UNIVERSITY OF PADOVA

DEPARTMENT OF INFORMATION ENGINEERING

Master degree in ICT for Internet and Multimedia

Cost of Learning in Mobile Edge Computing

Supervisor
Professor Federico Mason

Co-supervisor
Professor Federico Chiariotti

Master Candidate
Maddalena Boscaro

Academic Year
2023 - 2024

Alla mia famiglia,
per l'incondizionato affetto e incoraggiamento.
A chi mi vuole bene,
per essermi sempre stato accanto.
E ai miei amici,
per aver reso questo viaggio indimenticabile.

Abstract

In recent years, Mobile Edge Computing (MEC) has emerged as a promising network architecture that offers considerable computing capabilities in the proximity of mobile devices. In this way, MEC responds to the growing needs of various real-time applications that require both significant computational capacity and low latency. However, the efficient allocation of resources in MEC environments still remains a critical challenge, especially due to the shared nature of resources, often contested among multiple users.

In this context, Reinforcement Learning (RL) algorithms have proven to be an effective solution to optimize resource management. However, existing studies tend to overlook the consumption costs associated with the training of such algorithms. Indeed, in a system in which users and the learning process share the same pool of resources, competition may arise in order to meet their respective needs. On the one hand, it is essential to respond to the users' demands, on the other hand, it is indispensable to continue to improve the RL strategy in order to secure a higher return in the long run. This trade-off is called *cost of learning*.

In this thesis, we will analyze the cost of learning by comparing different resource allocation strategies within a simulated MEC environment. Furthermore, we will propose several effective strategies to adjust the frequency of training, thus significantly reducing the training impact and ensuring high user performance. Our results will show how the cost of learning is fundamental and non-negligible, especially in dynamic environments where continual training becomes necessary.

Sommario

Negli ultimi anni, il Mobile Edge Computing (MEC) è emerso come un'architettura di rete promettente, in grado di offrire considerevoli capacità computazionali in prossimità dei dispositivi mobili. MEC risponde così alle crescenti esigenze di diverse applicazioni in tempo reale che richiedono al contempo elevate capacità di calcolo e una bassa latenza. Tuttavia, l'allocazione efficace delle risorse in ambienti MEC rimane ancora una sfida critica, soprattutto a causa della natura condivisa di tali risorse, spesso contese tra più utenti.

In questo contesto, gli algoritmi di Reinforcement Learning (RL) si sono dimostrati una soluzione efficace al fine di ottimizzare la gestione delle risorse. Tuttavia, gli studi esistenti tendono spesso a trascurare i costi derivanti dall'addestramento di tali algoritmi. Infatti, in un sistema in cui gli utenti e il processo di apprendimento condividono lo stesso pool di risorse, può nascere una competizione al fine di soddisfare le rispettive necessità. Da un lato, è essenziale rispondere prontamente alle richieste degli utenti, dall'altro, è indispensabile continuare a migliorare la strategia RL al fine di assicurarsi nel lungo periodo un guadagno più elevato. Questo trade-off prende il nome di *cost of learning*.

In questa tesi analizzeremo il cost of learning confrontando diverse strategie di allocazione delle risorse all'interno di un ambiente MEC simulato. Inoltre, proporremo diverse strategie efficaci per regolare la frequenza degli addestramenti, riducendo significativamente i costi e garantendo così elevate prestazioni all'utente. I nostri risultati mostreranno come il cost of learning sia una misura fondamentale e non trascurabile, soprattutto in contesti dinamici in cui l'addestramento continuo diventa necessario.

Contents

1 Introduction

In recent years, Mobile Edge Computing (MEC) has emerged as a promising computing paradigm that provides cloud-computing capabilities at the edge of the mobile network [1]. By deploying multiple MEC servers in base stations (BS) in close proximity to users, MEC enables ultra-low latency, high bandwidth, and access to significant computational resources [2].

MEC systems address the growing demands of modern real-time applications (e.g. connected vehicles, remote surgery, real time gaming, etc.), which require vast computational resources while maintaining stringent latency requirements. In fact, current mobile devices cannot satisfy most of these needs due to hardware limitations such as limited processing power and insufficient battery life. Traditionally, applications have relied on cloud data centers for processing, but with the expected increase in data traffic and the diffusion of more specific communication services in the coming years, such an approach may prove insufficient [3]. The main challenge lies in the long network distance between the end user and the cloud data centers, which may result in unacceptable long delays, increased network congestion and degraded Quality of Experience (QoE) for the end users [4].

MEC addresses these issues by bringing computation closer to the user, but it also introduces its own set of challenges. Since MEC computational capacity is lower if compared to cloud data centers, and with multiple users often located inside the same BS coverage area, resource allocation becomes even more critical. Furthermore, with multiple MEC servers available, users must choose the optimal BS for offloading their tasks. This decision is essential to minimize latency and prevent resource contention, as too many users relying on the same server can lead to performance bottlenecks and reduce system efficiency [5].

For this reason, the scientific community has become increasingly involved in the study and development of new approaches for managing computational resources in MEC scenarios [6]. Among these, Reinforcement Learning (RL) algorithms have been very effective in addressing resource allocation challenges. RL is a Machine Learning (ML) paradigm where an agent learns to take actions by interacting with an environment that provides feedback on the quality of those actions. Through this

process, RL algorithms can adaptively find optimal resource management strategies, even in complex and dynamic conditions. This makes them highly suitable for MEC systems, where fluctuating user demands and changing network conditions require continual adaptation.

While many studies have focused on designing RL-based solutions for resource management in MEC environments, many of these approaches tend to overlook the resource consumption that is associated with optimizing the RL algorithm. In fact, in an RL-based resource allocation system, we encounter a dual problem: on the one hand, the need to allocate the maximum available resources to meet user demands, and, on the other, the requirement to invest resources into training and refining the RL strategy itself. Balancing these two competing demands is critical, as the resources spent on optimizing the learning process can detract from those available for user services. In this paper, we investigate this trade-off, known as *cost of learning*.

Although it remains a relatively unexplored area, recent research has introduced examples of computation-aware, learning-based frameworks, specifically for network management in network slicing scenarios [7][8]. These approaches use a RL agent to allocate backhaul link resources efficiently, balancing user demands with resources reserved for learning. The studies focus on the cost of learning, examining how RL training affects user performance and assessing the extent to which a learning-based approach is beneficial for network management.

In this thesis, we extend this investigation by exploring the cost of learning in a resource management application deployed on a MEC server. We compare different strategies for managing learning resources and identify the conditions under which investing in learning leads to significant performance improvements in the MEC system, ensuring that the benefits overcome the costs associated to training.

Practically, we model a MEC environment, which we use to conduct several simulations aimed at investigating the performance of various proposed solutions. The results reveal several key insights. First, the cost of learning proves to be a critical metric in systems where users and the learning process share the same pool of resources. This is particularly true in dynamic environments, where continual training is essential for adapting to evolving system conditions. In such scenarios, the impact of the learning process on user performance cannot be ignored, as it significantly influences the overall resource allocation. Second, we demonstrate the effectiveness of intelligent strategies for determining the most opportune moments to train the RL algorithm, allowing for a significant reduction in the associated learning costs. These strategies enable the system to minimize resource consumption on training while still maintaining high performance, offering a more efficient balance between learning and user resource demands.

The subsequent chapters of this thesis are organized as follows. In Chapter 2, we provide an overview of the RL fundamentals, focusing on its main concepts and methods, suggesting its potential in solving a wide range of problems. Chapter 3 introduces the resource management problem in MEC environments. This chapter outlines the key challenges of managing resources in such systems, providing some examples of both heuristic and RL-based resource allocation strategies. Furthermore, it motivates the entire thesis highlighting a lack in the existing literature regarding the consideration of the cost of learning. In Chapter 4, we present the model formulation for our MEC scenario. This chapter begins with a detailed description of the MEC environment, followed by its RL formulation

and the training algorithm employed. Chapter 5 focuses on the concept of cost of learning and introduces three different strategies for managing the resources allocated to the learning process. The first one is an ideal approach in which the agent is allowed to learn without any resource consumption. The second strategy involves a periodic allocation of resources, where training occurs at regular intervals. The third strategy is more adaptive, as it dynamically identifies the most opportune moments to train the RL agent. In Chapter 6, we presents the results of our simulations in both stationary and dynamic environments, with a particular focus on how each learning strategy impacts user performance. Finally, Chapter 7 concludes the thesis by summarizing our findings and discussing the implications of the results, along with potential directions for future research.

2 Reinforcement Learning

In this chapter, we cover the essential concepts of Reinforcement Learning (RL) whose theory is at the basis of the methodology presented in the thesis. We begin by outlining the basics of RL, including the concept of Markov Decision Processes (MDP), value function and Bellman equations. Then, we present the two main dynamic programming methods, specifically policy iteration and value iteration, which are techniques that are able to derive the optimal policy when the environment's model is known. After that, we discuss model-free methods, focusing specifically on value-based approaches, which are commonly employed when the environment's model is unknown, emphasizing the challenges of balancing exploration and exploitation in learning processes. Finally, we introduce the concept of Deep Reinforcement Learning (DRL), where deep neural networks are integrated with RL to address more complex environments. The chapter provides a thorough introduction to both traditional and more advanced RL methods, setting the stage for using these techniques in the system modeled later in this thesis.

2.1 The basics of Reinforcement Learning

Reinforcement Learning (RL) is a Machine Learning (ML) paradigm that enables the modeling of any decision making problems, including the natural learning processes of humans. Just like people and animals, it involves exploring and interacting with the environment, receiving feedback, and using that feedback to improve future responses.

The main element of RL is the *learning agent*. The agent observes the current state of the environment and takes actions that affect the state, aiming to achieve a specific goal. The agent receives feedback from its actions through the corresponding *reward*, which represents the immediate benefit of the action. The learner is not told which actions to take, but instead must discover which actions yield the higher reward by trying them. For deciding how to interact with the environment, the agent follows a *policy*, i.e., a function mapping each state of the environment with a specific action or

a probability distribution over the action space. In a general scenario, actions may affect not only the immediate reward but also the next transitions and, consequently, all following rewards. Therefore, the agent does not solely need to learn how good an action is in an immediate sense, but also the benefits of its decisions overs the long term.

2.1.1 Markov Decision Processes (MDP)

The RL paradigm uses the formal framework of Markov decision processes (MDP), a mathematical framework that models the interaction between a learning agent and its environment in terms of states, actions, and rewards. A MDP is defined by the tuple $(\mathcal{S}, \mathcal{A}, P, R)$ where:

1. $\mathcal{S}$ is the state space;
2. $\mathcal{A}$ is the action space;
3. $P : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ is the state transition probability function;
4. $R : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is the reward function.

By assumption, the state transitions and rewards are stochastic and follow the so called Markov propriety: i.e. the state transition probabilities and rewards depend only on the immediately preceding state s_{t-1} and action a_{t-1} taken by the agent [9].

To learn how to solve a specific task, the agent must interact with the environment multiple sequence of discrete time steps. Tasks are generally categorized into two types: episodic tasks and continuing tasks. Episodic tasks are divided into episodes, each episode being a sequence of states, actions, and rewards that starts in an initial state and ends in a terminal state. In continuing tasks, there is no natural termination and the interaction goes on indefinitely. At each time slot $t \in \mathbb{Z}^+$ the agent interacts with the environment by observing the following steps:

1. The agent observes the current state s_t of the environment.
2. The agent selects an action a_t according to a policy π, which is a distribution over each possible action available at a certain state. More specifically, π is defined as $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$, so that $\pi(s_t, a_t)$ represents the probability of taking action a_t in state s_t.
3. Subsequently, at time $t+1$, the environment shift to state s_{t+1} according to the function $P(s_t, a_t, s_{t+1})$, which gives the probability of moving from state s_t to state s_{t+1} due to action a_t, and the agent receives the reward $r_{t+1} = R(s_t, a_t, s_{t+1})$ that depends on the state transition.
4. The agent updates its policy π according with the observed state transition and the obtained reward.
5. This loop continues until a termination condition is met, such as reaching a terminating state or the maximum number of iterations.

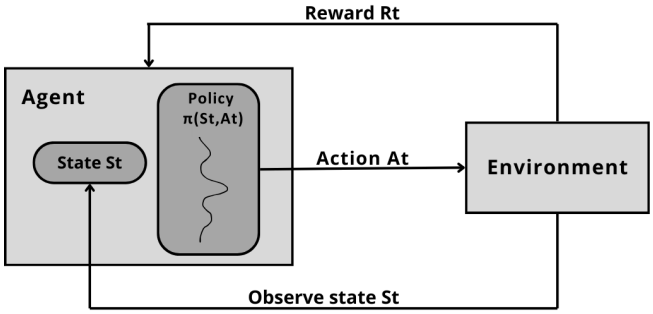


Figure 2.1: Markov Decision Process diagram.

The final goal of the agent is to maximize the total amount of rewards over time, which is expressed by the so-called *discounted return* $G(t)$:

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}, \tag{2.1}$$

where $\gamma \in [0, 1)$ is the so called discount factor, which controls the importance of future rewards relative to present ones.

One important property of the discounted returns, which is fundamental for the entire RL theory and algorithms, is the relationship between returns at successive time steps:

$$\begin{aligned} G_t &= \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \\ &= r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 r_{t+4} + ... = \\ &= r_{t+1} + \gamma(r_{t+2} + \gamma r_{t+3} + \gamma^2 r_{t+4} + ...) = \\ &= r_{t+1} + \gamma G_{t+1}. \end{aligned} \tag{2.2}$$

2.1.2 Value Functions and Bellman Equations

Given the agent policy π, we define the *state-value function* of a state s under a policy π, denoted $v_\pi(s)$, as the expected return G_t when starting in state $s_t = s$ and following π thereafter. It measures how advantageous is for the agent to be in any state. We can define $v_\pi(s)$ as:

$$v_\pi(s) = \mathbb{E}_\pi \left[G_t \middle| s_t = s \right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| s_t = s \right]. \tag{2.3}$$

The symbol $\mathbb{E}_\pi$ denotes the expectation of a random variable given that agent follows policy π. Similarly, we define the so called *q-value function* $q_\pi(s, a)$ which indicates how profitable is to take a certain action in a given state. Formally, it represents the expected return for taking action a in state s and

then following policy π thereafter.

$$q_\pi(s,a) = \mathbb{E}_\pi\left[G_t \middle| s_t = s, a_t = a\right] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| s_t = s, a_t = a\right]. \tag{2.4}$$

It is important to note that the value of the terminal state, if any, is always zero for both state-value and q-value functions.

Both the state-value and action-value functions can be estimated from experience. For instance, to estimate $v_\pi(s)$, the agent can record the average of all rewards obtained each time it visits state s by following policy π. As the number of visits increases, this average converges to the actual state value. The same procedure can be applied to $q_\pi(s,a)$ by maintaining separate averages for each action taken in each state. These approaches are named Monte Carlo methods since it involves averaging over many random samples of actual returns. We will see these methods more in details in the following sections.

Similar to what we observed for cumulative returns, both state-value and action-value functions satisfy recursive relationships. For any policy π and any state s, the following condition holds:

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi\left[G_t \middle| s_t = s\right] = \mathbb{E}_\pi\left[r_{t+1} + \gamma G_{t+1} \middle| s_t = s\right] \\ &= \sum_a \pi(a \mid s) \sum_{s'} P(s,a,s') \left(R(s,a,s') + \gamma \mathbb{E}_\pi\left[G_{t+1} \middle| s_{t+1} = s'\right]\right) \\ &= \sum_a \pi(a \mid s) \sum_{s'} P(s,a,s') \left(R(s,a,s') + \gamma v_\pi(s')\right), \end{aligned} \tag{2.5}$$

which expresses a relationship between the value of a state $v_\pi(s)$ and the values of its successor states $v_\pi(s')$. We can find a similar relation for the action-value function q_π:

$$q_\pi(s,a) = \sum_{s'} P(s,a,s') \sum_{a'} \pi(a' \mid s') \left(R(s,a,s') + \gamma q_\pi(s',a')\right). \tag{2.6}$$

The last two conditions (2.5) and (2.6) are known as the *Bellman Equations* and represent one of the essential tools for developing and deriving all RL fundamentals.

In RL, learning a task involves learning the *optimal policy* π_*, which is the policy that achieves the highest return. For finite MDP, it is always possible to find at least one optimal policy. We said that a policy π is better than or equal to a policy π' if its expected return is greater than or equal to that of π' for all states. In other words, $\pi \geq \pi'$ if and only if $v_\pi(s) \geq v_{\pi'}(s)$ for all $s \in \mathcal{S}$. Hence, π_* maximizes the state-value function and action-value function for all the states and actions. Hence, we can define the *optimal state-value function* v_*, and similarly, also the *optimal action-value function* q_* as:

$$v_*(s) = \max_\pi v_\pi(s) \quad \text{for all } s \in \mathcal{S}, \tag{2.7}$$

$$q_*(s,a) = \max_\pi q_\pi(s,a) \quad \text{for all } s \in \mathcal{S} \text{ and } a \in \mathcal{A}. \tag{2.8}$$

As done above for the Bellman equations, it is possible to derive the so-called *Bellman optimality equa-*

tions, which express the recursive relation between current optimal state value and successive optimal state values.

$$v_*(s) = \max_a \sum_{s'} P(s,a,s') \left[R(s,a,s') + \gamma v_*(s') \right], \tag{2.9}$$

$$q_*(s) = \sum_{s'} P(s,a,s') \left[R(s,a,s') + \gamma \max_{a'} q_*(s',a') \right]. \tag{2.10}$$

We can observe that, unlike in the previous conditions, equations (2.9) and (2.10) do not depend on the current policy explicitly. However, due to the max operator in the expressions, finding a closed-form solution is no possible and it is necessary to exploit iterative methods. The first and simplest of these is dynamic programming, which will be discussed in the next section. After a solution to the Bellman equations is obtained, determining an optimal policy becomes straightforward. For each state s, the agent can simply choose any action that maximize $q_*(s,a)$.

2.2 Dynamic Programming

Dynamic Programming (DP) consists of a collection of algorithms that derive the optimal policy solving the Bellman equations or similar problems by a recursive approach. DP is considered fundamental in the realm of RL for its theoretical implications. However, it is not practical due to two significant constraints. Firstly, DP algorithms need a perfect model of the environment, an assumption that is unrealistic for most RL problems. This is why they also fall under the category of so-called *model-based* methods. Secondly, these approaches are computationally expensive, making them impractical for many real-world applications.

For the sake of simplicity, let us assume starting from this chapter that the environment is a finite MDP: both sets $\mathcal{S}$ and $\mathcal{A}$ contain a finite number of elements. We also assume knowledge of the set of transition probabilities $P(s,a,s')$ for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$, $s' \in \mathcal{S}$. DP algorithms utilize Bellman equations, transforming them into updating rules that can progressively improve the approximation of the desired value functions. They consists of two distinct phases:

- **Policy Evaluation (Prediction)** is the process of calculating the value function $v_\pi(s)$ for a given policy π. The goal is to determine how good a policy is by computing the expected return for each state when following that policy.
- **Policy Improvement (Control)** is the process of improving a given policy π by making it greedy with respect to the current value function $v_\pi(s)$. The goal is to create a new policy π' that is better than or equal to the current policy π.

Depending on how these two phases are combined, two different algorithms can be distinguished: *Policy Iteration* and *Value Iteration*.

2.2.1 Policy Iteration

Policy iteration algorithm alternates between prediction and control until policy π converges to optimal policy π_*. During the policy evaluation phase, successive approximations of the value function

are computed for each state s. The old (and inaccurate) value of $v(s)$ is replaced with a new value obtained from the old values of the successor states of s, and the immediate rewards r, averaging over all possible transitions probabilities under policy π. Iterating this procedure infinitely guarantees reaching the exact value of $v(s)$ given the current policy. However, in practice we stop earlier. As shown in the pseudocode below, once the estimate of v is sufficiently accurate, i.e. when the accuracy of the estimation reaches parameter θ, we proceed to the policy improvement phase.

In the policy improvement phase, we seek to determine if there are any better actions to take for a certain state s. We find a greedy policy that selects the action achieving the highest expected return in the short term: after one step into the future. This process exploits the Bellman optimality equation, ensuring that policy improvement returns a strictly better policy unless the original policy is already optimal. This procedure is iterated for all states. If the policy remains stable at the end of a policy improvement process, i.e., none of the agent decision changes with respect to the previous step, the obtained policy is proven to be optimal. If the policy is not stable, the process repeats from the policy evaluation phase until convergence. The complete algorithm is given below.

Algorithm 2.1 Policy Iteration

Initialization: Policy π and value function $V(s)$ arbitrarily for all $s \in \mathcal{S}$, $V(\text{terminal}) \leftarrow 0$.

1. Policy Evaluation:
repeat
  $\Delta \leftarrow 0$
  for all $s \in \mathcal{S}$
    $v \leftarrow V(s)$
    $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s'} P(s,a,s')[R(s,a,s') + \gamma V(s')]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
  end for
until $\Delta < \theta$ ▷ Stopping criterion (small threshold θ)

2. Policy Improvement:
`policy_stable` $\leftarrow$ `true`
for all $s \in \mathcal{S}$
  `old_action` $\leftarrow \pi(s)$
  $\pi(s) \leftarrow \arg\max_a \sum_{s'} P(s,a,s')[R(s,a,s') + \gamma V(s')]$
  if `old_action` $\neq \pi(s)$
    `policy_stable` $\leftarrow$ `false`
  end if
end for
if `policy_stable` return $V \approx v_*$ and $\pi \approx \pi_*$ else return to step 2.
end if

2.2.2 Value Iteration

The strength of the policy iteration method lies in its convergence guarantees. However, the policy evaluation phase can be extremely time-consuming as it requires iterating through the entire state space multiple times. The value iteration algorithm addresses this issue by truncating the policy evaluation after just one update of the states, while still maintaining the convergence guarantees. In general, the update rule in Value Iteration is similar to the one used in the prediction phase of the Policy Iteration algorithm. However, the key difference is that in Value Iteration, we act greedily by choosing the maximum possible value of $v(s)$ for each state s. The process also terminates similarly: since theoretically the convergence to v_* can be achieved in an infinite number of iterations, the loop is stopped once the desired accuracy θ is reached. This process retains the same convergence properties as before while achieving convergence in significantly less time. The complete algorithm is given below.

Algorithm 2.2 Value Iteration

Initialize: Value function $V(s)$ arbitrarily for all $s \in \mathcal{S}$, $V(\text{terminal}) \leftarrow 0$.
repeat
  $\Delta \leftarrow 0$
  for all $s \in \mathcal{S}$
    $v \leftarrow V(s)$
    $V(s) \leftarrow \max_a \sum_{s'} P(s, a, s')[R(s, a, s') + \gamma V(s')]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
  end for
until $\Delta < \theta$ ▷ Stopping criterion (small threshold θ)

Output a deterministic policy $\pi \approx \pi_*$ such that:
$\pi(s) = \arg\max_a \sum_{s'} P(s, a, s')[R(s, a, s') + \gamma V(s')]$

2.3 Model-free methods

So far, we have explained how to model RL problems using MDPs. However, some problems can be formulated in an RL framework, but defining an MDP in environments with unpredictable behavior may prove impossible. The agent has no prior knowledge about the state transition probability function P and the reward function R, which are unknown. In these cases, using dynamic programming or model-based methods is not possible, and we must rely on *model-free* methods. In model-free methods, an explicit definition of the MPD is not necessary and the agent only requires *experience*: gathering real-world data from the environment by sampling sequences of states, actions, and rewards. Model-free algorithms are typically subdivided into three categories based on how they learn and represent the optimal policy and value functions:

- **Value-based methods** are approaches that focus on estimating and improving the value functions associated with states, or state-action pairs. These methods use value functions to derive the optimal policy, aiming to maximize cumulative rewards over time.

- **Policy-based methods** directly parameterize and optimize the policy without the need for value functions. These methods are particularly useful for high-dimensional or continuous action spaces.
- **Actor-critic methods** are hybrid methods that combine aspects of both approaches.

2.3.1 Value-based Methods

In this section, we will explore two of the best-known value-based methods: Monte Carlo (MC) and Temporal Difference (TD) Learning.

Monte Carlo Methods

Monte Carlo methods solve the RL problem by sampling and averaging complete returns for each state. We will see that concepts already seen in DP can be extended to the MC case, even though the agent learns exclusively through experience. We will consider episodic tasks in order to estimate $v_\pi(s)$, the value function of state s under policy π. As in DP algorithms, Monte Carlo involves multiple iterations of a policy evaluation phase, where $v_\pi(s)$ is computed for a fixed arbitrary policy π, followed by a policy improvement phase, where a new greedy policy is derived from the current value function. There are two main types of MC methods: first-visit MC, which averages the returns of the first visit to a state within each episode, and every-visit MC, which averages the returns of all visits to a state within each episode. In this discussion we will refer to the first visit case.

Algorithm 2.3 First-Visit Monte Carlo for estimating $\pi \approx \pi_*$

Initialize:
$\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$
$Q(s,a) \in \mathrm{R}$ arbitrarily for all $s \in \mathcal{S}$, for all $a \in \mathcal{A}$, $Q(\text{terminal}, .) \leftarrow 0$
an empty list `Returns`(s,a) for all $s \in \mathcal{S}$, for all $a \in \mathcal{A}$
for each episode
  Generate an episode following policy π: $(s_1, a_1, r_2, s_2, a_2, r_3, \ldots, s_T)$
  Initialize $G \leftarrow 0$
  for $t = T-1, T-2, \ldots, 0$
    $G \leftarrow \gamma G + r_{t+1}$
    if the pair s_t, a_t appears for the first time in the episode
      Append G to `Returns`(s_t, a_t)
      $Q(s_t, a_t) \leftarrow$ average(`Returns`(s_t, a_t)) ▷ Policy Evaluation
      $\pi(s_t) \leftarrow \mathrm{argmax}_a\, Q(s_t, a)$ ▷ Policy Improvement
    end if
  end for
end for
return Q and π

As in the DP case, the resulting state-value function will approximate the true value, which can only be reached asymptotically. However, unlike DP algorithms, model-free methods estimate the action-value function $Q(s,a)$ rather than $V(s)$, which comes with higher computational costs. In the

control phase the agent policy is improved directly from the action-value function Q, selecting the action with the highest action-value for each state.

As shown in the pseudocode above, the estimate of Q is given by averaging the returns:

$$Q(s_t, a_t) \leftarrow \text{average(Returns}(s_t, a_t)). \tag{2.11}$$

However, the majority of RL algorithms employ *incremental updates* to the action-value function (and value function), typically expressed as follows:

$$\texttt{New Estimate} \leftarrow \texttt{Old Estimate} + \alpha[\texttt{Target} - \texttt{Old Estimate}], \tag{2.12}$$

where the step-size, commonly named $\alpha \in (0, 1]$, is a parameter that controls the weight given to the new estimate of the value function, commonly referred to as the *target*. Incremental updates are particularly effective in non-stationary (or dynamic) scenarios where the reward distribution changes over time. This approach allows the algorithm to prioritize, in the computation of the estimate, recent data over older and potentially inaccurate data. In the MC case the target is given by G_t and the MC update can be expressed as follows:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[G_t - Q(s_t, a_t)]. \tag{2.13}$$

From this expression, we can identify a significant limitation of MC methods: the agent must wait until the end of the episode to determine the increment to $Q(s_t, a_t)$, as the expected return G_t is only known at that time. This delay means that MC methods do not immediately incorporate the most recent information about the environment, potentially leading to slower convergence. Additionally, MC is designed to work only on episodic tasks, which may not be suitable for continuous or non-episodic environments.

Despite these limitations, MC remains a valid and widely utilized approach due to its strong convergence properties and the ease of understanding and implementation.

TD Learning

Temporal Difference (TD) Learning can be seen as a combination of the MC and DP approaches. Like MC, they learn directly from data sample recorded through experience. However, similar to DP, TD learning updates value function estimates based on other estimates, a technique known as *bootstrapping*.

We will first focus on the prediction phase, which consists in estimating the value function. As anticipated, MC methods must wait until the end of the episode to determine the increment to $V(s_t)$, as soon as the expected return G_t is known. TD methods, instead, need to wait in general for n steps, hence they are called n-step TD methods. Consider now the simplest case of *one-step* TD, or TD(0). In this case, we only need to wait until the next time step:

$$V(s_t) \leftarrow V(s_t) + \alpha[R_{t+1} + \gamma V(s_{t+1}) - V(s_t)]. \tag{2.14}$$

In this way, at each time step, TD method compute a new target, by using the reward R_{t+1} and the estimate $V(s_{t+1})$.

TD updates resemble those of DP but with a crucial difference: DP updates are calculated based on a complete distribution of all possible successive states, while TD updates are based on a single successive state. For this reason, TD target is a *biased* estimate of $v_\pi(s)$: since it depends on one random action, transition and reward. On the other hand, the MC target G_t depends on many samples making it an *unbiased* estimate of $v_\pi(s)$. However, even if bootstrapping can introduce bias, TD(0) has been proved to converge to v_π for any fixed policy π, provided that the step size α satisfies the following conditions [9]:

$$\sum_{t=1}^{\infty} \alpha_t = \infty \ , \qquad \sum_{t=1}^{\infty} \alpha_t^2 < \infty. \tag{2.15}$$

Compared to MC methods, TD learning generally achieves faster convergence by using incremental updates at each step, without needing to wait for the end of an episode. Another advantage of TD learning over MC is the lower variance in updates. MC targets, G_t, average all rewards collected during an episode, incorporating the randomness of the entire sequence of states and rewards, which can lead to noisy updates, especially in long episodes. In contrast, TD learning uses bootstrapping, which considers only the immediate reward and the value of the next state. Indeed, $V(s_{t+1})$ is an averaged estimate itself that is continuously refined through multiple experiences, reducing the impact of any single noisy reward or state transition. Furthermore, TD methods do not require episodes to terminate, making them suitable even for continuous tasks.

Let us now proceed to the use of TD prediction methods for the control problem. We will see two different approaches: State-Action-Reward-State-Action (SARSA) and Q-learning. SARSA method derives from what described above in the prediction part. We need to estimate $q_\pi(s, a)$ replicating what was done for the value function $v_\pi(s)$ in equation (2.14). From here, the process always follows the same pattern: we continually estimate q_π following current policy π, while simultaneously updating π to become increasingly greedy with respect to q_π.

Algorithm 2.4 SARSA algorithm for estimating $Q \approx q_*$

Initialize: $Q(s, a) \in \mathbb{R}$ arbitrarily for all $s \in \mathcal{S}$, for all $a \in \mathcal{A}$, $Q(\text{terminal}, \cdot) \leftarrow 0$
for each episode
  Initialize s
  Choose action a from s using policy π
  repeat
    Take action a, observe reward r and next state s′
    Choose a′ from s′ using policy π
    $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)]$
    $s \leftarrow s'$
    $a \leftarrow a'$
  until s is terminal
end for
return Q and π

So far we have always encountered so-called *on-policy* methods. On-policy methods aim to evaluate or improve the policy that is actively being used to make decisions. In contrast, *off-policy* methods focus on evaluating or improving a policy that is different from the one used to generate the data. The latter is the case of Q-learning, which, contrary to SARSA, doesn't update using the estimate of successive Q-values provided by the current policy π. Instead, as shown below, it updates the Q-values by taking the maximum possible value among all following state-action pairs, regardless of the current policy:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[R_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t))]. \tag{2.16}$$

We said that it doesn't follow current policy π since action a' in equation (2.16) is used only in the computation of $Q(s_t, a_t)$ but it is not, or at least not necessarily, the action undertaken in the next time step $t+1$. In other words, this means that a' does not necessarily correspond to a_{t+1}, which is the action actually chosen following policy π. This approach enables the agent to directly learn the optimal policy π_* by maximizing updates through a greedy action selection strategy, which progressively aligns Q-values with the optimal action-value function q_*.

Algorithm 2.5 Q-Learning Algorithm for estimating $\pi \approx \pi_*$

Initialize: $Q(s, a) \in \mathbb{R}$ arbitrarily for all $s \in \mathcal{S}$, for all $a \in \mathcal{A}$, $Q(\text{terminal}, \cdot) \leftarrow 0$
Parameters: step-size α, discount factor γ, small $\epsilon > 0$
for each episode
  Initialize state s
  repeat
    Choose action a using policy π
    Take action a, observe reward r and next state s'
    $Q(s, a) \leftarrow Q(s, a) + \alpha \left[R + \gamma \max_{a'} Q(s, a') - Q(s, a)\right]$
    $s \leftarrow s'$
    $a \leftarrow a'$
  until s is terminal
end for
return Q and π

Both algorithms, SARSA and Q-learning, are proven to converges if and only if each state-action pair is visited an infinite number of times. Although, in practice, their efficiency improves with the frequency of visits to each state-action pair, ensuring that errors remain within an acceptable range. Consequently, in both cases it is important to explore many different states and actions in order to learn the optimal strategy.

SARSA, as an on-policy algorithm, learns the value of the policy it is currently following. Because of this, it converges to a policy that is inherently stochastic, meaning it maintains a level of exploration even after convergence. Consequently, the policy obtained through SARSA may be sub-optimal compared to the policy derived from Q-learning, which tends to be more deterministic. However, this characteristic makes SARSA more conservative and reliable in environments with noisy or uncertain rewards, albeit at the expense of slower convergence.

On the other hand, Q-learning greedy updates contribute to its potential for faster convergence to the optimal policy. Nevertheless, this approach can lead to an overestimation bias in the Q-values. This bias arises from the inherent properties of the algorithm's update rule, which can lead to overoptimistic value estimates. Especially in environments with high variance in rewards, the algorithm might consistently overestimates the true value of certain actions, potentially leading to suboptimal decisions.

2.3.2 The Exploration-Exploitation Dilemma

One of the challenges that arise in model-free approaches, and essentially every RL task, is managing the trade-off between *exploration* and *exploitation*. Exploration involves selecting various actions to visit different states, allowing the agent to gather new information about the environment. In contrast, exploitation focuses on leveraging the knowledge already acquired by choosing actions that have previously resulted in higher rewards.

As explained, the agent's main objective is to maximize expected returns. To achieve this, it must prioritize actions that have previously yielded higher rewards by exploiting past experiences. However, to continually discover better actions, the agent must keep exploring by trying alternatives that have not yet been selected. Nevertheless, this exploration carries the risk of choosing sub-optimal actions and takes time away from exploitation.

The dilemma lies in the fact that both options can lead to sub-optimal outcomes for different reasons that cannot be known in advanced. The key to solving this dilemma is to simultaneously pursue both exploration and exploitation, trying to balance the two strategies as effectively as possible. The agent must first trial a large number of different actions and then gradually begin to choose those that have proven to be best.

A method to balance the trade-off between exploration and exploitation is the ϵ-*greedy* algorithm, where ϵ is a parameter that controls the rate of exploration:

- with probability ϵ explore: choose a random action from the possible ones.
- with probability $1 - \epsilon$ exploit: choose the action corresponding to the higher Q-values.

Another widely used exploration strategy is the *Softmax* method which select actions probabilistically based on their estimated Q-values. Given a set of action $a_1, a_2, ..., a_n$ for a state s and the corresponding set of action values $Q(s, a_1), Q(s, a_2), ..., Q(s, a_n)$, the Softmax function calculates the probability $P(a_i)$ of selecting action a_i in state s:

$$P(a_i) = \frac{e^{Q(s,a_i)/\tau}}{\sum_{j=1}^{n} e^{Q(s,a_j)/\tau}} \ , \tag{2.17}$$

where τ is a temperature parameter that controls the level of exploration versus exploitation. A higher value of τ leads to greater exploration: as τ increases, the probability distribution over actions becomes more uniform, making the agent more likely to select less favored actions, as the probabilities are closer to each other.

2.4 Deep Reinforcement Learning

Deep Reinforcement Learning (DRL) represents a new field of Reinforcement Learning combing the strengths of deep learning with RL problem. This successful combination enabled the development of agents capable of tackling complex, high-dimensional tasks that were previously infeasible for traditional methods. Thus, DRL opens up many new applications in domains such as healthcare, robotics, communication, finance, and many more.

Similarly to tabular RL methods, we progressively update the action-value function using a target. However, in high-dimensional spaces, storing Q-values in tables becomes impractical. Neural networks (NNs) can handle high-dimensional inputs and automatically extract relevant features, which is crucial in environments with image-based inputs or sensory data. In this context, NNs serve as approximations of value and action-value functions, $v_\pi(s,;\theta)$ and $q_\pi(s,a;\theta)$ or policy $\pi(a \mid s;\theta)$. Here, the parameters θ represents the weights in deep neural networks.

2.4.1 Deep Q Network

Deep Q Network (DQN) [10] is DRL value-based method and follows the Q-learning approach. It uses two NNs to approximate the action-value function. Specifically, DQN utilizes:

- **Policy Network Q:** is used to select actions based on the current state. This is typically done using an ϵ-greedy policy.
- **Target Network $\hat{Q}$:** provides stable targets for the Q-value updates during training. It is a copy of the policy network but with weights that are updated less frequently. This procedure decreases the instability that can occur due to the rapid Q-values changes.

Another important tool in DQN is *experience replay*, a technique that involves storing past transitions in a memory buffer and then sampling random batches of experience during training. This approach helps to break the correlation between consecutive updates, leading to more stable learning.

Updating the action-value function involves training the policy network by minimizing the difference between the predicted Q-values $Q(s_i, a_i, \theta)$ and the target Q-values (or Q-target), renominated y_i in the pseudo-code. The Q-target is computed following Q-learning approach by taking the maximum Q-values, computed using the target network $\hat{Q}$, for a better stability.

Algorithm 2.6 Deep Q-Network (DQN) algorithm

Initialize:
Experience replay memory $\mathcal{D}$ to capacity N
Policy Network Q with random weights θ
Target Network $\hat{Q}$ with weights $\hat{\theta} \leftarrow \theta$
while not converged
  Initialize s
  Choose action a from s using policy ϵ-greedy(Q)
  Take action a, observe reward r and next state s'
  Store transition (s, a, r, s') in replay buffer
  if enough experience in $\mathcal{D}$:
    Sample a random mini-batch of N transitions from $\mathcal{D}$
    for every transition (s_i, a_i, r_i, s'_i) in mini-batch

$$y_i = \begin{cases} r_i & \text{if } s'_i \text{ is terminal} \\ r_i + \gamma \max_{a'} \hat{Q}(s'_i, a'; \hat{\theta}) & \text{otherwise} \end{cases}$$

    end for
    Calculate the loss $\mathcal{L} = 1/N \sum_{i=1}^{N} (y_i - Q(s_i, a_i, \theta))^2 = \sum_{i=1}^{N} (\delta))^2$
    Update Q using SGD algorithm by minimizing the loss $\mathcal{L}$ (update θ)
    Every C steps copy weights from Q to $\hat{Q}$ ($\hat{\theta} \leftarrow \theta$)
  end if
end while

The loss is computed as the Mean Squared Error (MSE) of the target Q-values y and the predicted Q-values averaging over each sample. The difference between the target Q-values and the predicted Q-values is also known as *TD error* δ. The TD error measures the gap between what the agent thought would happen (the predicted value) and what actually happened (the target value) giving insights of how good the agent is learning. The policy network is then updated using gradient descent methods, such as Stochastic Gradient Descent (SGD), in order to minimize the loss, and consequently, the TD error. Finally, every few steps, we updates the target network weights $\hat{\theta}$, copying the weights from the policy network.

3
Resource Management Problem

In this chapter, we present the problem of resource management in a Mobile Edge Computing (MEC) scenario, where the efficient allocation of computing resources is essential for meeting the diverse and dynamic demands of users. We begin by outlining a general MEC system, where it is necessary to offload computational tasks from end-user devices to edge servers. Such an offloading process represents a key tool to reduces latency and enhances real-time processing capabilities. In doing so, we model the system's inherent challenges, such as resource constraints, varying workloads, and the complexity of managing resources in a rapidly changing environment. Following this, we explore some examples of resource allocation techniques, from traditional heuristic methods to more dynamic RL-based approaches. We analyze the strengths and limitations of each approach, both in terms of immediate performance and adaptability to new dynamics characterizing MEC scenarios.

3.1 The Resource Management Problem in MEC

In the light of the growing development of data science, many data-intensive applications are increasingly spreading, such as connected vehicles, virtual/augmented reality and real-time gaming. However, these applications require huge amounts of computational resources and strict latency requirements, which current mobile devices cannot satisfy due to their hardware limitations. Not even running all applications on a central cloud will solve the weaknesses of mobile devices, which will easily lead to severe network congestion with a consequent drop in performance. Indeed, due to the long network distance between the end user and the cloud data center, involving switching, routing, and congestion, a higher end-to-end delay is expected. In this scenario, MEC has emerged as an extension of cloud computing that deploys the edge cloud server on the side close to mobile devices [11]. In this way, the MEC architecture can not only alleviate the high latency problem of traditional cloud computing, but also solve the limited computing power of mobile devices.[12]

A simplified representation of the MEC scenario is shown in Figure 3.1, consisting of cloud data

centers, edge servers, and mobile devices. Cloud data centers offer powerful computing resources, such as CPUs and GPUs, which support data-intensive applications for end users at the cost of higher latency. MEC facilities consists of multiple edge servers located in Small Base Stations (SBSs) and Macro Base Stations (MBSs). SBSs include various wireless technologies, such as eNodeB and WiFi Access Points (APs). Despite their proximity to mobile devices, which favours ultra-reliable connectivity and ultra-low latency, SBSs cannot support more complex applications on their own due to their limited computational capacity. However, MBSs are able to do so, by exploiting city-level data centers with significant computational resources [13]. In addition, they are commonly connected to SBSs via high-speed optical fiber, enabling reliable and low-latency communications.

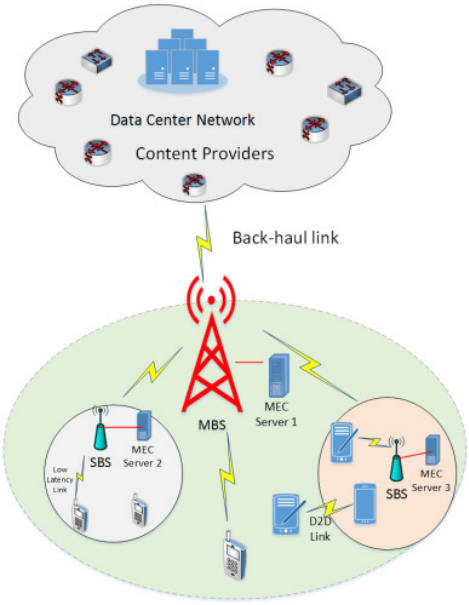


Figure 3.1: A simplified MEC architecture [12].

In case of multiple edge servers, the mobile device determines the best edge server to which it can send its service request. Nevertheless, when more and more mobile devices choose the same server, resource competition intensifies, leading to a decline in system performance [14][15]. For this reason, the scientific community has put increasing effort into the studying and development of new approaches for orchestrating computational and storage resources under the specific conditions that characterize MEC scenarios.

3.2 Heuristic Resource Management Methods

In the MEC scenario, it is crucial to efficiently allocate the resources required by the numerous tasks arriving at the edge server with the goal of optimizing performance, reducing latency and ensuring that the system can effectively handle the workload. Particularly, the action of assigning resources to perform tasks, commonly called *jobs*, is known as *job scheduling*. In a MEC server, the resources required by the jobs consist of computational resources such as CPUs and GPUs, while the jobs represent the processes generated by end users requests. In the following, we present some of the most well-known traditional algorithms for scheduling jobs in MEC servers or similar environments. These algorithms have been widely used in various computational environments and form the foundation for more advanced resource management techniques.

First In, First Out (FIFO)

First In, First Out (FIFO) is the simplest example of a scheduling algorithm. It schedules jobs in the exact order they arrive in the MEC server, without considering their priority, size, or resource requirements. The first job that enters the queue is the first to be processed, followed by the second, and so on. Because of its simplicity, FIFO generally yields poor performance. One significant drawback is that the average waiting time can become excessively high. Short tasks may be delayed for extended periods if they are queued behind longer processes, leading to inefficient resource utilization and potential bottlenecks in the system. This issue, often referred to as the *convoy effect* [17], can significantly degrade the overall system performance.

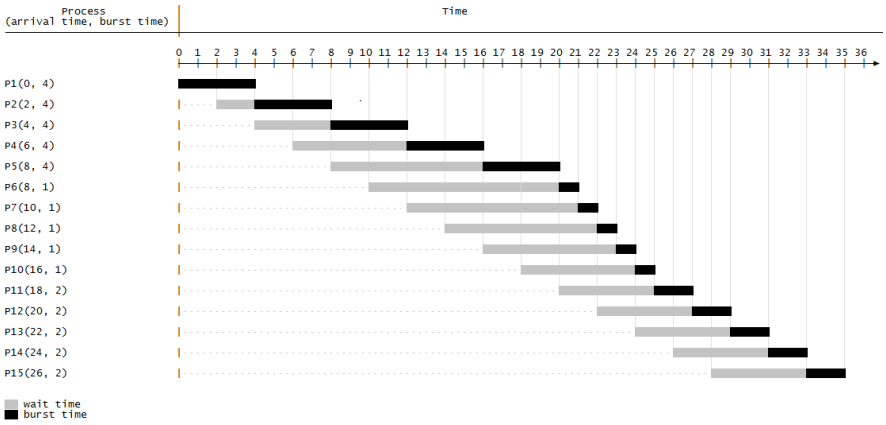


Figure 3.2: A FIFO scheduling example [18].

Shortest Job First (SJF)

Shortest Job First (SJF) is a scheduling policy that prioritizes jobs with the shortest execution time for processing. By selecting the shortest tasks first, SJF has the advantage of having the minimum average waiting time among all scheduling algorithms. As a result, SJF can lead to very efficient performance

and far superior to FIFO. However, this approach assumes to know in advance the execution time of each job, which may not always be practical. Additionally, SJF can lead to the problem of *starvation* [19], where longer jobs are repeatedly postponed in favour of shorter jobs that continue to arrive, preventing them from ever being executed.

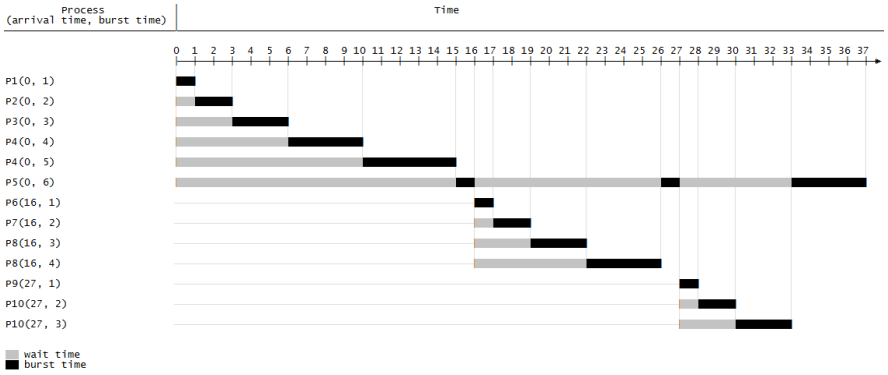


Figure 3.3: A SJF scheduling example [20].

Round-Robin (RR)

Round-Robin (RR) algorithm differs from the previous examples since it assigns to each job a fixed time slot, called *quantum*. The scheduler selects the first job in queue: if the task is not been completed within the time quantum, the process is paused and moved to the back of the queue to wait for its next turn. RR prevents the starvation problem and ensures that no job can monopolize the CPU, as all tasks have equal priority. However, if the quantum is large compared to the average job size, RR can lead to longer waiting times, especially for short jobs. On the other hand, if the time slot is very short, the continuous switching of jobs in execution can significantly increases the total overhead, causing additional delays.

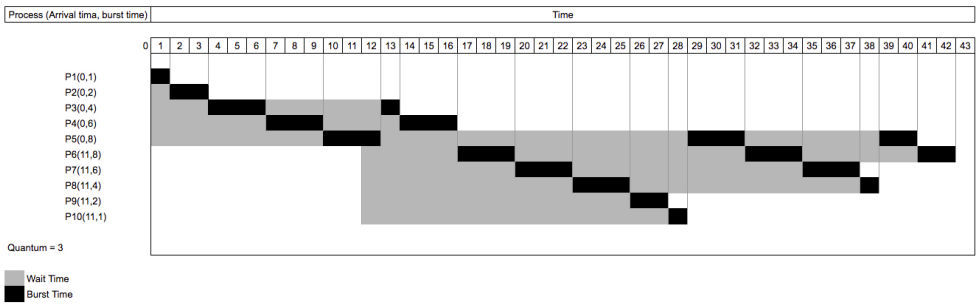


Figure 3.4: A RR scheduling example with quantum=3 [21].

3.3 RL-based Resource Management Methods

Traditional methods are static algorithms: they are simple and easy to implement, but cannot adapt to changing environments with varying workloads. This limitations have driven, in recent years, the development of more advanced techniques, particularly with the rise of Artificial Intelligence (AI). RL algorithms are able to excel at optimising resource allocation. In fact, thanks to the numerous feedbacks it receives from the environment, it is able to continuously improve the scheduling decisions, particularly due to its ability to adapt in complex environments dominated by uncertainty.

However, it is important to recognise a limitation of RL-based methods: the dependency they acquire during training in a specific scenario. While this approach makes RL algorithms highly effective in that particular context, it also limits their generalisability in other scenarios. For example, an RL model trained on a specific network or MEC server configuration may have difficulty adapting to different environments with varying workloads or system dynamics.

In this section, we will examine three distinct RL-based resource management methods, focusing on how they exploit the flexibility of RL and address the challenge of adapting to different environments.

3.3.1 DeepRM: Image-based DRL for Resource Management

DeepRM [23], is a DRL-based scheduler developed by Hongzi Mao et al. in 2016. The authors considers an online setting in which incoming jobs need to be scheduled within a cluster, which can be viewed as a MEC servers with d different types of resources, including CPU, GPU, and storage.

Jobs arrive in the system at discrete time steps and wait in a queue until they are scheduled for execution. For each job j, the resource profile r_j and duration T_j are known upon arrivals. The resource profile $r_j = \{r_{j,1}, r_{j,2}, ..., r_{j,d}\}$ specifies the amount of each of the d available system resources that the job requires. The duration T_j represent the total time required for the job to be executed. At each timestep, the scheduler chooses one or more jobs from the queue and allocate the necessary resources r_j. This resources are allocated continuously within the cluster from the start of the job's execution until its completion.

DeepRM learns to optimize various objectives such as minimizing average job slowdown or completion time. Specifically, the slowdown for job j is given by $S_j = C_j/T_j$, where C_j is the completion time of the job, i.e. the time from its arrival to the completion of the task.

The state of the system includes, the current allocation of cluster resources, as well as the resource profiles of jobs waiting to be scheduled. In addition, as we can see in Figure 3.5, the state is represented through a series of multiple images.

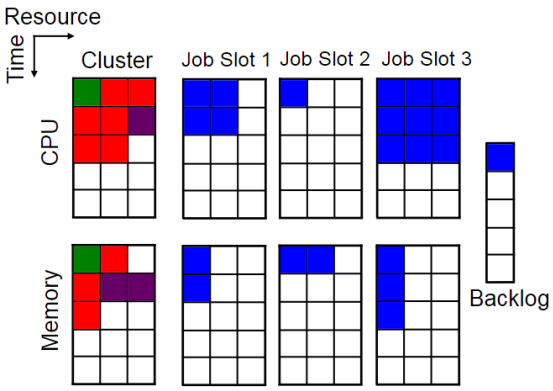


Figure 3.5: State space representation [23].

We can easily distinguish between the cluster images on the left, which use different colors, and the waiting job images on the right, which are shown in blue. We observe that the state includes a dedicate image per job and per each of the d resource types, for simplicity $d = 2$ (CPU and memory) in this example. Images have fixed dimensions: the number of rows corresponds to the number of time units being considered, while the number of columns represents the total number of resources available in the system. In the example, these are 5 and 3, respectively. In the images describing the cluster status, different colors represent different jobs: for example, the red job in Figure 3.5 uses two units of CPU, and one unit of memory, with a total duration of three timesteps.

On the other hand, the state of waiting jobs only includes detailed information for the first M jobs in the queue, with $M = 3$ in the example. The remaining jobs, instead, are summarized in the *backlog* component of the state, which simply counts the number of such jobs. This approach ensures a fixed state representation, making it more suitable as input for a neural network. It's also a reasonable strategy, indeed, an efficient scheduler aims to minimize delays by prioritizing those jobs that arrived earlier.

At each timestep, the agent can choose to schedule any set of the M jobs. However, this would lead to a very large action space, specifically, equal to 2^M. To simplify the problem, the authors decide to allow the agent to execute multiple actions within a single timestep. In this way, we can have multiple *decision epochs* in a single timestep.

The action space $\mathcal{A}$ is defined as $\mathcal{A} = \{0, 1, ..., M\}$, where $a = i$ indicates to schedule job i. The action $a = 0$, known as the *void action*, signals that the agent has finished scheduling jobs for the current timestep. Actions can be of three types: valid, invalid or void. Invalid actions may occur, for example, when there is not enough space to allocate the selected job.

Each valid action allocate the necessary resources in the first empty slots. Then, the agent observes a state transition: the scheduled job is moved to the appropriate position in the cluster image. This process is repeated for each decision epochs until the agent picks either a void or invalid action. Then, time actually proceeds: the cluster images shift up by one timestep and any newly arriving jobs are revealed to the agent.

DeepRM is designed to be customizable for achieving various objectives by defining different reward functions. In the paper, two alternative functions are analysed. In the case we want to minimize the average slowdown the reward at each timestep is computed as $\sum_{j \in \mathcal{J}} -1/T_j$, where $\mathcal{J}$ is the set of jobs currently in the system which includes both scheduled and awaiting jobs. Alternatively, in the case we want to minimize the average completion time it is sufficient to compute $-|J'|$, where J' is the set of unfinished jobs in the system. The agent receives reward only after a transition in time, meaning no rewards are given for intermediate steps.

The authors use the *REINFORCE* algorithm [24], a policy-based algorithm, in order to train the model. Furthermore, they chose a DRL approach, deciding to approximate the policy using an NN. The input of this network, i.e. the state representation described above, passes through a fully connected hidden layer with 20 neurons.

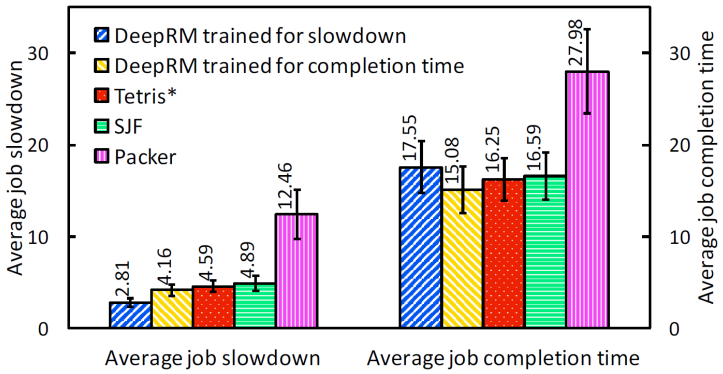


Figure 3.6: Performance of DeepRM compared to different methods [23].

In Figure (3.6), we can see the performance of DeepRM when trained specifically to optimize average job completion and average job slowdown. DeepRM is evaluated against various strategies, including SJF mentioned in section 3.2, and two baselines from *Tetris*: *packing heuristic* and *Tetris heuristics*. Packing strategy allocates jobs to resources by considering how well each job's resource demands align with the resources available. It starts by allocating jobs with the least alignment and progresses to allocate jobs with better alignment. On the other hand, the Tetris approach balances the trade-off between short jobs and an efficient use of available resources, similar to how Tetris pieces are placed to fill rows completely. The results show that, in a cluster scheduling problem, the DRL approach outperforms ad-hoc heuristics, demonstrating its potential as a valuable alternative for real-world applications.

3.3.2 DRL Resource Allocation in IoT Edge Computing

In paper [25], authors focus on the resource allocation problem in the MEC system for Internet of Things (IoT) applications. In this scenario, IoT devices upload sensor data, which are treated as jobs that need to be processed. Once arrived in the MEC system, they are queued waiting to be scheduled. The mobile edge *host level management* determines whether to allocate local resources to jobs

or forward them to the remote IoT cloud for further processing. Moreover, it is also responsible for adjusting and minimizing the virtual resources required by each application. For simplicity, in the paper it is considered only one mobile edge application deployed on the MEC system.

The state s encapsulates all information concerning the mobile edge application, which can be given as:

$$s = (C, Q, i_\psi, \tilde{n}_c, \tilde{n}_b), \tag{3.1}$$

where C represents the current allocation of computing resources, Q is the observation part of job queue, i_ψ is the value of adjusting indicator, $\tilde{n}_c$ is the number of computing resources that is requested by the mobile edge application in the next timestep, and $\tilde{n}_b$ is the number of jobs in the backlog part of waiting queue.

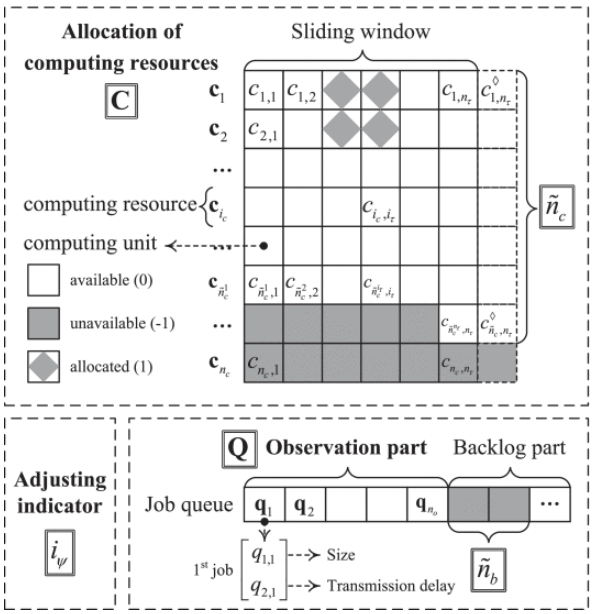


Figure 3.7: State space representation [25].

Similar to DeepRM, the current resource allocation C can be visualized as an image with dimensions $n_c \times n_\tau$ (Figure 3.7), where n_c is the number of total computing resources on the mobile edge host and n_τ denotes the number of time slices in the sliding window. Each row of C denotes one of the n_c computing resources, tracking their availability from the current time slice and looking ahead n_τ time slices into the future. Each single cell in C, referred to as *computing unit*, represents a computing resource during a specific time slice. These computing units can assume one of the three values from the set $\{-1, 0, 1\}$ depending on their current state: unavailable (-1), available (0), allocated (1). A cell may become unavailable because the system can choose to allocate fewer computing resources to the MEC application than the maximum number n_c. Specifically, every n_ϕ timesteps, the agent updates the value of $\tilde{n}_c \in \{1, 2, ..., n_c\}$, which determines how many computing resources will be requested and made available in the next timestep.

As regards the job queue Q, it is divided into two sections: the observation part and the back-

log part. The observation part contains the first n_o jobs, where each job is represented by a two-dimensional vector. The vector specifies the number of computing units required to process the job, and the transmission delay which is the time taken for the job to be transmitted from the device to the MEC system. The backlog part denotes the number of remaining jobs $\tilde{n}_b$ in the queue, with the maximum value of n_b. Finally, parameter i_ψ record the number of timesteps since the last update of $\tilde{n}_c$. Initially, i_ψ is set to 1 and increased by one at each timestep. When i_ψ reaches n_ϕ, the agent is required to adjust $\tilde{n}_c$.

In order to allocate multiple jobs at the same timestep, the authors devise a mechanism similar to the one described for DeepRM. They introduce the possibility of having multiple decision epochs in a single timestep. At each timestep, time is frozen until all jobs in the queue, both those in the observation and backlog part, are scheduled sequentially from first to last, or alternatively, the number of requested computing resources is adjusted. To manage both functions, action space $\mathcal{A}$ is divided into two subsets:

$$\mathcal{A} = \mathcal{A}_c \cup \mathcal{A}_\phi, \tag{3.2}$$

where $\mathcal{A}_c$ includes all actions $a = (\delta_\tau, \delta_c)$ that schedules the first job in queue, as well as action $a = (-1, \emptyset)$, which signifies that no computing units are allocated and the job is forwarded to the IoT cloud for processing. Specifically, when the agent selects action $a = (\delta_\tau, \delta_c)$, it allocates δ_c computing resources continuously from the δ_τ-th time slice until the job completion. On the other hand, $\mathcal{A}_\phi$ includes all actions $a = (-2, \delta_\phi)$ that updates the number of computing resources requested, $\tilde{n}_c$, by setting it equal to δ_ϕ, where $\delta_\phi \in \{1, 2, ..., n_c\}$.

The reward function is designed to achieved specific objectives, in this example authors consider minimizing both the average completion time of jobs (including processing and waiting time) and the average number of requested computing units. It follows that, the reward that the agent aims to maximize, is defined as the negative value of the weighted sum of the average job completion time and the average number of requested computing units.

To train the model described, the authors employed the DQN algorithm, using a multi-layer neural network to approximate the Q-values. They also compared the performance of the DRL approach with two benchmark policies. The first one, called *resource precedence policy*, prioritizes actions which allocate the most computing resources δ_c and then selects among them the action with the least time slice offset δ_τ. The second one, known as *time precedence policy*, follows the opposite order: first selects actions with smallest time slice offset δ_τ and then chooses the action that allocates the most computing resources δ_c. Both policies prioritize the maximum utilization of resources δ_c, in order to enhance the minimum duration of job processing, and the least time offset δ_τ leading to the minimum waiting time. In this way, these policies align with the same objectives pursued by the DRL agent.

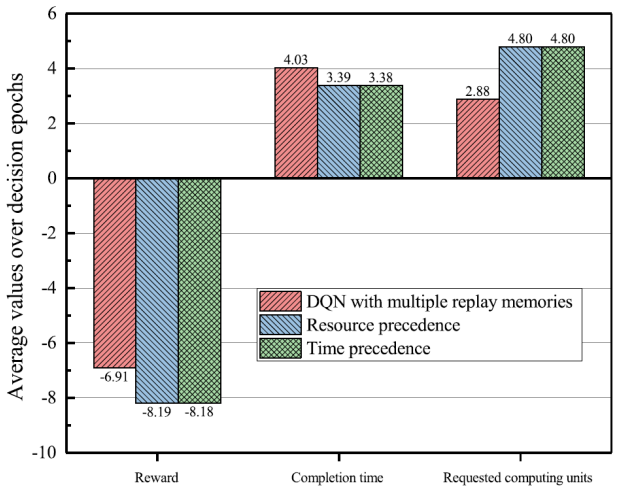


Figure 3.8: Performance of DQN algorithm compared to the baselines [25].

Figure (3.8) shows the average values of reward, completion time of jobs, and the number of requested computing units over the decision epochs. As can be seen, the average completion time achieved by the DRL algorithm is larger than those of the other reference policies. However, even if the learned policy may require more time on average, it needs less computing units from the MEC system. As a result, the average reward, computed as the weighted sum of the two objectives, is higher on the long run with respect to the other two approaches.

3.3.3 DRL Resource Allocation in Multi-User Wireless MEC system

The work, presented in paper [26], proposes two RL-based approaches for a multi-user MEC resource allocation system. The authors considered a scenario of one small cell where N users can communicated with a single eNodeB, in which a MEC server is deployed. Each user n generate a computation-intensive task $R_n = (B_n, D_n, \tau_n)$, where B_n is the size of the input data needed to process the job, D_n denotes the total number of required CPU cycles and τ_n is the maximum tolerable delay. D_n is directly proportional to B_n and represents the amount of computing resources required to complete task R_n.

Moreover, each job can be processed on the user's local CPU or on the MEC server via computation offloading. However, the system does not support malleability, i.e. once the task is assigned, it should be executed either locally or offline. The *offloading decision vector* $\mathcal{A} = [\alpha_1, \alpha_2, ..., \alpha_N]$ records for each user n whether the job is executed by local computing ($\alpha_n = 0$) or by offloading computing ($\alpha_n = 1$).

In this context, users can have different computation capacities ${f_n}^l$, expressed as CPU cycles per second. Users capacities are typically lower than the entire computational resource F on the MEC server. In order to process job R_n, the MEC server allocates part of the total resource F, which is denoted as f_n. The *computational resource allocation* $\boldsymbol{f} = [f_1, f_2, ..., f_N]$ represents the resource allocated for each task. Therefore, it follows that $\sum_{n=1}^{N} \alpha_n \cdot f_n \leq F$, meaning that the total amount of assigned

resource can not exceed the entire computational capacity F.

The system aims to minimize the average delay and the average energy consumption for both the local computing and the offloading computing. As regards the local computing, the local execution delay ${T_n}^l$ is defined as:

$$ {T_n}^l = D_n / {f_n}^l, \tag{3.3} $$

i.e. the requested amount of resources divided by the capacity of the local CPU. On the other hand, the local energy consumption ${E_n}^l$ is computed as:

$$ {E_n}^l = z_n \cdot D_n, \tag{3.4} $$

with z_n as the energy consumption per CPU cycle. The total cost of local computing, combining time (3.3) and energy cost (3.4), can be expressed as:

$$ {C_n}^l = \beta \cdot {T_n}^l + (1 - \beta) \cdot {E_n}^l, \tag{3.5} $$

where β is a parameter that balance the importance of both parts in the total cost.

Nevertheless, if user n decides to execute job R_n on the MEC server, the process unfolds differently. The user sends the input data to eNodeB through a wireless channel. Then, eNodeB forwards this data to the MEC server, which allocates the necessary computational resource to process the task. According to this model, the execution delay ${T_n}^o$ for user n when utilizing offloading computing can be expressed as follows:

$$ {T_n}^o = B_n / r_n + D_n / f_n, \tag{3.6} $$

that is the sum of the transmission delay and execution time, where r_n is the uplink rate in the wireless channel for user n. The energy consumption ${E_n}^o$, instead, is computed as follows:

$$ {E_n}^o = \frac{P_n \cdot B_n}{r_n} + \frac{{P_n}^i \cdot D_n}{f_n}, \tag{3.7} $$

the first term represents the energy consumed during the data upload, computed as the transmission power P_n multiplied by the time required for the transmission. The second term expresses the energy utilized by the user in the idle state, which is given by ${P_n}^i$ (the idle power consumption) multiplied by the duration of the task execution on the MEC server. Combining both contribution of time 3.6 and energy 3.7, we obtain the total cost of offloading computing:

$$ {C_n}^o = \beta \cdot {T_n}^o + (1 - \beta) \cdot {E_n}^o. \tag{3.8} $$

Finally, we can express the *sum cost* C_{all} of all users in the system as:

$$ C_{all} = \sum_{n=1}^{N} (1 - \alpha_n) {C_n}^l + \alpha_n \cdot {C_n}^o. \tag{3.9} $$

The authors define the state of the system as a vector of two components $s = (C_{all}, F_{av})$. Here, F_{av}

represents the available computational capacity of the MEC server, which is calculated by subtracting the total allocated resources from the server's entire resource capacity F, mathematically formulated as:

$$F_{av} = F - \sum_{n=0}^{N} f_n. \tag{3.10}$$

The actions are modeled in order to capture both the decision of how to allocate resources and whether to utilize local or offloading computing. This is represented by an action vector a that combines the offloading decision vector $\mathcal{A}$ and the resource allocation $\boldsymbol{f}$:

$$a = [\alpha_1, \alpha_2, ..., \alpha_N, f_1, f_2, ..., f_N]. \tag{3.11}$$

For each step, after taking action a, the agent receives a reward R that is correlated to the objective function, i.e. minimizing the total cost C_{all}. Since the DRL agent aims to maximize the rewards, the latter is negatively correlated to C_{all} and computed as follows:

$$R = \frac{{C_n}^l - C_{all}}{{C_n}^l} \tag{3.12}$$

The authors tested two different RL approaches, tabular Q-learning and DQN, comparing them against two baseline methods. The first baseline is a *full local* strategy in which all users execute their jobs on their local CPUs. The second method follows a *full offloading* approach, meaning that all users forward their tasks to the MEC server, which allocates equal resource to each task.

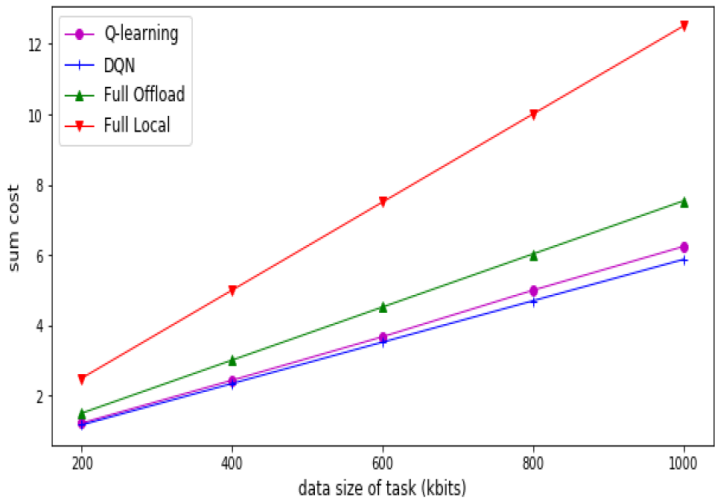


Figure 3.9: Performance comparison of the proposed methods, evaluated based on the sum cost C_{all} relative to the size B_n of the offloading data [26].

Figure (3.9) illustrate the sum cost C_{all} of the MEC system with respect to the data size B_n of offloading task, where the number of users is fixed to 5. As depicted, the cost associated with all methods

rises as the data size increases, since a larger amount of time and energy is needed for offloading larger data quantities. Among the methods compared, the proposed DQN approach achieves the best performance, as its cost escalates more gradually compared to the other strategies. It is also important to note why DQN outperforms tabular Q-learning. In Q-learning, the state-action value function $Q(s, a)$ is stored in a table, where each entry corresponds to a specific state-action pair. This approach works well for problems with small, discrete state and action spaces. However, in environments like this one, with continuous states, the number of possible state-action pairs can increases exponentially. To address this, the continuous state space must be discretized, dividing it into a finite number of bins, each representing a specific state. Nevertheless, if the discretization is too low the performance will suffer since the agent may end to treat two very different states as the same simply because they fall into the same bin. DQN eliminates the need for discretization by directly working with continuous state representations allowing it to handle complex environments with continuous and high-dimensional states much more effectively than tabular Q-learning. Eventually, the simulation results demonstrate that both the proposed RL schemes outperform the baseline, highlighting their effectiveness in optimizing resource allocation, minimizing time delay and reducing energy consumption.

4

Model Formulation

As demonstrated in the previous chapter, RL strategies can strongly enhance the performance of resource management systems in mobile edge scenarios. Due to their ability to handle the uncertainty of environments, adapting dynamically to schedule different tasks, they easily outperform traditional heuristic strategies. However, the methods presented so far assume that the environment statistics are stationary when measuring final performance. In the case of RL algorithms, such statistics are represented by the transition probability $P(\cdot)$ and the reward function $R(\cdot)$. In stationary environments, RL algorithms benefit from the knowledge acquired during training, but how can they adapt to systems whose statistics constantly change? In a dynamic environment, a one-time training approach based on past data would no longer be sufficient for ensuring high performance. Instead, it is necessary to exploit a *continual learning* approach, where the agent's knowledge is continuously refined to let the agent adapt to new conditions.

Adopt a continual learning approach in resource allocation scenarios could be critical, since the operation of training the algorithm may involve the consumption of resources. Particularly, in an MEC scenario, like the one considered in this thesis, training an algorithm makes it necessary to generate additional jobs, which are added to those normally generated by the users, possibly degrading the immediate performance. The underlined problem, that we named *cost of learning*, involves critical challenges: how is it advantageous to adopt a continual learning approach in a dynamic system where training an algorithm has a direct impact on the system itself? How many resources can be re-allocated to training without negatively impacting the system's overall performance? The balance between learning and resource efficiency is essential to ensure that the gains, derived from an RL approach, do not exceed the cost of the training process itself. Indeed, in some critical cases, it may happen that the training cost of an RL algorithm is too high, making the use of an heuristic approach, which does not require additional resource consumption, a more practical and efficient alternative.

In the remaining of this thesis, we will study the cost of learning in an MEC scenario where user and training jobs compete for the same resources. By doing so, we will examine how the process of

learning can impact overall system performance by analysing the trade-offs between the effectiveness of RL strategies and the consumption of resources required for their training. In this chapter, we introduce a specific MEC scenario as our case study, formulate the problem of allocating resources in the target scenario as an RL task, and provide a detailed explanation of the training algorithm used to solve it.

4.1 MEC scenario

This section describes the MEC system used in this thesis, designed to manage the resource allocation of jobs. Additionally, we detail the job arrival process and the resource requirements needed for their execution.

As shown in Figure 4.1, we focus on a MEC system deployed at a base station within a single-cell cellular network. We model the MEC computing resources as a cluster equipped with various types of resources, such as CPU, GPU or memory. Additionally, we assume that the cluster can be treated as a unified resource pool, overlooking any effects of resource fragmentation.

In the coverage area of the base station, N user devices are randomly distributed and need to process their data by exploiting the MEC resources. In this context, the data processing tasks determine the computational jobs received by the cluster. After arriving at the MEC system, job are queued waiting to be scheduled. Moreover, each job requires a predetermined amount of computing resources and time to be processed, which are known upon arrival. Hence, the MEC scheduler monitors the status information of the system, including the available resources in the cluster and the awaiting jobs in the queue. Based on the monitored information, the scheduler decides whether and which of the waiting jobs to allocate first.

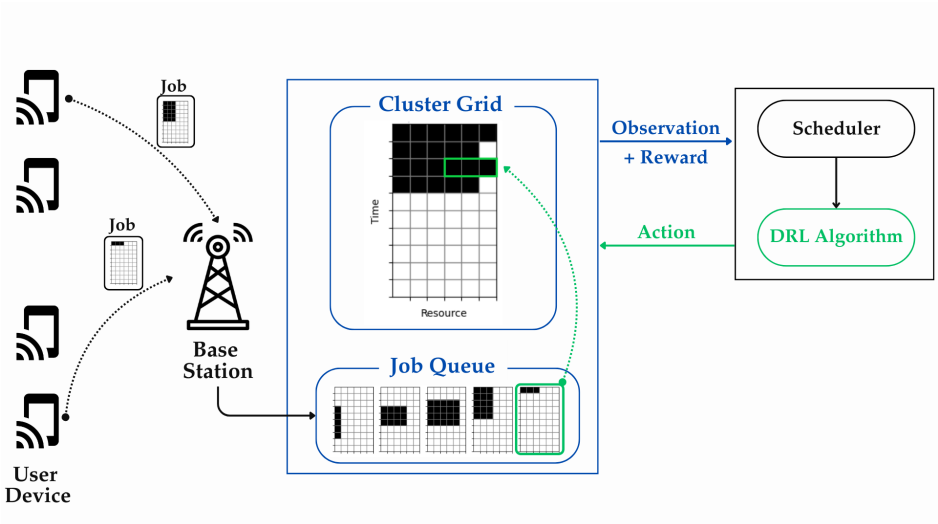


Figure 4.1: Illustration of the system model.

We assume that each user $n \in \{1, 2, ..., N\}$ generates jobs according to a Bernoulli process $\mathcal{X}_n \sim$ Bernoulli(p), where p is calculated as:

$$p = \frac{\lambda}{N}, \tag{4.1}$$

where N is the total number of users in the system and λ is the *average arrival rate*, i.e. the rate at which jobs arrive in the system. The latter is computed as:

$$\lambda = \frac{\rho}{\mu}, \tag{4.2}$$

in which ρ is the desired *average load*, that is the average resource utilization of the cluster. In our analysis, we make ρ vary between 10% to 30% of the total cluster capacity. Instead, μ is the *average service time*, i.e. the rate at which the system processes jobs. In our case μ corresponds to the job's average resources, since the MEC system schedules one job per unit time.

Jobs enter the cluster in a real-time fashion, progressing in discrete time intervals. Each incoming job j is identified by three parameters: its execution time e_j, resource demand r_j and current time spent in the queue w_j. Additionally, each job j has a maximum allowable waiting time $w_{j,max}$ it can remain in the queue. If the job's current waiting time w_j exceeds this threshold, the job is discarded from the system. This constraint ensures that jobs are either scheduled within a reasonable time or removed to prevent indefinite queuing and resource wastage. At each timestep, the scheduler makes decisions regarding which job to allocate among those currently in the queue. Once resources are assigned to a job, they remain allocated continuously until the job completion. The main objective is to minimize latency, ensuring that as many jobs are executed as quickly as possible to reduce waiting times and improve responsiveness.

We assume two classes of jobs: short and long jobs. The 80% of the jobs falls into the short category and are associated with an execution time given by the uniform distribution: $e_{short} \sim U(0.05 \cdot T_{max}, 0.15 \cdot T_{max})$, where T_{max} is the maximum execution time of a job. The remaining 20% are long jobs and are associated with an execution time given by the uniform distribution $e_{long} \sim U(0.4 \cdot T_{max}, 0.6 \cdot T_{max})$. The resource demand r is the same for both classes, $r \sim U(0.25 \cdot R_{max}, 0.5 \cdot R_{max})$, where R_{max} is the maximum resource capacity of the system. Since shorter jobs are more likely to be scheduled first in order to reduce system congestion, we assign to shorter jobs a smaller maximum waiting time compared to long jobs. Specifically, all short jobs j are associated with half of the maximum waiting time of the long jobs, so that $w_{j,max} = W_{\text{short, max}} = W_{\text{long, max}}/2$. This ensures that short jobs are prioritized for quick execution, while longer jobs are allowed more time in the queue, balancing fairness and efficiency in job scheduling.

4.2 RL Formulation

In the following section we describe the state space representation for a RL-based scheduler that has to optimize the proposed MEC scenario. In doing so, we elucidate the process behind the action selection and the computation of rewards, crucial for guiding the RL agent towards optimal scheduling strategies.

4.2.1 State Space

Extending the image-based scheduling methodology proposed in [23], we make the system state encapsulate both the current cluster conditions and the pending job requirements in terms of both resource demand and execution time. The status can be visualised through separate matrices or images, as illustrated in Figure 3.1. The first matrix, called *cluster grid*, depicts the association between the cluster resources and the jobs already scheduled by the learning agent. It is a $(T_{max} \times R_{max})$ matrix where cluster capacity is captured by the number of columns, equal to R_{max}. The resource allocation is not described only for the current time-step but also for the future, up to a total of T_{max} timesteps. On the other hand, the *queue's state* encapsulates the requirements of each awaiting job j in terms of execution time e_j, resource units r_j, and remaining waiting time w_j. In Figure 3.1, jobs are visually represented as blocks, where the height corresponds to the execution time e and the width to the resource demand r. Furthermore, some jobs are shifted downwards by a few timesteps, indicating the amount of time w they have already spent in the queue. This comprehensive depiction allows the system to track each job's waiting time alongside its resource and time requirements. For example, Job 1 in Figure 4.2 has an execution time of 3 timesteps, demands 4 resource units, and has spent 5 time units waiting to be scheduled.

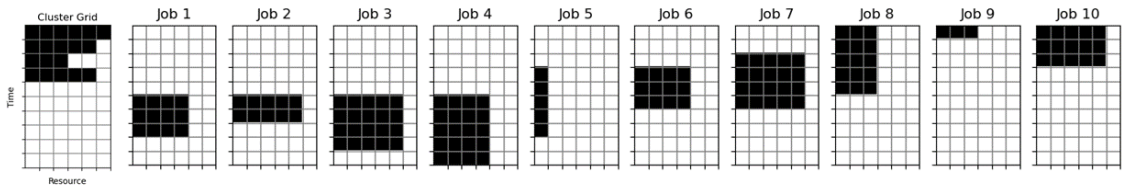


Figure 4.2: State space representation.

We opted to limit the queue to a maximum of M jobs: hence, if a new job arrives and the queue is full, it is promptly discarded. This decision not only helps to maintain a consistent state space representation, but also simplifies the learning process by limiting the action space, as discussed in section 4.2.2 . We observe that the queue's state, although visually intuitive, could be encoded in a less expensive representation, not requiring images. The necessary information for each job can be condensed into three values: the job's height (execution time e), the job's width (resource demand r), and the shift in time (waiting time w). By doing so, we significantly reduces the complexity of the input while retaining the necessary information for decision-making. For instance, the job depicted in Fig 4.3 can be represented by the tuple (3, 4, 5). This indicates that the job needs $e = 3$ time units

to be processed, requires $r = 4$ resource units, and its current waiting time is $w = 5$. If the queue is not fully occupied, the unused slots are filled with placeholder values, which are set to (–1,–1,–1), in order to maintain a consistent input format for the learning agent.

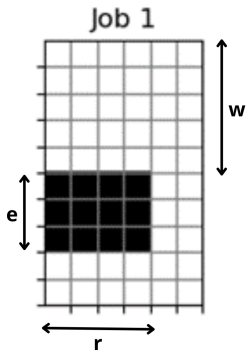


Figure 4.3: Job representation where e = execution time, r = resources, w = waiting time.

In summary, the final state is described by two matrices:

- a ($T_{max} \times R_{max}$) matrix describing the cluster allocation grid.
- a ($M \times 3$) matrix where each row contains the three values corresponding to each job or empty slot in the queue.

Both matrices are then flattened into a one-dimensional array whose total length is ($T_{max} \times R_{max}$) + ($M \times 3$).

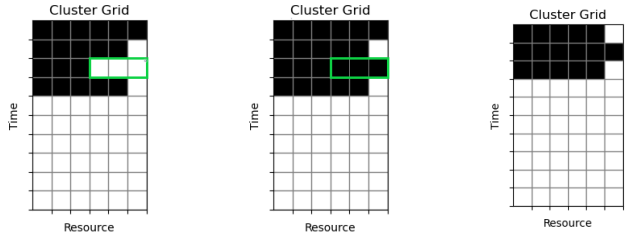
4.2.2 Action Space

At each timestep, the learning agent may consider allocating a pending job in the queue or not scheduling any jobs at all. To model every possible decision of the agent, we define the action space $\mathcal{A}$ as:

$$\mathcal{A} = \{1, 2, ..., M\} \cup \{0\}, \tag{4.3}$$

where M represents the maximum number of jobs allowed in the queue, action $a = i$ indicates scheduling job in position i, while $a = 0$, denoted as the *void action*, indicates that no job is scheduled in the current timestep. Therefore, in each time unit, the learning agent can choose from $M + 1$ possible actions.

We also need to distinguish between two types of actions: valid and invalid. Valid actions are those where a pending job is successfully allocated in the cluster grid. In other words, these actions ensure that the job's resource and execution time requirements can be fully met until completion. On the other hand, invalid actions occur in all remaining cases: when the action is void ($a = 0$), when the chosen action corresponds to an empty slot in the queue, or finally, when the corresponding job cannot be inserted correctly into the cluster image.



(a) Current state (b) Action taken (c) Next state

Figure 4.4: Transition example.

If the agent choose a valid action, the corresponding job is scheduled in the earliest possible timestep, and the system evolves into a new state accordingly. We can see an example of this in Figure 4.4 where the agent chooses action 9 (corresponding to Job 9 in Figure 3.1). The scheduler takes this action as input and identifies the earliest position within the current cluster grid (highlighted in green in Figure 4.4a). Subsequently, the job is positioned in its final location (Figure 4.4b) and removed from the queue. On the other hand, in the case of an invalid action, no jobs would be allocated and the system state would remain the same. Following these steps, regardless of whether the action was valid or invalid, the following steps are performed before the agent receives a new state observation: the cluster image shifts up by one timestep (Figure 4.4c) and any pending job which exceeds the maximum waiting time W_{max}, is permanently removed from the queue. Lastly, the waiting time of all the pending jobs in the queue in increased by one unit and all newly arrived jobs are disclosed to the agent.

4.2.3 Rewards

The reward corresponding to a certain action depends on the category to which the job belongs: short or long. For the sake of simplicity, we will only explain the case of short jobs, remembering that in the case of long jobs we have $W_{\text{long, max}} = 2 \cdot W_{\text{short, max}}$. The procedure to compute the rewards is the following:

- If action a is valid, i.e., job a is successfully scheduled, the agent receives a positive reward that depends on how much time job a has spent in the queue, as denoted by w_a. Specifically, the reward is equal to a fixed value R_{short} until $w_a < W_{\text{short, max}}/2$, and decrease it linearly over time, until reaching the value of 0 when $w_a = W_{\text{short, max}}$. We can summarise the reward function r_a, associated to valid action a, as follows:

$$r_a(w_a) = \begin{cases} R_{\text{short}}, & \text{if } w_a < W_{\text{short, max}}/2, \\ 2 - 2 \cdot w_a / W_{\text{short, max}}, & \text{if } W_{\text{short, max}}/2 \leq w_a < W_{\text{short, max}}, \\ 0, & \text{if } w_a = W_{\text{short, max.}} \end{cases} \tag{4.4}$$

- If action a is invalid the reward associated to it is always equal to 0. Hence, it results that $r_a = 0$.
- If certain conditions are met, we add a penalty to the reward, depending on the status of the

system. This occurs when incoming jobs find the queue full or if any pending jobs, after the action is taken, exceed the maximum waiting time $W_{\text{short, max}}$. In both cases, the job is discarded, and a penalty is applied to the total reward associated with the action performed in that timestep. The total value of the penalty is $-D \cdot R_{\text{short}}/2$, where D is the number of jobs discarded during that time unit.

In summary, the reward r_a associated with action a depends on whether the action is valid or invalid, as well as on the number of discarded jobs D:

$$r_a = \begin{cases} r_a(w_a) - D \cdot R_{\text{short}}/2, & \text{if } a \text{ is valid,} \\ 0 - D \cdot R_{\text{short}}/2, & \text{if } a \text{ is invalid.} \end{cases} \tag{4.5}$$

In order to give long jobs an advantage over short jobs, we decided to give the first ones a higher reward $R_{\text{long}} = 2 \cdot R_{\text{short}}$. This choice was driven by the need to prevent starvation, avoiding the agent to prefer short jobs over long ones. Indeed, short jobs can be more easily processed by the system. Consequently, with equal rewards, the agent would likely learn to always allocate short jobs, as such strategy would cumulatively yield a higher return over time.

4.3 Training algorithm

This section presents a detailed overview of the training algorithm used for optimizing the RL agent. In particular, we consider a DRL approach, which, as explained in Chapter 2, allows the agent policy to be approximated by a Neural Network (NN). We discuss the specific techniques and methodologies behind the learning process, providing a comprehensive understanding of how the model is trained to optimise its performance.

4.3.1 Double Deep Q-Network (D-DQN)

In order to train our DRL model we utilized the Double Deep Q-Network (D-DQN) algorithm, an improved version of the Deep Q-Network (DQN), previously discussed in section 2.4.1. D-DQN differs from DQN in the use of the two NNs, named policy and target network, respectively, assigning them distinct and specific roles, for more efficient results. The policy network Q is responsible for the *action selection*, while the target network $\hat{Q}$ is dedicated to the *action evaluation*.

In the DQN case, only the target network is used. Specifically, for updating the agent policy, the target Q-value y is calculated by taking the maximum Q-values estimated using the target network $\hat{Q}$:

$$y = r + \gamma \max_{a'} \hat{Q}(s', a', \hat{\theta}). \tag{4.6}$$

In other words, the target network is used both to estimate future Q-values and select the action a', associated with the maximum of such Q-values. However, this approach can lead to an overestimation of Q-values, a known issue in Q-learning methods, which can result in suboptimal policies. This overestimation occurs because using the same architecture for both selecting and evaluating actions,

potentially amplifies biases in the learning process. The D-DQN algorithm, address such a issue by modifying the update equation given in (4.6) as follows:

$$y = r + \gamma \hat{Q}(s', \underset{a'}{\operatorname{argmax}}\, Q(s', a', \theta), \hat{\theta}), \tag{4.7}$$

where target network $\hat{Q}$ is used to estimate future Q-values while the policy network Q is used to select the action a' by which the estimation takes place. This separation of functions helps mitigate the overestimation bias present in traditional DQN, leading to more accurate and stable learning.

Another enhancement we introduced alongside D-DQN is the use of Prioritized Experience Replay (PER), an improvement over Experience Replay mechanism described in section 2.4.1. The PER buffer, like its traditional counterpart, is used to store experiences and sample batches of them during training. However, the innovation of PER lies in not sampling experiences purely at random. Instead, it prioritizes experiences based on their importance, giving higher sampling probability to more significant experiences.

In our implementation, we prioritize experiences that yield higher rewards, by setting for each sample i in the PER buffer $\mathcal{B}$ the following weights α:

$$\alpha_i = exp\left(r_i - \max_{j \in \mathcal{B}} r_j\right) \tag{4.8}$$

This approach ensures that the agent focuses more on experiences that have greater impact on learning, potentially accelerating the convergence to an optimal policy.

4.3.2 Neural Network Architecture

As outlined in section 4.2.1, the Resource Management system is modeled using a bipartite state representation: the cluster grid, and the state of the queue. The cluster grid keeps track of the resource allocation and availability, while the queue state represents the jobs awaiting scheduling. Moreover, they are represented differently: the state of the cluster is an image that visually depicts the current resource allocation, instead, the queue's state is a multi-dimensional vector of parameters (e, r, w) representing the status of each job. To effectively process these distinct input types, we implemented a Multi-Branch Neural Network (MBNN) [27] whose allow us to handle different types of inputs separately, leveraging the unique characteristics of each input stream.

As we can observe from Figure 4.5, we implement a MBNN based on fully connected layers that incorporates two parallel branches. The first branch receives in input the flattened cluster grid and includes 3 layers with 512, 256, and 128 neurons, respectively. The input dimensions are progressively reduced and the Leaky RELU is used as non-linear activation functions within the hidden layers. The second branch processes the waiting job vector representation and follows a similar structure with 2 layers of 128 and 32 neurons, alternated by Leaky RELU functions. After both branches process separately the states, their outputs are concatenated and fed into a merging layer with 32 neurons that further processes the combined features. Finally, the merged representation is passed through

an output layer that generates $M + 1$ Q-values predictions: one for each possible action.

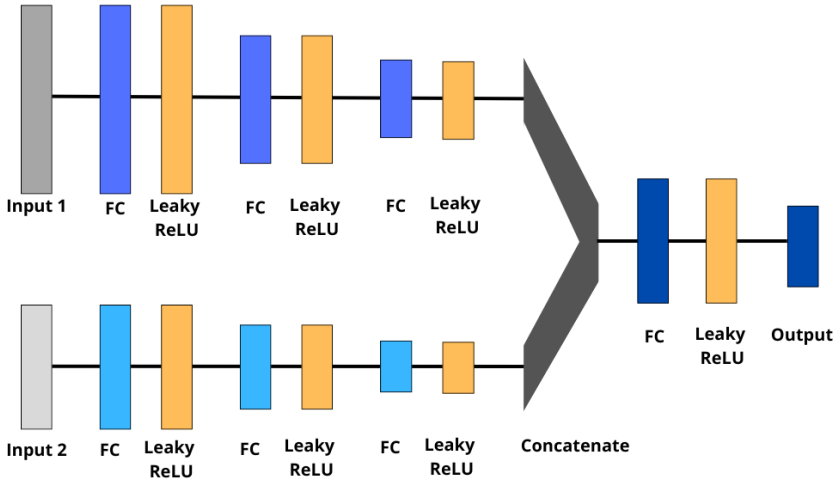


Figure 4.5: Multi-Branch Neural Network architecture.

4.3.3 Exploration Strategy: Epsilon-Greedy

In section 2.3.2, we explained the Exploration-Exploitation dilemma and the importance of choosing an exploration strategy as effective as possible. We described the epsilon-greedy algorithm balancing the amount of exploration and exploitation through parameter ϵ. However, using a constant epsilon is often not so efficient. Generally, when approaching a new task, a learning agent needs first to acquire more exploration samples, to get familiar to the environment statistics. Indeed, at early stages of the training, the Q-value estimations are very noisy, and taking the maximum over all Q-values is unlikely to lead to the optimal action. While, later, when the Q-values have nearly converged, continuing exploring by taking sub-optimal actions no longer makes sense and it is more convenient to only slightly deviate from the optimal policy. Following this idea, we progressively reduce epsilon over time to favor exploitation as the knowledge of the environment increases. At the same time, we want to avoid the exploration to decrease too quickly: in such cases, some state-action pairs may not be visited enough, leading the agent to get stuck in sub-optimal strategies.

With these goal in mind, we make the value of epsilon decrease following a reverse Sigmoid function scaled according to the number of episodes (Figure 4.6). In particular, epsilon starts at a value of 0.9 and then progressively decreases, reaching the value of 0.1.

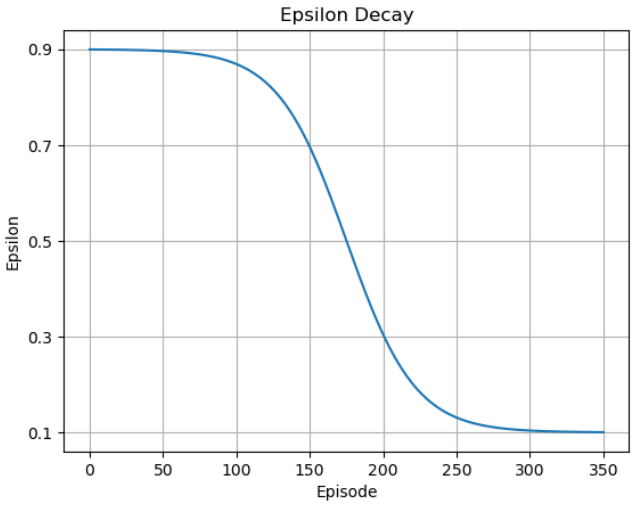


Figure 4.6: Epsilon decay function.

5

Cost of Learning Formulation

In the MEC scenario presented in the previous Chapter 4, one of the possible task to be scheduled is the training of an AI entity, e.g. classification, regression or RL algorithm. Like any other job, it consumes resources over a certain amount of time, thus reducing the resources available to the other users of the system. In our simulations, this type of job, referred to as *training job*, is handled similarly to any other generic job, with the exception that it has a fixed size: it occupies R_{max} units of resources (i.e. the entire resource capacity of the cluster) continuously for one unit of time.

Hence, we define the cost of learning as the overhead incurred by introducing training jobs in order to train the resource management system itself. In the case the AI unit has the task of optimizing the MEC system itself, we are asked to simultaneously manage regular incoming jobs and allocate resources for learning the optimal strategy. This dual objective problem represents a unique challenge, that is often neglected in the scientific literature, and the central focus of our case study.

In our model, we use a DRL algorithm to learn an effective strategy for job scheduling. The DRL agent can refine its scheduling policy by asking resources to the same cluster that it has to manage, potentially impacting the overall performance of the allocation system. This raises the problem of deciding when and whether to reserve resources for training the DRL agent instead of allocating the full cluster to the end users. One potential solution is to implement an additional learning-based strategy i.e, introducing another DRL agent with the goal of allocating MEC resources between training and users jobs. However, this would require extra resources to train the new DRL agent, bringing back the challenge of balancing resource allocation between users and the learning process, thus creating a recursive problem.

Another possibility is to include the decision directly in the action space of the DRL agent. In this scenario, the DRL agent would need to learn not only how to allocate resources for generic incoming jobs but also for its own training jobs, identifying states in which learning is more beneficial. However, as the policy improves over time, the priority of learning decreases and the same happens to the reward associated with the agent's resource demand. The result is a non-stationary environment,

which potentially prevents the agent's policy from converging towards the optimal solution. As we will see in the rest of the chapter, to avoid these issues, we decided to manage separately user jobs and training jobs. Practically, user jobs are scheduled using the policy learned by the DRL agent, while training jobs are allocated according to predetermined heuristics.

In the following sections, we will first describe a scenario with an ideal learning process, which does not consider the allocation of training jobs. Then, we will explain a first strategy to take the cost of learning into account, in which training jobs are scheduled periodically, according to a regular time frame. We refer to this approach as the *Periodic Training Strategy* (PTS). Hence, we will propose a second approach, which we call the *Advanced Training Strategy* (ATS), that leverages the knowledge of Q-values to decide when to schedule the training jobs, which makes the training process depends on the current estimation of each state-action pair. In particular, by utilizing Q-values, we can make more informed decisions when allocating training jobs. Finally, we will described a possible scenario of continual learning, in which the system statistics vary over time. In such a context, allocating resources continuously for training becomes essential to ensure that the policy adapts effectively to the changes in the environment.

5.1 Ideal Learning Process

The ideal learning process is the best possible option for a learning algorithm because it assumes that the training has no cost for the system. Normally, in a RL-based system model, as seen in the three examples in section 3.3, an ideal approach in which the agent does not consumes any resources to learn the optimal policy is assumed. In this context, it is crucial to study the performance of an ideal system that can be considered as the maximum achievable result, and thus represents an important comparison criteria.

In the following, we will outline the process by which the agent is able to learn from experience using the D-DQN algorithm, Prioritized Experience Replay (PER), and the exploration strategy explained in section 4.3. This procedure serves as a foundation for all the methods we will present in the next sections, of which only the modifications to the general structure will be discussed.

In our simulations, we decided to train the agent using episodic tasks with a fixed number of timesteps per episode. At the beginning of a new episode, the environment is reset: both the cluster grid and the queue are emptied, waiting for a new assignment cycle. At each timestep, a certain number of jobs arrive in the system, the arrival rate of which is regulated by the selection of the desired average load. For each incoming job, there is an 80% probability that it will be categorized as a short job. Otherwise, it will be classified as a long job. The specific dimensions of the job, i.e. execution time and resource demand, are uniformly chosen according to the description provided in Section 4.1. Once jobs arrive, they are immediately placed in the queue. If all the M positions in the queue are taken, all remaining jobs are discarded, causing a penalty in the final calculation of the reward r.

At this stage, the state s is disclosed to the agent, who must decide which job to select from those in the queue or chooses to skip the allocation for the current time unit. Since we use the epsilon greedy strategy with a decaying epsilon (section 4.3.3), we have that:

- With probability ϵ, the DRL scheduler chooses a random action from the available $M+1$ options.
- With probability $1-\epsilon$, the agent chooses the action that, according to the Q-values computed by the policy network, is expected to yield the highest return G.

After taking the selected action a, we can observe four possible scenarios:

1. The job in position a can be inserted correctly in the cluster grid.
2. The job in position a cannot fit entirely in the cluster grid.
3. The job in position a correspond to an empty slot in the queue.
4. Action a is the void action.

Only the first scenario corresponds to a valid action, while the remaining ones are invalid since they do not allocate any jobs.

Subsequently, the system transitions to a new state s'. In the case 1, the selected job is inserted in the first available slot of the cluster and removed from the queue. On the other hand, if the action is invalid nothing changes: in case 2, job a remains in the queue and no job is scheduled, similarly for cases 3 and 4. Concurrently, the cluster image is shifted up by one timestep, and any pending job that has waited longer than the maximum allowed time ($W_{\text{max,short}}$ or $W_{\text{max,long}}$, depending on the class) is permanently removed from the queue, at the cost of a penalty. Finally, the remaining jobs are shifted downwards, and any newly arrived job is presented to the agent.

The reward r associated with action a is calculated by summing to the reward obtained from scheduling (or not) job a, and the penalties incurred for jobs discarded due to a full queue or exceeding the maximum waiting time.

Then, the collected experience sample (s, a, r, s') is inserted in the PER buffer together with the weight associated to it. Once the buffer is completely filled, the agent can be trained through multiple batches of experience selected according to the priority. For each transition tuple in the batch, the target Q-values is computed following the D-DQN algorithm as explained in section 4.3.1. Subsequently, we compute the loss as the Mean Squared Error (MSE) of the TD error δ, that is the difference between the target Q-values y and the predicted Q-values $Q(s, a)$.

The policy network is then updated using the Adam optimization algorithm [28] in order to minimize the loss, and consequently, reduce the TD error. Adam, which stands for Adaptive Moment Estimation, is an advanced optimization technique that, adjusting the learning rate dynamically, helps the optimization process converge faster and more reliably. This contributes to make it one of the most commonly used alternative to traditional Stochastic Gradient Descent (SGD). Finally, we perform a soft updates of the target network weights $\hat{\theta}$, using the following formula:

$$\hat{\theta} = (1-\tau) \cdot \hat{\theta} + \tau \cdot \theta, \tag{5.1}$$

where τ controls the rate at which the target network's weights are updated towards the policy network's weights θ. By updating the target network slowly, we prevent drastic changes in the Q-values,

which can lead to unstable training, moreover, soft updates can help to reduce oscillations in the loss function, leading to a smoother convergence [29].

5.2 Periodic Learning Process

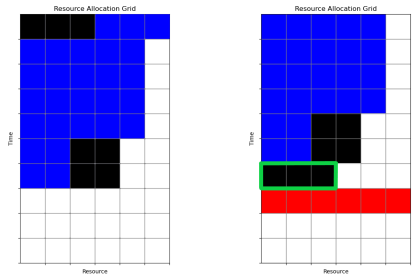
The Periodic Training Strategy (PTS) is the first example we present which considers the cost of learning in the resource management system. By simulating the insertion of training jobs in the cluster grid, we can understand how the resource and time demands, needed for training the D-DQN algorithm, can impact the overall system performance. Each training job requires the maximum available resources R_{max}, fully occupying the cluster for a single time unit. The insertion of a training job correspond to an actual improving of the current allocation strategy by training the network with b batches of experience samples, where b is a tunable hyperparameter.

We define the *training period* as the fixed time interval between successive scheduling of training jobs. It is paramount to determine the optimal training period for the scheduling task. Indeed, as the training period decreases, the more we optimize the strategy. However, the system may be overloaded by such training jobs, which take important resources away from the system users. On the other hand, if the training period is too high, the learning algorithm could converge too slowly, ending up using a suboptimal policy for long periods of time.

As anticipated the DRL agent itself is not responsible for the allocation of training jobs since it would lead to a non-stationary problem, that might prevent the convergence towards the optimal policy. At the beginning of each training period, the training job is immediately allocated in the cluster, independently of the agent's decisions. For this reason, a training job is never placed into the queue with other jobs and receives no reward after its allocation.

The transition from state s, before the allocation, and state s^*, after the allocation of the training job, deserves proper attention. The symbol (*) denotes a *training state*, i.e. the state at the beginning of the training period, in which a training job is allocated in the cluster. To ensure that the Markov property of the MDP is always satisfied, the future state must depend only on the current state and the chosen action. Consequently, each transition sample (s, a, r, s') stored in the PER buffer must be consistent with this principle.

For instance, consider an experience tuple (s, a, r, s'^*), where initial state s transitions to a training state s'^* as the new training period begins (Figure 5.1).

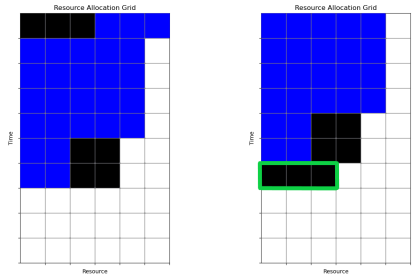


(a) State s (b) State s'^*

Figure 5.1: Invalid transition from s to s'^*.

In the image above, short jobs are represented in black, long jobs in blue, and training jobs in red. The agent, following action a, schedules a (1×3) job (highlighted in green in the Figure 5.1b) and then time shifts upwards. However, it also observes the allocation of a training job, which is not a direct result of any agent decision. In our MDP formulation, there is no a for which the state transition probability $P(s, a, s'^*)$ is greater than zero. In other words, s'^* is a state that cannot be reached from s through any possible action. The agent would never see s'^* as a result of its actions, even though it observes it in the experience sample, causing a misinterpretation of the data.

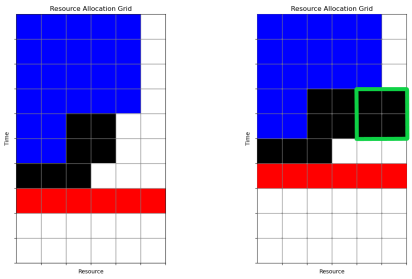
To ensure accurate learning, the correct transition to be stored in the PER buffer is the one illustrated in Figure 5.2, where only the agent's action (scheduling job a) and the subsequent time shift are considered.



(a) State s (b) State s'

Figure 5.2: Valid transition from s to s'.

Conversely, a transition like (s^*, a, r, s') is valid and satisfies the Markov property. The agent starts in state s^*, performs action a and observes the resulting state s', a direct consequence of its own decision. As illustrated in Figure 5.3, we can see that the scheduler allocates a (2×2) job (highlighted in green), followed by a time shift of one cell upward. This transition is correct, since the training job is already allocated in the cluster before the action is taken.



(a) State s^* (b) State s'

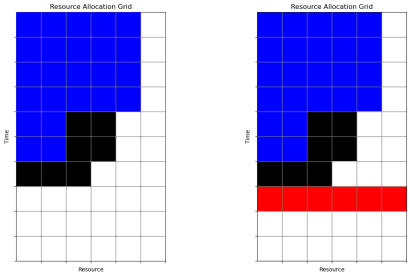
Figure 5.3: Valid transition from s^* to s'.

To ensure an accurate and efficient training of the algorithm, we meticulously store only those transitions that satisfy the Markov property and align with the underlying logic of the environment. By considering only the valid transitions, we avoid the introduction of extraneous or misleading information that could interfere with the learning process and deviate the agent from the learning of the optimal strategy.

5.3 Advanced Learning Process

The Advanced Training Strategy (ATS) originates as an improved version of PTS discussed in the previous section. Unlike PTS, where the training states are all states occurring at the beginning of the period, ATS leverages the knowledge of the Q-values to determine which states are the best for allocating resources for training. States with higher Q-values are generally less congested or in general more favorable to the insertion of a job.

To identify which state should be intended for training, for each timestep, we simulates the insertion of a training job in the current state s. We obtain two possible initial states: s and s^*, as shown in Figure 5.4.



(a) State s (b) State s^*

Figure 5.4: Two possible initial states: s and s^*.

Then, we compute the Q-values for both states using the policy network Q and compute the following measure:

$$\psi = \beta \cdot \left[\max_a Q(s^*, a)\right] + (1 - \beta) \cdot \left[\max_a Q(s^*, a) - \max_a Q(s, a)\right], \tag{5.2}$$

where β is a parameters that balances the contributions of the two components. The first term represents the maximum Q-values over all possible actions for the training state s^*, reflecting the potential reward of taking the optimal action in that state. The second term measures the difference between the maximum Q-values of state s^* and state s. This difference intuitively quantifies the penalty (or cost) on the expected reward when a greedy action is performed in training state s^*. In essence, ψ measures the weighted sum of the expected reward, which represent the intrinsic quality of the training state, and the cost associated to it.

We record the value of ψ for each initial state throughout the learning process. Once the buffer is full, we compare the current value of ψ with the values in the buffer. If the current ψ is in the 99th percentile, meaning the present state is more favorable for training than 99% of the past states, a training job is allocated for the current timestep, and the network is trained using D-DQN algorithm. The decision to use the 99th percentile was based on several simulations that demonstrated its effectiveness in our environment. However, it is possible that, in different environments and systems, this value may need to be revised and adjusted to the optimal setting.

At the early stages of the learning process, when the Q-values have not already converged, it is possible that they may not be able to provide an accurate estimate of the quality of a given state. To address this, we develop a method to determine when it is appropriate to begin relying on the Q-value estimates, and thus when we can start using ATS as described earlier.

During training, we do not only records the value of ψ, but also track the TD error δ and a newly defined measure ϕ, which is used together with δ for determining the moment in which the Q-values become reliable. In D-DQN algorithm, the TD error corresponds to the difference between the target Q-value y and the observed Q-value, which is computed as follows:

$$\begin{aligned} \delta &= y - Q(s, a) \\ &= r + \gamma \hat{Q}(s', \operatorname*{argmax}_{a'} Q(s', a')) - Q(s, a). \end{aligned} \tag{5.3}$$

Instead, the measure ϕ is given by the difference between the maximum Q-values for training state s^* and the average Q-value across all possible actions for state s:

$$\phi = \max_a Q(s^*, a) - \mathbb{E}_a[Q(s, a)]. \tag{5.4}$$

Hence, ϕ represents the penalty on the expected reward for taking a greedy action in training state s^*, compared to the predicted Q-value of state s. When the value of the TD error becomes smaller than ϕ, it suggests that the Q-values are sufficiently reliable to proceed with using ATS. In contrast, if the TD error exceeds ϕ, it signals that the Q-values may be less reliable or more uncertain. In such cases, it is advisable to go back to PTS. Indeed, the periodic alternative, not relying on Q-value estimates,

can avoid the potential risks associated with inaccuracies in the Q-value predictions.

5.4 Continual Learning Process

Our methodology assumes more importance in environments characterized by non-stationary behaviors, which makes it necessary to periodically adapt the learning agent to new conditions. In a stationary scenario, it is generally more convenient to train the agent as fast as possible and then enjoy the benefits of the optimal strategy on the long terms. Instead, in non-stationary scenarios, where training phases are periodically re-initiated, the training effectiveness and the cost of learning problems are particularly critical. The strategies designed to manage systems that continuously evolve fall within the so-called continual learning paradigm.

Continual learning is a branch of ML that involves training models to learn new information constantly over time while maintaining the knowledge they have acquired from previous tasks. Continual learning is essential in dynamic environments, where the statistics can suddenly change without any prior notice. The main purpose is to allow models to adapt to new conditions exploiting the past experience. Particularly, in continual tasks, it is no longer feasible to ignore or overlook the resource consumption associated with training, as it persists over time and becomes an integral part of the system's ability to adapt to different situations.

In our simulations, we evaluate the proposed strategies considering a continual learning scenario where the target MEC environment is characterized by an increasing average load over time. Theoretically, increasing the average load alters the parameters of the transition matrix P in the underlying MDP that models the scenario. Since the transition probabilities change as the system dynamics (e.g., resource availability, job arrival rates) evolve with the load, the optimal decision-making policy may no longer be effective under these new conditions. As a result, the policy needs continuous adjustment to remain aligned with the updated MDP and ensure efficient system performance.

We compare four different approaches. The first is the Shortest Job First (SJF) method, described in Section 3.2, which serves as the benchmark for all our experiments. The second approach uses a pre-trained DRL agent that has learned the optimal strategy in a stationary environment and continues to apply it without considering the increase of the average load. We will refer to this approach as the *fixed strategy*. The last two approaches utilise pre-trained DRL agents, but allowed to continue learning from new data to adapt the agent to to the changing conditions. Specifically, the third method employs PTS, while the fourth uses ATS as detailed in the previous section.

Our objective is dual: first, we want to demonstrate the effectiveness of the continual learning approach in improving performance. Secondly, we want to highlight the resource consuming nature of continual learning. Indeed, continual learning offers adaptability and improves the agent's ability to respond to changing conditions, but this comes at the cost of significant computational overhead, which can affect overall system efficiency. By analyzing the advantages offered by the different approaches, we will offer a thorough knowledge of the trade-offs associated with applying continual learning in real-world systems.

6

Results

In this chapter, we present and discuss the simulation results conducted to evaluate the cost of learning in the MEC system. In particular, we investigate the trade-off between the necessity to guarantee users access to the greatest number of resources and the overhead needed to optimize the learning agent that manages the resource allocation process. Indeed, the optimization of the allocation technique comes at the expense of immediate resource availability. We examine this balance closely, comparing different strategies and identifying the conditions that allows the learning system to continuously adapt to new scenarios without overly degrading user performance. In doing so, we will prove that adopting learning-based strategies comes with a cost, and evaluating the learning effectiveness besides the pure performance is fundamental for the design of real systems. Indeed, if the impact of the cost of learning is too high, it may be more convenient to adopt traditional solutions that do not involves a training cost.

This chapter is organized into two main sections, each focusing on different scenarios in terms of stationarity. The first section examines performance in a stationary MEC facility, where system statistics (i.e., the jobs' arrival process) remain constant over time. In this context, we investigate the optimal training period for the learning agents, identifying the value that best balances the cost of learning with system performance. Additionally, we compare various resource allocation strategies during the learning process, highlighting their strengths and weaknesses in terms of the rate by which user jobs are allocated in the system.

The second section shifts the focus to a dynamic MEC facility, where the jobs' arrival process change over time. Here, we explore different DRL agents adapt to evolving conditions, studying how effectively they respond to such changes in comparison to traditional methods. In addition, we pay particular attention to the speed and accuracy of this adaptation, as well as the overall impact on system performance.

6.1 Performance Analysis in Stationary Environments

In this section we examine the behaviour of various resource allocation strategies in a MEC facility with stationary statistics. We consider the model defined in Chapter 4, where the average load is fixed to the 30% of the total cluster capacity. Specifically we set:

$$\rho = 0.3 \cdot T_{\text{max}} \cdot R_{\text{max}}, \tag{6.1}$$

where $T_{\text{max}} \cdot R_{\text{max}}$ is the cluster capacity representing the resource availability for a total of T_{max} time units. For the sake of simplicity, from now on, we will refer to the average load in its normalised form where only the percentage of capacity utilisation is made explicit.

In Table 6.1, we summarize the main parameters of the system along with their corresponding values used throughout the simulations. The first block gathers the parameters necessary for the comprehensive definition of the environment. Instead, the second block shows the hyperparameters utilized for the learning process.

Parameter	Symbol	Value
Number of users in the system	N	1000
Average load	ρ	0.3
Number of jobs in queue	M	10
Maximum cluster's height	T_{max}	20
Maximum cluster's width	R_{max}	20
Maximum short job's waiting time	$W_{\text{short, max}}$	4
Maximum long job's waiting time	$W_{\text{long, max}}$	8
Maximum short job's reward	R_{short}	1
Maximum long job's reward	R_{long}	2
Number of time units per episode	N_{step}	1000
Discount factor	γ	0.95
Batch size	B	16
Number of batches per training step	b	10
Weights' soft update parameter	τ	0.005
Learning rate of Adam optimizer	α	10^{-3}
ATS parameter	β	0.4

Table 6.1: Parameters of the system

In our simulations, we evaluate and compare the performance of different approaches by considering three key metrics. First, we track the average reward over time during the training of the agent, recording the achieved performance across the episodes. This metric provides immediate insights into how effectively the agent learns and how quickly it converges to the optimal policy. Secondly, along with the average reward, we show the cumulative average reward, calculated as the integral of all rewards accumulated over the entire training phase. Cumulative rewards offer a smoother version of the results, and provide a more general view of the overall performance trend, reducing abrupt changes and fluctuations. In addiction, we measure the average reward during an inference

period, where the agent's performance are evaluated applying the learned policy without any further learning. This evaluation allow us to asses the final gains and to conduct an immediate comparison between the proposed methods.

6.1.1 Periodic Training Strategy (PTS): the effect of the training period

In this analysis, we aim to determine the optimal value to use for the training period by exploring the trade-off between the frequency of the policy updates and the resultant consumption of resources. Frequent updates, i.e. short training periods, may improve the strategy but consume resources intended to user jobs. The goal is to find the training period which maximizes system performance while minimizing the resource overhead associated with frequent policy updates. In order to do so, we tracked the performance of the DRL agent in a MEC scenario with a fixed average load $\rho = 0.3$.

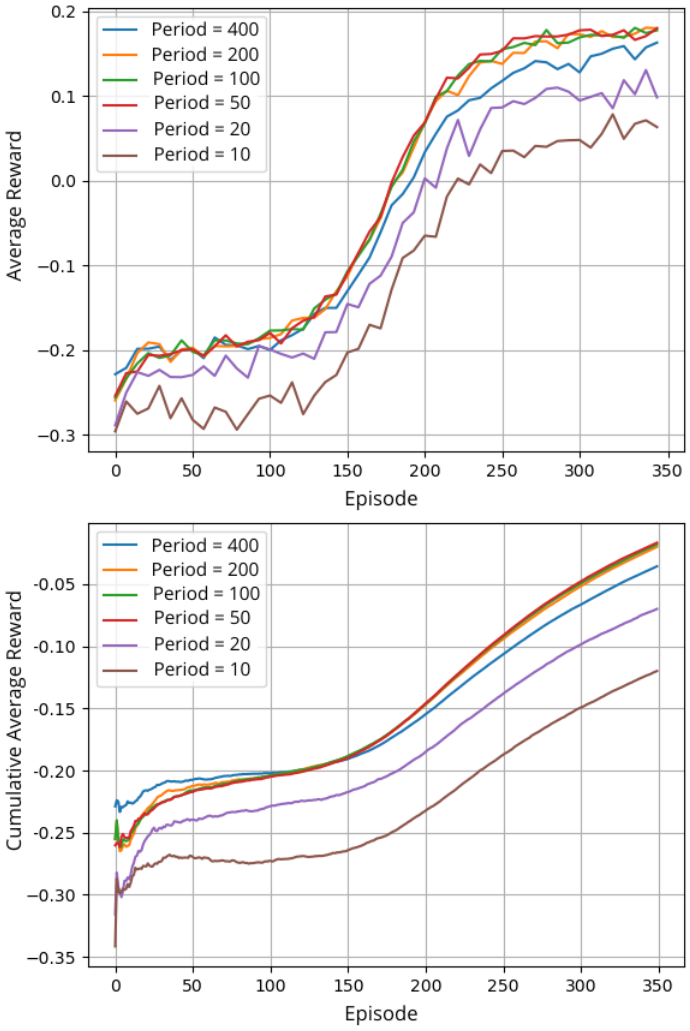


Figure 6.1: The effect of the training period: average and cumulative average rewards.

As shown in Figure 6.1, we trained the agent using PTS (described in section 5.2), experimenting with different training periods to identify the most effective configuration. We simulate the system over 350 episodes, each consisting of 1000 time units, employing the epsilon-greedy strategy with epsilon decay described in section 4.3.3. We observe that, especially when examining the average cumulative rewards, longer training periods, such as updating every 400 steps, perform better in the short term. Although, once convergence is reached, they stabilise at relatively low values. This behaviour is quite unexpected in environments where the cost of learning is neglected and the final performance depends directly on the speed of the learning process. Instead, in our case, where training consumes important system resources, it makes complete sense. Indeed, initially, less frequent training is more convenient in terms of rewards, as the underloaded cluster can allocate more resources to users even with unrefined strategies. However, in the long run, less frequent training becomes inefficient, because a suboptimal policy continues to be applied, preventing further performance improvements.

On the other hand, very short training intervals, such as 10 or 20 steps, perform poorly because they consume an excessive amount of resources. As a result, the system is unable to meet many user demands, even when employing refined strategies, leading to numerous reward penalties. The best results are obtained with agents using intermediate training periods, such as 200, 100 or 50 steps. These values represent an optimal balance between the cost of learning and policy improvements, maximising performance without overloading the system.

In Figure 6.2 below, which illustrate the performance of PTS in the inference phase, i.e. without further policy updates, we can determine which value of the training period is the more beneficial in the long run. We can note that the optimal value is equal to 50, and that the performance collapses once this threshold is exceeded. For this reason, we can assume that a training period of 50 represents the upper limit for which, once exceeded, the cluster is no longer able to cope with the overload, leading to a drop in performance.

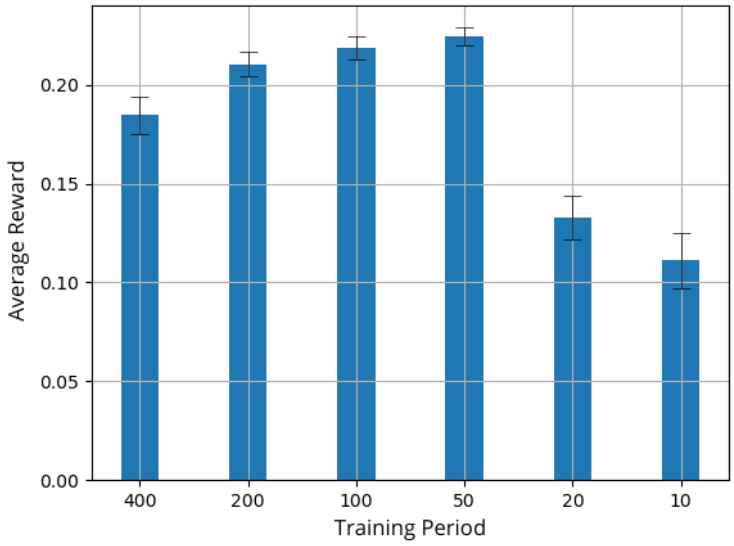


Figure 6.2: Average reward performance in inference.

6.1.2 A comparative analysis between different resource allocation methods

In the following analysis, we compare the performance of three different version of our learning-based framework and a benchmark policy, which implements an heuristic rule to allocate the jobs. As benchmark, we utilize the SJF scheduling strategy, which prioritizes jobs based on their execution time. Although SJF is a relatively simple algorithm, it is very effective in minimising the average waiting time for tasks, making it one of the most widely adopted scheduling methods for MEC or similar scenarios.

The upper performance limit achievable by RL-based approaches is instead represented by an *ideal scenario* where the cost of learning is completely neglected. In this case, the process of updating the policy would not involve the scheduling of any jobs in the MEC facility, becoming a process that occurs without any associated resource consumption. This ideal assumption provides a theoretical benchmark for evaluating the efficiency of effective RL methods, allowing us to evaluate to what extent the cost of learning affects the performance of a real system. We compare the SJF benchmark and the ideal approach with the PTS and ATS methodologies detailed in Section 5.3. We recall that PTS employs a periodic scheduling of training jobs, while ATS dynamically identifies optimal states for allocating training jobs by leveraging Q-values. We simulate the resource allocation system over 1000 episodes, each consisting of 1000 time units. Additionally, we employ the epsilon-greedy strategy with epsilon decay for the first 350 episodes, and then maintaining a constant epsilon of 0.1 for the remaining episodes.

As illustrated in Figure 6.3, all RL-based approaches surpass the performance of SJF around episode 170 and continue to improve, eventually achieving an average reward of approximately 0.2.

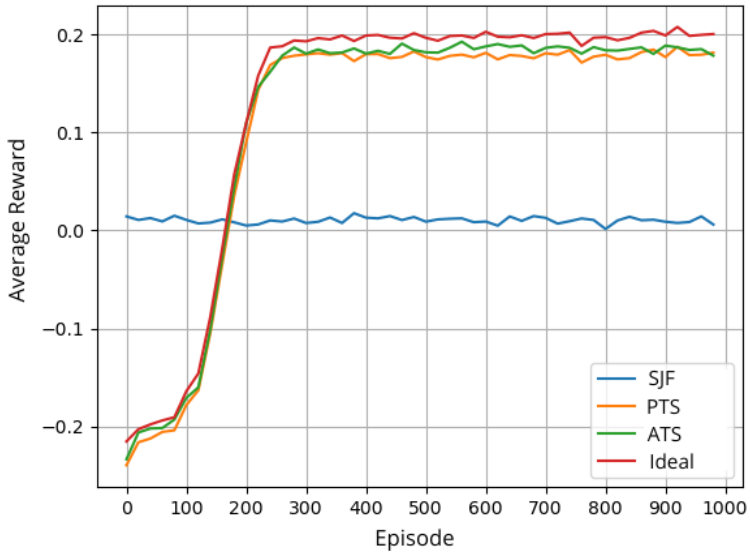


Figure 6.3: Average rewards during training.

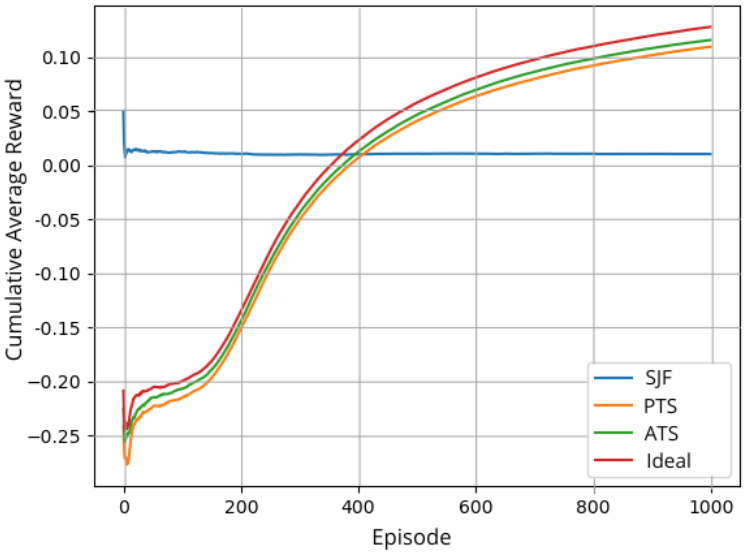


Figure 6.4: Cumulative average rewards.

The cumulative average reward graph (Figure 6.4) shows that all three methods recover from the initial disadvantage around episode 400, overcoming the initial SJF advantage. However, we can note that if the environment remains stable for less than about 400 episodes, the use of an RL strategy is not really beneficial, as the initial losses are not sufficiently compensated during this period.

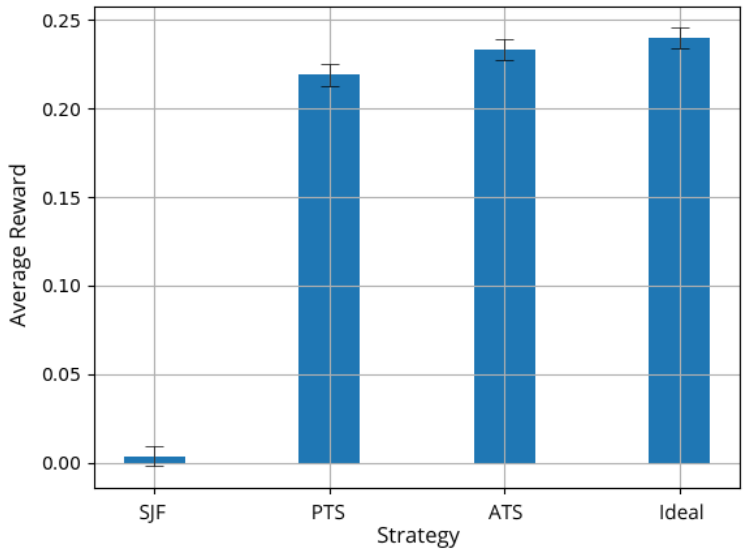


Figure 6.5: Average reward performance in the inference phase.

As expected, the ideal case outperforms all other approaches. In fact, although it trains as frequently as PTS, the system does not have to handle the jobs associated with the agent training and,

therefore, can accomplish more user requests. This allows us to quantify concretely the cost of learning, demonstrating that neglecting this factor, in real systems with similar characteristics, can lead to significant oversimplifications of the problem.

Furthermore, we observe that ATS achieves better results compared to PTS. This difference is particularly evident in Figure 6.5, where the performance of ATS approach almost reaches that of the ideal case. This confirms the fact that, ATS, based on the knowledge of Q-values, is able to recognize the states in which it is more convenient to train. Hence, the ATS approach successfully identifies the the moments in which the system is less busy, minimizing the resource consumption associated with training, and effectively reducing the cost of learning.

6.2 Performance Analysis in Dynamic Environments

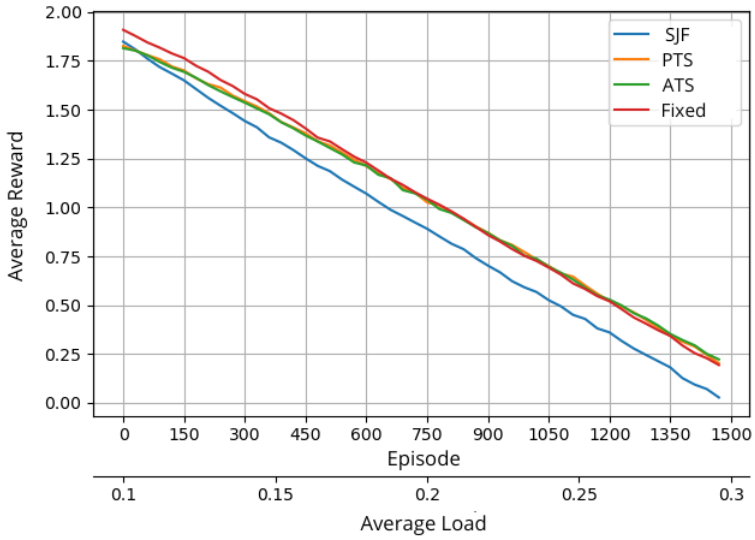
In this section, we analyse the performance of the proposed DRL methods in a dynamic environment, in which the rate at which new user requests are generated evolve over time. These changing conditions require the agent to continuously adapt, demanding an ongoing investment of resources for training. As discussed in Section 5.4, this kind of scenario can be managed via a continual learning approach, where the agent iteratively refines its strategies while retaining the knowledge and experience accumulated from past interactions.

In this work, we simulate the learning process of the agent in a MEC scenario with varying average load. Initially, the load is set to 0.1, increasing linearly until it reaches 0.3. Under these conditions, the agent has to manage the available resources in an increasingly overloaded system, where the number of job requests arriving per unit time grows steadily. We track the performance over 1500 episodes, each consisting of 1000 timesteps, comparing the four approaches that were discussed in the first half of this chapter. The benchmark is again represented by the SJF method, which serves as a reference point for evaluating the baseline performance in the dynamic system. Hence, we implement three learning-based strategies (section 6.1.2), each trained under stationary conditions, with an average load $\rho = 0.1$, using respectively: the ideal strategy, the PTS, and the ATS approach. The result is three distinct agents, each of which is subsequently deployed in a dynamic environment to adapt to the progressively increasing load.

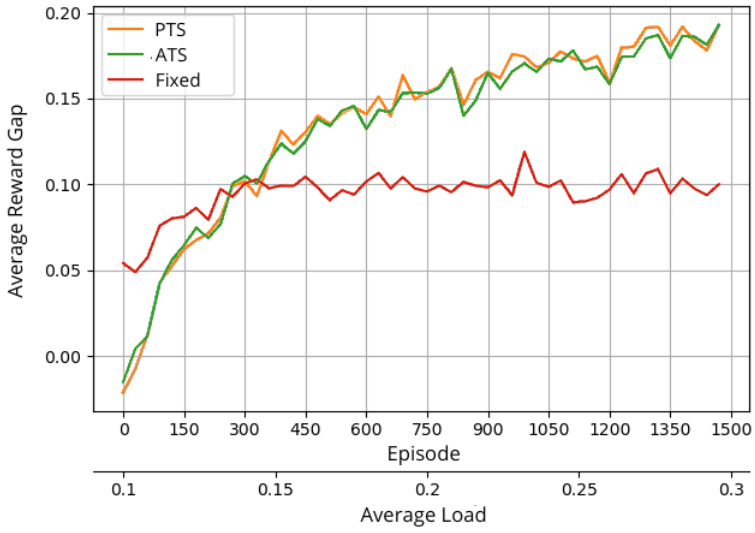
The policy network derived from the ideal strategy is directly deployed in the dynamic environment without further training. Due to its lack of adaptation, we refer to this approach as the *fixed strategy*. In contrast, the agents trained with PTS and ATS continue to interact with the evolving environment, adapting to changing statistics. Both of these adaptive agents maintain a fixed exploration rate ϵ, equal to 0.1, enabling them to continuously explore and learn from the environment as it evolves.

In Figure 6.6a, is shown the average reward over time. As expected, the reward decreases, as the average load increases, following a perfect linear dependence. For a more detailed performance comparison, Figure 6.6b shows the reward gap between the SJF baseline and the RL-based methods, this allow to overlook the decrease in performance due to the greater load, focusing on the benefits introduced by continual learning.

We can note that, initially, the continual learning approaches, namely the PTS and ATS approaches, achieve lower rewards compared to the fixed strategy. This can be explained by the fact that the agent needs more time and experience to adapt to the new conditions. Furthermore, even minimal exploration during the learning process puts continual learning methods at a disadvantage, as it lead to the use of suboptimal policies. In contrast, the fixed strategy benefiting from a purely greedy approach, and with no need to allocate resources for training, allows the system to handle more user requests, resulting in higher immediate rewards.



(a) Average rewards during training.



(b) Comparison of the reward gap: the difference between SJF and RL methods' performance.

However, as shown in Figure 6.6b, around episode 300, when the average load approaches 0.15, the performance of PTS and ATS begins to outperform that of the fixed strategy. From this point onward, their performance continues to improve, creating an increasingly larger gap in rewards compared to the fixed strategy alternative.

To further evaluate the methods, we conducted a final test by simulating resource allocation in a stationary environment with $\rho = 0.3$ after the training phase concluded. Figure 6.7 demonstrates that ATS outperforms PTS, as well as the fixed strategy, even in this scenario.

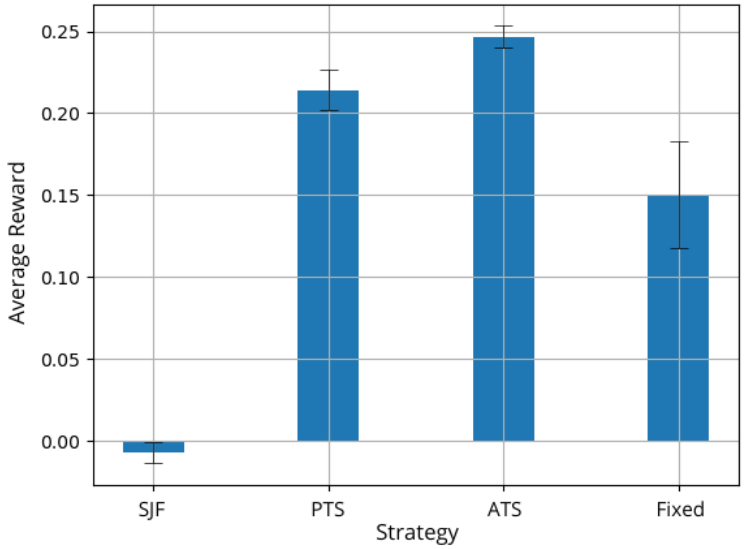


Figure 6.7: Average reward performance in inference.

This result confirm how continual learning is necessary to allow the agent to refine its policy according with new dynamics. Besides, considering effective training strategy that take the system conditions into account for performing training actions is strongly beneficial for the overall performance. In our system, the most effective approach is represented by ATS, despite maintaining some level of exploration and consuming resources for learning, strategically trains the agent during less busy periods when the MEC system is less burdened. By doing so, ATS minimizes the impact on user requests, allowing the system to achieve high performance during both the training and the inference phase.

7 Conclusion

In this thesis, we presented an in-depth analysis of the cost of learning in a MEC environment, focusing on scenarios where resource management is performed by a deep reinforcement learning (DRL) agent. Specifically, we developed a novel framework in which users and learning process shares a single resource pool, which involves complex and realistic constraints for resource allocation.

We studied diverse strategies to determine the amount of resources spent on training. In particular, the Periodic Training Strategy (PTS), where training jobs are executed at regular, predefined intervals, and the Advanced Training Strategy (ATS), where the more convenient training condition are determined based on the Q-values associated to that environment state. The performance of these strategies was tested in both stationary and dynamic environments, analyzing the cost of learning: in terms of both resource consumption and impact on user experience.

The results for the stationary scenario demonstrated that significant resources are spent during training, a factor that has often been overlooked in existing literature. This suggests that future work in similar RL-based systems must account for these training costs, as neglecting them could lead to inaccurate and optimistic estimation of system performance.

In dynamic environments, the role of the cost of learning becomes even more pronounced. Indeed, in such changing conditions, a continual learning approach is essential to enable the agent to continuously adapt and refine its policy. However, this kind of strategy need an ongoing consumption of resources. As a result, relying on pre-trained agents without considering the need for continual updates is no longer viable. Moreover, ignoring the costs associated with continual learning can lead to a substantial overestimation of the system's real performance.

The thesis also demonstrated the effectiveness of adaptive strategies that intelligently schedule training jobs when the system is in the most favourable states. This approach exhibited performance levels nearly equivalent to the ideal conditions, where no training cost is considered, indicating that intelligent training strategies can effectively mitigate the impact of the cost of learning.

These results encourage further research efforts in this domain. While this work contributes valu-

able insights into the cost of learning in a single MEC server, it would be interesting to expand the model to consider more complex environments, e.g., considering a MEC scenario consisting of a distributed system of servers. This expansion would allow a more comprehensive understanding of resource management challenges in real-world scenarios.

Another direction for future research could be the analysis of new performance metrics, including, the average completion time or the average waiting time of jobs arriving in the system. Even if these measures are already incorporated in the computation of the rewards, they can provide a more direct understanding of the impact of learning in user experience. In addition, with the growing concerns around sustainability, also the energy consumption of RL-based MEC systems should be investigated.

Finally, conducting real-world experiments or deploying the proposed framework in more realistic MEC models could provide empirical data to validate and refine our results. Such practical validation would ensure that the proposed strategies are not only theoretically valuable but also applicable in real-world environments, where unexpected factors and constraints must be addressed.

In conclusion, this thesis provides a first step towards understanding the cost of learning in MEC systems and offers practical strategies for resource management in RL-based environments. Future work should continue to refine these strategies, with a focus on optimizing both system performance and the cost associated with training of learning agents.

References

[1] H. Li, G. Shou, Y. Hu, and Z. Guo, "Mobile edge computing: Progress and challenges," in *2016 4th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud)*, 2016, pp. 83–84.

[2] H. T. Dinh, C. Lee, D. Niyato, and P. Wang, "A survey of mobile cloud computing: Architecture, applications, and approaches," *Wireless Communications and Mobile Computing*, vol. 13, no. 18, pp. 1587–1611, 2013. eprint: `https://onlinelibrary.wiley.com/doi/pdf/10.1002/wcm.1203`.

[3] N. Abbas, Y. Zhang, A. Taherkordi, and T. Skeie, "Mobile edge computing: A survey," *IEEE Internet of Things Journal*, vol. 5, no. 1, pp. 450–465, 2018.

[4] B. Jamil, H. Ijaz, M. Shojafar, K. Munir, and R. Buyya, "Resource allocation and task scheduling in fog computing and internet of everything environments: A taxonomy, review, and future directions," *ACM Comput. Surv.*, vol. 54, no. 11s, 2022.

[5] M. A. Sharf and O. O. Aldawibi, "Mobile edge computing issues and proposed solution," in *2022 IEEE 2nd International Maghreb Meeting of the Conference on Sciences and Techniques of Automatic Control and Computer Engineering (MI-STA)*, 2022, pp. 560–563.

[6] Z. Ning, P. Dong, X. Wang, *et al.*, "Mobile edge computing enabled 5g health monitoring for internet of medical things: A decentralized game theoretic approach," *IEEE Journal on Selected Areas in Communications*, vol. 39, no. 2, pp. 463–478, 2021.

[7] F. Mason, F. Chiariotti, and A. Zanella, "No free lunch: Balancing learning and exploitation at the network edge," *Proc. of IEEE ICC 2022*, pp. 631–636, 2022.

[8] S. Lahmer, F. Mason, F. Chiariotti, and A. Zanella, "Fast context adaptation in cost-aware continual learning," *IEEE Trans. Mach. Learn. Commun. Netw.*, vol. 2, pp. 479–494, 2024.

[9] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. MIT press, 2018.

[10] V. Mnih, K. Kavukcuoglu, D. Silver, *et al.*, "Playing atari with deep reinforcement learning," 2013.

[11] Y. Mao, J. Zhang, and K. B. Letaief, "Dynamic computation offloading for mobile-edge computing with energy harvesting devices," *IEEE Journal on Selected Areas in Communications*, vol. 34, no. 12, pp. 3590–3605, 2016.

[12] S. Safavat, N. N. Sapavath, and D. B. Rawat, "Recent advances in mobile edge computing and content caching," *Digital Communications and Networks*, vol. 6, no. 2, pp. 189–194, 2020.

[13] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief, "A survey on mobile edge computing: The communication perspective," *IEEE Communications Surveys & Tutorials*, vol. 19, no. 4, pp. 2322–2358, 2017.

[14] X. Sun and N. Ansari, "Edgeiot: Mobile edge computing for the internet of things," *IEEE Communications Magazine*, vol. 54, no. 12, pp. 22–29, 2016.

[15] L. Zhao, J. Liu, Y. Shi, W. Sun, and H. Guo, "Optimal placement of virtual machines in mobile edge computing," in *GLOBECOM 2017 - 2017 IEEE Global Communications Conference*, 2017, pp. 1–6.

[16] J. Wang, L. Zhao, J. Liu, and N. Kato, "Smart resource allocation for mobile edge computing: A deep reinforcement learning approach," *IEEE Transactions on Emerging Topics in Computing*, vol. 9, no. 3, pp. 1529–1541, 2021.

[17] M. Blasgen, J. Gray, M. Mitoma, and T. Price, "The convoy phenomenon," *SIGOPS Oper. Syst. Rev.*, vol. 13, no. 2, pp. 20–25, Apr. 1979.

[18] Maxtremus, "Fifo schedule example," in *Wikimedia Commons, the free media repository*, Available from `https://commons.wikimedia.org/wiki/File:Fifo_schedule.png`, 2015.

[19] R. A. Milito and H. Levy, "Modeling and dynamic scheduling of a queueing system with blocking and starvation," *IEEE Transactions on Communications*, vol. 37, no. 12, pp. 1318–1329, Dec. 1989.

[20] Maxtremus, "Sjf schedule example," in *Wikimedia Commons, the free media repository*, Available from `https://commons.wikimedia.org/wiki/File:Shortest_job_first.png`, 2015.

[21] B. Meiri, "Round robin schedule example," in *Wikimedia Commons, the free media repository*, Available from `https://commons.wikimedia.org/wiki/File:Round_Robin_Schedule_Example.jpg`, 2017.

[22] D. Olliaro, M. Ajmone Marsan, S. Balsamo, and A. Marin, "The saturated multiserver job queuing model with two classes of jobs: Exact and approximate results," *Performance Evaluation*, vol. 162, p. 102 370, 2023.

[23] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, "Resource management with deep reinforcement learning," in *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, ser. HotNets '16, Atlanta, GA, USA: Association for Computing Machinery, 2016, pp. 50–56.

[24] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine Learning*, vol. 8, no. 3, pp. 229–256, May 1992.

[25] X. Xiong, K. Zheng, L. Lei, and L. Hou, "Resource allocation based on deep reinforcement learning in iot edge computing," *IEEE Journal on Selected Areas in Communications*, vol. 38, no. 6, pp. 1133–1146, 2020.

[26] J. Li, H. Gao, T. Lv, and Y. Lu, "Deep reinforcement learning based computation offloading and resource allocation for mec," in *2018 IEEE Wireless Communications and Networking Conference (WCNC)*, 2018, pp. 1–6.

[27] S. Aslani, M. Dayan, L. Storelli, *et al.*, "Multi-branch convolutional neural network for multiple sclerosis lesion segmentation," *NeuroImage*, vol. 196, pp. 1–15, 2019.

[28] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, 2017. arXiv: `1412.6980 [cs.LG]`.

[29] T. P. Lillicrap, J. J. Hunt, A. Pritzel, *et al.*, *Continuous control with deep reinforcement learning*, 2019. arXiv: `1509.02971 [cs.LG]`.