storing the collected data into the memory. After collecting the data, the algorithm uses the entire memory to update the DNNs. First, it uses the rewards stored in the memory to build a list of discounted cumulative rewards $\hat{R}$ to be used as a target to fit the Critic network (line 15). Then, the advantage function is estimated using GAE for each time-step along the memory, based on the predictions of the Critic network V_ϕ (line 16). Here the estimate advantage list $\hat{A}$ will be used to compute the Actor loss using Equation (15) as in line 13. Next, we forward the states from the memory through the value function network V_ϕ to create the list R. This is used to find the loss of the Critic network by computing the Mean-Squared Error (MSE) between R and $\hat{R}$:

$$MSE(R, \hat{R}) = \frac{1}{|R|} \sum_{i=1}^{|R|} (R_i - \hat{R}_i) \tag{18}$$

where $|R|$ is the number of elements in the list R. Finally, the algorithm updates the DNNs of the Actor and the Critic. We use the gradient ascent algorithm with the Adam optimizer to update the Actor network π_θ since this is a maximization problem towards the parameter θ that produces the highest return (line 19). Then, we use gradient descent with the Adam optimizer to fit the value function represented by the Critic network (line 20). We repeat this process for a number of epochs until convergence.

Algorithm 2 PPO strategy for cost minimization

1: **Input:** Number of epochs E, Steps per epoch T, clip hyperparameter ϵ_c, GAE factor λ
2: Create the Actor Network (the policy π) and initialize its weights θ with random values
3: Create the Critic Network (the value function V) and initialize its weights ϕ with random values
4: Create the memory list
5: **for** e ← 1 ... E **do**
6: **for** t ← 1 ... T **do**
7: $s \leftarrow$ Get the state vector as defined in section IV-B
8: $a \leftarrow$ Sample the allocated frequency and transmission power from the probability distribution predicted by the Actor network π_θ
9: Compute the size of the part allocated for local execution using equation (3)
10: Compute the delay of the offloaded part using equation (5)
11: Calculate the system cost Φ using equation (6)
12: $r \leftarrow -\Phi$
13: Store the experience in the memory
14: **end for**
 # Using the experience stored in the memory we compute the following:
15: $\hat{R} \leftarrow$ Compute the discounted cumulative rewards
16: $\hat{A} \leftarrow$ Estimate the advantage function using (GAE) (equation (16)) with the help of the critic network V_ϕ
17: $L_{actor} \leftarrow$ Compute the Actor loss using equation (15)
18: $L_{critic} \leftarrow MSE(R, \hat{R})$
 # R obtained by forwarding the states through V_ϕ
19: Update the Actor network π_θ by backpropagating L_{actor} using stochastic gradient ascent with the Adam optimizer
20: Update the Critic network V_ϕ by backpropagating L_{critic} using gradient descent with the Adam optimizer
21: **end for**

V. Experiments and Simulation Results

In this section, we conduct simulation experiments to evaluate and compare the performance of our proposed strategies: DQN and PPO. We used Python as our programming language with Pytorch, an open source library for machine and deep learning, to implement our DNNs and strategies. We run our experiments on the Cedar Compute Canada cluster [38], which is a High Performance Computing (HPC) resource having a total of 94,528 CPU cores, 1352 GPU devices and from 125GB to 3022GB of RAM and a total of 22926TB of storage capacity. The simulations are conducted with the help of the MIMO LAB Dataset [39], an open dataset collected on a 64-antenna base station. It consists of the CSI measurements of a Massive MIMO system and contains 252004 samples. Each sample represents one channel measurement resulting in a complex channel matrix of the form $H \in \mathbb{C}^{64\times100}$.

In our simulation, we set the number of antennas to $N = 64$, the channel bandwidth to $W = 1\ MHz$ and the variance of the AWGN to $\sigma^2 = 10^{-9}\ W$. In addition, we assume that the maximum CPU frequency of the mobile device is $f_{\max} = 1.5\ GHz$ and the maximum transmission power is $p^{tr}_{\max} = 2\ W$, the number of cycles required to execute one bit of data is $\omega = 500\ cycles/bit$ and the coeeficient of the CPU power consumption is $\eta = 10^{-26}$ [18]. Moreover, we assume that the sizes of the input tasks are uniformly distributed between $m_{\min} = 10\ Mbits$ and $m_{\max} = 30\ Mbits$ as the work in [1]. Finally, the width of the time slots is set to $t_0 = 1\ ms$.

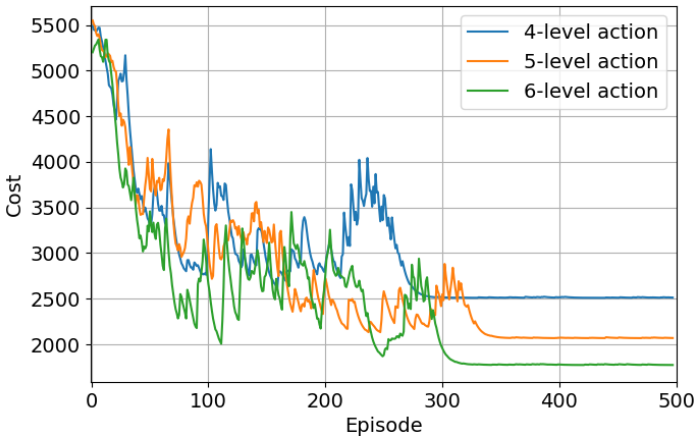

Fig. 4: Convergence of the DQN algorithm under different action levels with $\alpha = 0.5$

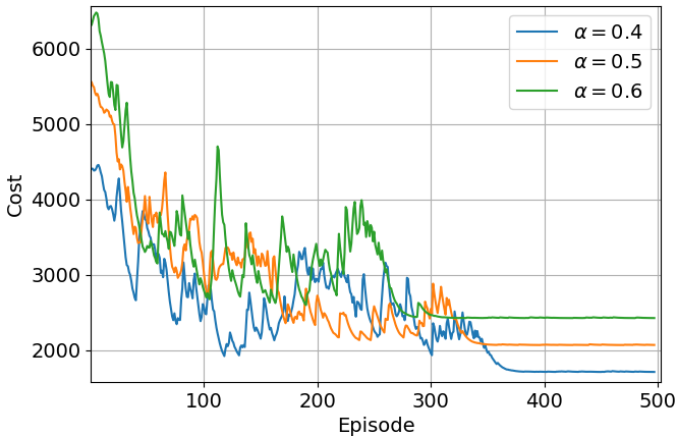

Fig. 5: Performance of the DQN algorithm for different values of α

A. DQN simulation results

To implement DQN, we used a fully connected feed-forward Neural Network with 1 hidden layer having 64 neurons and ReLU (Rectified Linear Unit $ReLU(x) = max(0, x)$) as the activation function. We set $\gamma = 0.99$ and the learning rate for the Adam optimizer to 10^{-3}. The replay buffer capacity is set to $C = 1000$ and the mini-batch size to $B = 900$. Figure 4 shows the convergence of the DQN strategy for multiple levels of action. The algorithm reaches stable performance after 300 episodes of training in average. The results show that when the level increases, the cost decreases. This is because the more we sample from the continuous interval of the action, the more the action space becomes larger, and more the algorithm has the chance to find a better action to minimize the cost.

In Figure 5, we set the action level to 5 and we investigate the effect of the weight factor α used in equation 6 on the DQN strategy. The results show that when we give more weight to the delay ($\alpha = 0.4$), the system cost decreases but the algorithm takes more time to converge. On the other hand, if we give more weight to the energy, the system cost increases and the algorithm converges faster.

Figure 6 shows the effect of the mini-batch size on the performance of the DQN strategy. The mini-batch is sampled randomly from the replay buffer to train the neural network. From the results, we can see that changing the mini-batch size has different effects on the convergence performance. For instance, when the mini-batch size is 900, the algorithm converges gradually towards a minimum cost and then it shows a stable performance. When the mini-batch size is 950, the algorithm does not even converge and it seems to have a random behaviour. Meanwhile, when the batch-size is 1000, the algorithm seems to converge again but quickly and towards another different cost with huge margin difference with the 900 performance.

From the previous results, we notice that, besides the limitation of the sampling mechanism, the DQN algorithm is very sensitive to the hyper-parameters tuning, changing one parameter can lead to a different behaviour and therefore a different performance.

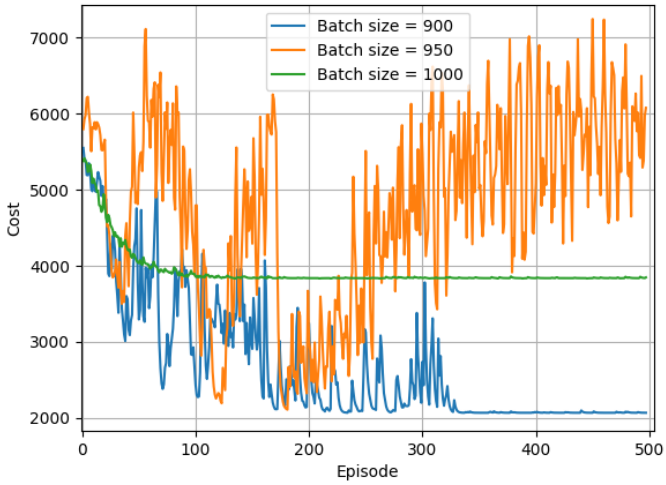

Fig. 6: Performance of the DQN algorithm under different batch sizes

B. PPO simulation results

In the implementation of our PPO-based solution, we used two fully connected feed-forward neural networks for the Actor and the Critic respectively. Each neural network has two hidden layers consisting of 64 neurons each, and having tanh (the hyperbolic tangent function) as the activation function. In addition, we set $\gamma = 0.99$, the GAE hyperparameter $\lambda = 0.97$, and the clip ratio $\epsilon_c = 0.2$. For the learning rates, we set the Actor learning rate to 3×10^{-3} and the Critic learning rate to 10^{-4}. Finally, we run the algorithm for $E = 250$ epochs with $T = 1000$ steps per epoch.

First, we study the convergence of our PPO strategy under $\alpha = 0.5$. As depicted in Figure 7, the algorithm reaches considerably stable performance after 400 episodes and it continues improving with a slow rate after that. Next, we

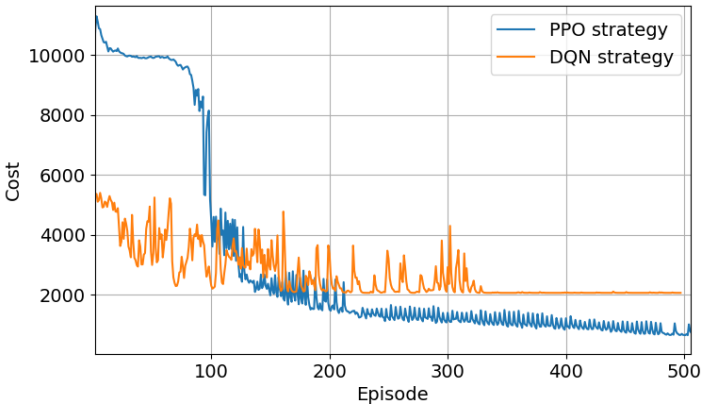

Fig. 7: Convergence of the PPO algorithm compared to the DQN algorithm

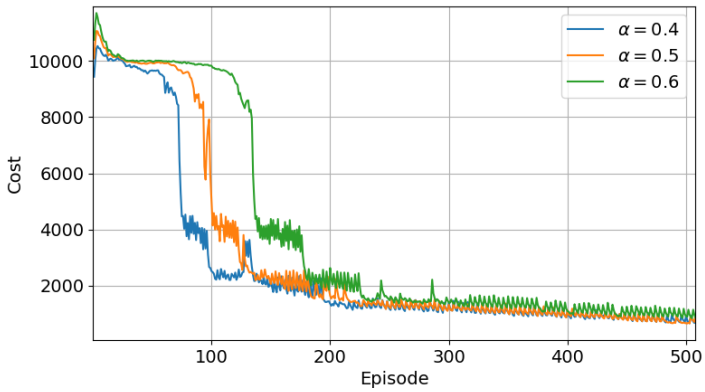

Fig. 8: Convergence of the PPO algorithm for various values of the weight factor α

examine the performance of the PPO algorithm compared to the benchmark DQN algorithm. As clearly shown in the figure, the PPO strategy outperforms the DQN-based solution. We can also notice that our PPO algorithm may reach even better performance with more training time in contrast to the DQN algorithm, which stagnates after 300 episodes.

In Figure 8, we investigate the effect of the weight factor α on the performance of the PPO algorithm. The results show a noticeable stable performance for different values of α where the algorithm converges towards the same minimum cost in contrast with the DQN strategy which is highly sensitive to the hyperparameters.

Figure 9 shows the performance of our PPO algorithm at different values of the parameter T (steps per epoch) which also represents the length of the PPO memory. For every epoch, the training memory will contain T steps to train the Actor and the Critic networks. The results show that the PPO strategy is not very sensitive to the parameter as it converges towards the same minimum cost for its different values.

Unlike DQN, our PPO algorithm provides a solution for the continuous action space and it gives better convergence results and stable performance. However, it is heavy in terms

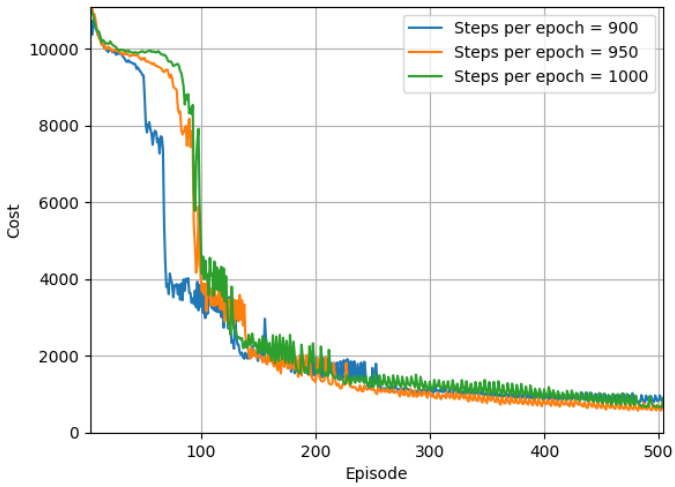

Fig. 9: Convergence of the PPO algorithm under different steps per epoch

of computation because of its architecture and takes more time to converge.

C. Baseline Methods

In order to examine the performance of the proposed solutions, we compare them with different schemes by calculating the system cost resulting from applying each of the following strategies:

- The PPO strategy: the trained model of our PPO algorithm.
- The DQN strategy: the trained model of the DQN algorithm.
- The greedy strategy: chooses the action with the highest performance.
- The random strategy: selects random action for each task size.

Figure 10 shows the comparison results between the four strategies. We vary the task size $m(Mbits)$ inside our interval of simulation and we measure the system cost resulting from each scheme. The results show that our proposed solutions give the best performance among the other strategies. They also show that the PPO strategy gives the best result compared to the benchmark DQN strategy.

VI. Conclusion

In this paper, we addressed the computation offloading problem in a mutative MIMO-based MEC environment considering the stochastic time-varying wireless channels and computation task arrivals. We formulated our problem with the objective of minimizing the long-term cost in terms of energy consumption and offloading delay under the constraints of the limited computation capacity and transmission energy of the MDs. A MDP has been designed for the problem, and two DRL-based strategies have been introduced to learn a dynamic offloading policy without any prior knowledge of the environment. These

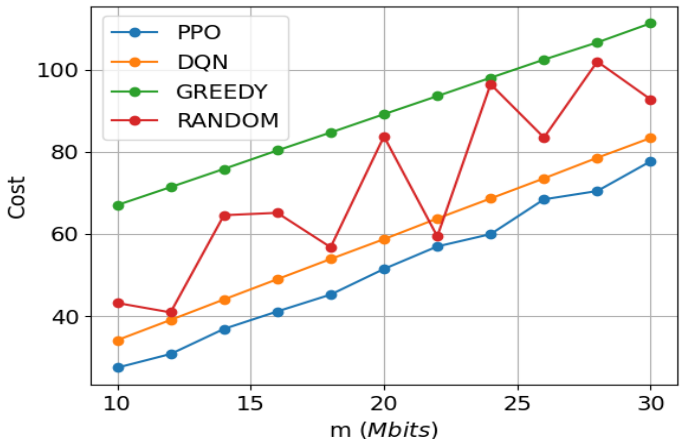

Fig. 10: Performance comparison under different task sizes

two strategies are: the DQN strategy with discrete action space, also used as benchmark, and the PPO strategy with continuous action space. From our simulations using a real dataset, we found that the DQN strategy, although it gives good results, is highly sensitive to hyperparameters tuning in the learning phase compared to the PPO strategy, which exhibits stable performance and much better results. Moreover, both DRL algorithms achieve superior performance over other baseline strategies.

REFERENCES

[1] L. Huang, X. Feng, C. Zhang, L. Qian, and Y. Wu, "Deep reinforcement learning-based joint task offloading and bandwidth allocation for multi-user mobile edge computing," *Digital Communications and Networks*, vol. 5, no. 1, pp. 10–17, 2019. [Online]. Available: https://doi.org/10.1016/j.dcan.2018.10.003

[2] J. Wang, L. Zhao, J. Liu, and N. Kato, "Smart resource allocation for mobile edge computing: A deep reinforcement learning approach," *IEEE Transactions on Emerging Topics in Computing*, pp. 1–1, 2019.

[3] X. Chen, H. Zhang, C. Wu, S. Mao, Y. Ji, and M. Bennis, "Optimized computation offloading performance in virtual edge computing systems via deep reinforcement learning," *IEEE Internet of Things Journal*, vol. 6, no. 3, pp. 4005–4018, 2019.

[4] B. Dab, N. Aitsaadi, and R. Langar, "Q-learning algorithm for joint computation offloading and resource allocation in edge cloud," *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pp. 45–52, 2019.

[5] T. Q. Dinh, Q. D. La, T. Q. Quek, and H. Shin, "Learning for computation offloading in mobile edge computing," *IEEE Transactions on Communications*, vol. 66, no. 12, pp. 6353–6367, 2018.

[6] T. Alfakih, M. M. Hassan, A. Gumaei, C. Savaglio, and G. Fortino, "Task offloading and resource allocation for mobile edge computing by deep reinforcement learning based on SARSA," *IEEE Access*, vol. 8, pp. 54 074–54 084, 2020.

[7] J. Xu, L. Chen, and S. Ren, "Online learning for offloading and autoscaling in energy harvesting mobile edge computing," *arXiv*, 2017.

[8] L. Ale, N. Zhang, X. Fang, X. Chen, S. Wu, and L. Li, "Delay-aware and energy-efficient computation offloading in mobile edge computing using deep reinforcement learning," *IEEE Transactions on Cognitive Communications and Networking*, 2021.

[9] H. Meng, D. Chao, and Q. Guo, "Deep reinforcement learning based task offloading algorithm for mobile-edge computing systems," *ACM International Conference Proceeding Series*, pp. 90–94, 2019.

[10] J. Cao, W. Feng, N. Ge, and J. Lu, "Delay characterization of mobile-edge computing for 6G time-sensitive services," *IEEE Internet of Things Journal*, vol. 8, no. 5, pp. 3758–3773, 2021.

[11] N. Shlezinger, G. C. Alexandropoulos, M. F. Imani, Y. C. Eldar, and D. R. Smith, "Dynamic metasurface antennas for 6G extreme massive MIMO communications," *IEEE Wireless Communications*, vol. 28, no. 2, pp. 106–113, 2021.

[12] R. Chataut and R. Akl, "Massive MIMO systems for 5G and beyond networks—overview, recent trends, challenges, and future research direction," *Sensors (Switzerland)*, vol. 20, no. 10, pp. 1–35, 2020.

[13] M. Zeng, W. Hao, O. A. Dobre, Z. Ding, and H. V. Poor, "Massive MIMO-assisted mobile edge computing: Exciting possibilities for computation offloading," *IEEE Vehicular Technology Magazine*, vol. 15, no. 2, pp. 31–38, 2020.

[14] X. Chen, L. Jiao, and W. Li, "Efficient Multi-User Computation Offloading for," *IEEE/ACM Transactions on Networking*, vol. 24, no. 5, pp. 2795–2808, 2016.

[15] "A Q-learning based method for energy-efficient computation offloading in mobile edge computing," *Proceedings - International Conference on Computer Communications and Networks, ICCCN*, vol. 2020-August, 2020.

[16] X. Chen, H. Zhang, C. Wu, S. Mao, Y. Ji, and M. Bennis, "Performance optimization in mobile-edge computing via deep reinforcement learning," *arXiv*, pp. 1–6, 2018.

[17] R. Zhao, X. Wang, J. Xia, and L. Fan, "Deep reinforcement learning based mobile edge computing for intelligent Internet of things," *Physical Communication journal*, pp. 1–18, 2020.

[18] Y. Wang, M. Sheng, X. Wang, L. Wang, and J. Li, "Mobile-edge computing: Partial computation offloading using dynamic voltage scaling," *IEEE Transactions on Communications*, vol. 64, no. 10, pp. 4268–4282, 2016.

[19] F. Messaoudi, A. Ksentini, and P. Bertin, "On using edge computing for computation offloading in mobile network," *2017 IEEE Global Communications Conference, GLOBECOM 2017 - Proceedings*, vol. 2018-January, pp. 1–7, 2017.

[20] M. Chen and Y. Hao, "Task offloading for mobile edge computing in software defined ultra-dense network," *IEEE Journal on Selected Areas in Communications*, vol. 36, no. 3, pp. 587–597, 2018.

[21] M. E. Khoda, M. A. Razzaque, A. Almogren, M. M. Hassan, A. Alamri, and A. Alelaiwi, "Efficient computation offloading decision in mobile cloud computing over 5G network," *Mobile Networks and Applications*, vol. 21, no. 5, pp. 777–792, 2016. [Online]. Available: http://dx.doi.org/10.1007/s11036-016-0688-6

[22] M. Zeng, M. Zeng, W. Hao, O. A. Dobre, and H. V. Poor, "Delay minimization for massive MIMO assisted mobile edge computing," *IEEE Transactions on Vehicular Technology*, vol. 69, no. 6, pp. 6788–6792, 2020.

[23] Y. Hao, Q. Ni, H. Li, and S. Hou, "Energy-efficient multi-user mobile-edge computation offloading in massive MIMO enabled HetNets," *IEEE International Conference on Communications*, vol. 2019-May, 2019.

[24] C. Ding, J. bo Wang, H. Zhang, M. Lin, and J. Wang, "Joint MU-MIMO precoding and resource allocation for mobile-edge computing," *IEEE Transactions on Wireless Communications*, vol. 20, pp. 1639–1654, 2021.

[25] H. Sami and A. Mourad, "Dynamic on-demand fog formation offering on-the-fly IoT service deployment," *IEEE Transactions on Network and Service Management*, 2020.

[26] H. Sami, A. Mourad, H. Otrok, and J. Bentahar, "Demand-driven deep reinforcement learning for scalable fog and service placement," *IEEE Transactions on Services Computing*, 2021.

[27] H. Sami, H. Otrok, J. Bentahar, and A. Mourad, "AI-based resource provisioning of IoE services in 6G: A deep reinforcement learning approach," *IEEE Transactions on Network and Service Management*, 2021.

[28] W. Zhang, Y. Wen, K. Guan, D. Kilper, H. Luo, and D. O. Wu, "Energy-optimal mobile cloud computing under stochastic wireless channel," *IEEE Transactions on Wireless Communications*, vol. 12, no. 9, pp. 4569–4581, 2013.

[29] R. S. Sutton, F. Bach, and A. G. Barto, *Reinforcement Learning: An Introduction*. MIT Press Ltd, 2018.

[30] V. François-lavet, P. Henderson, R. Islam, M. G. Bellemare, V. François-lavet, J. Pineau, and M. G. Bellemare, "An introduction to deep reinforcement learning. (arxiv:1811.12560v1 [cs.lg]) http://arxiv.org/abs/1811.12560," *Foundations and trends in machine learning*, vol. II, no. 3 - 4, pp. 1–140, 2018.

[31] "A machine learning approach for task and resource allocation in mobile-edge computing-based networks," vol. 8, no. 3, pp. 1358–1372, feb 2021.

[32] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," *Journal of Artificial Intelligence Research*, apr 1996. [Online]. Available: http://arxiv.org/abs/cs/9605103

[33] X. Xu, L. Zuo, and Z. Huang, "Reinforcement learning algorithms with function approximation: Recent advances and applications," *Information Sciences*, vol. 261, pp. 1–31, Mar. 2014. [Online]. Available: https://doi.org/10.1016/j.ins.2013.08.037
[34] V. François-Lavet, R. Fonteneau, and D. Ernst, "How to discount deep reinforcement learning: Towards new dynamic strategies," pp. 1–9, 2015. [Online]. Available: http://arxiv.org/abs/1512.02011
[35] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing Atari with deep reinforcement learning," pp. 1–9, 2013. [Online]. Available: http://arxiv.org/abs/1312.5602
[36] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," pp. 1–12, 2017. [Online]. Available: http://arxiv.org/abs/1707.06347
[37] J. Schulman, P. Moritz, S. Levine, M. I. Jordan, and P. Abbeel, "High-dimensional continuous control using generalized advantage estimation," *4th International Conference on Learning Representations, ICLR 2016 - Conference Track Proceedings*, pp. 1–14, 2016.
[38] "Cedar - cc doc," https://docs.computecanada.ca/wiki/Cedar, (Accessed on 07/05/2021).
[39] "Massive mimo channel state information measurements." [Online]. Available: https://homes.esat.kuleuven.be/~sdebast/measurements/measurements_index.html

Abdeladim Sadiki received the bachelors degree in embedded systems from the National Institute of Post and Telecommunications, Morocco, in 2018. He is currently pursuing his M.Sc. degree in Quality Systems Engineering at Concordia University, Institute for information Systems Engineering (CIISE). His research interests include reinforcement learning, software engineering and services computing.

Jamal Bentahar received the Ph.D. degree in computer science and software engineering from Laval University, Canada, in 2005. He is a Professor with Concordia Institute for Information Systems Engineering, Concordia University, Canada. From 2005 to 2006, he was a Postdoctoral Fellow with Laval University, and then NSERC Postdoctoral Fellow at Simon Fraser University, Canada. He was an NSERC Co-Chair for Discovery Grants for Computer Science (2016-2018). His research interests include computational logics, model checking, reinforcement and deep learning, multi-agent systems, and services computing.

Rachida Dssouli is a professor and the founding of Concordia Institute for Information Systems Engineering at Concordia University, Canada. She earned her Ph.D. degree in Computer Science (1987) from (DIRO) University of Montreal, Canada. Her research interests are in software engineering, communication software engineering, requirements engineering, systems engineering, services engineering, and formal methods. She applied her research to telecommunication, eHealth and avionics software.

Abdeslam En-Nouaary received the Ph.D. degree in computer science from the University of Montreal in 2001. He is currently a Full Professor at INPT, Rabat, Morocco. Before joining INPT, he was an Associate Professor at Concordia University, Montreal, Canada, from 2001 to 2008. His main research interests are modeling, validation, and prototyping of distributed, real-time, and embedded systems.