DEEP REINFORCEMENT LEARNING FOR THE COMPUTATION OFFLOADING IN MIMO-BASED EDGE COMPUTING

Abdeladim Sadiki

A thesis
in
The Department
of
Concordia Institute of Information Systems Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Applied Science in Quality Systems Engineering
Concordia University
Montréal, Québec, Canada

July 2022

Concordia University
School of Graduate Studies

This is to certify that the thesis prepared

By: **Abdeladim Sadiki**

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

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science in Quality Systems Engineering

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining commitee:

Dr. Abdessamad Ben Hamza _____ Chair

Dr. Abdessamad Ben Hamza _____ Examiner

Dr. Zachary Patterson _____ Examiner

Dr. Jamal Bentahar _____ Supervisor

Dr. Rachida Dssouli _____ Co-supervisor

Approved _____

Dr. Mohammad Mannan, Graduate Program Director

_____ 2022 _____

Dr. Mourad Debbabi, Dean of Gina Cody School of Engineering and Computer Science

Abstract

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

Abdeladim Sadiki

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 thesis, 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 introduce a new formulation of the problem as a Markov Decision Process (MDP) and propose two Deep Reinforcement Learning (DRL) algorithms to learn the optimal offloading policy without any prior knowledge of the environment dynamics. First, a Deep Q-Network (DQN)-based algorithm to solve the curse of the state space explosion is defined. Then, a more general Proximal Policy Optimization (PPO)-based algorithm to solve the problem of discrete action space is introduced. Simulation results show that our DRL-based solutions outperform the state-of-the-art algorithms. Moreover, our PPO algorithm exhibits stable performance and efficient offloading results compared to the benchmarks DQN and Double DQN (DDQN) strategies.

Acknowledgments

First and foremost I would like to express my sincere gratitude to my supervisors Dr. Jamal Bentahar and Dr. Rachida Dssouli for their invaluable support, patience and feedback during my whole graduate program. I could not have undertaken this journey without your endless encouragements and precious guidance. Thank you very much.

I would like to extend my gratitude to my beloved parents and my family members for their warm love, emotional and spiritual support during this experience. No amount of words will be enough to tell how grateful I am to you. Thank you for everything you have done for me.

I would also like to thank Concordia University for the funding and the facilities they put under my disposal to achieve this work.

Last, but not least, I would like to thank all my friends for their help, the cherished time we spent together and all the good memories we have shared.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter introduces the general context of our research, which tackles the computation offloading problem in Multi-Access Edge Computing (MEC) servers assisted with the Multiple-Input Multiple-Output (MIMO) technology. Next, it discusses the challenges of the problem and the motivations behind using Reinforcement Learning (RL) to find an optimal solution. It also states the main contributions of this research. Finally, it provides the organizational structure of the thesis and its outline.

1.1 Context

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 [30], some of which may also have a delay-sensitive characteristic [54]. 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 [17].

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) [21]. Although

MCC could improve the battery life of MDs as well as the application experience [25], 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 [11, 54].

To address the above issues, a recent promising approach, known as Multi-access Edge Computing (MEC), has been proposed (Figure 1.1). 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 [9, 21, 56]. 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 [10]. 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 [36].

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 [14], massive Multiple-Input Multiple-Output (MIMO) is a key enabler technology to deliver the needs of the next generation networks [49]. 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 [15]. 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 [59].

1.2 Challenges and Motivations

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 [10]. 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 [11]. MDs are also limited in terms

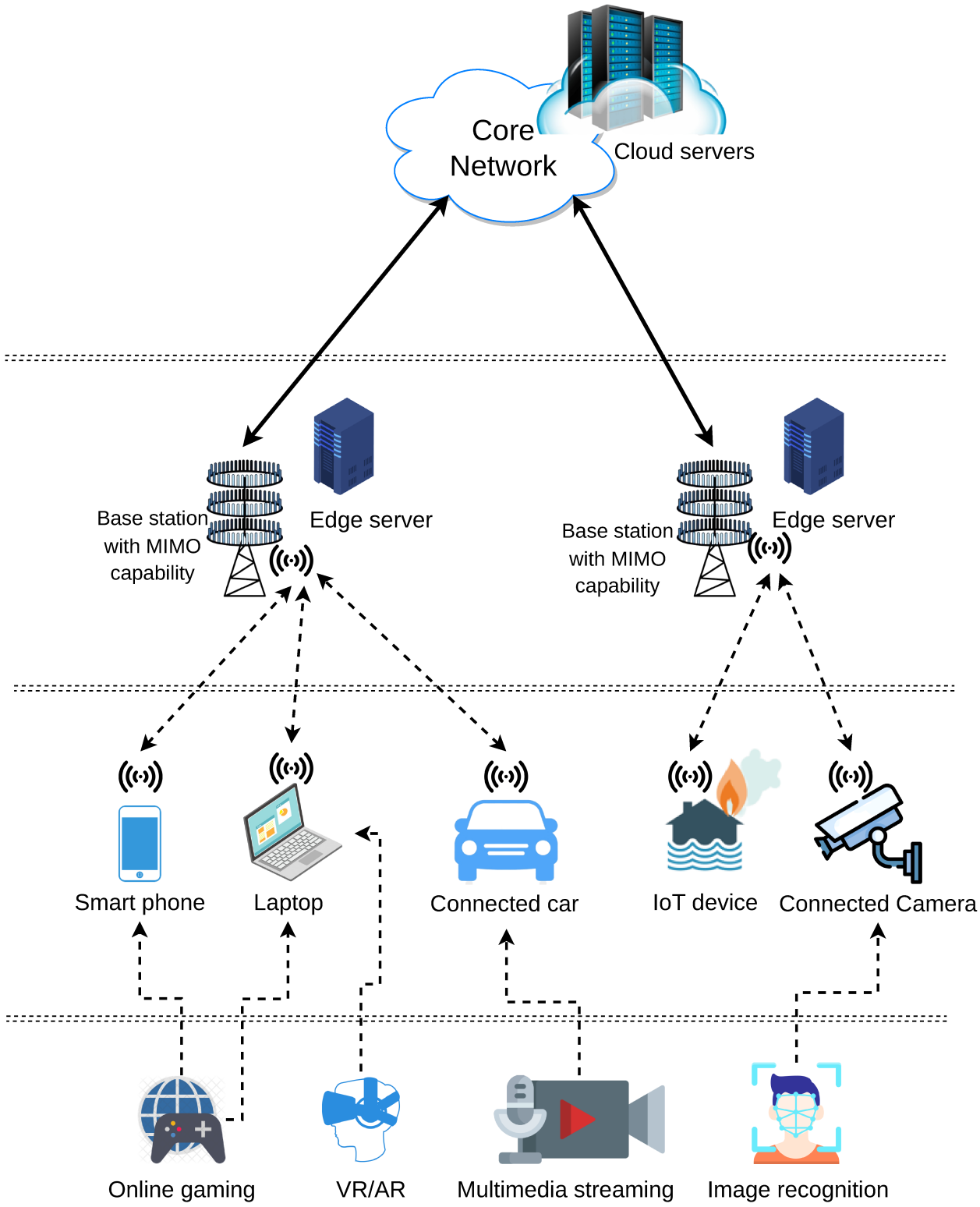


Figure 1.1: Multi-Access Edge Computing Architecture

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 [10, 11]. 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 [19, 25, 30]. The objective functions were designed as reducing the energy consumption, satisfying the latency requirement or finding a trade-off between the energy and latency [4]. 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 [4]. Accordingly, emerging Reinforcement Learning (RL) has been recognized as an effective approach to overcome this issue [4]. Some of the existing proposals use Q-learning [4, 21] to develop dynamic computation offloading methods without any prior knowledge of the MEC dynamics. Yet, due to the curse of state space explosion [18], 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 [30, 62]. 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, [59] considered studying MEC computation offloading under the next generation massive MIMO technology.

1.3 Contributions

In this work, 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 formalize 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 solve the formulated problem, we design a Markov Decision Process (MDP) where the state space, action space and reward function are carefully formalized.
- A DQN-based algorithm is implemented based on our formulated MDP to learn a computation offloading policy that can minimize the system cost.
- A novel PPO-based solution is introduced to derive better offloading policy over the continuous power allocation action space.
- A series of simulations are conducted to compare the performance of the proposed DRL-strategies. The results show that our PPO-based algorithm gives better performance over the DQN strategy that is also used as benchmark. We also compare our strategies with the DDQN solution proposed in [17]. The results reveal that our DRL-based strategies outperform the state-of-the art schemes.

1.4 Thesis Outline

The remaining parts of the thesis are organized as follows. In Chapter 2, we provide the background knowledge required to understand this research, we review the key concepts of RL as well as the most relevant papers related to the computation offloading in MEC with various RL applications. In Chapter 3, the mathematical model of the computation offloading in MIMO-based MEC is proposed and our RL formulation of the problem is given. then, the two DRL-based strategies (DQN and PPO) are explained in details. In Chapter 4, we go through the implementation details of our strategies. We present the dataset and the software technologies used, and we go through the conducted experiments and the simulation results for each strategy, in addition to a performance compassion with other methods. In Chapter 5, we give a summary of our research and some directions for future work.

Chapter 2

Background and Literature Review

In this chapter, we review some background knowledge required for the rest of the thesis. We start by presenting an overview of Reinforcement Learning and its core concepts in Section 2.1. Then, Section 2.1.2 presents Deep Reinforcement Learning since it is adopted to solve our computation offloading problem. Next, in Section 2.2, we provide a review of the relevant related work to the Computation Offloading problem in Multi-Access Edge Computing. We also discuss some proposals that have considered the problem in a Multiple-Input Multiple-Output Edge Computing network. Finally, Section 2.3 summarizes the chapter.

2.1 Reinforcement Learning

2.1.1 Key Concepts

Reinforcement Learning (RL) is sub-field of machine learning where an agent learns the optimal behaviour (through trial and error) in a given environment by executing actions and observing the outcomes. In the RL framework (Fig. 2.1), the agent has to make decisions by observing the current state of the environment. The environment reacts to the agent's actions by sending a reward (positive or negative) and the new state to the agent. The agent then tries to maximize the cumulative rewards through a sequential decision-making process. This formalism incorporates the non determinism aspect as well as the sense of cause and effect which can be applied to a wide range of Artificial Intelligence (AI) problems [28].

One of the most important aspects of RL is that the agent does not require *a priori*

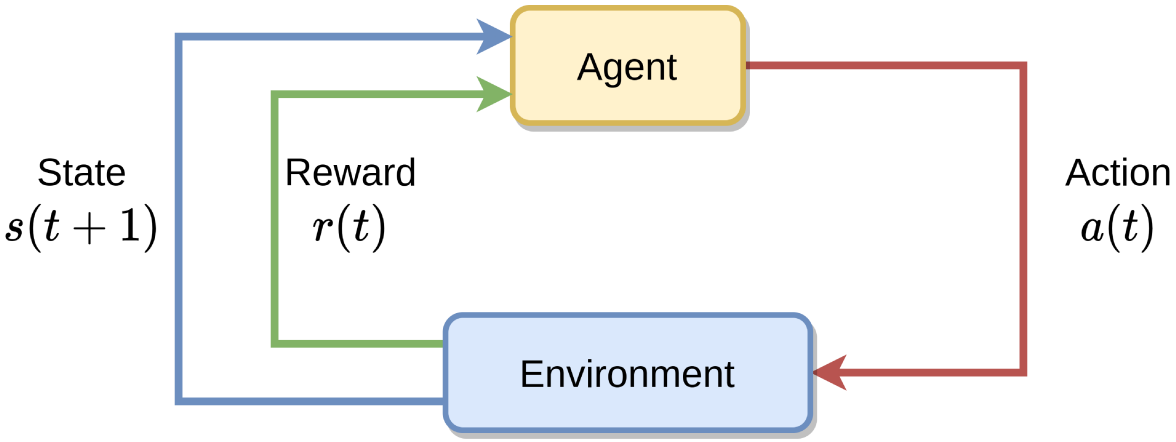


Figure 2.1: Reinforcement Learning Framework

full knowledge of the environment (in contrast to dynamic programming) to learn the best behaviour. The agent, while collecting information through its interaction with the environment, acquires the good behaviours incrementally. The challenge though is that the agent needs to deal with the *exploration/exploitation* dilemma, which is to find a balance between *exploration* (acquiring new experience) and *exploitation* (use of the previously acquired experience) while learning and maximizing its reward [28].

The RL problem is formulated as a discrete time stochastic control process where, at each time step t, the agent observes the state of the environment $s(t) \in \mathcal{S}$. Next, the agent has to take an action $a(t) \in \mathcal{A}$. The agent then receives a reward $r(t)$ and the next state of the environment $s(t+1) \in \mathcal{S}$ (Fig. 2.1).

Markov Decision Process

The underlying mathematical theory behind RL is based on the notion of Markov Decision Process (MDP). An MDP 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 $t+1$.

- $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \longrightarrow \mathbb{R}$ 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.

From the definition, An MDP is used as a mathematical formulation to model the decision making process by which, depending on the current state of the environment and the chosen action, the agent receives a reward. By satisfying the Markov property, MDPs make it easier to compute the transition distribution from one state to the next. The Markov property states that a future state will be determined only by the current state, with all past states being ignored or treated as unimportant [53].

Policy Definition

A policy can be defined as the strategy that the agent uses to select actions. It means that, at each state, the agent decides what actions to take based on its policy. In formal terminology, a policy is a function (usually denoted by π) that maps states to actions. There are two main categories of policies:

- **Deterministic Policies** The policy is given by $\pi : \mathcal{S} \rightarrow \mathcal{A}$, where for each state s, $\pi(s) = a$ is the action to take at that state.

- **Stochastic Policies** In this case, The policy is described by $\pi : \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$ where for each state-action pair (s, a), $\pi(s, a)$ denotes the probability of choosing action a in state s.

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 (usually denoted by τ).

Reward and Return

The reward function defined above in the MDP section is extremely important in RL, it outputs a scalar value depending on the current state, the action chosen and the

next state. It is the mean by which the agent tries to recognize the good behavior. The discounted cumulative reward resulting from the trajectory is called a return:

$$G(t) = \sum_{k=0}^{+\infty} \gamma^k r(t+k+1)$$

The end goal of RL algorithms is to find an optimal policy π^* that maximizes the expected return [27].

Value Functions

In RL, the notion of value functions is used to attribute a value to states (or state-action pairs) in order to describe how good is a state for the agent to be in (or how good is the execution of a given action in a given state). This value is defined in terms of the expected return if the agent starts in that state (or state-action pair). It is worth noting that the expected future rewards the agent will receive depend on what actions it will perform. As a result, value functions are defined according to a particular policy [2].

The value of a state s with respect to a policy π, denoted $V^\pi(s)$ (aka the **state-value function** or the V-function), is defined as the expected return when starting in s and following π forever after. Formally:

$$V^\pi(s) = \mathbb{E}_\pi[G(t)|s(t) = s]$$

where $\mathbb{E}_\pi$ denotes the expected value of the return following the policy π, and t denotes any time step.

Similar to the state-value function, another function is introduced to measure the state-action pair, i.e, how good to perform action a in state s following a policy π. This function is called the **action-value function** (aka the Q-function), denoted $Q^\pi(s, a)$ and defined as the expected return when starting from state s, performing action a and following the policy π thereafter. Formally:

$$Q^\pi(s) = \mathbb{E}_\pi[G(t)|s(t) = s, a(t) = a]$$

Optimal Value Functions

Among all possible value-functions following different policies, the **optimal value function**, denoted $V^*(s)$ is defined as the function that gives the high value for all

states:

$$V^*(s) = \max_{\pi} V^{\pi}(s)$$

The **optimal action-value function**, denoted $Q^*(s, a)$ can also be defined as the maximum expected return, over all policies, the agent gets when starting from state s and taking action a :

$$Q^*(s, a) = \max_{\pi} Q^{\pi}(s, a)$$

Note that $V^*(s)$ is the maximum expected total reward when starting from state s, so $V^*(s)$ is the maximum of $Q^*(s, a)$ over all possible actions. The relation between $V^*(s)$ and $Q^*(s, a)$ is defined formally as follows:

$$V^*(s) = \max_{a} Q^*(s, a)$$

Optimal Policy

As $Q^*(s, a)$ gives the expected return when starting from state s, taking action a and following the optimal policy π^* forever after, there is an important connection between $Q^*(s, a)$ and π^*. The optimal policy $\pi^*(s)$ will select the action that maximizes the expected return when starting from state s [3]. Therefore, if the optimal Q-function $Q^*(s, a)$ is know, the optimal policy $\pi^*(s)$ can be derived directly via:

$$\pi^*(s) = \operatorname*{argmax}_{a} Q^*(s, a)$$

Advantage Functions

Another important quantity in RL is called the Advantage function. It describes how, on average, a chosen action is better than the other actions. Formally, the Advantage function. denoted $A^{\pi}(s, a)$, with respect to a policy π is the quantity that describes how much better to perform action a in state s, over choosing an action randomly, and following the policy π forever after [3]. Mathematically, it is defined as the difference between the Q-function and the V-function:

$$A^{\pi}(s, a) = Q^{\pi}(s, a) - V^{\pi}(s)$$

The Advantage function is significantly important in Policy gradient methods like the Proximal Policy Optimization algorithm which we are going to use to solve our Computation Offloading problem.

Bellman Equations

The Bellman equation is one of the key components of many Reinforcement Learning algorithms, and it can be found all throughout the RL literature. Its importance relies on describing the relation between the value function in a state and the value function of the next state. The Bellman equation states that the value function can be decomposed into two parts, the immediate reward plus the discounted value of the next state. Mathematically:

$$V^{\pi}(s) = \mathbb{E}[r(t+1) + \gamma V^{\pi}(s(t+1))|s(t) = s]$$

Similarly for the Q-function:

$$Q^{\pi}(s,a) = \mathbb{E}[r(t+1) + \gamma \mathbb{E}_{a'} Q^{\pi}(s(t+1), a')|s(t) = s, a(t) = a]$$

The Bellman equation can be derived also for optimal value functions. Rather than computing the expectation following a policy, we take the action that leads to the maximum value [3]:

$$V^{*}(s) = \max_{a} \mathbb{E}[r(t+1) + \gamma V^{*}(s(t+1))|s(t) = s]$$

$$Q^{*}(s,a) = \mathbb{E}[r(t+1) + \gamma \max_{a'} Q^{*}(s(t+1), a')|s(t) = s, a(t) = a]$$

The particularity of the Bellman equation is that it simplifies the computation of the value function. It basically transforms the computation of the value function into a Dynamic Programming problem where the optimal solution can be found by breaking the computation problem down into simpler, recursive sub-problems.

Off-Policy vs On-Policy Learning

RL algorithms can also be categorized depending on the learning policy the agent is using. In **off-policy** methods, the behaviour policy (the agent uses to interact with the environment) may not necessarily be the same as the target policy (the policy that algorithm needs to find). For example, the agent could behave randomly on the environment (the behaviour policy) and off-policy methods can still find the optimal policy (the target policy). The advantage of this approach is to allow the agent to use experience replay from older samples collected using different previous policies. This

improves exploration and sample efficiency because it does not require to recollect new experience whenever the policy has been updated. An example of this approach is the *Q-learning* algorithm [27] [51].

On the other hand, **on-policy** based methods attempt to improve the same policy the agent is using to interact with its environment. This means that the behaviour policy is the same as the target policy. In this case, the agent collects samples using its current policy. Then, the same policy will be updated using the collected experience. The new updated policy will be used in its turn to collect new data and the past experience will be discarded. Therefore, the same policy will get improved gradually until convergence towards an optimal policy [51]. The *Proximal Policy Optimization* algorithm is an example of on-policy based methods.

Model-Based vs Model-Free RL

In the context of RL, a model of the environment is represented as a function which predicts the states transitions and rewards [3]. In terms of MDP, the model of the environment is the combination of the transition probability matrix $\mathcal{T}$ and the reward function $\mathcal{R}$.

A model of the environment can be used for planning, allowing the agent to think ahead considering the various possible future choices and explicitly decide which option to go with. From there, the agent can derive an optimal policy [3]. Methods that use this kind of approach are called **Model-based** methods. A recent well-known example of model-based algorithms is *MuZero* [46] which is built upon the famous *AlphaZero* [50] algorithm. The main issue with this approach is that the model of the environment is usually unknown. In this case, the agent has to learn the model first by approximating the states transitions and rewards function while it interacts with the environment. However, as the learned model might just be approximation of the "real" model, the learned policy might be biased towards the learned model, and behaves terribly on the real model. Therefore, the optimal policy might never be found, which makes model-based learning fundamentally challenging to accomplish [3].

On the other hand, **Model-free** methods don't use a model to infer the policy. In fact, their objective is to approximate the policy without estimating the environment dynamics (states transitions and rewards function). In this case, the agent, using only

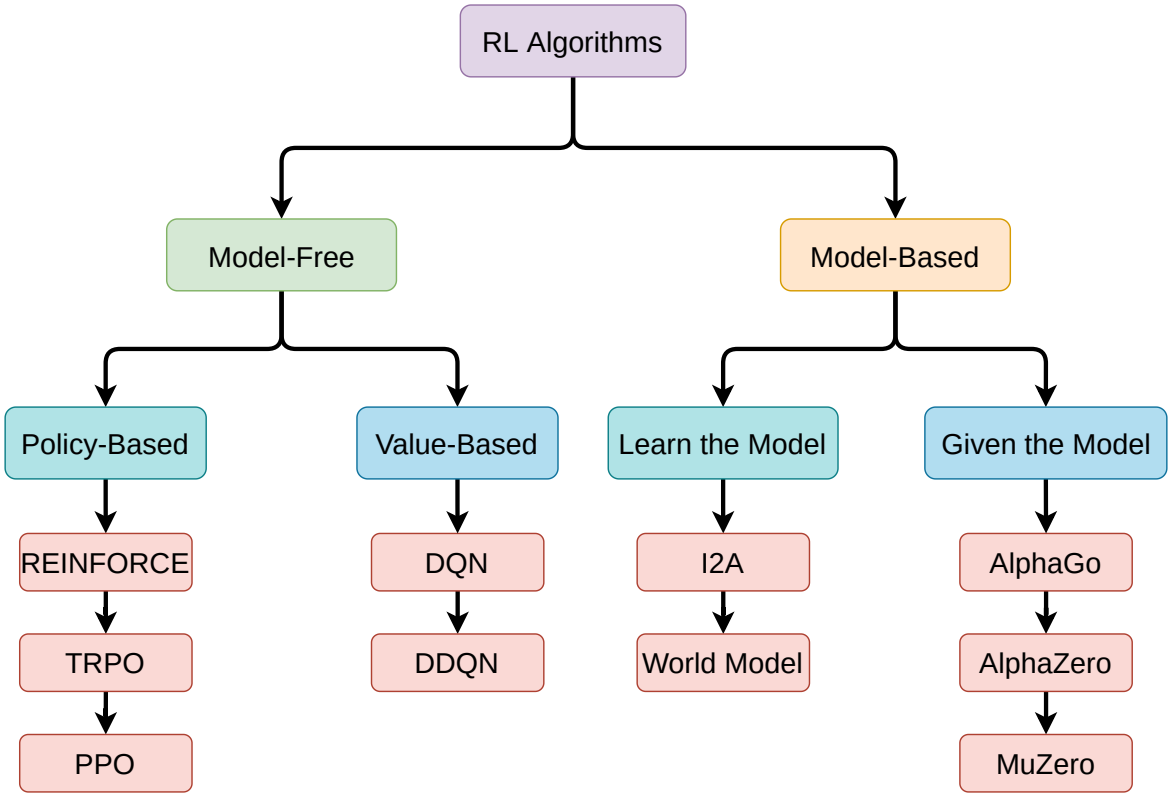


Figure 2.2: Examples of Model-Free and Model-Based Reinforcement Learning Algorithms

the experience gained through its interaction with the environment, either estimates the V-functions and then derive the policy or estimates directly the policy. The main advantage of this approach is that a model of the environment is not required to learn an optimal policy, as a result, it can be applied to different environments and can readily adapt to new and unforeseen situations [6]. This is the reason why the majority of modern RL algorithms are model-free and have been thoroughly developed and validated in comparison to the model-based methods [3][6].

2.1.2 Deep Reinforcement Learning

Deep Learning (DL) is a machine learning technique concerned with algorithms trying to mimic how the human brain learns using a type of structure called Artificial **Deep Neural Networks** (**DNN**). The recent advent of DL pushes the machine learning field to achieve results that were not possible before. For many tasks, such

as computer vision, speech recognition, language translation, deep learning models outperforms all the conventional previous machine learning solutions [34] and even exceeds human performance. The fascinating property of DNN is their ability to learn and recognize patterns of features inside high-dimensional input of data (eg. images) using complex operations through a network with multiple layers of abstraction.

DL has been applied to the field of RL as well, which gave birth to the field of **Deep Reinforcement Learning** (**DRL**). The combination of RL algorithms with DL techniques creates astounding breakthroughs in the AI world like the famous recent *DeepMinds's MuZero* algorithm that can play Chess, Go, Shogi and Atari games with superhuman performance [46].

DL techniques allow RL to scale to previously intractable decision-making situations where the state space and action space are high-dimensional (for example images, sounds or raw data from a robot's sensor) [12]. In this setting, RL relies on DL to represent the policy or other value functions as DNN in order to find the optimal policy. The advantage of using neural networks to estimate the learned functions relies on the fact that DNN are well suited to deal with high-dimensional data, which is the case of most real-world problems, and they can learn gradually as the data comes in from the agent's experience within the environments.

DRL has already been used to solve a variety of problems, including robotics, where the robot can learn an optimal policy based on camera inputs [35]. Self driving cars, where the agent can learn the various aspects of driving a car such as object and lane detection, trajectory optimization, steering, acceleration, breaking and so many more [33]. Natural language processing, where the agent can learn how to answer questions [20] and how to summarize a text [42]. Healthcare, where the RL agent can learn policies to a wide range of applications such as automated medical diagnosis, dynamic treatment regimes for chronic diseases, drug discovery, etc [58]. Engineering tasks such as energy optimization [39], industrial process control [40] and Multi-Access Edge Computing which has been studied in this work.

2.2 Computation Offloading in Multi-Access Edge Computing

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 [23]. 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 [55], 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 [37] 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 [19]. 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 [16] 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 [32], 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.

2.2.1 Computation Offloading in MEC assisted with MIMO Technology

Most of the previous mentioned works 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 [60]. For this reason, researchers begin to investigate MEC assisted with the MIMO technology. In [59], 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 [29] 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 [59], the simulation results revealed that the use of MIMO communication in MEC reduces sufficiently the energy consumption and time delay during computation offloading.

2.2.2 Reinforcement Learning for Computation Offloading in MEC

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 [18]. 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 [10]. In fact, they mainly use heuristics to yield feasible solutions, which makes them subject to two major limitations [43] [44] [45]:

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 [21], 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 [4]. 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 [10], which is mostly the case in a MEC environment. DRL is heavily used in recent proposals to address this issue. In [18], 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 [17] using the Double DQN (DDQN) algorithm, which gave significant improvement in computation offloading performance. The authors in [30] 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 [62] 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 Chapter 4) show that the PPO algorithm has better performance.

2.3 Summary

In this chapter, we provided the foundation background and key concepts of the field of RL needed for the rest of this thesis. We also presented a review of the most relevant studies related to our work, including the computation offloading in MEC and in MEC assisted with the MIMO technology. Different RL and DRL solutions that have been proposed for computation offloading are also reviewed. In the next chapter, we will present the theoretical approach adopted to tackle our computation offloading problem, the system model, the application of the RL framework and our DRL-based algorithms used to find an optimal solution.

Chapter 3

Computation Offloading Modeling and DRL-based Algorithms

This chapter starts by presenting the system model and the mathematical formulation of the MEC computation offloading problem in a MIMO based architecture. Next, the problem is formulated in the light of the RL framework where a Markov Decision Process (MDP) of the problem is provided. Then, two DRL-based strategies are proposed. The first is based on the Deep Q-Network (DQN) algorithm and the other is based on the state-of-the-art Proximal Policy Optimization (PPO) algorithm. Finally, the chapter is concluded with a brief summary and a comparison of our proposed strategies.

3.1 System Model and Problem Formulation

In a MEC architecture, multiple users are connected to a base station equipped with a high performance server (Fig. 3.1). The servers are deployed at the edge of the network to provide cloud-like services in close proximity of end-users. Thus, the users offload their computation-intensive tasks to the close edge server they are connected to, instead of remote cloud infrastructure. Thereby, enabling a wide range of applications and services to work in real or near-real time performance.

In our problem formulation, we consider a set of K mobile single-antenna users $\mathcal{U} = \{u_1, \ldots, u_K\}$ communicate with a massive MIMO based MEC server through a Base Station (BS) equipped with N antennas ($N >> K$). For simplicity but without

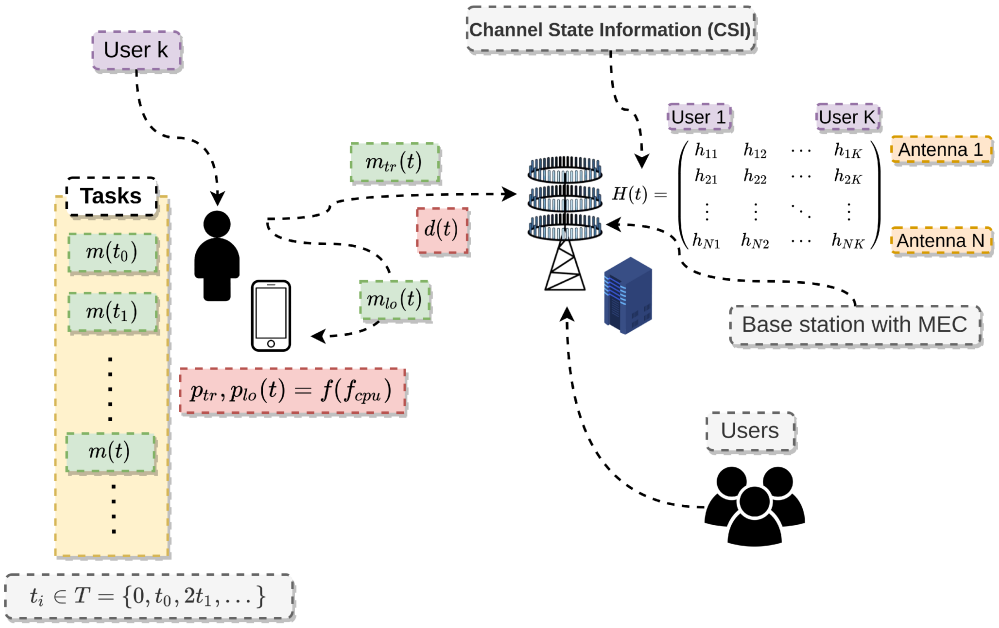


Figure 3.1: Problem description

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, $T = \{0, t_0, 2t_0, \dots\}$. The variables used in our problem formulation are listed in Table 3.1.

3.1.1 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 $\boldsymbol{x}(t) = [x_1(t), x_2(t), \dots, x_K(t)]^\top \in \mathbb{C}^{K\times 1}$ be the complex transmitted signal vector from all the users to the BS. Table 3.2 summarizes all the variables used in the wireless channel model.

The $(N \times 1)$ received signal vector in the uplink at the BS $\boldsymbol{y}(t) \in \mathbb{C}^{N\times 1}$ is expressed by:

$$\boldsymbol{y}(t) = \boldsymbol{H}(t)\boldsymbol{x}(t) + \boldsymbol{b}(t)$$

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

Symbol	Meaning
BS	Base station
N	Number of antennas at the BS
n	Antenna n at the base station , $1 \leq n \leq N$
$\mathcal{U}$	Set of all users connected to the BS
K	Number of users
u_k	User k in the network, $1 \leq k \leq K$
t_0	Length of a time slot
T	System time $\{0, t_0, 2t_0, \dots\}$
t	Time step $t \in T$

Table 3.1: Problem Formulation Notations

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

$h_{nk}(t) \in \mathbb{C}$ represents the channel response between the antenna of user k and the n^{th} BS antenna, and $\boldsymbol{b}(t) = [b_1(t), b_2(t), ..., b_N(t)]^\top \in \mathbb{C}^{N\times 1}$ is the Additive White Gaussian Noise (AWGN) which follows a complex normal distribution with zero mean and variance σ^2 : $\boldsymbol{b}(t) \sim \mathcal{CN}(0, \sigma^2 \mathbf{I}_N)$ ($\mathbf{I_N}$ 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. Hence, the ZF-detection matrix at the BS is given by the pseudo-inverse of its CSI Matrix $\boldsymbol{H}$ [60]:

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

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

$$\Gamma_k(t) = \frac{|\boldsymbol{v}_k(t)\boldsymbol{h}_k(t)|^2}{\sigma^2 |\boldsymbol{v}_k(t)|^2} \tag{2}$$

Symbol	Meaning
$\mathbb{C}$	Set of complex numbers
x_k	Transmitted complex signal of user k
$\boldsymbol{x}$	Transmitted signal vector from all the users to the BS
$\boldsymbol{y}$	Received signal in the uplink at the BS
$\boldsymbol{H}$	Channel state information matrix between the users and the BS
h_{nk}	Channel response between user k and the n^{th} BS antenna
$\boldsymbol{b}$	Additive White Gaussian Noise (AWGN) vector
σ^2	Variance of the AWGN
$\mathcal{CN}$	Complex normal distribution
$\mathbf{I}_N$	Identity matrix with N rows and N columns
$\boldsymbol{H}^H$	Hermitian matrix of H
$()^T$	Matrix transpose
$\boldsymbol{V}$	ZF-detection matrix
Γ_k	Normalized effective channel gain for user u_k
C	Achievable channel capacity at user u_k
W	Channel bandwidth
p^{tr}	Transmission power
P^{tr}	Maximum transmission

Table 3.2: Wireless channel model Notations

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

$$C_k(t) = W.log_2(1 + p_k^{tr}(t)\Gamma_k(t)) \tag{3}$$

where W is the channel bandwidth in *Hertz* 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.

3.1.2 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 [4]. 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 [55], 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. Table 3.3 contains all the variables used in our computation model.

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:

$$m_k^{lo}(t) = \frac{t_0 f_k(t)}{\omega_k} \tag{4}$$

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 [55] and [61] by:

$$p_k^{lo}(t) = \eta f_k(t)^3 \tag{5}$$

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:

$$d_k(t) = \frac{m_k^{tr}(t)}{C_k(t)} \tag{6}$$

3.1.3 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:

$$\Phi_k(t) = \alpha(p_k^{lo}(t) + p_k^{tr}(t)) + (1 - \alpha)d_k(t) \tag{7}$$

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

Symbol	Meaning
m	Size of a task
f	Frequency of the CPU
F	Maximum frequency of the CPU
m^{lo}	Size of the part of the task that is computed locally
ω	Number of CPU cycles required to execute one bit of data
p^{lo}	Local power consumption
η	CPU hardware architecture coefficient
m^{tr}	Size of the part offloaded to the edge server
d	Time delay to get the computation results

Table 3.3: Computation model Notations

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:

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

s.t

$$f_k(t) \in [0, F_k] \ \ \forall t \in T \tag{8}$$

$$p_k^{tr}(t) \in [0, P_k^{tr}] \ \ \forall t \in T \tag{9}$$

3.2 Modeling the Problem using the RL Framework

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:

3.2.1 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: $\boldsymbol{s}_k(t) = (m_k(t), h_{1k}(t), h_{2k}(t), ..., h_{Nk}(t))$ where $h_{ik}(t), i \in \{1, 2, ..., N\}$ the CSI between the user u_k and the i-th BS antenna.

3.2.2 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 $\boldsymbol{a}_k(t) = (f_k(t), p_k^{tr}(t))$ 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 [5] 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: $\{(f_k(t), p_k^{tr}(t)) | f_k(t) \in \mathcal{F} \text{ and } p_k^{tr}(t) \in \mathcal{P};\ \forall t \in \mathcal{T}\}$

3.2.3 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 7.

3.3 DRL-based Computation Offloading-based Algorithms

3.3.1 Deep Q-Network (DQN)-based Solution

Q-learning is one of most popular RL algorithms [27]. 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*:

$$Q^*(s(t), a(t)) = \mathbb{E}[r(t) + \gamma \max_{a(t+1)} Q^*(s(t+1), a(t+1))] \quad (10)$$

The basic idea behind Q-learning is to use Equation 10 as a simple iterative update:

$$Q(s(t), a(t)) \longleftarrow Q(s(t), a(t)) + \beta[r(t) + \gamma \max_{a(t+1)} Q(s(t+1), a(t+1)) - Q(s(t), a(t))] \quad (11)$$

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 11 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 [31].

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 [57]. This is known as the curse of high dimensionality problem [18]. 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 3.2).

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 [26]. Deep Q Network (DQN) [38] is one of the effective algorithms that combines DL with RL. As shown in Figure 3.2, our DQN strategy uses DNN with weights θ to approximate the Q 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.

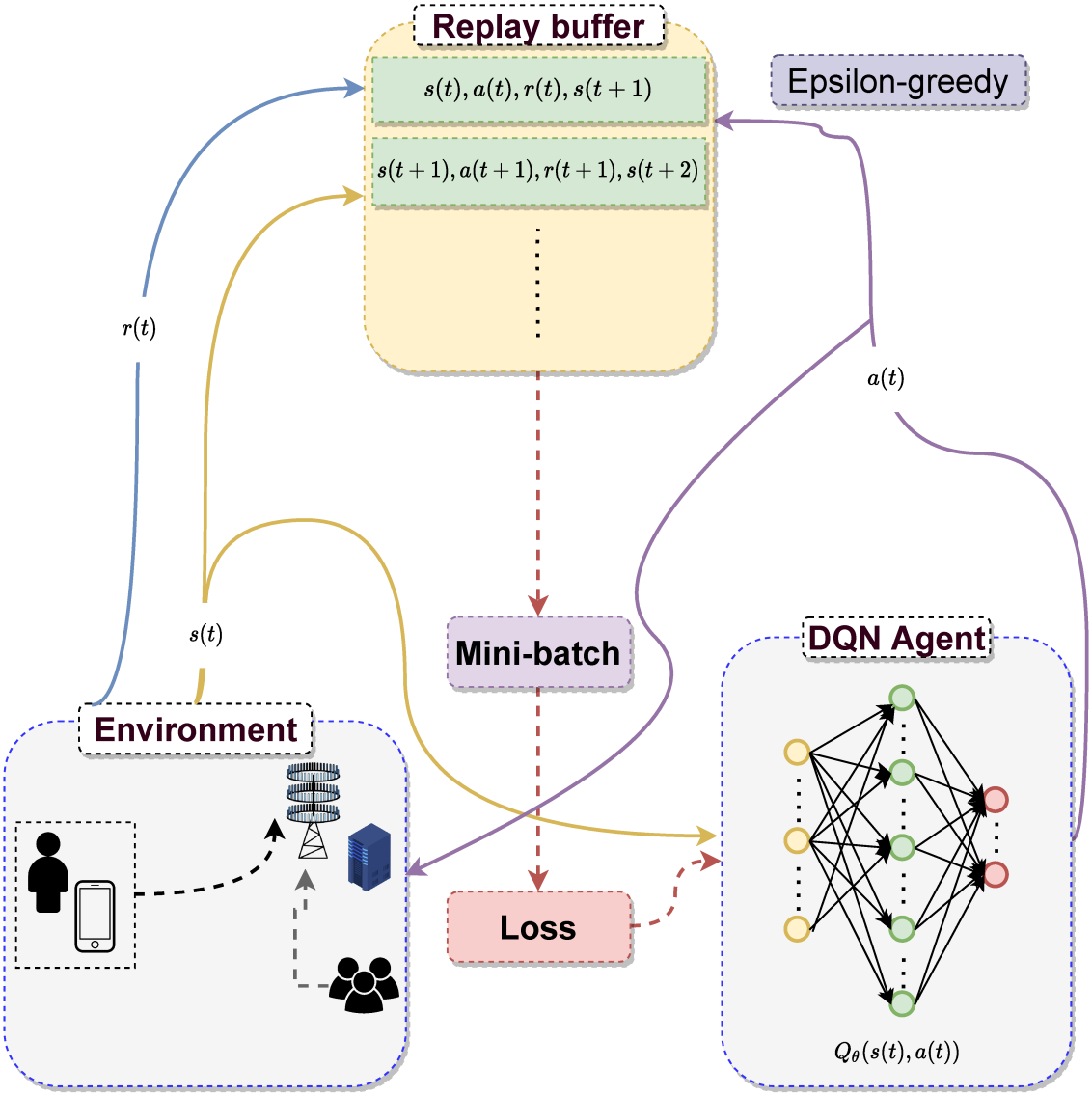


Figure 3.2: The structure of our Deep Q-Network (DQN) strategy

Algorithm 1 DQN strategy for cost minimization

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

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).

3.3.2 Proximal Policy Optimization (PPO)-based Algorithm

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 [48], 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 3.3, 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 [48]:

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

and the gradient estimator is given by :

$$g = \mathbb{E}_t[\nabla_\theta \log \pi_\theta(a(t)|s(t))\hat{A}(t)] \tag{13}$$

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:

$$A(t) = Q(s(t), a(t)) - V(s(t)) \tag{14}$$

However, this approach can be unstable due to the large updates when performing multiple optimization steps on the policy [48]. 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 $\pi_{\theta_{old}}$:

$$\rho_t(\theta) = \frac{\pi_\theta(a(t)|s(t))}{\pi_{\theta_{old}}(a(t)|s(t))} \tag{15}$$

The new objective function is given by:

$$L = \mathbb{E}_t[\min(\rho_t(\theta)\hat{A}(t), clip(\rho_t(\theta), 1-\epsilon_c, 1+\epsilon_c)\hat{A}(t))] \tag{16}$$

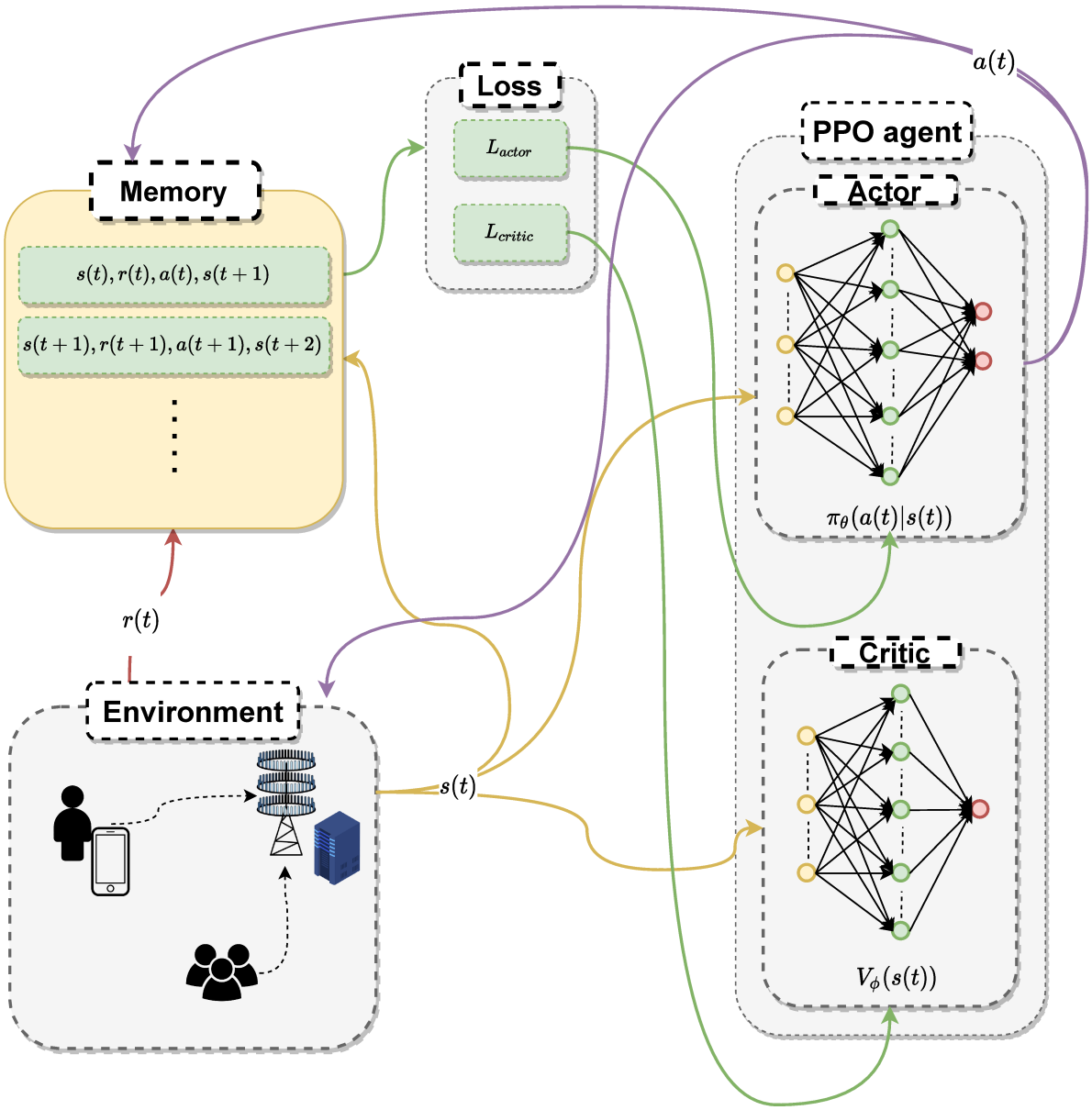


Figure 3.3: The structure of our Actor-Critic PPO strategy

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 [47], PPO uses a truncated version of the Generalized Advantage Estimator (GAE):

$$\hat{A}(t) = \delta(t) + (\gamma\lambda)\delta(t+1) + ... + (\gamma\lambda)^{T-t+1}\delta(T-1), \tag{17}$$

$$\text{where } \delta(t) = r(t) + \gamma V(s(t+1)) - V(s(t)) \tag{18}$$

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 3.2). 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 (7) 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 (16) 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 $\hat{R}$:

$$MSE(R, \hat{R}) = \frac{1}{|R|} \sum_{i=1}^{|R|} (R_i - \hat{R}_i) \tag{19}$$

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.

3.4 Summary

In this chapter, we presented the system design of the computation offloading problem in MIMO-based MEC, specifying the wireless channel model, the computation model and the system cost. Moreover, in order to apply the RL methodology, we formulated the problem as an MDP defining its state space, action space and reward function. Then we proposed two DRL-based solutions to solve the problem. The first strategy is based on the DQN algorithm to deal with the large state space. The second strategy is based on the PPO algorithm which, in addition to what the DQN algorithm provides, solves the issue of the discrete action space. In the next chapter, we will present the implementation details of these algorithms and the results of the simulations and experiments.

Algorithm 2 PPO strategy for cost minimization

Input: Number of epochs E, Steps per epoch T, clip hyperparameter ϵ_c, GAE factor λ
Create the Actor Network (the policy π) and initialize its weights θ with random values
Create the Critic Network (the value function V) and initialize its weights ϕ with random values
Create the memory list
for e $\leftarrow$ 1 ... E **do**
    for t $\leftarrow$ 1 ... T **do**
        $s \leftarrow$ Get the state vector as defined in section 3.2
        $a \leftarrow$ Sample the allocated frequency and transmission power from the probability distribution predicted by the Actor network π_θ
        Compute the size of the part allocated for local execution using equation (4)
        Compute the delay of the offloaded part using equation (6)
        Calculate the system cost Φ using equation (7)
        $r \leftarrow -\Phi$
        Store the experience in the memory
    end for
    # Using the experience stored in the memory we compute the following:
    $\hat{R} \leftarrow$ Compute the discounted cumulative rewards
    $\hat{A} \leftarrow$ Estimate the advantage function using (GAE) (equation (17)) with the help of the critic network V_ϕ
    $L_{actor} \leftarrow$ Compute the Actor loss using equation (16)
    $L_{critic} \leftarrow MSE(R, \hat{R})$
    # R obtained by forwarding the states through V_ϕ
    Update the Actor network π_θ by backpropagating L_{actor} using stochastic gradient ascent with the Adam optimizer
    Update the Critic network V_ϕ by backpropagating L_{critic} using gradient descent with the Adam optimizer
end for

Chapter 4

Experiments and Simulation Results

This chapter presents the conducted simulation experiments used to evaluate and compare the performance of our proposed strategies: DQN and PPO. We start by introducing the dataset used to conduct the experiments in Section 4.1. Then, We present a brief review of the different technologies and frameworks used in the implementation of the DRL-based algorithms in Section 4.2. The results of the simulation along with the performance comparison with other baseline methods are provided in Section 4.3. Finally, Section 4.4 concludes this chapter.

4.1 Dataset

To conduct the simulation experiments and evaluate the performance of our proposed strategies, the open MaMIMO dataset [22] [1] was used. It was collected at KU Leuven ESAT-TELEMIC using a massive MIMO tesbed. It contains the CSI measurements of a massive MIMO system with many single-antenna users connected to a base station equipped with 64 antennas transmitting and receiving simultaneously. 252004 CSI samples in total were generated with location accuracy of less than 1 mm and each CSI measurement is represented by a matrix of the form $H \in \mathbb{C}^{64\times100}$.

[1] `https://homes.esat.kuleuven.be/~sdebast/measurements/measurements_index.html`

4.2 Technology Stack Used for the Implementation

Python

Python is an open source object oriented programming language that supports the development of a diverse variety of applications. It becomes one of the most popular programming languages among the Artificial Intelligence (AI) community developers in the last decade due to its numerous benefits that make it particularly ideal for machine learning and deep learning projects.

Python has acquired a large ecosystem of machine learning frameworks which can be applied out of the box and make the development of complex machine learning and deep learning projects relatively easy to build. In addition, Python's syntax is simple which makes it easy to use and read. It helps also in rapidly prototype and iterate the machine learning models and makes the project accessible even for non-or-novice programmers who are usually part of any real-world application. Moreover, Python comes with a variety of visualization tools that represents the large amount of data in machine learning projects in a human-readable format. Furthermore, Python enjoys a big thriving community of developers who are providing excellent support and high-quality documentation online.

Thanks to its countless advantages, Python remains undoubtedly the best choice for AI applications compared to other programming languages. It is being used by many big firms and companies due to its power and scalability to handle massive machine learning and deep learning projects.

PyTorch

PyTorch is an open source Python framework for machine learning, developed by the Facebook AI Research team to facilitate the path from research to production [7]. It is an optimized library to process calculations on tensors using GPUs and CPUs in Deep Learning (DL) applications. PyTorch is based on four architectural design principles that provide great flexibility in terms of ease of use and implementation speed, as stated on the Pytorch paper [41]:

- **Be Pythonic**: Python's ecosystem is very popular among data scientists and

machine learning practitioners in general, that's why PyTorch uses Python and keeps its programming interface intuitive and consistent. Also to easily integrate with Python's tools for plotting, debugging and data processing.

- **Put researchers first**: PyTorch's aim is to make it as easy and productive for researchers as possible. This is achieved by handling all the computations complexities of deep learning models internally and expose intuitive APIs to the programmer.
- **Provide pragmatic performance**: Although PyTorch offers great flexibility and implementation speed, it delivers compelling performance. It also provides additional tools for researchers if they want to control and improve the performance of their code.
- **Worse is better**: The field of AI is continually progressing, having a simple internal implementation of PyTorch even if it is slightly incomplete, will make it easy to maintain, fair enough to implement new features given the limited engineering resources and capable to adapt to new situations.

PyTorch's particularity is that it uses dynamic tensor computations compared to static dataflow graphs that is being used by many popular deep learning frameworks such as TensorFlow [8]. Table 4.1 gives a brief head to head comparison between PyTorch and TensorFlow. Although static computation graphs offer enhanced performance and scalability, they are hard to use, not easy to debug and not flexible. Therefore, PyTorch's goal is to overcome those limitations (by leveraging dynamic eager computation with automatic differentiation and GPU acceleration) without sacrificing performance. This achived trade-off makes PyTorch a very popular framework among the AI research community [41].

Compute Canada

We run our experiments on *Compute Canada* which is a powerful High Performance Computing (HPC) platform for research. It integrates high performance computers (close to a petaFLOP of computing performance), long-term storage that can be accessed online with high speed read and write over Canada's high performance

Framework	PyTorch	TensorFlow
Developed By	FAIR Lab (Facebook AI Research Lab)	Google Brain Team
Computation mechanism	Dynamic tensor execution	Static dataflow graphs
Focus	Research	Industry
Visualization	Tensorboard	Tensorboard
Debugging	Python's standard debuggers	tfdbg library (TensorFlow debugger tool)
Popular RL projects	rlpyt, SLM-Lab, jetson-reinforcement, ...	TF-Agents, trfl, deep-rl-tensorflow, ...

Table 4.1: Comparison between PyTorch and TensorFlow

networks, resources and tools for data analysis, and many research facilities around the country. [13]

Researchers, independently of their disciplines, their skill levels, their system or their location, can access the various resources of Compute Canada through dedicated, secure, in-house software programs developed by highly qualified specialists working in the organization. In addition, they develop new software, both on the system level and the application level that can work inside this sophisticated and complex architecture. Furthermore, Compute Canada provides a full ecosystem of computing facilities, ranging from small lab computing to very large systems to the GPGPU architectures and cloud computing. It also provides a vast amount of data storage (petabytes of capacity) to accommodate the large size of data-sets used in the scientific research such as machine learning, gene sequencing, medical imaging, satellite data, etc. [13]

4.3 Simulation Results

In this section, we conduct simulation experiments to evaluate and compare the performance of our proposed solutions: DQN and PPO-based algorithms. We used Python with the Pytorch library and we run our experiments on the Cedar Compute

Parameter	Value
N	64
W	$1MHz$
σ	10^{-9}
f_{max}	$1.5GHz$
p_{max}^{tr}	$2W$
ω	$500cycles/bit$
η	10^{-26}
t_0	$1\ ms$

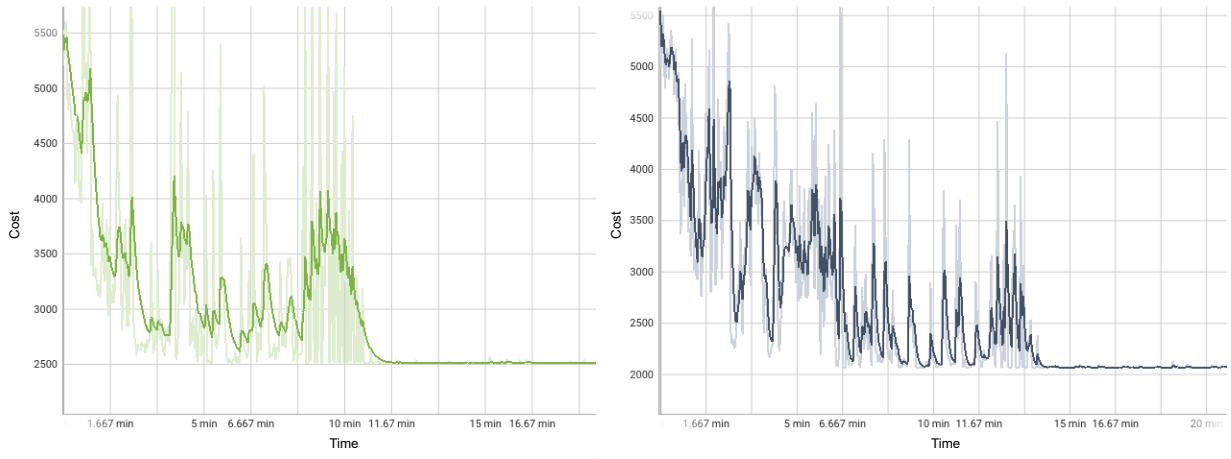
Table 4.2: Simulation Parameters

Canada cluster [1], 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 (Section 4.1).

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 $p_{\max}^{tr} = 2\ W$, the number of cycles required to execute one bit of data is $\omega = 500\ cycles/bit$ and the coefficient of the CPU power consumption is $\eta = 10^{-26}$ [55]. 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 [30]. Finally, the width of the time slots is set to $t_0 = 1\ ms$. All the parameters are listed in Table 4.2. We compared our solutions with the following benchmarks: DDQN, GREEDY and RANDOM.

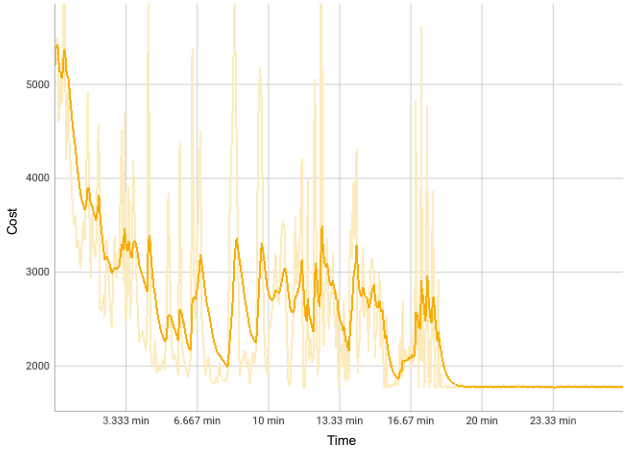
4.3.1 DQN-based Algorithm

To implement the DQN-based algorithm, 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 4.1 shows the convergence of the DQN strategy for multiple levels of action. The algorithm reaches stable performance



(a) 4-level action

(b) 5-level action



(c) 6-level action

Figure 4.1: Convergence time of the DQN strategy under different action levels

after 300 episodes of training in average.

Figure 4.2 shows all the levels on the same graph. 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 4.3, we set the action level to 5 and we investigate the effect of the weight factor α used in equation 7 on the DQN strategy. The figure shows that DQN converges for the different values of α. The results in Figure 4.4 show the 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

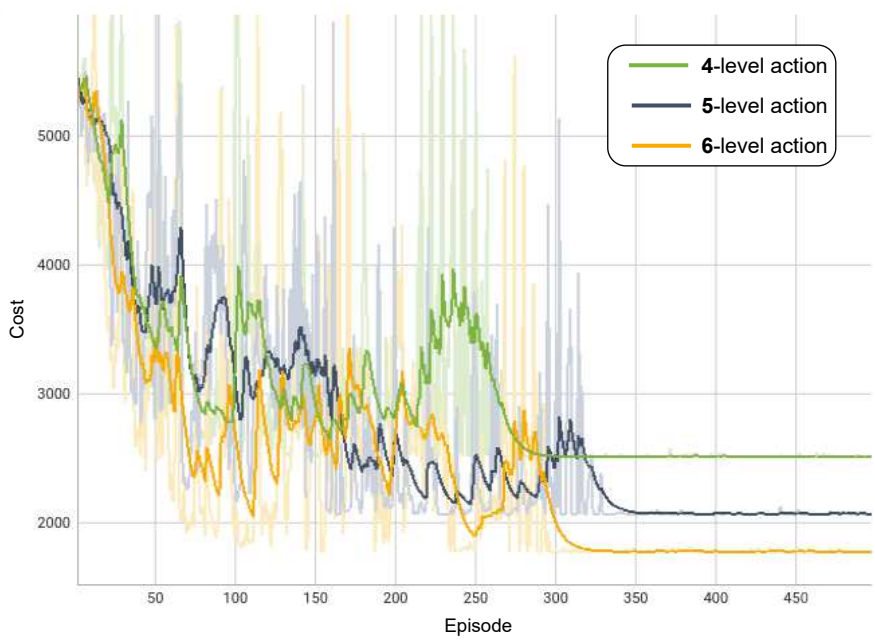
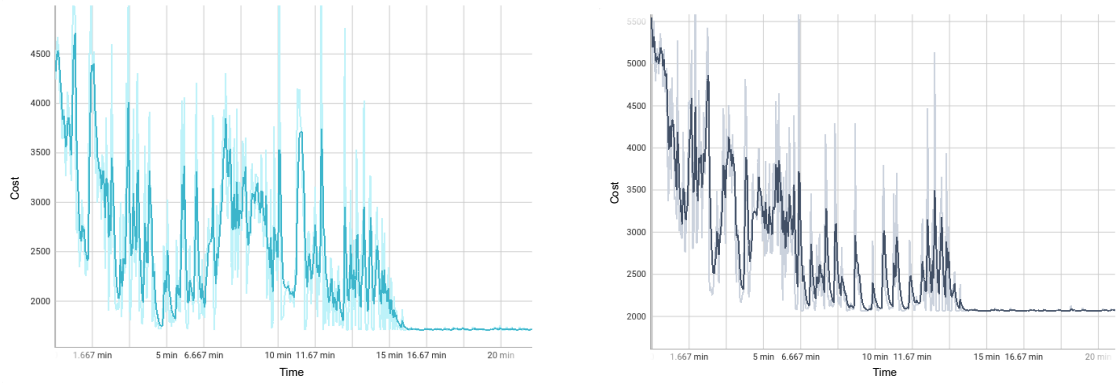


Figure 4.2: Comparison between the effect of different action levels on the DQN strategy convergence

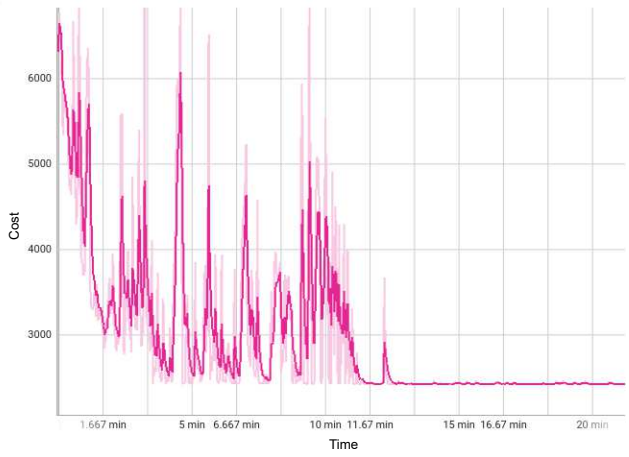
to the energy, the system cost increases and the algorithm converges faster.

Figure 4.8 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 on the figure 4.6, 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.



(a) $\alpha = 0.4$ (b) $\alpha = 0.5$



(c) $\alpha = 0.6$

Figure 4.3: Convergence time of the DQN strategy under different values of α

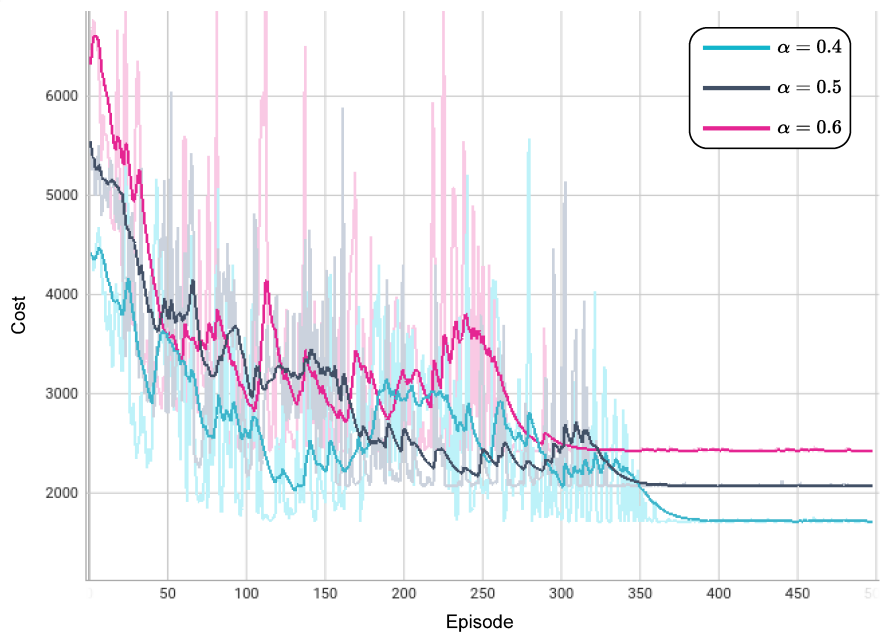


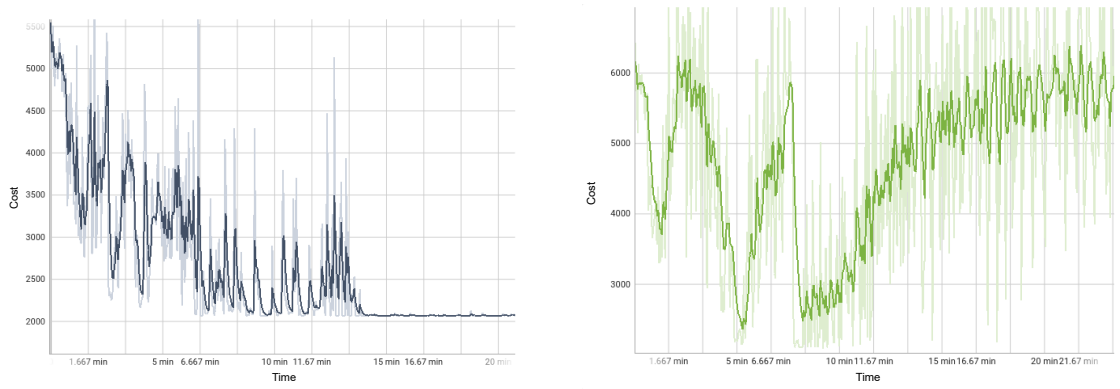
Figure 4.4: Comparison between the performance of the DQN algorithm for different values of α

4.3.2 PPO-based Algorithm

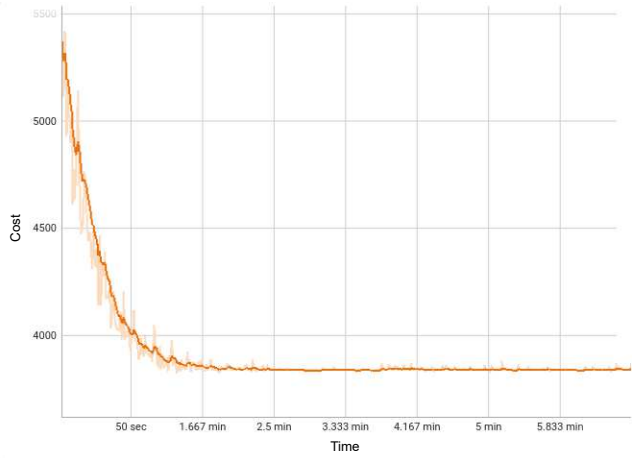
In the implementation of our PPO-based solution, we used two fully connected feed-forward neural networks for the Actor and the Critic respectively. Each neural network has two hidden layers consisting of 64 neurons each, and having tanh (the hyperbolic tangent function) as the activation function. In 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 4.7, the algorithm reaches considerably stable performance after 400 episodes and it continues improving with a slow rate after that. Next, we 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 4.10, we investigate the effect of the weight factor α on the performance



(a) $B = 900$ (b) $B = 950$



(c) $B = 1000$

Figure 4.5: Convergence time of the DQN strategy under different values of min-batch sizes

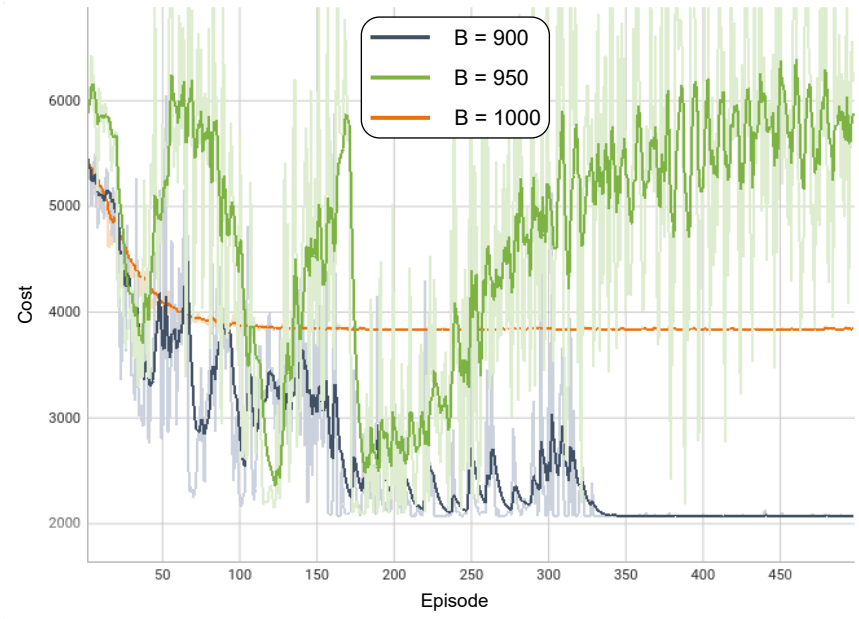


Figure 4.6: Comparison of the performance of the DQN algorithm under different mini-batch sizes

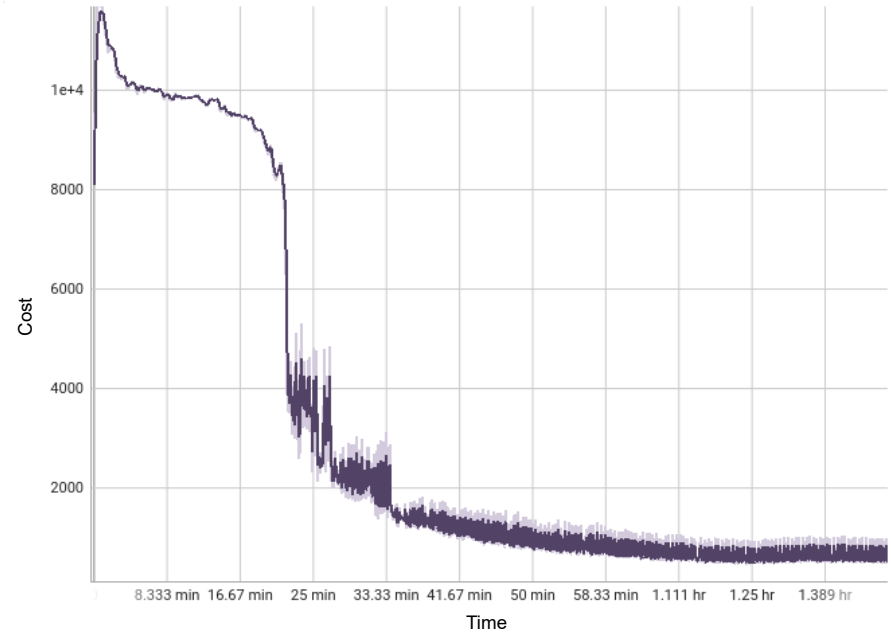
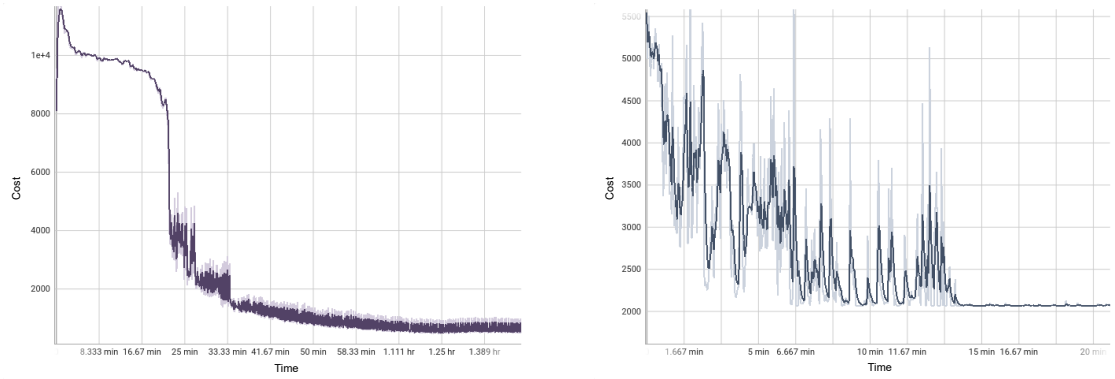


Figure 4.7: Convergence time of the PPO strategy



(a) PPO strategy

(b) DQN strategy

Figure 4.8: Comparison between the convergence performance of the PPO and DQN strategies

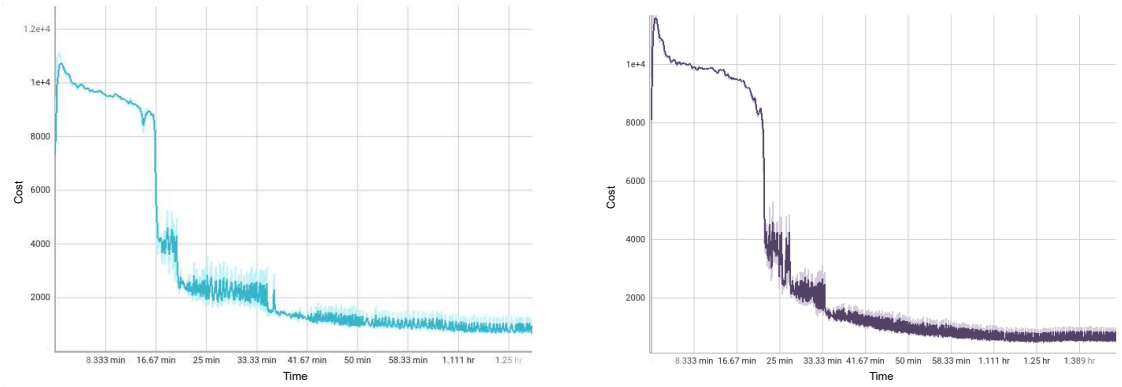
of the PPO algorithm. The results show a noticeable stable performance for different values of α where the algorithm converges towards the same minimum cost in contrast with the DQN strategy which is highly sensitive to the hyperparameters.

Figure 4.12 shows the performance of our PPO algorithm for different values of the parameter S (steps per epoch) which also represents the length of the PPO memory. For every epoch, the training memory will contain S steps to train the Actor and the Critic networks. The results show that there is no significant effect on the convergence time of the PPO algorithm when slightly changing S in contrast with the DQN algorithm. Moreover, PPO converges towards the same minimum under the three different values of S, which means that it is more stable and not very sensitive to hyper-parameters tuning.

4.3.3 Performance Comparison

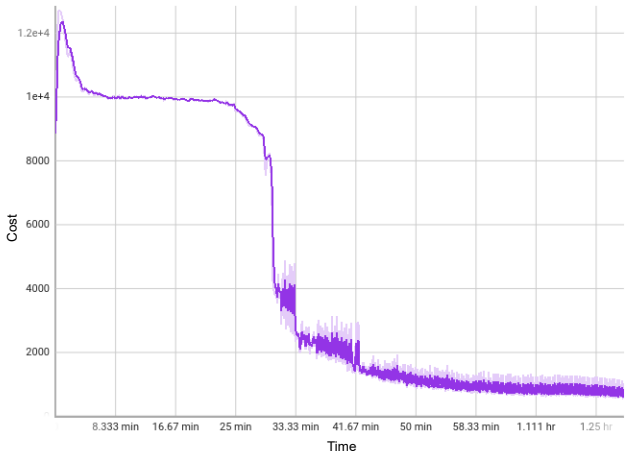
In order to examine the performance of the proposed solutions, we compare them with different schemes by calculating the system cost resulting from applying each of the following strategies:

- The PPO strategy: the trained model of our PPO algorithm.
- The DQN strategy: the trained model of the DQN algorithm.
- The DDQN strategy: proposed in [17], which is an adaptation of the DQN



(a) $\alpha = 0.4$

(b) $\alpha = 0.5$



(c) $\alpha = 0.6$

Figure 4.9: Convergence time of the PPO strategy under different values of α

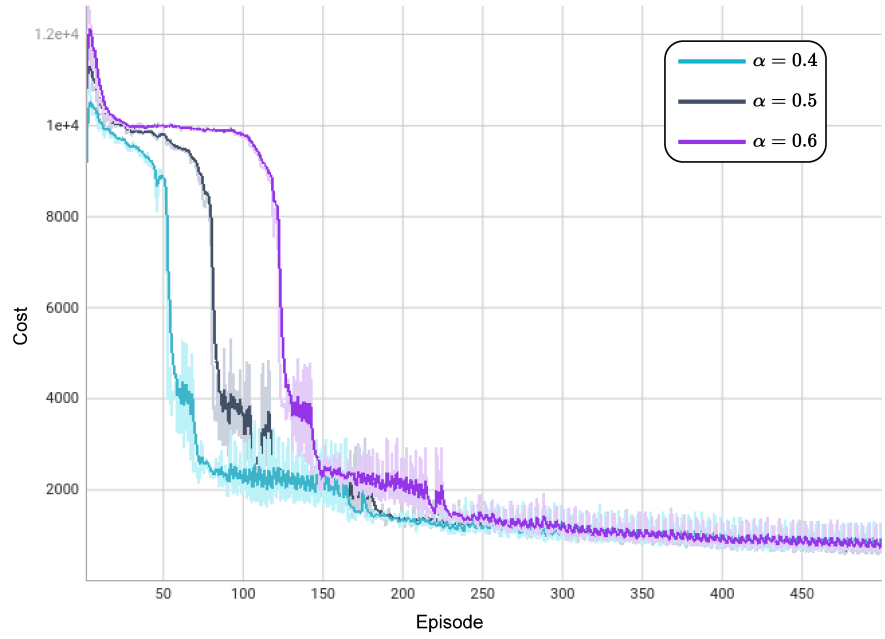


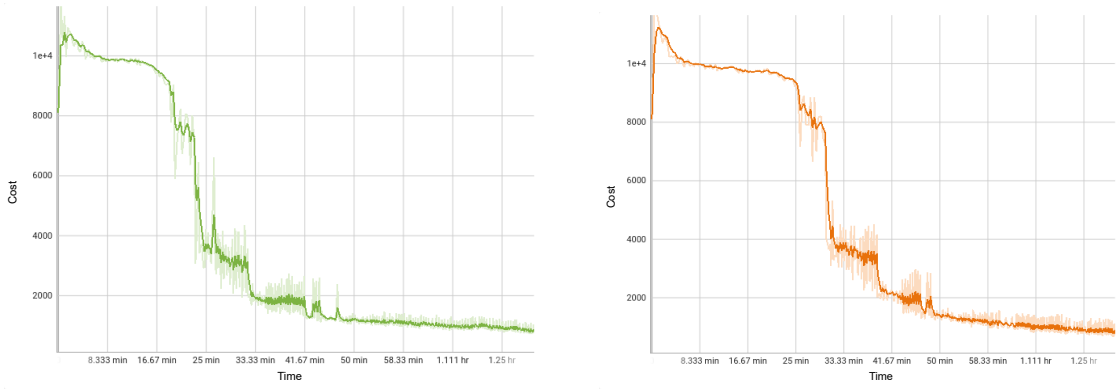
Figure 4.10: Comparison of the performance of the PPO algorithm under different values of α

algorithm to solve the problem of overestimation. It uses two separate neural networks to represent the action-value function and each of which is used to update the parameters of the other [52].

- The GREEDY strategy: chooses the action with the highest performance.
- The RANDOM strategy: selects random action for each task size.

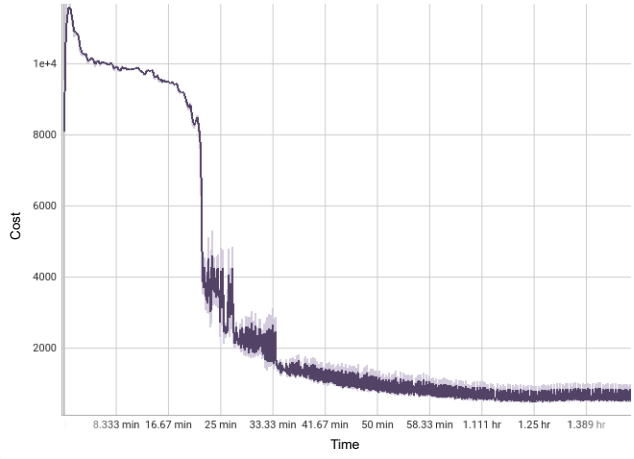
Figure 4.13 shows a side-by-side comparison of the PPO, DQN and DDQN strategies with regards to the training performance. the PPO strategy outperforms both benchmark solutions in terms of the minimum cost. Furthermore, the DDQN strategy does not show any improvement over the DQN strategy as they converge towards the same minimum cost. On the other hand, the PPO algorithm takes way more time to converge compared to the two benchmarks, which is reasonable because of its Actor-Critic architecture. Also, the double neural network architecture introduced in the DDQN algorithm makes the training time taking longer compared to the DQN algorithm.

Figure 4.14 shows the comparison results between the five strategies. We vary the task size $m(Mbits)$ inside our interval of simulation and we measure the system cost



(a) $S = 900$

(b) $S = 950$



(c) $S = 1000$

Figure 4.11: Convergence time of the PPO strategy under different values of S

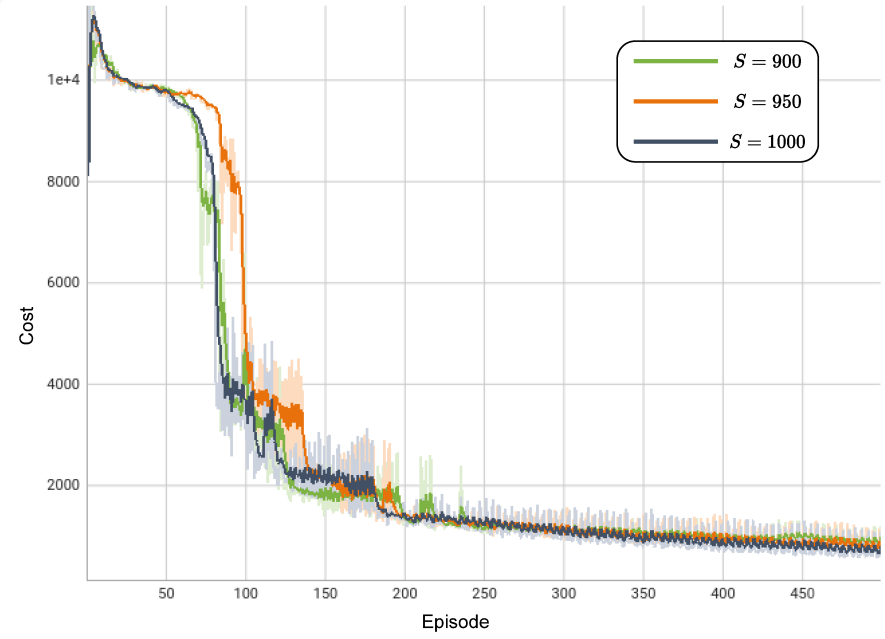
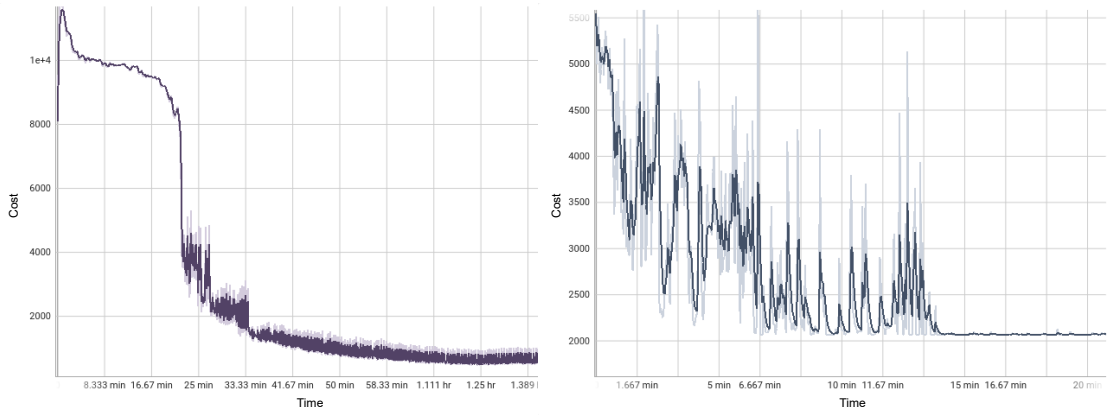


Figure 4.12: Comparison of the performance of the PPO algorithm under different values of S

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 and DDQN strategies.

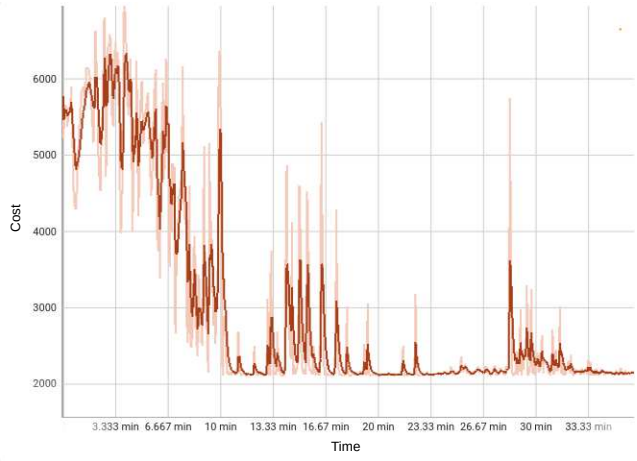
4.4 Summary

In this chapter, we presented the implementation details of our proposed solutions including the dataset, the software technologies as well as the hardware used to train our DRL algorithms. Moreover, we studied the convergence of both strategies under different hyper-parameters and compared their performance against other benchmark methods. Based on the obtained results, our PPO-based algorithm showed better performance compared to the DQN algorithm in terms of efficiency and stability. Furthermore, both DRL strategies outperform the benchmark methods in terms of the overall cost minimization.



(a) PPO strategy

(b) DQN strategy



(c) DDQN strategy

Figure 4.13: Training performance of the PPO, DQN and DDQN strategies

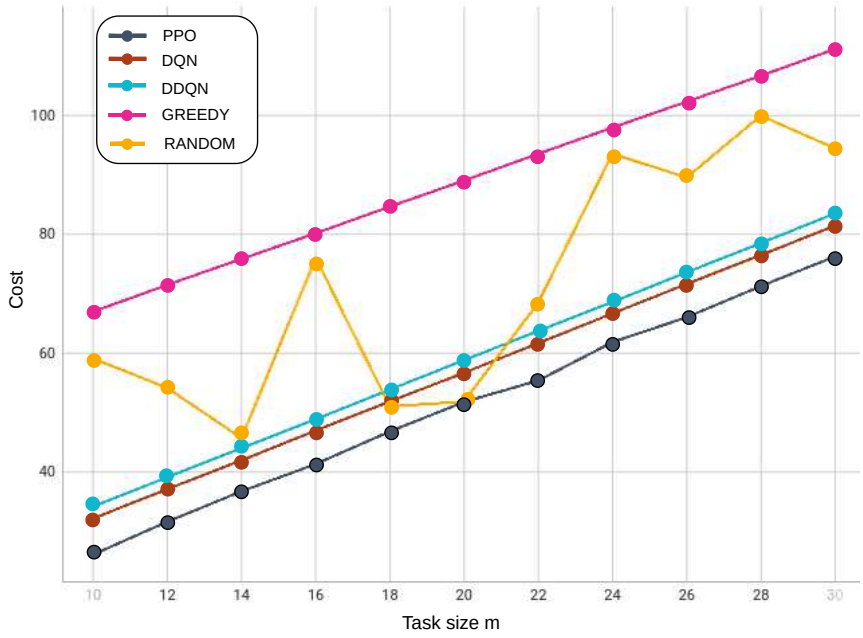


Figure 4.14: Performance comparison under different task sizes

Chapter 5

Conclusion and Future Work

MEC is a promising technology for next-generation 5G and 6G mobile networks to bring cloud-like capabilities to the edge of the network and closer to the customers. By offloading the computation-intensive tasks to the edge server, high demanding applications enjoy a reduced latency and near real-time performance. Defining the optimal computation offloading strategy is a key component towards better network performance. After going through the literature review and studying the promising results of DRL in problem optimization, we tried in this work to find an optimal DRL-agent for the computation offloading problem.

In this thesis, 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 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 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. For instance, the PPO strategy improved the average

cost by 12% compared to the DQN and DDQN strategies, as they both converge towards the same minimum cost, and by 37%, 73% compared to the RANDOM and GREEDY strategies respectively. Also, the average cost of the DQN strategy is improved by 21% compared to the RANDOM strategy, and by 52% compared to the GREEDY strategy.

The challenges we went through during this research are related to the modeling of the real-world MIMO-based MEC environment as well as the difficulties of applying DRL algorithms on it. Capturing the stochastic wireless channel variations of the MIMO-based MEC and the dynamically generated tasks at each mobile user into a mathematical model was challenging. In addition to the difficulties of defining a suitable MDP model of our problem, including the state space, action space and the reward function, applying DRL algorithms was also challenging because of the curse of the state space explosion and the continuous action space. Furthermore, training the neural networks during the simulations was not a straightforward task, because they are computationally intensive, need more time to converge and sometimes hard to tune due to the decent number of hyper-parameters. Despite the mentioned challenges, our solution to overcome those limitations is provided in details and the decisions we made are also explained.

In summary, we have been able to analyze and apply DRL algorithms to create smart software agents that can control the computation offloading in the stochastic multi-user MIMO-based MEC network. As future work, our research can be extended in many ways. One can study the offloaded tasks allocation on the MEC server taking into account more environment details such as its power and computation capacities. one can also explore the optimal computation offloading strategy to multiple base stations based on other characteristics such as their geo-location and band-with capability. Another potential way to extend this work is to investigate the case where multiple edge servers are connected to the same base station and how the offloaded tasks can be shared between them.

Bibliography

[1] Cedar - cc doc. `https://docs.computecanada.ca/wiki/Cedar`. (Accessed on 07/05/2021).

[2] 3.7 Value Functions, Dec 2018. [Online; accessed 13. Jan. 2022].

[3] Part 1: Key Concepts in RL — Spinning Up documentation, Mar 2020. [Online; accessed 18. Jan. 2022].

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

[5] A machine learning approach for task and resource allocation in mobile-edge computing-based networks. 8(3):1358–1372, feb 2021.

[6] What Is Model-Free Reinforcement Learning?, Dec 2021. [Online; accessed 20. Jan. 2022].

[7] PyTorch documentation — PyTorch 1.10.1 documentation, Jan 2022. [Online; accessed 29. Jan. 2022].

[8] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mane, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viegas, Oriol

Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. 2016.

[9] Nasir Abbas, Yan Zhang, Amir Taherkordi, and Tor Skeie. Mobile edge computing: A survey. *IEEE Internet of Things Journal*, 5(1):450–465, 2018.

[10] Laha Ale, Ning Zhang, Xiaojie Fang, Xianfu Chen, Shaohua Wu, and Longzhuang Li. Delay-aware and energy-efficient computation offloading in mobile edge computing using deep reinforcement learning. *IEEE Transactions on Cognitive Communications and Networking*, 2021.

[11] Taha Alfakih, Mohammad Mehedi Hassan, Abdu Gumaei, Claudio Savaglio, and Giancarlo Fortino. Task offloading and resource allocation for mobile edge computing by deep reinforcement learning based on SARSA. *IEEE Access*, 8:54074–54084, 2020.

[12] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. A Brief Survey of Deep Reinforcement Learning. aug 2017.

[13] Susan Baldwin. Compute Canada: Advancing computational research. *Journal of Physics: Conference Series*, 341(1), 2012.

[14] Jianyu Cao, Wei Feng, Ning Ge, and Jianhua Lu. Delay characterization of mobile-edge computing for 6G time-sensitive services. *IEEE Internet of Things Journal*, 8(5):3758–3773, 2021.

[15] Robin Chataut and Robert Akl. Massive MIMO systems for 5G and beyond networks—overview, recent trends, challenges, and future research direction. *Sensors (Switzerland)*, 20(10):1–35, 2020.

[16] Min Chen and Yixue Hao. Task offloading for mobile edge computing in software defined ultra-dense network. *IEEE Journal on Selected Areas in Communications*, 36(3):587–597, 2018.

[17] Xianfu Chen, Honggang Zhang, Celimuge Wu, Shiwen Mao, Yusheng Ji, and Medhi Bennis. Optimized computation offloading performance in virtual edge

computing systems via deep reinforcement learning. *IEEE Internet of Things Journal*, 6(3):4005–4018, 2019.

[18] Xianfu Chen, Honggang Zhang, Celimuge Wu, Shiwen Mao, Yusheng Ji, and Mehdi Bennis. Performance optimization in mobile-edge computing via deep reinforcement learning. *arXiv*, pages 1–6, 2018.

[19] Xu Chen, Lei Jiao, and Wenzhong Li. Efficient Multi-User Computation Offloading for. *IEEE/ACM Transactions on Networking*, 24(5):2795–2808, 2016.

[20] Eunsol Choi, Daniel Hewlett, Alexandre Lacoste, Illia Polosukhin, Jakob Uszkoreit, and Jonathan Berant. Coarse-to-Fine Question Answering for Long Documents Eunsol : Hierarchical Question Answering for Long Documents. 2016.

[21] Boutheina 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)*, pages 45–52, 2019.

[22] Sibren De Bast and Sofie Pollin. MaMIMO CSI-based positioning using CNNs: Peeking inside the black box. *2020 IEEE International Conference on Communications Workshops, ICC Workshops 2020 - Proceedings*, 2020.

[23] Shuiguang Deng, Hailiang Zhao, Weijia Fang, Jianwei Yin, Schahram Dustdar, and Albert Y. Zomaya. Edge intelligence: The confluence of edge computing and artificial intelligence. *IEEE Internet of Things Journal*, 7(8):7457–7469, 2020.

[24] Changfeng Ding, Jun bo Wang, Hua Zhang, Min Lin, and Jiangzhou Wang. Joint MU-MIMO precoding and resource allocation for mobile-edge computing. *IEEE Transactions on Wireless Communications*, 20:1639–1654, 2021.

[25] Thinh Quang Dinh, Quang Duy La, Tony Q.S. Quek, and Hyundong Shin. Learning for computation offloading in mobile edge computing. *IEEE Transactions on Communications*, 66(12):6353–6367, 2018.

[26] Vincent François-Lavet, Raphael Fonteneau, and Damien Ernst. How to discount deep reinforcement learning: Towards new dynamic strategies. pages 1–9, 2015.

[27] Vincent François-lavet, Peter Henderson, Riashat Islam, Marc G Bellemare, Vincent François-lavet, Joelle Pineau, and Marc 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*, II(3 - 4):1–140, 2018.

[28] Vincent Francois-Lavet, Peter Henderson, Riashat Islam, Marc G. Bellemare, and Joelle Pineau. An Introduction to Deep Reinforcement Learning. *arXiv*, Nov 2018.

[29] Yuanyuan Hao, Qiang Ni, Hai Li, and Shujuan Hou. Energy-efficient multi-user mobile-edge computation offloading in massive MIMO enabled HetNets. *IEEE International Conference on Communications*, 2019-May, 2019.

[30] Liang Huang, Xu Feng, Cheng Zhang, Liping Qian, and Yuan Wu. Deep reinforcement learning-based joint task offloading and bandwidth allocation for multi-user mobile edge computing. *Digital Communications and Networks*, 5(1):10–17, 2019.

[31] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[32] Mahbub E. Khoda, Md Abdur Razzaque, Ahmad Almogren, Mohammad Mehedi Hassan, Atif Alamri, and Abdulhameed Alelaiwi. Efficient computation offloading decision in mobile cloud computing over 5G network. *Mobile Networks and Applications*, 21(5):777–792, 2016.

[33] B. Ravi Kiran, Ibrahim Sobh, Victor Talpaert, Patrick Mannion, Ahmad A. Al Sallab, Senthil Yogamani, and Patrick Pérez. Deep Reinforcement Learning for Autonomous Driving: A Survey. *arXiv*, Feb 2020.

[34] Yann Lecun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[35] Sergey Levine, Peter Pastor, Alex Krizhevsky, and Deirdre Quillen. Learning Hand-Eye Coordination for Robotic Grasping with Deep Learning and Large-Scale Data Collection. *arXiv*, Mar 2016.

[36] Hao Meng, Daichong Chao, and Qianying Guo. Deep reinforcement learning based task offloading algorithm for mobile-edge computing systems. *ACM International Conference Proceeding Series*, pages 90–94, 2019.

[37] Farouk Messaoudi, Adlen Ksentini, and Philippe Bertin. On using edge computing for computation offloading in mobile network. *2017 IEEE Global Communications Conference, GLOBECOM 2017 - Proceedings*, 2018-January:1–7, 2017.

[38] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with deep reinforcement learning. pages 1–9, 2013.

[39] Elena Mocanu, Decebal Constantin Mocanu, Phuong H. Nguyen, Antonio Liotta, Michael E. Webber, Madeleine Gibescu, and J. G. Slootweg. On-Line Building Energy Optimization Using Deep Reinforcement Learning. *IEEE Trans. Smart Grid*, 10(4):3698–3708, May 2018.

[40] Rui Nian, Jinfeng Liu, and Biao Huang. A review On reinforcement learning: Introduction and applications in industrial process control. *Comput. Chem. Eng.*, 139:106886, Aug 2020.

[41] Adam Paszke, Sam Gross, James Bradbury, Zeming Lin, Zach Devito, Francisco Massa, Benoit Steiner, Trevor Killeen, and Edward Yang. PyTorch : An Imperative Style , High-Performance Deep Learning Library. (NeurIPS), 2019.

[42] Romain Paulus, Caiming Xiong, and Richard Socher. A Deep Reinforced Model for Abstractive Summarization. *arXiv*, May 2017.

[43] Hani Sami and Azzam Mourad. Dynamic on-demand fog formation offering on-the-fly IoT service deployment. *IEEE Transactions on Network and Service Management*, 2020.

[44] Hani Sami, Azzam Mourad, Hadi Otrok, and Jamal Bentahar. Demand-driven deep reinforcement learning for scalable fog and service placement. *IEEE Transactions on Services Computing*, 2021.

[45] Hani Sami, Hadi Otrok, Jamal Bentahar, and Azzam Mourad. AI-based resource provisioning of IoE services in 6G: A deep reinforcement learning approach. *IEEE Transactions on Network and Service Management*, 2021.

[46] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, Timothy Lillicrap, and David Silver. Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model. *arXiv*, Nov 2019.

[47] John Schulman, Philipp Moritz, Sergey Levine, Michael I. Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *4th International Conference on Learning Representations, ICLR 2016 - Conference Track Proceedings*, pages 1–14, 2016.

[48] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. pages 1–12, 2017.

[49] Nir Shlezinger, George C. Alexandropoulos, Mohammadreza F. Imani, Yonina C. Eldar, and David R. Smith. Dynamic metasurface antennas for 6G extreme massive MIMO communications. *IEEE Wireless Communications*, 28(2):106–113, 2021.

[50] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. *arXiv*, Dec 2017.

[51] Richard S. Sutton, Francis Bach, and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press Ltd, 2018.

[52] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double Q-Learning. *30th AAAI Conference on Artificial Intelligence, AAAI 2016*, pages 2094–2100, 2016.

[53] N I Ver. Deep Reinforcement Learning Variants of Multi-Agent Learning Algorithms Alvaro Ovalle Casta ˜. 2016.

[54] Jiadai Wang, Lei Zhao, Jiajia Liu, and N. Kato. Smart resource allocation for mobile edge computing: A deep reinforcement learning approach. *IEEE Transactions on Emerging Topics in Computing*, pages 1–1, 2019.

[55] Yanting Wang, Min Sheng, Xijun Wang, Liang Wang, and Jiandong Li. Mobile-edge computing: Partial computation offloading using dynamic voltage scaling. *IEEE Transactions on Communications*, 64(10):4268–4282, 2016.

[56] Jie Xu, Lixing Chen, and Shaolei Ren. Online learning for offloading and autoscaling in energy harvesting mobile edge computing. *arXiv*, 2017.

[57] Xin Xu, Lei Zuo, and Zhenhua Huang. Reinforcement learning algorithms with function approximation: Recent advances and applications. *Information Sciences*, 261:1–31, 2014.

[58] Chao Yu, Jiming Liu, and Shamim Nemati. Reinforcement Learning in Healthcare: A Survey. *arXiv*, Aug 2019.

[59] Ming Zeng, Wanming Hao, Octavia A. Dobre, Zhiguo Ding, and H. Vincent Poor. Massive MIMO-assisted mobile edge computing: Exciting possibilities for computation offloading. *IEEE Vehicular Technology Magazine*, 15(2):31–38, 2020.

[60] Ming Zeng, Ming Zeng, Wanming Hao, Octavia A. Dobre, and H. Vincent Poor. Delay minimization for massive MIMO assisted mobile edge computing. *IEEE Transactions on Vehicular Technology*, 69(6):6788–6792, 2020.

[61] Weiwen Zhang, Yonggang Wen, Kyle Guan, Dan Kilper, Haiyun Luo, and Dapeng Oliver Wu. Energy-optimal mobile cloud computing under stochastic wireless channel. *IEEE Transactions on Wireless Communications*, 12(9):4569–4581, 2013.

[62] Rui Zhao, Xinjie Wang, Junjuan Xia, and Liseng Fan. Deep reinforcement learning based mobile edge computing for intelligent Internet of things. *Physical Communication journal*, pages 1–18, 2020.