DRL-BASED TASK OFFLOADING FOR DEADLINE-SENSITIVE APPLICATIONS IN MULTI-ACCESS EDGE COMPUTING

by

Hui Huang

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

at

Dalhousie University
Halifax, Nova Scotia
June 2024

To my dear parents, Guoting Huang & Nihong Wei

Table of Contents

List of Tables

List of Figures

Abstract

The proliferation of computation-intensive applications, such as online gaming and autonomous driving, has imperatively urged resource-constrained mobile devices to alleviate the computation and energy consumption pressure with the aid of external computing resources. Recently, a cutting-edge computing paradigm, Multi-access Edge Computing (MEC), has emerged as a promising solution to mitigate the resource shortage problem with mobile devices by selectively offloading a portion of computation-intensive tasks to physically-close edge servers. Over the past years, a series of task offloading schemes, including non-learning-based offloading and learning-based offloading, have been extensively studied. However, we notice that many of the computation-intensive applications are also deadline-sensitive. Namely, the computation tasks from these applications often have deadlines to satisfy. Nevertheless, the existing task offloading schemes face several limitations that hinder their applicability in deadline-sensitive MEC systems. In the thesis, we aim to employ Deep Reinforcement Learning (DRL) to address the task offloading problems in multi-tier deadline-sensitive MEC systems. First, we propose an innovative task offloading scheme for partially observable MEC systems, referred to as PDMO, which incorporates partially observable DRL and Dynamic Voltage and Frequency Scaling (DVFS) to minimize the energy consumption of mobile devices while guaranteeing deadline satisfaction. Second, we devise a novel Multi-access Edge-assisted Learning-based Offloading (MELO) scheme to effectively optimize the completion time of tasks in a highly dynamic MEC system. Lastly, we propose a unique offloading scheme for safety-critical tasks, Constrained Reinforcement Learning based Offloading (CRLO). With CRLO, a safety layer is integrated to the policy network of the learning-based policy generator, which effectively eliminates risky offloading decisions that could lead to deadline misses. Additionally, to achieve more efficient offloading decisions, Informer, a computationally-efficient long-sequence forecasting model, is utilized to forecast temporally-dependent system states for the upcoming time window. The experimental results indicate that all of the proposed learning-based offloading schemes outperform the baseline methods in terms of energy consumption or task completion time.

List of Abbreviations

MEC Multi-access Edge Computing

MDP Markov Decision Process

DS Deadline-Sensitive

NDS Non-Deadline-Sensitive

MTS Multiple-Time-Slot

STS Single-Time-Slot

LSTM Long Short-Term Memory

RNN Recurrent Neural Network

DDPG Deep Deterministic Policy Gradient

PPO Proximal Policy Optimization

DRL Deep Reinforcement Learning

CRL Constrained Reinforcement Learning

MINLP Mixed Integer NonLinear Programming

QoS Quality-of-Service

DQN Deep Q-Network

RL Reinforcement Learning

DVFS Dynamic Voltage and Frequency Scaling

TD3 Twin Delayed Deep Deterministic Policy Gradients

TSF Time-Series Forecasting

DPM Dynamic Power Management

IC Integrated Circuits

AET Actual Execution Time

WCET Worst-Case Execution Time

CC Cycle Conserving

LA Look Ahead

DRA Dynamic Reclaiming Algorithm

ASR Aggressive Speed Reduction

DL Deep Learning

IoT Internet-of-Thing

MIMO Multiple-Input-Multiple-Output

C-RAN Cloud-Radio-Access-Network

EDF Earliest-Deadline-First

FIFO First-In-First-Out

CST Context Switch Time

5G Fifth Generation

6G Sixth Generation

PDMO Partially-observable DRL-based Multi-tier Offloading

MELO Multi-access Edge-assisted Learning-based Offloading

CRLO Constrained Reinforcement Learning-based Offloading

Acknowledgements

I would like to extend my deepest gratitude to my supervisor, Qiang Ye, for his exceptional guidance, unwavering support, and invaluable mentorship throughout the duration of my doctoral journey, especially amid the pandemic. His expertise, patience, and encouragement have been pivotal in shaping the direction and success of this thesis.

I am also indebted to my esteemed committee members for their insightful feedback, scholarly advice, and constructive criticism, which have significantly enhanced the quality and depth of this research. My heartfelt gratitude goes to my parents for their sacrifices and boundless love, without which my academic journey would have been unimaginable.

I would like to express my sincere appreciation to Yitong Zhou, Yuxuan Jiang, Fudong Li, and all other colleagues and peers for their camaraderie, encouragement, and intellectual exchange, which have enriched my academic experience and fostered a supportive community. Finally, I aspire to give back to the world in the same generous spirit that my advisor, mentors, friends, and partners have shown me through their unwavering support and guidance.

Chapter 1

Introduction

In this chapter, we start with an overview of MEC systems. Subsequently, we delve into the existing task offloading schemes for MEC systems. Thereafter, we analyze the limitations associated with the current offloading schemes. Finally, we summarize the contributions of our research in addressing these identified limitations.

1.1 Overview of MEC Systems

The ubiquity of mobile devices, such as smartphones, tablets, and laptops, has resulted in a proliferation of mobile applications. These mobile applications, e.g., image processing, augmented reality, and autonomous driving, demand instantaneous interactions and complex computations to satisfy high-level Quality of Service (QoS) [47, 56, 59]. Generally, meeting the QoS requirements of such applications necessitates a substantial amount of computation resources. However, due to portable and commercial considerations, mobile devices are often configured with limited hardware resources, such as low-speed CPU and small-capacity memory. The execution of multiple computation-intensive applications concurrently on mobile devices may result in difficulty in satisfying strict time constraints of deadline-sensitive tasks and energy requirements of mobile devices. Consequently, how to complete computation-intensive tasks efficiently on resource-constrained mobile devices has become a challenging problem for both academia and industry.

To alleviate the computation burden on mobile devices, MEC has been introduced as a viable solution by selectively offloading a portion of computational tasks to nearby edge servers [88,101,102]. Technically, MEC comprises two types of devices: resource-constrained mobile devices and resource-rich edge servers. By offloading some of computation-intensive tasks from mobile devices to edge servers, MEC significantly reduces energy consumption of mobile devices and facilitates completion of all the tasks generated on mobile devices. Despite the effectiveness of MEC in addressing

the computation problem on mobile devices, determining whether a task should be processed locally or offloaded to edge servers is non-trivial in practice. For example, if a large amount of data needs to be uploaded to an edge server when a task is offloaded, the power used for data uploading could exceed that consumed when the task is processed locally. Furthermore, due to finite computation resources of edge servers, a limited number of tasks are allowed to be offloaded. The reason is that offloading tasks to the overloaded edge servers inevitably lead to unacceptable queueing delays. Additionally, when a large amount of data needs to be transferred for edge computing, mobile devices can experience excessively long transmission delays, potentially causing considerable constraint violations. Therefore, blindly offloading deadline-sensitive and energy-consuming tasks to edge servers could not only result in considerable deadline misses, but also negatively degrade the overall performance of MEC systems.

1.2 Task Offloading for MEC Systems

In general, developing effective task offloading methods is about finding satisfactory answers to the following three critical questions. The first question is whether a task should be offloaded or not. The second question is related to the scenario where multiple edge servers are involved in MEC systems. In this case, mobile device needs to decide where to offload the task. The final question is concerned with how to determine the optimal offloading policies that can effectively satisfy all performance requirements.

Great efforts, such as game theory, heuristics, and dynamic programming, have been made by researchers to answer the questions [15, 26, 33, 75]. Fundamentally, these schemes are not learning-based. They strive to improve offloading decisions through the utilization of exhaustive search methods or mathematical optimization techniques, which depend on either high computational complexity or reliable mathematical models. For instance, Jošilo *et al*. proposed a game-theory-based model to solve the problem of offloading delay-sensitive task in multiple-wireless-link MEC scenarios, considering the minimization of both task completion time and energy

consumption [48]. Sundar *et al*. proposed a heuristic offloading algorithm that transforms the original offloading problem into its binary-relaxed surrogate to minimize completion time for deadline-sensitive tasks [81]. In [76], Rodriguea *et al*. developed a mathematical model to determine the suboptimal offloading decisions for a group of deadline-sensitive applications in varying 5G/6G networks, aiming to minimize the total service delay. Jiang *et al*. proposed an offloading approach using Lyapunov optimization to maximize the long-term quality of user experience while simultaneously minimizing the energy consumption of mobie devices [46].

In recent years, considerable DRL-based based offloading approaches have also been proposed in the literature to optimize task offloading policies. With RL-based offloading methods, an RL agent interacts with the environment in a trial-and-error manner. Based on the accumulated feedback from the environment, the appropriate offloading policy could be found without prior knowledge of the system. The benefits of RL-based methods over non-learning-based methods can be summarized as follows. First, instead of creating an intricate mathematical model to optimize offloading policies iteratively, RL agents learn the most appropriate offloading polices adaptively via a trial-and-error exploration mechanism without knowing the prior knowledge of MEC environment. Moreover, the overhead of policy inference in RL-based offloading approaches is much lower than conventional mathematical models, as only one forward propagation of the policy network is required for policy generation after model training. Gao *et al*. proposed a multi-agent actor-critic offloading approach to enhance task completion rate and reduce average system cost simultaneously in a large-scale MEC system that embraces a large amount of cooperative-competitive heterogeneous mobile devices [32]. Wang *et al*. proposed a meta reinforcement learning-based offloading method to improve the adaptability of learning model when offloading heterogeneous tasks in different MEC environments [89]. Dai *et al*. investigated the optimization of offloading for computation-intensive tasks with a set of time constraints in vehicular networks and proposed asynchronous DRL-based offloading approch to jointly optimize offloading decisions, resource allocation, and renting cost [21]. In [1] and [22], the authors considered a mobile-edge-cloud offloading architecture in a 5G network and proposed the energy-efficient DRL-based offloading approaches to mitigate long-term

energy consumption of mobile devices.

1.3 Research Gap

Although there have been many attempts to enhance offloading decisions, they still suffer from several drawbacks. First of all, the aforementioned DRL-based offloading schemes only consider Non-Deadline-Sensitive (NDS) tasks that have no strict deadlines. When handling Deadline-Sensitive (DS) tasks with a hard deadline, the schemes cannot properly guarantee the time constraint satisfcation. Secondly, these schemes cannot lower CPU frequency to make full use of idle CPU slack to reduce the energy consumption of mobile devices. Thirdly, the existing schemes assume that complete information about MEC systems is available at each decision point. However, in practice, many real-world environments involve unobservable system states. If a system leverages incomplete information to make task offloading decisions, massive suboptimal offloading policies will inevitably incur, resulting in serious deterioration of system performance. Fourthly, considering the time-varying nature of wireless network and edge server workload, task offloading becomes highly complicated and is typically formulated as a Mixed-Integer NonLinear Programming (MINLP) problem, which generally leads to NP-hard complexity [95]. Therefore, acquiring optimal offloading policies with minimal computational overhead using conventional non-learning-based offloading approaches is often impractical due to their high computational complexity, especially when offloading deadline-sensitive tasks that demand fast policy generation. Alternatively, RL-based offloading methods are promising. However, most of them are designed for systems with a limited number of actions. In systems with a large or even continuous action space, the existing RL-based offloading schemes often struggle to find the optimal offloading policy. Fifthly, most existing RL-based offloading methods assume that both the training and inference operations of the learning algorithm are carried out on mobile devices, resulting in heavy computation workload on the mobile devices [16, 100]. Specifically, although reinforcement learning is an effective method for autonomous task offloading, the training and inference operation of RL inevitably consume a noticeable amount of computational resources [28].

Namely, when mobile devices must complete time-sensitive tasks, allocating computation resources of mobile devices to RL may not be feasible. Finally, most existing RL-based offloading models focus on the decision-making for every single time slot, which is computationally expensive in highly-dynamic MEC systems, where the RL agent frequently updates its input states and resolves the corresponding offloading problems. To address this problem, MTS offloading schemes are devised to regularly determine the offloading decisions for a set of tasks that will arrive in the upcoming MTS period [41, 109]. However, it is still challenging for the RL agent to produce appropriate offloading policies if it only relies on the observations perceived at the begining of each MTS period. Namely, Single-Time-Slot (STS) observation does not accurately reveal the state variation pattern in the subsequent MTS period. Without sufficient and precise information as inputs of the policy network, the RL agent may fail to generate the best offloading policies.

1.4 Research Contributions

The objective of the thesis is to utilize deep reinforcement learning, DVFS, and Time-Series Forecasting (TSF), to minimize energy consumption or task completion time in multi-tier MEC systems. In this thesis, we focus on enhancing the efficiency of MEC systems by minimizing energy consumption and task completion time, while ensuring the deadline constraint of tasks are satisfied.

1.4.1 Contributions of PDMO

In the thesis, to bridge the research gap presented in Section 1.3, we first propose an intelligent offloading scheme, called PDMO, for the efficient offloading of deadline-sensitive and non-deadline-sensitive tasks in an MEC system with only one edge server. The primary objective of PDMO is to minimize the overall energy consumption of mobile devices while ensuring the deadlines oftasks are satisfied. To achieve this goal, we devise a novel learning algorithm, called Partially-Observable Twin Delayed Deep Deterministic policy gradient (POTD3), which is capable of effectively approximating the missing observations of system state. Furthermore, we employ

a hybrid task scheduling approach based on DVFS to optimize energy consumption for both periodic deadline-sensitive tasks and aperiodic non-deadline-sensitive tasks. The main contributions of PDMO can be summarized as follows:

(i) We propose an innovative DRL-based offloading scheme for partially-observabl MEC systems, PDMO, which aims to meet the deadline of deadline-sensitive tasks while minimizing the total energy consumption of mobile devices by uploading a selected set of tasks to edge servers. Specifically, the offloading problem is first formalized as a POMDP due to the existence of unobservable states. Then, the problem is solved with a modified Deep Deterministic Policy Gradient (DDPG) method, POTD3. Our simulation results indicate that PDMO outperforms the existing methods.

(ii) We propose a comprehensive task scheduling algorithm for a task set involving both deadline-sensitive and non-deadline-sensitive tasks, which not only greatly reduces the energy consumption of mobile devices, but also guarantees that the deadline of deadline-sensitive tasks can be met. Specifically, the DVFS technique, which enables mobile devices to adjust CPU frequency dynamically for saving mobile energy, is used in the scheduling algorithm.

(iii) To minimize the energy consumed by learning algorithms, we propose a holistic decision-making mechanism for the tasks that arrive within a fixed time interval. With this innovative mechanism, the frequency of invoking the DRL process is significantly reduced, ultimately conserving more energy for task processing and offloading.

1.4.2 Contributions of MELO

Although PDMO serves as an effective method to determine the optimal offloading policy, it is constrained by the following critical limitations. Firstly, it is specifically proposed for simpler MEC systems involving only one edge server, thereby limiting the full utilization of resources at the edge when multiple edge servers are available. Secondly, the learning algorithm employed by PDMO tends to generate substantial

sub-optimal offloading policies due to the overestimation that occurs during the learning process. Lastly, the training of PDMO takes place on the resource-limited mobile device, resulting in fewer computational resources available on the mobile device for processing periodic deadline-sensitive tasks. To address these limitations, we propose MELO scheme for a multiple-edge-server MEC system to minimize the completion time of periodic deadline-sensitive tasks while ensuring compliance with their deadlines. Particularly, we consider an MEC system that is composed of 3 tiers: mobile devices, edge servers, and cloud server, where a mobile device could offload part of its deadline-sensitive computational tasks to one of the multiple edge servers or cloud server, thereby largely shortening the completion time of these tasks. Furthermore, considering the potentially extensive action space in MEC systems, we adopt Twin Delayed Deep Deterministic Policy Gradients (TD3) as the learning algorithm in MELO because TD3 excels in managing large or continuous action space while effectively mitigating overestimation issues during the learning process [29]. Additionally, in order to allocate more computational resources to deadline-sensitive tasks on mobile devices, we introduce a novel learning framework called Edge-Assisted Learning Architecture (EALA). This architecture involves transferring the entire training phase of TD3 to a learning server deployed at the edge. As a result, a mobile device only requires a computationally-light agent for policy generation. To the best of our knowledge, it is the first attempt to utilize EALA for deadline-sensitive task offloading in deadline-sensitive MEC systems. The main contributions of MELO are summarized as follows:

(i) We propose a novel edge-assisted DRL offloading scheme, MELO, which utilizes TD3 to optimize offloading policies tailored for deadline-sensitive applications within a multiple-edge-server MEC system. Technically, the challenge inherent in our MEC system lies in effectively scheduling multiple periodic deadline-sensitive tasks concurrently across multiple edge servers to minimize long-term costs. In our research, we tackle this challenge by formulating the offloading problem as a Markov Decision Process (MDP) and subsequently solving the MDP-based problem with the TD3 algorithm.

(ii) To ensure that mobile devices have sufficient computation resources for the execution of their own applications, we propose a new learning architecture called EALA. This architecture handles all the computation-intensive operations of TD3 learning on edge server, thereby significantly reducing the computational burden on mobile devices

(iii) To assess the effectiveness of our proposed offloading model, we conduct extensive experiments, varying several parameters such as the task arrival rate of edge servers and the number of edge servers. The experimental results demonstrate the fast convergence of our offloading model. Additionally, MELO outperforms all baseline offloading approaches in terms of various metrics, including the completion time of deadline-sensitive tasks and the number of deadline misses.

1.4.3 Contributions of CRLO

Most existing RL-based offloading methods determine the appropriate policy during each time slot which means that system state is updated and thereafter an offloading decision is made at each time slot. This could lead to a significant amount of system overhead, which negatively affects the execution of local tasks. To tackle this issue, MTS offloading schemes have been proposed to determine offloading decisions for a set of tasks that will arrive during the upcoming MTS period [41,109]. We notice that most existing MTS offloading schemes, such as PDMO and MELO, tend to generate offloading policies based on the system state at the beginning of the MTS period or the historical statistics. Nevertheless, since the system state evolves over the MTS period, the RL agent is inherently unable to make the best offloading decisions. A few MTS schemes, such as [32], [84], and [109], have employed Recurrent Neural Network (RNN) or Long Short-Term Memory (LSTM) to forecast system states during the MTS period. However, RNN and LSTM are only effective for short-term prediction, such as the prediction of the system states over the upcoming 48 time slots or less. This implies that when the MTS period is longer than 48 time slots, the forecasting method may fail to accurately predict the system states [111]. Furthermore, if there are no effective safety measures to regulate the dangerous behaviors of RL agents,

directly applying RL-based offloading schemes to safety-critical applications, such as autonomous driving, could lead to highly risky offloading decisions, which potentially results in catastrophic consequences [43, 62]. Therefore, some safety measures should be incorporated into the offloading decision-making process to guarantee that the generated offloading policies are "safe". To tackle these limitations, we introduce a novel MEC offloading scheme for safety-critical applications, termed CRLO. This approach adopts the philosophy of the EALA structure, enables long-term system state prediction, and incorporates a safety layer into the decision-making module. Specifically, we introduce an innovative EALA-based multi-tier structure for MEC task offloading, which involves mobile devices, edge servers, a learning server, and cloud servers. In this structure, mobile devices are tasked with executing local tasks and inferring offloading decisions. Edge servers handle the processing of offloaded tasks. The learning server, situated at the network edge, is dedicated to training the learning model and predicting future system states. Note that, with this structure, training and inferring are separate. Namely, the resource-consuming training of the learning algorithm is done on the learning server while the trained model is used by mobile devices to make offloading decisions. The separation of training and inferring leaves as many resources on mobile devices as possible for local task processing. The cloud servers are used as the last resort when mobile devices and edge servers cannot handle the tasks to be processed.

CRLO, in particular, adopts a variant of TD3 featuring a novel policy network to conduct long-term system state prediction and generate secure offloading decisions [29]. Essentially, this particular variant of TD3 follows the actor-critic architecture. The policy network in the architecture is used to produce the appropriate action based on the current state. With CRLO, a long-term forecasting algorithm, Informer, is utilized to enable system state prediction for a time window that is longer than 48 time slots [111]. Then the predicted system states are used as the additional input for the policy network of TD3. Essentially, a long-term forecasting module is added to the input side of the policy network of TD3 in order to generate effective offloading policies. On the output side, a safety layer is added to regulate the policies generated by TD3. If a policy violates the safety constraints, the policy will be calibrated to

ensure only safe offloading operations are performed. Compared with the existing RL-based offloading methods based the actor-critic architecture [22, 51, 74], CRLO tends to result in more efficient and safer offloading decisions thanks to the long-term prediction module and safety layer. The main contributions of CRLO can be summarized as follows:

(i) We propose a novel safety-critical RL-based offloading scheme for MEC systems, CRLO. In CRLO, a novel policy network is devised by integrating the original policy network of TD3 with a long-sequence forecasting model and a safety layer. Through these enhancements, CRLO can generate effective offloading policies based on predicted long-term system states. In addition, with the safety layer, the number of unsafe offloading decisions is significantly reduced. To the best of our knowledge, CRLO is the first DRL-based offloading scheme for deadline-sensitive MEC systems, which takes policy safety into consideration.

(ii) We devise an innovative multi-layer offloading framework, in which resource-consuming learning modules of the MEC system are relocated to a resource-sufficient learning server. Different than the existing schemes where edge servers are responsible for both task processing and learning [32, 70, 89], this framework comprises two types of edge servers: computing edge server and learning edge server. The former processes tasks with strict deadlines offloaded from associated mobile devices, while the latter handles resource-consuming functionalities, such as training the RL model and completing transformer-based forecasting. This orchestration enables mobile devices and computing edge servers to fully focus on task processing, ultimately resuling in minimum deadline violations.

(iii) We conduct extensive experiments to study the performance of CRLO. Our experimental results indicate that CRLO converges to the optimal offloading policy after being trained for 500 epochs or less in the scenerios under investigation. In addition, CRLO outperforms the baseline offloading methods in terms of task completion time and the number of deadline misses.

1.5 Thesis Organization

The remainder of the thesis is organized as follows. In Chapter 2, we first present the preliminaries of our research work, including DVFS, deep reinforcement learning, and time-series forecasting model. Afterwards, we review the related work from two perspectives: non-learning-based offloading and learning-based offloading. In Chapter 3, the system models used in PDMO are described in Section 3.1, which is followed by the problem formulation of task offloading. In Section 3.2, we present the proposed learning algorithm, POTD3, and the associated offloading scheme, PDMO. Section 3.3 includes the detailed simulation results, and the major conclusions of PDMO are presented in Section 3.4. In Chapter 4, we provide the overview of our offloading MEC system in Section 4.1, followed by the description of three essential models of MELO. Afterwards, the offloading problem is formally formulated. In Section 4.2, we firstly elaborate on the decoupled learning architecture of EALA, and then present MELO in detail. At the end of this section, we analyze the complexity of the proposed offloading model. In Section 4.3, we first prove the convergence of MELO and then carry out the performance comparisons with other baseline methods. Lastly, we evaluate the scalability of MELO by using different number of edge servers. In Section 4.4, we present the major conclusions of MELO. In Chapter 5, we first provide an overview of the system models of CRLO in Section 5.1. In Section 5.2, we introduce two fundamental techniques employed in CRLO: long-sequence forecasting and constrained reinforcement learning. Subsequently, we introduce a novel policy network that integrates these techniques to overcome the limitations outlined in Section 1.4.3. The offloading problem is then reformulated as a CMDP and subsequently solved using CRLO. Section 5.3 presents comprehensive simulation results, while the conclusions of CRLO are summarized in Section 5.4. In Chapter 6, we present our conclusions and future work.

Chapter 2

Related Work

In this chapter, we give an overview of the existing studies that are related to our research. Specifically, we first discuss DVFS techniques. Thereafter, we present the fundamentals of deep reinforcement learning and time-series forecasting. Finally, we outline the existing task offloading schemes for MEC systems.

2.1 Dynamic Voltage and Frequency Scaling

The processors equipped in mobile devices have emerged as the primary energy-consuming module, their performance closely depends on power dissipation. When executing a large number of computation-intensive tasks simultaneously, the power consumption of processors can exceed 50% of the total battery power [3, 69]. Power consumption is broadly classified into two categories: dynamic power consumption and leakage power consumption. Dynamic power is consumed during instruction execution, while leakage power, caused by leakage current, is a growing concern due to the significant amount of power consumption generated constantly by active processor cores even during idle periods. In addition, two types of power management techniques have been developed to reduce dynamic and leakage power consumption: DVFS and Dynamic Power Management (DPM) [7]. DVFS is a technique that adjusts voltage and frequency of processors adaptively at runtime to mitigate overall power dissipation and heat generation in Integrated Circuits (IC). Conversely, DPM activates or shuts down processors dynamically according to system states while ensuring task deadlines [85]. Specifically, DVFS techniques is more suitable for lowering dynamic power consumption by dynamically scaling CPU frequency. On the other hand, DPM is primarily responsible for reducing leakage power consumption.

DVFS techniques have been successful in optimizing power consumption of deadline-sensitive systems, e.g., realtime operating systems. The techniques that reduce power consumption primarily depend on frequency adjustment between Actual Execution

Table 2.1: Task Set with Three Periodic DS Tasks.

Task ID	WCET	Period
1	2	8
2	2	10
3	3	12

Time (AET) and Worst-Case Execution Time (WCET). As WCET is typically longer than AET, DVFS techniques can slow down the operating frequency to save energy while still meeting all task deadlines. Pillai and Shin et al. presented two DVFS techniques: Cycle Conserving (CC) and Look Ahead (LA) [69]. The CC technique scales the clock frequency based on system usage while assuming worst-cases scenarios at the beginning. The frequency is then gradually slowed down based on the AET of tasks. In contrast, LA starts by running with a relatively low frequency and postpones most of tasks until the end, where they are executed with a high frequency to meet the task deadlines. In [6], another two DVFS techniques, Dynamic Reclaiming Algorithm (DRA) and Aggressive Speed Reduction (ASR), are investigated. The former adjusts operating frequency depending on a queue structure, while the latter is an extension of DRA.

Table 2.2: Two Sets of AET for Three Periodic DS Tasks.

Task ID	AET-1	AET-2
1	1	1
2	1	1
3	2	1

As the work [39], an example of algorithm CC is shown in Fig. 2.1. Correspondingly, a set of periodic deadline-sensitive tasks and the related two sets of AET are given in Table 2.1 and Table 2.1. We can observe that, after Task 1 completes its work with the highest frequency (2/8+2/10+3/12=0.7), the system utilization is recalculated based on the AET of Task 1 before determining the subsequent frequency of Task 2 (1/8+2/10+3/12=0.575). Iteratively, the frequency of Task 3 is calculated based on the AET of Task 1 and Task 2. Note that all three deadline-sensitive tasks can meet their deadlines strictly, even though the frequencies are slowed down for each

scheduling point. Thus, deadline-sensitive systems using the CC technique can dramatically reduce energy consumption by gradually lowering the operating frequency at runtime.

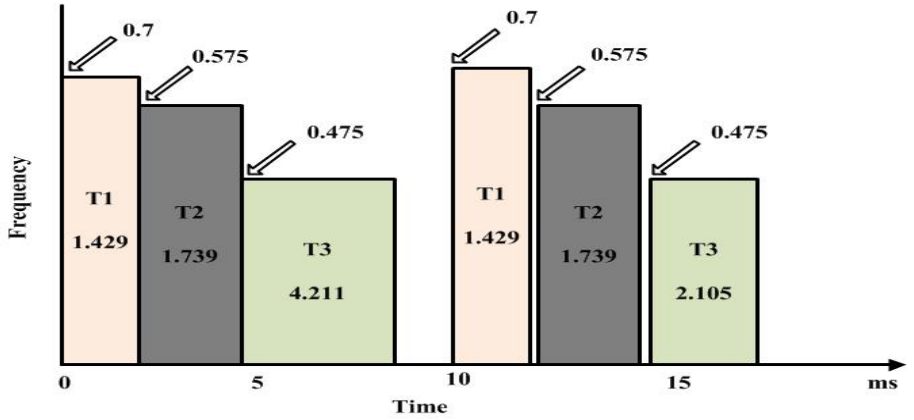


Figure 2.1: Cycle Conserving Algorithm.

2.2 Deep Reinforcement Learning

Deep reinforcement learning is an emerging subfield of machine learning that incorporates Deep Learning (DL) and Reinforcement Learning (RL) to train RL agents to learn from experience and generate optimal problem solution in complex environments [30,38,57]. In DRL, deep neural networks are used to approximate the policy of RL agents or the value function, allowing them to learn directly from raw sensory input data. By using deep neural networks, the RL agent can effectively learn complex features from high dimensional state space, such as images and videos. It makes DRL suitable for applications in many areas, such as online gaming and augmented reality, where complex decision-making is demanding. For instance, in [8] and [66], the researchers proposed a DRL-based approach to learn how to play Atari games relying on pixel image inputs. The approaches show that DRL methods outperform previous methods and achieve near-human-level performance on serveral games. Chen *et al.* proposed an interpretable DRL-based approach to address complex urban scenarios in end-to-en autonomous driving [13]. Gu *et al.* proposed a novel DRL-based algorithm for robotic manipulation in multiple robots scenarios. Liu *et al.* proposed a DRL

approach for clinical decision-making in healthcare systems [60]. Particualarly, the clinical decision-making problem is transformed into a Markov decision process and then solved with a trained DRL agent that learns the optimal treatment strategies relying on patient data.

The environment of standard DRL can be characterized as a four-tuple as: $< \mathcal{S}, \mathcal{A}, \Gamma, \mathcal{R} >$, where $\mathcal{S}$ and $\mathcal{A}$ denote state space $\mathcal{S} = \{s(t)|\forall t \in \mathbb{N}\}$ and action space $\mathcal{A} = \{a(t)|\forall t \in \mathbb{N}\}$. Γ denotes state transition function, $\Gamma = \{\Gamma : \mathcal{S} \times \mathcal{A} \times S \to [0, 1]\}$, which indicates the probability of the transition to the next state $s(t+1)$ when performing state-action pair $(s(t), a(t))$. $\mathcal{R}$ is the reward function $\mathcal{R} = \{\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}\}$. Specifically, the RL agent learns the optimal behaviors through trial and error and upgrades its policy network over time to maximize the total accumulated reward intentionally [83]. This means that, unlike previous work, expert knowledge is unnecessary for these intelligent and autonomous agents. In each learning iteration, the RL agent initially interacts with the environment and receives the current environmental state, $s(t)$. Based on the state $s(t)$ and policy network π, an action $a(t)$ is chosen to perform on the environment. Once this action is completed, two factors, the corresponding immediate reward $\mathcal{R}(t)$ and the status of the next state $s(t+1)$, are used for updating the parameterised mapping relationship of state-action pairs. The Fig. 2.2 shows the cycle of the DRL learning between the RL agent and its environment.

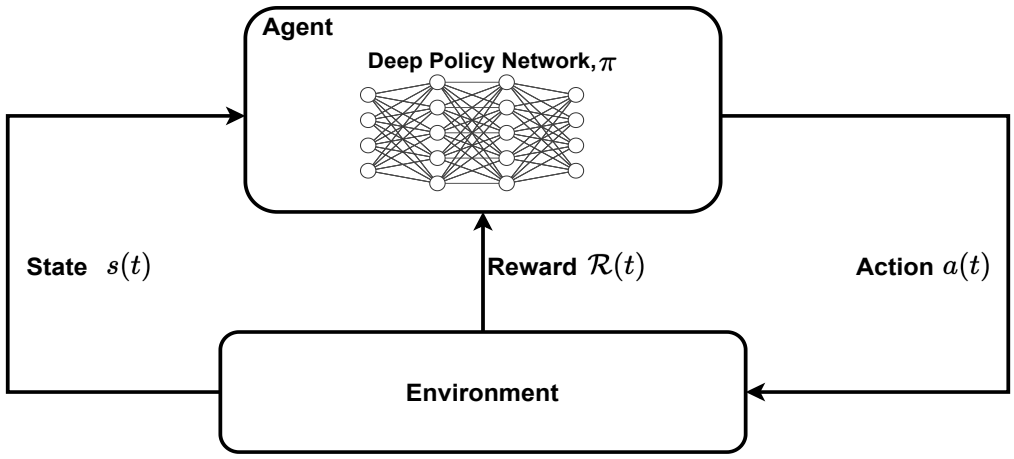


Figure 2.2: Deep Reinforcement Learning.

One representative DRL is DDPG that is proposed to solve the optimization

problems with large action in reinforcement learning tasks [57]. Specifically, DDPG combines the actor-critic approach with deep learning, making it suitable for environments with high-dimensional action spaces. The DDPG algorithm comprises an actor network and a critic network. The function of the actor network is to acquire a deterministic policy that aims to produce the optimal action corresponding to a provided state. Alternatively, the role of critic network is to evaluate the reward linked to state-action pairs. Through the assessment of actions taken across different states, the critic network provides feedback to the actor network. This feedback is then used for the refinement of the policy, resulting in a better convergence and more stable learning compared to pure policy-based DRL methods.However, due to the overestimation in DDPG, TD3 is proposed as an extension of DDPG to effectively mitigate the overestimation bias during the learning process by employing two critic networks [29]. Compared to other actor-critic based DRL algorithms, e.g., DDPG and A2C, TD3 has been shown to achieve superior performance on serveral benchmark environments [57,65]. Technically, TD3 is composed of one actor network π and two critic networks (Q^1 and Q^2). Correspondingly, three target networks, $\hat{\pi}$, $\hat{Q}^1$, and $\hat{Q}^2$, are involved to efficiently stabilize the training process. Additionally, to collect transition samples for the neural network trainings, a replay buffer $\mathcal{B}^{cr}$ is defined. In the training phase, six neural networks are firstly initialized with random parameters and then updated periodically until reaching the convergence. More specifically, based on an environment state $s(t)$, action $a(t)$ is selecting using the policy network π, $a(t) = \pi(s(t))$, which is then performed on the environment. Once the action $a(t)$ is completed, the RL agent captures the environmental feedback $\mathcal{R}(t)$, which is regarded as an indicator of the performance of action $a(t)$, and observes the next environmental state $s(t+1)$ simultaneously. Next, the transition sample of this one-time interaction, $(s(t), a(t), \mathcal{R}(t), s(t+1))$, is stored in the replay buffer $\mathcal{B}^{cr}$ for further model training. Essentially, the actor and critics have distinct updating patterns. Specifically, in each iteration, a mini-batch of transitions is randomly sampled from the replay buffer $\mathcal{B}^{cr}$ to update the critic networks Q^1 and Q^2. During every episode ϱ, the actor and three target networks are updated based on the critic networks that are trained during the current iteration.

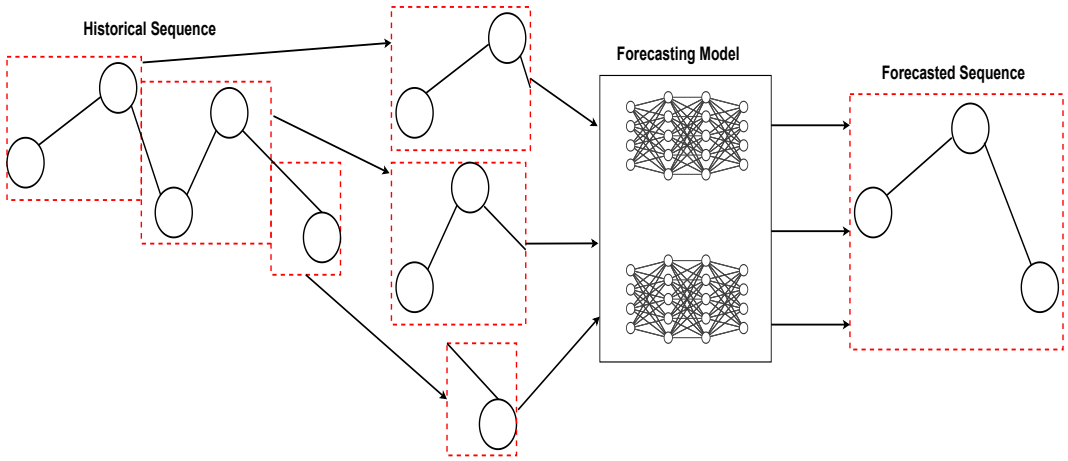


Figure 2.3: Time-series Forecasting.

2.3 Time-series Forecasting

A TSF model is a mathematical or computational framework used to predict future values of a time-series based on historical data. TSF models aim to capture patterns, trends, and seasonality in the data to make accurate predictions [78, 99]. TSF is a longstanding task with diverse applications, such as wireless network monitoring, smart grid management, and traffic jam control [19, 55, 111]. There exist three categories of TSF models. The first one is statistical-based TSF methods that rely on statistical techniques to capture patterns and make predictions. In contrast, machine learning-based TSF models, including but not limited to linear regression, decision trees, random forests, and gradient boosting machines, learn from historical data to formulate predictions and can uncover more complex and hidden relationships in the dataset. The last one is DL-based TSF models, such as RNN, LSTM, and Transformer, which excel at capturing temporal dependencies within the data and are particularly well-suited for sequential time-series data. Over the past several decades, TSF solutions have evolved from traditional statistical methods and machine learning techniques to DL-based solutions [50,58]. The mechanism of time-series forecasting is exhibited in Fig.2.3. In the time-series applications, the enormous amount of data from the historical data can be utilized to make prediction about the near future fluctuation. Nevertheless, most existing forecasting approaches are specifically

designed for the short-sequence scenarios, where the length of sequences is less than 48 points [52, 97]. Namely, the performance is significantly undermined in terms of the predictions on long sequences. Hence, the long-sequence forecasting models are introduced to extend the forecasting capacity [111]. The details of long-sequence forecasting model will be elaborated in Section 5.2.1.

2.4 Existing Task Offloading Schemes

A variety of different issues in MEC have been investigated over the past years [9, 10, 20, 42, 112]. The existing MEC-based offloading methods can be mainly categorized into two groups: non-learning-based computation offloading and learning-based computation offloading. Most former approaches achieve the optimal strategy by solving the optimization problem with a complex mathematical model, while the latter approaches learn the optimal strategy automatically either using a set of training data or interactions with the environment. The details of these two groups of offloading methods are systematically presented in the following subsections.

2.4.1 Non-learning-based Task Offloading

Non-learning-based task offloading typically employs predefined rules, heuristics, or straightforward algorithms to determine when and how tasks should be offloaded. These approaches, which do not involve learning mechanisms, are often straightforward to implement and may suffice for applications characterized by predictable workload patterns and stable environments. Many non-learning-based task offloading approaches have been considerably studied in the literature to optimize the performance of MEC systems.

Apostolopoulos *et al.* investigated a highly dynamic MEC environment that consists of ground and UAV-mounted MEC servers and proposed a novel data offloading framework to partially offload the data of end users [4]. Particularly, the framework considered risk-aware behavior of users and uncertainties of the computing environment. The offloading problem is formulated as a non-cooperative game that provably and uniquely exists a Pure Nash Equilibrium to maximize the user's satisfaction.

Chen *et al.* considered a green MEC scenario with multi-user multi-task, and developed a centralized and distributed greedy maximal scheduling algorithm in the environment for cooperating fog and cloud computing enviornment. The algorithm aims to effectively reduce task delay and energy consumption of user equipment [15]. Specifically, they formally formulated the multi-user multi-task computation problem, and leveraged a Lyapunov optimization algorithm to determine the near-optimal energy harvesting policy, e.g., the amount of energy being harvested for each mobile device, as well as the schedule of task offloading, e.g., the portion of tasks being processed on the mobile devices and edge servers. Du *et al.* considered the computation offloading problem in a mixed fog/cloud system and proposed a multi-tier offloading approach, which aims to jointly optimize task offloading decisions and the allocation of computation resources while ensuring user fairness and tolerable computation delay [25]. The offloading problem is formulated as a relaxation problem that is theorectically proved as a NP-hard problem. To solve the problem, a low-complexity algorithm, which incorporates semidefinite relaxation, randomization, fractional programming theory, and Lagrangian, was proposed to solve the NP-hard problem and then achieve a suboptimal solution. In [26], the authors aimed to develop a cost-driven computation offloading model for an edge-cloud environments that considers communication expense asymmetry of non-resident tasks. The offloading problem in the model is first proved to be an NP-hard problem and then an efficient algorithm was developed to solve it. Additionally, to handle a homogenous case when the two types of communication costs between pairwise interactive tasks are symmetric, an optimal offloading algorithm can be designed by converting the problem into a classical min-cut problem, which is evaluated by a PageRank-based application with a manipulated edge-cloud setting. Geng *et al.* considered the energy optimization on both the mobile devices and the remote server and proposed a multicore-based heuristic computation offloading approach that takes into account processing capabilities, energy consumption, and network bandwidth to determine when and how to offload multiple dependent tasks from mobile devices having a big.LITTLE structure to the resource-rich remote servers [33]. They solve the offloading problem with a proposed

heuristic algorithm that can obtain the best performance in terms of offloading decisions and task scheduling. Ji *et al*. aimed to relax the assumptions imposed on radio channels and network queue sizes in the MEC systems with various uncertainties, and proposed an energy-efficient computation offloading approach that considers the intrinsic uncertainties in network to minimize the worst-case expected energy consumption of a local device [45]. This approach developed an ϵ-bound approximation algorithm based on column generation to efficiently identify the optimal offloading decisions, and the algorithm is evaluated on an Android smartphone. Jošilo *et al*. considered the coordination of wireless mobile devices that generate computationally intensive tasks periodically, and proposed a game theorectical-based offloading model to determine when to perform tasks and whether or not to offload the tasks to an edge server through wireless link so that the overall cost is minimum [48]. Then, the existence of pure strategy Nash Equilibrium is proved by using a proposed polynomial-time complexity algorithm. In addition, an asymptotically tight bound on the approximation ratio of the proposed offloading model is specified, based on a given upper bound on the price of anarchy of the game. Zhang *et al*. jointly considered the computation and caching resources at the mobile edge servers, and proposed a novel offloading approach for handling latency-sensitive tasks to optimize multiple metrics, e.g., computation offloading, content caching, and resource allocation. The related offloading problem is formulated as a MINLP problem [104]. Then, the MINLP problem is solved by using an asymmetric search tree and branch and bound method to obtain a set of accurate offloading decisions and resource allocation policies. In [108], the authors studied the offloading problem in caching-restricted MEC systems for a set of dependent tasks to minimize the task completion time. In our research, an effective convex programming-based algorithm is first proposed to solve the hard problem, which cannot be tackled with existing algorithms with constant approximation. Then, a favorite successor based algorithm is devised to address a special case referred to a predefined parameter in the offloading problem. Naouri *et al*. proposed a three-layer task offloading approach, called DCC, comprising the device layer, cloudlet layer, and cloud layer [67]. In DCC, tasks with high computing requirements are offloaded to the cloudlet layer and cloud layer, while tasks with low

computing but high communication costs are executed on the device layer. This design minimizes data transmission to the cloud, effectively reducing processing delay. Specifically, a greedy task graph partition offloading algorithm, where task scheduling considers device computing capabilities using a greedy optimization approach to minimize communication costs, is devised. To demonstrate the effectiveness of the aprroach, a facial recognition system is implemented as a use case scenario. Furthermore, simulation results indicate effectiveness of DCC approach.

Despite the effectiveness of aforementioned non-learning-based schemes, they suffer to several limitations. For instance, the approaches need to consume massive computation resources to reach a satisfactory optimum, which is infeasible for most mobile systems. Secondly, these approaches struggle to cope with the highly dynamic nature of MEC systems and to promptly generate optimal offloading decisions. Therefore, these concerns have prompted RL agents to incorporate machine learning algorithms to learn the best offloading automatically to adapt to the variations of MEC systems.

2.4.2 Learning-based Task Offloading

Learning-based task offloading strategies leverage machine learning algorithms to dynamically determine task offloading decisions within MEC systems. These strategies involve learning from historical data, instantaneous observations, or system feedback to optimize the task allocation and offloading process.

To overcome the limiations of non-learning-based offloading approaches, researchers have increasingly explored machine learning techniques to address the challenges of task offloading in edge computing. Cao *et al*. studied multi-user distributed computation offloading to reduce the system-wide execution cost in a multi-user multi-channel cloudlet-based edge computing environment [11]. They proposed a game-theoretic machine learning based approach that formulates the distributed offloading problem as a noncooperative game for all users in the MEC system, and demonstrate the existence of pure-strategy Nash equilibrium point. This point could be obtained with the aid of a stochastic learning process. The proposed offloading algorithm was analyzed

theoretically, and simulation results exhibited that it is more effective than all baseline methods. Elgazzar *et al*. focused on resource-constrained mobile devices and proposed a cloud-assisted mobile service provisioning framework that allows for dynamic offloading of resource-intensive tasks to resourceful and reliable remote servers [27]. This approach significantly shortens the latency of computational tasks and reduces the total amount of transmission data, while also satisfying user-defined energy constraints. The offloading model is optimized using a learning-based decision-maker called Follow-Me-Provider, which generate the offloading policies based on a series of system states, such as the available resources of mobile systems and immediate network conditions. In [37], Hao *et al*. considered offloading computation-intensive, data-sensitive, and delay-sensitive intelligent tasks in a multi-user multi-server mobile edge computing environment. They developed a task offloading architecture that includes computation task cognitive layer, edge resoruce cognitive layer, and global management cognitive layer. The proposed cognitive DL-based task offloading approach optimizes the offloading policies under multi-user multi-edge scenarios to achieve a lower task duration and energy consumption compared to the baseline offloading methods. Additionally, the offloading approach specifies where to run the offloading scheme and how to run it. Kalantarian *et al*. aimed to jointly reduce system power consumption and improve battery life of lightweight wearable health-monitoring devices [49]. To achieve these goals, a novel dynamic computation offloading approach is proposed to alter the partitioning of data processing between health monitoring device and mobile application based on the desired classification accuracy. The experimental results demonstrate that the power usage can be significantly reduced by selecting appropriate offloading decisions based on current system parameters. Furthermore, the effectiveness of the offloading model is also evaluated in the deadline-sensitive monitoring systems in terms of energy optimization. Pradhan *et al*. explored computation offloading for Internet-of-Things (IoT) applications in the Multiple-Input-Multiple-Output (MIMO) Cloud-Radio-Access-Network (C-RAN) architecture [71]. Particularly, a low-complexity supervised DL-based offloading approach was proposed to promote the offloading decisions for computational tasks of

the IoT devices in MIMO C-RAN so that the transmit power consumption of the devices can be minimized while the latency requirements of the computation-intensive tasks are satisfied. Additionally, to enhance the adaptability of the offloading model to various MEC environments, they employed deep transfer learning to facilitate the learning process when turning into a new environment. Xu *et al*. proposed a DL-based aprroach (DeepWear) for wearable devices to enhance performance and reduce energy consumption [94]. DeepWear strategically offloads DL tasks from a wearable device to its paired handheld device through local network connectivity like Bluetooth. Unlike remote-cloud-based offloading, DeepWear operates without the need for Internet connectivity, consumes less energy, and ensures robust privacy protection. Furthermore, DeepWear incorporates several innovative techniques including context-aware offloading, strategic model partitioning, and pipelining support to effectively leverage the processing capacity of nearby paired handheld devices. Lastly, this approach has been implemented on the Android OS and evaluated on COTS smartphones and smartwatches using real DL models. The experiment results demonstrate that DeepWear achieves up to 5.08X and 23.0X execution speedup, as well as 53.5% and 85.5% energy savings compared to wearable-only and handheld-only strategies, respectively. However, aforementioned learning-based offloading approaches are supervised and require a large amount of manually-labeled data for the training of a robust decision model, which is generally infeasible in many real-world MEC systems.

Recently, instead of relying on the labeled data, considerable RL-based offloading methods that learn a automatic decision mode using the interactions between the RL agent and its external environment, have been widely studied. Zhan *et al*. proposed a decentralized DRL-based computation offloading approach to allow a set of mobile users, who makes offloading decisions independently, to fairly compete for limited resources in MEC systems so that overall energy consumption is minimized [100]. To address the multi-user offloading problem, the proposed approach utilizes game theory and formulates the problem as a partially-observable Markov decision process, which is then solved using a multi-agent policy gradient deep reinforcement learning. Unlike previous research work, this approach does not require the disclosure of privacy infromation of mobile users, such as netowrk connectivity and preferences.

The authors proposed an online DRL-based offloading approach to jointly optimize computation offloading decisions, Non-Orthogonal Multiple Access (NOMA) transmission and resource allocation while the total energy consumption of mobile devices is minimized [24, 72]. To handle time-varying channel conditions, the DRL offloading model learns the near-optimal offloading policies to adapt to network dynamics so that the objective of energy optimization is achieved. In the research [64], Min *et al*. proposed a Q-learning-based computation offloading approach for IoT devices that selects a suitable MEC server and offloading rate. Specifically, to facilitate the learning process, they applied a deep Q-network to estimate the Q-value of each action in Q-learning. In [53], Li *et al*. devised a Deep Q-Network (DQN) computation offloading model for a multi-user MEC system to mitigate both offloading delay and energy consumption for all users. Nevertheless, Q-learning-based offloading approaches explicitly struggle to address the offloading problems with high-dimensional and continuous state/action spaces. To address the issue, extensive actor-critic DRL-based offloading approaches have been proposed to tackle offloading problems with large action spaces. Wang *et al*. proposed a customized Proximal Policy Optimization (PPO) approach referred to as Hybrid-PPO, which is augmented by a parameterized discrete-continuous hybrid action space [90]. Utilizing Hybrid-PPO, a novel DRL-based multi-server multi-task collaborative partial task offloading strategy, aligned with meticulously constructed formal models, is developed. The experimental evaluation demonstrates superior offloading efficiency, surpassing current state-of-the-art schemes in terms of convergence rate, energy expenditure, time consumption, and adaptability across diverse network conditions. Qiu *et al*. proposed a novel online DRL-based offloading approach in a blockchain-empowered network that considered both mining tasks and data processing tasks [74]. To be specific, the online offloading problem is formulated as a Markov decision process to maximize the long-term offloading performance. Additionally, deep reinforcement learning is leveraged to solve the problem and reduce the computational complexity. Particularly, a novel DDPG algorithm that integrates DDPG method with an adaptive genetic algorithm is designed to learn an intelligent offloading scheduler for the tasks with different characteristics.

Tan *et al.* formulated the multi-user offloading problem as a distributed decision-making problem and proposed a multi-agent decebtralized computation offloading approach to incentivize agents to reconcile the private and global objectives by finding a balance between competition and cooperation with respect to multiple metrics of MEC system, such as offloading failure rate, communication overhead, and energy consumption [82]. The mechanism is shown to have Nash equilibrium with optimal resource allocation in the static scenario. To handle dynamic scenarios, a novel multi-agent online learning-based algorithm was proposed to learn the partial, delayed and inaccurate state information, and a reward signal that reduces the need for detailed information. Ren *et al.* introduces a fast environment-adaptive DRL-based offloading approach, FEAT, which is proposed to adapt to unseen environments with minimal fine-tuning [75]. The approach involves splitting MEC states into internal and environmental states Subsequently, two main components of FEAT are developed: a set of internal state-dependent meta-policies and an environmental state-embedded steerer. The meta-policies learn skills within the internal state space, allowing for their reuse in different environments, while the steerer learns to select appropriate meta-policies based on embedded environmental states. When encountering an unseen environment with the same internal state space, FEAT requires only minimal fine-tuning of the steerer using the newly embedded environmental state, with few internal state explorations. In the context of edge-enabled Internet of Things (IoT), Lu *et al.* proposed an improved DDPG-based computation offloading approach to enhance Quality of Experience (QoE), such as service latency, energy consumption, and task success rate [63]. Due to the sensitivity of the critic network of the DDPG algorithm, the learning model integrates a Double-Q learning and Dueling Network, called Double-Dueling-Deterministic Policy Gradients (D3PG), with the original DDPG to accelerate and stabilize the learning phase. The simulation results demonstrate the superior stability and fast convergence of the proposed offloading approach compared to existing methods.

Nevertheless, these approaches may not effectively process periodic deadline-sensitive tasks due to the utilization of STS observation as inputs of policy network, resulting in substantial suboptimal offloading decisions. Furthermore, they lack effective

measures to ensure safety during the learning process, consequently leading to considerable suboptimal offloading policies and deadline misses. To tackle these issues, many safe DRL-based offloading approaches have been proposed. Gao *et al.* proposed an Attention-weighted Recurrent Actor-Critic (ARMAAC) based decentralized computation offloading scheme for large-scale mixed cooperative -competitive MEC systems to reduce service latency for mobile users [32]. Particularly, the proposed approach consists of three main components. Firstly, a recurrent actor-critic framework is used to assist mobile device agents in memorizing historical information to better understand future states. Secondly, an attention mechanism is intorduced to capture critical features the observations of mobile devices. To meet the constraints of offloading problem, ARMAAC leverages reward shaping to penalize the unsafe offloading policies. Finally, the mdoel takes into consideration the stable and convergence difficulties that arise due to the sensitivity correlation between the actor and critic networks. Tang *et al.* considered non-divisible, delay-aware tasks, and workload dynamics of edge servers, and formulate the task offloading problem to minimize the processing delay and dropped rate of tasks [84]. Specifically, the proposed approach involves a model-free DRL-based distributed algorithm, allowing each device to determine its offloading decision independently, without knowledge of task models or the decisions of other devices. To enhance the estimation of the long-term cost, the techniques such as LSTM, dueling DQN, and double-DQN into the algorithm, are technically integrated. Furthermore, reward shaping is utilized to guarantee the constraints of the offloading problem. Simulation results demonstrate that the proposed algorithm effectively leverages the processing capacities of edge nodes, leading to a significant reduction in the ratio of dropped tasks and average delay compared to several existing algorithms. Gao *et al.* proposed a novel offloading strategy for MEC systems, termed Com-DDPG, which leverages multiagent reinforcement learning to improve offloading performance [31]. Within the transmission radius of the Internet of Vehicles (IoV), multiple agents collaborate to learn environmental changes, such as numebr of mobile devices and task queues, and determine appropriate offloading policies. Specifically, the authors explore task dependency, priority, and resource consumption from the perspective of server clusters and multiple task dependencies.

Subsequently, the proposed method formulates communication behavior among multiple agents, with the policy determined through reinforcement learning executed as an offloading policy to produce corresponding outcomes. Furthermore, to enhance information exchange among agents, a LSTM network is employed as an internal state predictor to provide a comprehensive system state. Lastly, a Bidirectional Recurrent Neural Network (BRNN) is utilized to learn and improve features obtained from the communication of agents. In [41], Huang *et al.* introduced a 6G-empowered DRL-based offloading scheme named MELO, designed to facilitate appropriate offloading decisions for periodic deadline-sensitive tasks. The offloading problem is specifically formulated as a Markov Decision Process, followed by the utilization of a DRL algorithm, TD3, to address the problem. Furthermore, this approach integrates 6G as the communication infrastructure to adequately support data transmission between mobile devices and edge servers. Aimed at optimizing resource allocation on mobile devices, a novel learning architecture, EALA, is proposed. With EALA, the training and inference operations of the learning algorithm are decoupled, with training executed on edge servers and inference performed on mobile devices. To mitigate deadline violations during the learning phase, reward shaping is implemented to punish the offloading policies that result in deadline misses.

Chapter 3

Energy Consumption Minimization with DRL-based Task Offloading

In this chapter, we present the details of PDMO, a novel task offloading scheme for MEC that aims to minimize energy consumption. Specifically, the system model for PDMO is first discussed. Afterwards, the components of PDMO, including the proposed learning algorithm called POTD3, are descirbed. Finally, the performance of PDMO is compared with that of serveral baseline schemes.

3.1 System Model

In this section, we present three system models adopted in PDMO: scheduling model, job completion time model, and energy consumption model. The scheduling model is responsible for determining the processing priorities of local tasks and the data transmission priorities of offloaded tasks in mobile transmitter. The job completion time model is utilized to calculate the completion time of tasks being processed on both mobile device and remote servers. The energy consumption model is employed to measure the overall energy consumption of both local and offloaded tasks when transmitting task data. The details of the problem formulation are also described in this section. A list of the key notations uniquely used in Chapter 3 are provided in Table 3.1.

3.1.1 Overview

In real-world scenarios, many mobile applications such as autonomous driving, involve both periodic Deadline-Sensitive (DS) tasks and aperiodic Non-Deadline-Sensitive (NDS) tasks [106]. In our research, we consider a set of periodic DS tasks $\mathcal{T} = \{\tau_1, \tau_2, \ldots\}$ and a set of aperiodic NDS tasks $\Phi = \{\varphi_1, \varphi_2, \ldots\}$ on mobile devices. In general, each DS task τ_n (note that $\tau_n \in \mathcal{T}$) is represented using a 4-tuple: $\tau_n(p_n, w_n, d_n, l_n)$. Since task τ_n is periodic, it is repeated once in a while. We use p_n to denote the period of task τ_n. It is noteworthy that the task τ_n is executed only once in per

Table 3.1: Key Notations in Chapter 3

Notation	Description
$\mathcal{T}, \Phi$	DS task set and NDS task set
τ_n, φ_m	DS task n, $\tau_n \in \mathcal{T}$ and NDS task (job) m, $\varphi_m \in \Phi$
τ_n^i	i-th job of τ_n in a hyperperiod
LQ, TQ	The local and transfer queue for ready jobs
$\mathcal{H}_p$	Length of hyperperiod w.r.t set $\mathcal{T}$
Q_{nr}	NDS task queue in a hyperperiod
x	Total number of jobs in a hyperperiod
$\mathcal{J}_{l,r}, \mathcal{J}_{l,nr}$	DS and NDS job set for local computing
$\mathcal{J}_{e,r}, \mathcal{J}_{e,nr}$	DS and NDS job set for edge computing
$\mathcal{J}_{c,r}, \mathcal{J}_{c,nr}$	DS and NDS job set for cloud computing
$\hbar_u$	Schedulability test result for $\mathcal{J}_{l,r}$
$\mathcal{U}_s$	System utilization of local devices
R_e, R_c	Transmission rates of edge and cloud network
ω_e, ω_c	Speedup variables of edge and cloud server
ζ_e, ζ_c	Instantaneous workload of edge and cloud server
$\mathcal{O}$	Set of offloading profile
Λ_t	Offloading profile at t-th hyperperiod, $\Lambda_t \in \mathcal{O}$
$\mathcal{P}_{trans}$	Transmission power of a mobile device
ϑ	Average job arrival rate at edge server
t_s	Scheduling points in a hyperperiod
T_d	Length of idle time at scheduling point
f_l, a_l, d_l	Frequency, AET and data size of a local job
d_e, d_c	Data size of a job being processed at edge and cloud
$f_n^{i,}, f_m$	Frequencies of $\tau_n^i \in \mathcal{J}_{l,r}$ and $\varphi_m \in \mathcal{J}_{l,nr}$
F,f_k	Set of frequencies and the maximum frequency in F

period. The second element of the 4-tuple, w_n, denotes the WCET of τ_n, which is a time window required to complete a task in the worst case when CPU runs at the highest frequency level [37]. In deadline-sensitive systems, a task often requires less execution time than its WCET. This leads to the term of AET, which is defined as the specific execution time of a task in a period. In this chapter, we use a_n to denote the AET of task τ_n. The third element of the tuple, d_n, represents the volume of the data that need to be transferred between the local device and the remote server when the task needs to be offloaded. If a DS task τ_n cannot be completed within its period p_n, it is considered to miss its deadline. The last element of the tuple, l_n, is a status variable that indicates whether the task τ_n misses its deadline. In our research, a 3-tuple $\varphi_m = (w_m, d_m, k_m)$ is used to denote an NDS task. Note that w_m and d_m are similar to w_n and d_n; k_m is a status variable that indicates whether an NDS task is finished with a hyperperiod (note that the definition of hyperperiod will be presented below). In the scheduling phase, DS tasks are assigned with a higher priority than NDS tasks.

In our research, we assume that each periodic DS task τ_n arrives at the beginning of its period. To effectively schedule periodic DS tasks, Islam *et al.* proposed a scheduling method based on the concept of hyperperiod [43]. The adoption of hyperperiod provides an effective method to calculate the CPU utilization of a mobile device, which can be used to scale CPU frequency in order to reduce energy consumption of the mobile device. In our research, we adopt a scheduling method based on the concept of hyperperiod. Technically, the hyperperiod of a set of periodic tasks, $\mathcal{H}_p$, is defined as a time window within which every DS task in the task set could be executed at least once. Formally, $\mathcal{H}_p$ can be defined using Eq. (3.1):

$$\mathcal{H}_p = LCM(p_1, p_2, ...), \tag{3.1}$$

where LCM is a function that calculates the least common multiple of a series of numbers. Mathematically, $\mathcal{H}_p$ is the least common multiple of all task periods regarding the task set $\mathcal{T}$. Within each hyperperiod, a DS task τ_n is executed $\mathcal{H}_p/p_n$ times. Note that if a DS task rarely arrives (i.e. the period of a DS task is significantly longer than that of other DS tasks), the resulting hyperperiod could be overly long.

With PDMO, one learning cycle corresponds to one hyperperiod. At the end of each learning cycle, learning reward is generated and the offloading policy is thereafter adjusted. This process continues untill PDMO converges. If the hyperperiod is too long, PDMO will need a lengthy period to converge. In the research, we consider MEC systems in which the periods of DS tasks are in the range of 4 to 32 timeslots (unless specified otherwise). For clarity, each execution of task τ_n in a hyperperiod is called a job corresponding to the task τ_n. Additionally, we use τ_n^i to denote the i-th job of τ_n in a hyperperiod. Correspondingly, w_n^i, d_n^i, and l_n^i represent WCET, AET, and the deadline-miss status variable of job τ_n^i. Note that $i \leq \mathcal{H}_p/p_n$.

Unlike DS tasks, NDS tasks have a flexible arrival pattern. Since each aperiodic NDS task is executed only once, each NDS task corresponds to one NDS job. Therefore, we can use φ_m to denote both an NDS task and the job corresponding to the task. In the rest of this chapter, unless stated otherwise, we use τ_n^i and φ_m to denote a DS job and an NDS job, respectively. Noticeably, each hyperperiod involves both DS and NDS jobs. The total number of jobs in a hyperperiod, denoted as x, can be calculated using Eq. (3.2):

$$x = \sum_{n=1}^{|\mathcal{T}|} \mathcal{H}_p/p_n + |Q_{nr}|, \tag{3.2}$$

where Q_{nr} denotes the queue for the arrived NDS tasks during a hyperperiod and $|Q_{nr}|$ represents the number of NDS tasks in the queue.

In this research, we consider a 3-tier offloading system that involves mobile device, edge server, and cloud serve. Fig. 3.1 shows the architecture of the system under investigation. With this system, a job is either processed locally on a mobile device or offloaded to an edge server (or a cloud server if the edge server associated with the mobile device is overloaded). Technically, the proposed offloading scheme, PDMO, is invoked on a mobile device at the beginning of each hyperperiod to determine where the jobs of subsequent hyperperiod should be processed. Since DS jobs are periodic, PDMO is aware of the DS jobs that will arrive in the upcoming hyperperiod. However, for NDS jobs, PDMO only knows the NDS jobs that have arrived at a mobile device. Consequently, PDMO only makes an arrangement for the DS jobs that need to be processed in the upcoming hyperperiod and the NDS jobs that have arrived in the

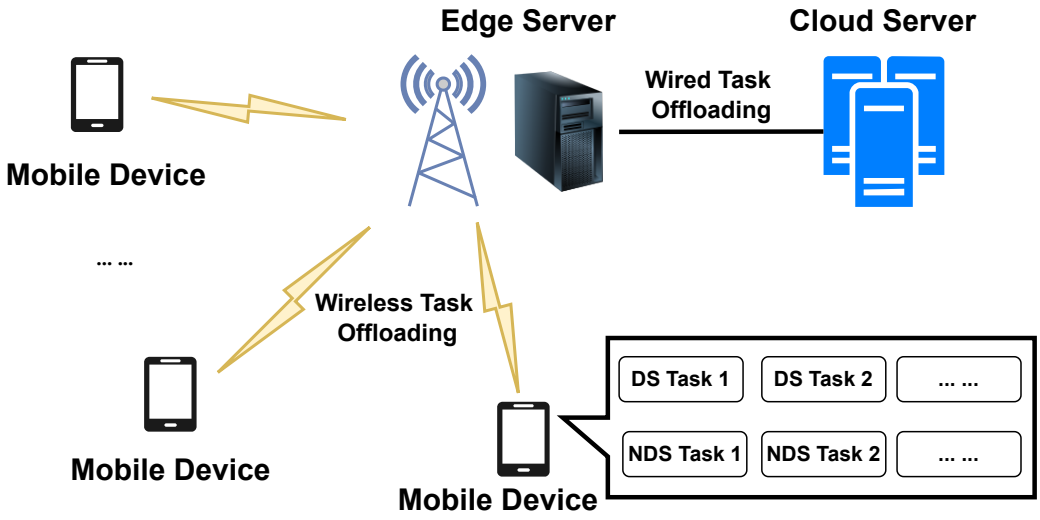


Figure 3.1: Architecture of the MEC System under Investigation for PDMO

past. At the beginning of a hyperperiod, PDMO classifies these jobs into six job subsets: $\mathcal{J}_{l,r}$, $\mathcal{J}_{e,r}$, $\mathcal{J}_{c,r}$, $\mathcal{J}_{l,nr}$, $\mathcal{J}_{e,nr}$ and $\mathcal{J}_{c,nr}$. Note that $\mathcal{J}_{l,r}$, $\mathcal{J}_{e,r}$ and $\mathcal{J}_{c,r}$ denote the DS job subset for the local mobile device, the edge server and the cloud server, respectively. $\mathcal{J}_{l,nr}$, $\mathcal{J}_{e,nr}$ and $\mathcal{J}_{c,nr}$ represent the NDS job subset for the local mobile device, the edge server and the cloud server, respectively.

3.1.2 Scheduling Model

For scheduling purposes, the proposed scheme maintains two priority queues, LQ and TQ, to store the jobs processed locally and those being transferred to an edge/cloud server. Each queue consists of two components, DS and NDS component, which include DS and NDS jobs respectively. The DS component is always ahead of the NDS component so that DS jobs are first processed. Within the DS component, jobs are sorted based on the Earliest-Deadline-First (EDF) algorithm [79]. In contrast, the jobs in the NDS component are sorted on a First-In-First-Out (FIFO) basis. Note that an NDS job is preempted in the scenario where a DS job arrives during the execution possession of the NDS job.

As aforementioned, at the beginning of a hyperperiod, every job is assigned to one

of the subsets. Once subset $\mathcal{J}_{l,r}$ is available, the system utilization of local device, $\mathcal{U}_s$, can be calculated using the AET and WCET information of the jobs in $\mathcal{J}_{l,r}$. Note that, at the beginning of a hyperperiod, only the WCET information is available. Each time a DS job is completed, its AET becomes available and the system utilization $\mathcal{U}_s$ is updated using Eq. (3.3):

$$\mathcal{U}_s = (\sum_{\tau_n^j \in prec(\tau_n^i)} a_n^j + \sum_{\tau_n^j \notin prec(\tau_n^i)} w_n^j)/\mathcal{H}_p, \tag{3.3}$$

where τ_n^i is the next DS job to be completed; $prec(\tau_n^i)$ denotes a job set that includes the DS jobs that have been completed before job τ_n^i in a hyperperiod; a_n^j and w_n^j denote the AET and WCET of task τ_n^j. At the beginning of a hyperperiod, $\mathcal{U}_s$ is also obtained using Eq. (3.3). However, since $prec(\tau_n^i)$ is an empty set at the time, only the WCET of the jobs in the subset $\mathcal{J}_{l,r}$ is used to calculate the system utilization $\mathcal{U}_s$. To simplify the calculation of $\mathcal{U}_s$, the AET a_n^j in Eq. (3.3) denotes the time interval used to complete the job τ_n^j when CPU runs at the highest frequency level. In practice, CPU frequency can be scaled down adaptively to reduce energy consumption. When CPU frequency is lowered to run an DS job τ_n^i, the execution time of the job is a_n^i/f_l, where the time interval used to complete the job τ_n^i when CPU runs at the highest frequency level; f_l is the ratio of the lowered CPU frequency to the highest CPU frequency. Whenever the system utilization $\mathcal{U}_s$ is less than one, the local CPU is not fully utilized and the available slack could be used to lower CPU frequency. At the beginning of a hyperperiod, no DS job has been executed and therefore only static slack exists. After jobs start to be executed locally, dynamic slack is potentially reclaimed at runtime due to the difference between AET and WCET of each job. Both static and dynamic slack can be used by PDMO to lower CPU frequency for energy minimization.

To achieve this goal, the runtime CPU frequency is lowered adaptively according to the total amount of static and dynamic slack. Note that blindly lowering CPU frequency could be harmful to DS jobs because the lowered frequency will incur longer execution time, inevitably resulting in more deadline misses. To guarantee that the deadlines of the DS jobs in the subset $\mathcal{J}_{l,r}$ are met, the schedulability test and DVFS technique are used in the proposed scheme as previous work [5,80]. The schedulability

test leads to a positive result if the system utilization $\mathcal{U}_s$ is less than one. Formally, we use $\hbar_u$ to denote the schedulability of subset $\mathcal{J}_{l,r}$. Note that $\hbar_u$ can be calculated using Eq. (3.4):

$$\hbar_u = \begin{cases} 1, & if \quad \mathcal{U}_s \ \leq \ 1, \\ 0, & otherwise. \end{cases} \tag{3.4}$$

Once the schedulability test is passed (i.e. $\hbar_u = 1$), the deadlines of DS jobs can be met rigorously. Otherwise, the local mobile device cannot complete all the DS jobs assigned to it on time.

The DVFS technique can be leveraged to lower CPU frequency according to the slack in the local mobile device while ensuring the deadelines are met. In our research, we use $\mathcal{F}$ to denote the set of frequency levels available on the mobile device, where $\mathcal{F} = (f_1, f_2, ...f_k)$, $f_1 < f_2 ... < f_k$. For simplicity, we assume that f_n is the ratio of a specific CPU frequency to the highest CPU frequency. For example, if the highest frequency is 1 GHz and f_1 corresponds to the lowest possible frequency on the mobile device (e.g. 0.6 GHz), then f_1 is equal to 0.6. Since f_k corresponds to the highest frequency, f_k is equal to 1. In the scenario that $\mathcal{U}_s \leq 1$ and the schedulability test is passed, the DVFS technique could be used to lower the CPU frequency. With PDMO, the CPU frequency for the local mobile device, f_l, should be set to $\mathcal{U}_s \cdot f_k$. However, since mobile devices typically provide a set of discrete frequency levels, f_l is set to an element in $\mathcal{F}$, which is equal to or slightly greater than $\mathcal{U}_s \cdot f_k$. In our research, we use $\lceil \mathcal{U}_s \cdot f_k \rceil$ to denote the value assigned to f_l.

With PDMO, if a DS job τ_n^i runs at the local mobile device, its deadline can be surely met because of the employment of schedulability test. However, an offloaded DS job may miss its deadline due to the unpredictable network delay and queuing delay at the edge/cloud server. At the end of each hyperperiod, the deadline miss information is returned by PDMO to learn the system and thereafter make better offloading decisions. Since DS jobs are periodic, there might be some idle CPU time between DS jobs. With PDMO, the free time is utilized to schedule NDS jobs. The details of the job scheduling process at the beginning of a hyperperiod is outlined in Algorithm 1. Specifically, this algorithm involves the following three modules:

(i) PDMO first calculates the length of the hyperperiod $\mathcal{H}_p$ according to Eq. (3.1). Thereafter, at the beginning of a hyperperiod, every job is assigned to one of six subsets. Finally, LocalScheduling is used to schedule local jobs and TransScheduling is employed to schedule the jobs to be offloaded to edge/cloud servers.

(ii) With LocalScheduling, the system utilization $\mathcal{U}_s$ is first calculated based on subset $\mathcal{J}_{l,r}$. If the schedulability test is passed (i.e. $\mathcal{U}_s < 1$), the first job in the queue LQ is allowed to execute at the decision point t_s with a modified frequency $f_l = \lceil \mathcal{U}_s \cdot f_k \rceil$, where t_s denotes a critical time when either a job is completed or a new job arrives. To avoid too many frequency adjustments, NDS jobs would not be scheduled if the length of idle time T_d is shorter than a predefined threshold T^*. Generally, the length of T^* should be based on the Context Switch Time (CST), which is the time used to replace the currently-running process with another process on a mobile device. For example, T^* could be set to a value that is equal to a multiple of CST, such as 10 * CST. Since DS jobs have higher priority than NDS jobs, the NDS job $\varphi_m \in \mathcal{J}_{l,nr}$ is permitted to be executed only if all DS jobs in LQ are completed.

(iii) With TransScheduling, the DS and NDS jobs in TQ are sorted at each decision point. After TQ is sorted, the first job in TQ is offloaded to an edge server or a cloud server. Note that DS jobs should always be ahead of NDS jobs in TQ. Namely, an NDS job is preempted in the scenario where a DS job arrives when the NDS job is transmitting its data.

Fig. 3.2 includes an example that illustrates how DS/NDS jobs are scheduled simultaneously. In this example, there are four DS tasks and three NDS tasks. The details of these tasks are outlined in Table 3.2. Specifically, we assume that the frequency levels range from 0 to 1.0, with a step increase being 0.1. The transmission rate of the local device is set to 10 Mbps. The threshold T^* is set to 0.5 seconds. The hyperperiod $\mathcal{H}_p$ is equal to $LCM(4, 6, 12, 6) = 12$ seconds and the total number of jobs x is equal to $12/4 + 12/6 + 12/12 + 12/6 + 3 = 11$. The six job subsets are

Algorithm 1: Hybrid Job Scheduling with DVFS.

Input: DS and NDS task sets: $\mathcal{T}$, Φ
Output: operating frequency f_l, scheduling sequences $\mathcal{S}_l$ and $\mathcal{S}_t$

```
Function MainScheduling;
Calculate H_p using Eq. (3.1);
Allocate jobs to subsets: J_{l,r}, J_{e,r},J_{c,r}, J_{l,nr}, J_{e,nr}, J_{c,nr};
LocalScheduling(J_{l,r},J_{l,nr});
TransScheduling(J_{e,r},J_{c,r},J_{e,nr},J_{c,nr});
Function LocalScheduling(J_{l,r},J_{l,nr});
Update system utilization U_s based on AETs of τ_n^i ∈ J_{l,r};
for t = 1 to H_p do
    if t = t_s then
        Sort jobs in queue LQ;
        Pop the head job in LQ, τ̂ ← τ_n^i = LQ.head ;
        if τ̂ ∈ J_{l,r} then
            Update U_s using Eq. (3.3);
            Execute τ̂ with frequency f_l ← ⌈U_s · f_k⌉ ;
        end
        else
            while LQ ≠ ∅ and T_d ≥ T* do
                τ̂ ← φ_m = LQ.head;
                Execute τ̂ with the lowest frequency frequency f_1;
            end
        end
    end
end
Function TransScheduling(J_{e,r},J_{c,r},J_{e,nr},J_{c,nr});
for t = 1 to H_p do
    if t = t_s then
        Sort jobs in queue TQ;
        Pop the first job in queue TQ, τ̂ ← LQ.head ;
        Conduct data transmission of job τ̂ with rate R_e;
    end
end
```

$\mathcal{J}_{l,r} = \{\tau_1^1, \tau_1^3, \tau_2^1, \tau_3^1\}$, $\mathcal{J}_{l,nr} = \{\varphi_1, \varphi_2\}$, $\mathcal{J}_{e,r} = \{\tau_1^2, \tau_4^1\}$, $\mathcal{J}_{e,nr} = \{\varphi_3\}$, $\mathcal{J}_{c,r} = \{\tau_2^2, \tau_4^2\}$ and $\mathcal{J}_{c,nr} = \{\emptyset\}$. The AET of these jobs are shown in Table 3.3.

With LocalScheduling, the system utilization $\mathcal{U}_s$ is first calculated according to subset $\mathcal{J}_{l,r}$: $\mathcal{U}_s = 1/12+1/12+1/12+2/12 \approx 0.42$. Then, for the job τ_1^1, the frequency

Table 3.2: Details of Tasks

Task No.	Period(s)	WCET (s)	Data(Mb)
τ_1	4.0	1.0	5.0
τ_2	6.0	1.0	20.0
τ_3	12.0	2.0	15.0
τ_4	6.0	2.0	10.0
φ_1	N/A	4.0	15.0
φ_2	N/A	1.0	5.0
φ_3	N/A	1.0	30.0

is set to $\lceil 0.42 \cdot 1 \rceil = 0.5$. At the first decision point $t_s = 0$, $LQ = (\tau_1^1, \tau_2^1, \tau_3^1, \varphi_1, \varphi_2)$. The head job τ_1^1 in the queue LQ is first scheduled. Since the AET of job τ_1^1 is 0.5 seconds, the execution time of job τ_1^1 is calculated, $T_1^1 = 0.5/0.5 = 1.0$ second. Before scheduling job τ_2^1, the system utilization $\mathcal{U}_s$ is recalculated based on the AET of job τ_1^1: $\mathcal{U}_s = 0.5/12 + 1/12 + 1/12 + 2/12 \approx 0.38$. For the job τ_2^1, the CPU frequency is set to $\lceil 0.38 \cdot 1 \rceil = 0.4$. Similarly, the job τ_1^3 runs with a frequency of 0.4 for 3.75 seconds. At the decision point $t_s = 7.25$, the first NDS job φ_1 in the queue LQ is allowed to execute at the lowest frequency 0.1 for 0.75 seconds because the idle time interval $T_d = 0.75$ is greater than the threshold $T^* = 0.5$. At the decision point $t_s = 8.0$, preemption takes place. The NDS job φ_1 is suspended and preempted by the DS job τ_1^3. After the system utilization $\mathcal{U}_s$ is recalculated according to the AET of job τ_1^1, job τ_2^1 and job τ_3^1, it is adjusted to 0.4. Therefore, the DS job τ_1^3 is executed at the frequency 0.4. The scheduling process continues until the end of the hyperperiod.

With TransScheduling, the queue LQ is sorted at each scheduling point. At time slot 0, the priority queue TQ contains one DS job (i.e. the job τ_4^1) and one NDS job (i.e. the job φ_3). The DS job τ_4^1 that has highest priority is first to transmit its data. After the job τ_4^1 finishes data transmission, the NDS job φ_3 is scheduled to carry out data transmission. At the decision point $t_s = 4.0$, the DS job τ_1^2 arrives and thereafter starts uploading its data. The DS job τ_2^2 and τ_4^2 are scheduled in the same fashion.

Table 3.3: AET of Jobs.

Job No.	τ_1^1	τ_1^2	τ_1^3	τ_2^1	τ_2^2	τ_3^1
AET (s)	0.5	edge	0.5	1.0	cloud	1.5
Job No.	τ_4^1	τ_4^2	φ_1	φ_2	φ_3	
AET (s)	edge	cloud	2.0	0.5	edge	

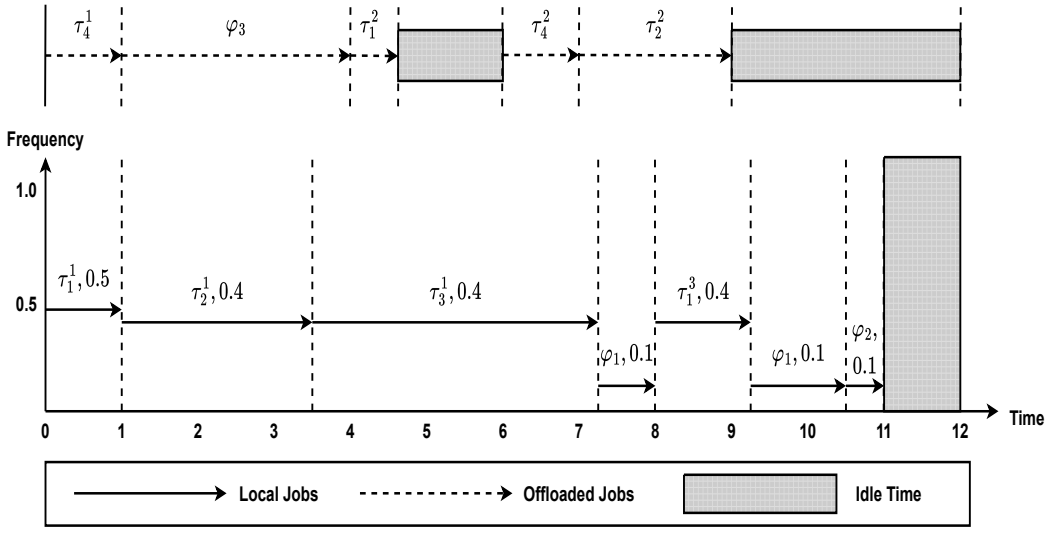


Figure 3.2: Hybrid Job Scheduling Example.

3.1.3 Job Completion Time Model

As shown in Fig. 3.1, the MEC system under investigation involves mobile device, edge server, and cloud server. To be specific, mobile device communicates with edge server via a wireless connection. In contrast, the communication between edge and cloud server is carried out via a wired link. The reason to consider a 3-tier system in our research is that edge server typically have limited computation resources, which might not be sufficient to accommodate the jobs offloaded from mobile device. To relieve the burden on edge server, a portion of the jobs can be offloaded to a cloud server that is assumed to have more computation resources. In the model, we use a vector $\Lambda = (\lambda_1^1, ...\lambda_n^i, ..., \lambda_1, ..\lambda_m, ...)$ to denote the offloading policy for both DS and NDS jobs in a hyperperiod. Note that λ_n^i and λ_m denote the offloading decision for the DS job τ_n^i and NDS job φ_m, respectively. Given an offloading policy, if the offloading decision λ_n^i or λ_m is equal to 0, the job τ_n^i or φ_m is processed on the local mobile device. In contrast, if the offloading λ_n^i or λ_m is equal to 1, then, the job τ_n^i or φ_m is offloaded to the edge server. If the job λ_n^i or λ_m is equal to 2, then, the job τ_n^i or φ_m will be offloaded to the cloud server. The number of elements in the offloading policy Λ is equal to the sum of the number of DS jobs and the number of NDS jobs in queue $\mathcal{Q}_{nr}$. In our research, we use $\mathcal{O}$ to denote the set of all possible offloading policies. Obviously, $\Lambda \in \mathcal{O}$.

Furthermore, we use $\mathcal{P}_{trans}$ to denote the transmission power of the mobile device when a job is offloaded to the edge server. In addition, we use c_g to represent the channel gain between the mobile device and edge servers. Finally, σ^2 is the receiver noise of the mobile device that can be modeled with a circularly symmetric complex Gaussian distribution [54]. With these definitions, the upload rate of the mobile device can be calculated using Eq. (3.5):

$$R_e = W \log_2 \left(1 + \frac{\mathcal{P}_{trans} \cdot c_g}{\sigma^2}\right), \tag{3.5}$$

where W denotes the network bandwidth. Finally, the transmission rate of the wired link between a base station (where an edge server is colocated) and a cloud server is denoted as R_c. Since the result returned from the edge/cloud server to the mobile

device is typically much smaller than the amount of data that need to be uploaded. We assume that the transmission delay associated with the returned result can be ignored. Therefore, the download rate of the mobile device does not need to be modelled.

Jobs Processed on Mobile Device

As mentioned previously, to reduce the energy cost of local jobs, the DVFS technique is applied to adjust CPU frequency dynamically. For a local DS job $\tau_n^i \in \mathcal{J}_{l,r}$, the CPU frequency f_l is referred to the system utilization $\mathcal{U}_s$. In contrast, the CPU frequency for an NDS job $\varphi_m \in \mathcal{J}_{l,nr}$ depends on the length of idle time T_d and its WCET w_m. In summary, the CPU frequency f_l for a job processed on the mobile device can be calculated using Eq. (3.6):

$$f_l = \begin{cases} \lceil \mathcal{U}_s \cdot f_k \rceil, & \tau_n^i \in \mathcal{J}_{l,r}, \mathcal{U}_s \leq 1, \\ f_1, & \varphi_m \in \mathcal{J}_{l,nr}, T_d \geq T^*. \end{cases} \tag{3.6}$$

Based on the CPU frequency f_l, the completion time of a local job, T_l^c, can be calculated using Eq. (3.7):

$$T_l^c = \begin{cases} a_n^i / f_n^i, & \tau_n^i \in \mathcal{J}_{l,r}, \\ a_m / f_1, & \varphi_m \in \mathcal{J}_{l,nr}, \end{cases} \tag{3.7}$$

where f_n^i is the CPU frequency for the job τ_n^i; f_1 denotes the lowest CPU frequency for the NDS job φ_m. All of these CPU frequencies are determined using Eq. (3.6).

Jobs Offloaded to Edge Server

If a job in the queue TQ is offloaded to the edge server, the execution time of this job, T_e, consists of four parts: 1) the queuing time at the local device, T_e^w; 2) the time used to transfer data between the mobile device and the edge server, T_e^t; 3) the queuing time at the edge server, T_e^q; 4) the computation time on the edge server, T_e^c. The length of the first part T_e^w depends on the number of jobs ahead of the job under

investigation. Note that the job under investigation can be either an DS job τ_n^i or an NDS job φ_m. Formally, T_e^w can be calculated using Eq. (3.8):

$$T_e^w = \begin{cases} \sum\limits_{\tau_l^j \in prec(\tau_n^i)} d_l^j / R_e, & \tau_n^i \in \mathcal{J}_{e,r} \\ \sum\limits_{\tau_l^j, \varphi_k \in prec(\varphi_m)} (d_n^j + d_k)/R_e, & \varphi_m \in \mathcal{J}_{e,nr} \end{cases} \tag{3.8}$$

where $prec(\varphi_m)$ is the jobs ahead of NDS job φ_m in the queue TQ; $\tau_l^j \neq \tau_n^i$; $\varphi_k \neq \varphi_m$.

In general, edge servers are physically close to mobile devices. Therefore, the propagation delay induced by packet transmission between edge servers and mobile devices can be omitted. Consequently, the time used to transfer data between the mobile device and the edge server, T_e^t, dominates the data transmission delay, which can be computed by the upload rate of the mobile device and the size of the data to be uploaded. Hence, the delay T_e^t is calculated using Eq. (3.9):

$$T_e^t = d_e / R_e, \tag{3.9}$$

where d_e denotes the size of data that need to be offloaded.

In the MEC system, the available computation resources of the edge server are shared by numerous associated mobile devices. Hence, the queuing time of a job on the edge server, T_e^q, is determined by the total number of offloaded jobs from connected mobile devices and the scheduling method for the queue. In our research, we focus on one of the mobile devices connected to the edge server. The DS and NDS jobs on the mobile device are generated according to the simulation settings that are given in Section 3.3.1. When a job from the mobile device is offloaded to the edge server, it is added to the job queue of the edge server. In order to calculate the queuing time of the job that is offloaded from the mobile device in a manageable manner, the jobs offloaded from other mobile devices to the edge server are modelled using Poisson distribution. Namely, we assume that the jobs from other connected mobile devices arrive at the edge server according to a Poisson distribution [61]. The average arrival rate of the jobs from other mobile devices is denoted as ϑ. Furthermore, we assume that the edge server uses EDF to schedule the offloaded jobs in its queue. With these assumptions, varying arrival rate ϑ leads to different queueing time T_e^q.

Note that edge servers are computationally faster than mobile devices. A job being processed on the edge server consumes less computation time. Thus, the computation time of a job completed by the edge server, T_e^c, can be defined as a fraction of its AET on the mobile device. Formally, the edge computation time of a job T_e^c can be calculated using Eq. (3.10):

$$T_e^c = a_l/(\zeta_e \cdot \omega_e) \tag{3.10}$$

where a_l is the AET of a job if it is processed locally; ω_e denotes the speedup of the edge server over the mobile device; ζ_e is a parameter based on the current workload on the edge server (e.g. if two jobs are being processed simultaneously on the edge server, then $\zeta_e = 1/2$).

To this end, the completion time of a job being offloaded to the edge server, T_e, can be calculated using Eq. (3.11):

$$T_e = T_e^w + T_e^t + T_e^q + T_e^c. \tag{3.11}$$

Jobs Offloaded to Cloud Server

In our research, we assume that the computation resources on the cloud server can satisfy all demands of the offloaded jobs. Namely, a job starts to be processed immediately after it arrives at the cloud server. Consequently, the queueing delay on the cloud server, T_c^q, is equal to zero. Hence, the execution time of a job offloaded to the cloud server comprises three components: 1) the queuing time at the local device, T_c^w; 2) the time consumed to transfer data between the mobile device and cloud server, T_c^t; 3) the computation time at the cloud server, T_c^c.

Similar to the scenario where a job offloaded to the edge server, an offloaded job cannot be processed until all the jobs ahead of the current job in the queue TQ have been dealt with. As a result, the local queuing time, T_c^w, can be calculated using an equation similar to Eq. (3.8). Note that, a cloud-processing job cannot be transferred to the cloud server directly. Instead, it is first forwarded to a base station in the mobile network via a wireless connection, then the base station relays the job to the cloud

server. Since the propagation delay between the base station and the cloud server is relatively large, it cannot be ignored any more. Overall, if a job is offloaded to the cloud server, the time used to transfer data between the mobile device and the cloud server, T_c^t, can be calculated using Eq. (3.12):

$$T_c^t = T_e^t + 2 \cdot L_c, \tag{3.12}$$

where T_e^t is the time used to transfer the data between the mobile device and the base station (where the edge server is located); $\cdot L_c$ is one-way propagation delay of wired link between the base station and the cloud server.

Typically, cloud server is more powerful than edge server. Therefore, the speedup of the cloud server over the mobile device, ω_c, is greater than ω_e. The computation time of a job completed by the cloud server can be calculated using Eq. (3.13):

$$T_c^c = a_l/(\zeta_c \cdot \omega_c) \tag{3.13}$$

where ζ_c is a parameter that is based on the current workload on the cloud server (e.g. if two jobs are being processed simultaneously on the edge server, then $\zeta_e = 1/2$). Thus, the completion time of a job offloaded to the cloud server, T_c, is the sum of three components:

$$T_c = T_c^w + T_c^t + T_c^c. \tag{3.14}$$

3.1.4 Energy Consumption Model

For a job processed locally on the mobile device, the energy consumption results from the processing of the job. For a job offloaded to the edge or cloud server, only the energy consumed by the transceiver of the mobile device needs to be taken into consideration.

Let us first consider the energy cost of a job processed locally. In our research, we focus on the energy consumed by the CPU for local jobs. For modern processors, running a job consumes two types of power, static power and dynamic power. Technically, static power is consumed even if no instruction is executed, while dynamic

power is involved when instructions of a job are executed [106]. Therefore, the energy consumption of a local job, E_l^c, is the sum of the static energy consumption, E_s, and the dynamic energy consumption, E_d. Formally, E_l^c can be calculated using Eq. (3.15):

$$E_l^c = E_s + E_d = (\mathcal{P}_s + \mathcal{P}_d) \cdot T_l^c, \tag{3.15}$$

where $\mathcal{P}_s$ and $\mathcal{P}_d$ denote the power consumption rate of static and dynamic power, respectively. Specifically, $\mathcal{P}_s$ and $\mathcal{P}_d$ can be calculated using the following equations [43]:

$$\mathcal{P}_s = V \cdot I_s + |V_{bs}| \cdot I_j. \tag{3.16}$$

$$\mathcal{P}_d = c \cdot V^2 \cdot f_l. \tag{3.17}$$

where I_s and V_{bs} denote the subthreshold current and the body bias voltage; I_j is the reverse bias junction current; c is a coefficient; V represents the voltage related to the frequency f_l. Apparently, $\mathcal{P}_d$ is proportional to the CPU frequency f_l. Consequently, $\mathcal{P}_d$ decreases as f_l goes down. As a result, when f_l is lower, the energy consumption of a local job, E_l^c, will be lower.

In terms of the power consumption for a job offloaded to the edge or cloud server, we focus on the energy consumed by the transceiver of the mobile device in order to send/receive the job data. This type of energy consumption is related to the size of data transferred between the mobile device and the edge/cloud server, and the instantaneous transmission rate. Hence, the energy consumed by the transceiver, E_l^t, can be calculated using Eq. (3.18) [14]:

$$E_l^t = (d_l / R_e) \cdot \mathcal{P}_{trans}, \tag{3.18}$$

where $\mathcal{P}_{trans}$ denotes the power consumption rate of the transceiver.

3.1.5 Problem Formulation

In this subsection, we formulate the multi-tier offloading problem that jointly optimizes the total energy consumption of mobile devices and the completion rate of NDS

jobs under the condition that the deadline constraints of DS jobs can be satisfied. We use T_n^i and T_m to denote the completion time of the DS job τ_n^i and the NDS job φ_m, respectively. For a local DS job $\tau_n^i \in \mathcal{J}_{l,r}$, T_n^i depends on the number of jobs in set $prec(\tau_n^i)$. Formally, it can be calculated using Eq. (3.19):

$$T_n^i = a_n^i / f_n^i + \sum_{\tau_n^j \in prec(\tau_n^i)} a_n^j / f_n^j, \tag{3.19}$$

where f_n^j can be obtained using Eq. (3.6). For the offloaded DS job $\tau_n^i \in \mathcal{J}_{e,r}, \mathcal{J}_{c,r}$, the completion time T_n^i can be calculated using Eq. (3.11) and Eq. (3.14). To determine whether a DS job misses its deadline, the completion time T_n^i is compared with its period p_n. The status variable that indicates whether the job τ_n misses its deadline, l_n^i, can be set using Eq. (3.20):

$$l_n^i = \begin{cases} 1, & if\ T_n^i \ \geq \ p_n, \\ 0, & otherwise, \end{cases} \tag{3.20}$$

where $l_n^i = 1$ indicates that the job τ_n^i misses its deadline and $l_n^i = 0$ indicates that the deadline is met. Based on this definition, the total number of deadline misses of DS jobs in a hyeperiod, $\mathcal{M}_r$, can be calculated using Eq. (3.21):

$$\mathcal{M}_r = \sum_{\tau_n^i \in \mathcal{J}_{l,r}, \mathcal{J}_{e,r}, \mathcal{J}_{c,r}} l_n^i. \tag{3.21}$$

If an NDS job, $\varphi_m \in \mathcal{J}_{l,nr}$, is processed locally, it is not dealt with until all local DS jobs and the local NDS jobs ahead of φ_m in LQ are completed. Therefore, its completion time, T_m, can be calculated using Eq. (3.22):

$$T_m = T_m^c + \sum_{\tau_n^j, \varphi_k \in pred(\varphi_m)} T_n^j + T_k^c, \tag{3.22}$$

where T_m^c and T_k^c denote the local completion time of the NDS job φ_m and φ_k, respectively; T_n^j is the local completion time of the DS job τ_n^j. If the completion time $T_m > \mathcal{H}_p$, the completion-status variable k_m is set to 1. Otherwise, the variable k_m is set to 0. When an NDS job, $\varphi_m \in \mathcal{J}_{e,nr}, \mathcal{J}_{c,nr}$, is offloaded to the edge/cloud

server, the computation result from server can be returned to the local device anytime. However, the completion-status variable k_m is set to 1 if the overall completion time is greater than the hyperperiod; otherwise, k_m is set to 0. Overall, the incompletion rate of NDS jobs, $\mathcal{C}_{nr}$, is defined as:

$$\mathcal{C}_{nr} = \sum_{\varphi_m \in Q_{nr}} k_m \ / \ |Q_{nr}|. \tag{3.23}$$

Furthermore, we use E_n^i to denote the energy consumption of a DS job τ_n^i that is successfully completed within its period. And we use E_m to denote the energy consumption of an NDS job φ_m that is successfully completed within its hyperperiod. Namely, $l_n^i = 0$ and $l_m = 0$. Thus, the total energy consumption of the jobs in a hyperperiod, E_l, is the sum of the local computation cost and the data transmission cost. Formally, E_l can be calculated using Eq. (3.24):

$$E_l = \sum_{\tau_n^i \in \mathcal{J}_{l,r}, \mathcal{J}_{e,r}, \mathcal{J}_{c,r},\ l_n^i = 0} E_n^i + \sum_{\varphi_m \in Q_{nr},\ l_m = 0} E_m, \tag{3.24}$$

where E_n^i and E_m are calculated with Eq. (3.15) and Eq. (3.18).

Finally, the multi-tier offloading optimization problem can be formulated as a minimization problem using Eq. (3.25):

$$\begin{aligned} \min_{\Lambda} \quad & \kappa \cdot E_l + \alpha \cdot \mathcal{M}_r + \upsilon \cdot \mathcal{C}_{nr} \\ \text{subject to} \quad & C1 : \lambda_n^i, \lambda_m \in \{0, 1, 2\} \\ & C2 : \sum_{\tau_n^i \in \mathcal{J}_{l,r}} w_n^i / \mathcal{H}_p \leq 1 \\ & C3 : \omega_c \gg \omega_e > f_l, \end{aligned} \tag{3.25}$$

where κ, α, and υ are three tunable coefficients predefined to balance the energy consumption, the deadline misses of DS jobs, and the completion rate of NDS jobs. Note that, compared with κ and α, υ should be set to a small value so that the weight associated with the completion rate of NDS jobs is not high. In this minimization problem, the constraint $C1$ ensures that a job is allocated to only one tier and offloading policy Λ is a candidate in set $\mathcal{O}$. Furthermore, the constraint $C2$ guarantees that

the sum of the computation resources allocated to DS jobs, $\tau_n^i \in \mathcal{J}_{l,r}$, do not exceed the total amount of computation resources of the mobile device. The constraint $C3$ guarantees that the computation speed of the edge/cloud server is faster than that of the local device. In our research, we found that it is non-trivial to find a solution to this optimization problem mathematically. Therefore, we attempt to employ deep reinforcement learning to generate a satisfactory solution. The details of the proposed scheme are presented in Section 3.2.

3.2 PDMO: Energy-aware DRL-based Task Offloading

In our research, we attempt to utilize deep reinforcement learning to solve the minimization problem for multi-tier MEC offloading. Since the MEC environment is not completely observable, it is challenging to find an optimal offloading policy. To achieve satisfactory offloading policy, we first propose a partially-observable DRL algorithm, POTD3, which considers both observable and unobservable states of the MEC system. Thereafter, we design a dynamic DRL-based computation offloading scheme, PDMO, which leverages POTD3 to learn the near-optimal offloading strategy. With PDMO, the energy consumption and the number of deadline misses are minimized while the completion rate of NDS jobs is maximized.

3.2.1 POTD3: A Novel Learning Algorithm

DDPG is a general model-free DRL approach based on the actor-critic structure that was proposed to deal with optimization problems with a high-dimension action space [57]. Technically, the DRL learning method comprises four neural networks: actor network π_{θ^μ}, actor target network $\pi_{\theta^{\mu'}}$, critic network π_{θ^Q} and its target network $\pi_{\theta^{Q'}}$. The actor network is used to select a possible action while the critic network is employed to evaluate the performance of this action. The actor target network is used to stabilize the learning process. Currently, DDPG has been extensively applied to communication applications [91, 98]. However, for time-variant environments that involve uncertainty, DDPG has been proven to be unstable and ineffective. In particular, DDPG induces extensive overestimation of Q-values, implicitly leading to many

sub-optimal policies. Moreover, most DDPG-based applications assume that the complete information of the MEC system under investigation is available at decision points. However, this is unrealistic for systems involving unknown features. Aiming to tackle the overestimation problem, Scott *et al*. proposed an enhanced DDPG approach, called TD3, which eliminates overestimation by utilizing a double-actor-critic neural network structure [29]. The second weakness of DDPG can be potentially addressed by converting the problem formulation from MDP to Partially Observable Markov Decision Process (POMDP).

In the multi-tier MEC offloading problem, part of the system states are unobservable. Therefore, DDPG is not an appropriate learning algorithm for the system under investigation. In our research, we proposed a novel learning algorithm for the MEC offloading problem, POTD3, which is derived from TD3. Compared with DDPG, POTD3 includes two additional measures. With the first additional measure, one more critic network is added to the learning model. Namely, POTD3 includes two pairs of critic and target network, which are denoted as $\pi_{\theta^{Q_1}}$, $\pi_{\theta^{Q_2}}$, $\pi_{\theta^{Q'_1}}$, $\pi_{\theta^{Q'_2}}$. The benefit of this additional measure is that the number of overestimated Q-values could be reduced significantly during the training phase. To be specific, one of the critic networks is randomly selected as the estimator of the Q-value, while the other is used to evaluate the Q-value. With the update rule, the value target can prevent additional overestimation when using standalone Q-learning target as DDPG [29].

With the second additional measure in POTD3, the MEC offloading problem is formulated as a POMDP. In this manner, a DRL agent can perform effectively on the environments involving uncertainty. Technically, a POMDP is commonly defined as a 5-tuple, (S, A, T, R, O, Γ), where S and A denote the state space $S = \{s_t | \forall t \in \mathbb{N}\}$ and the action space $A = \{a_t | \forall t \in \mathbb{N}\}$ respectively. T denotes the state transition function, $T = \{T : S \times A \times S \rightarrow [0, 1]\}$, which indicates the probability of the transition to the next state s_{t+1} when the state-action pair (s_t, a_t) is performed. R is the reward function $R = \{R : S \times A \rightarrow \mathbb{R}\}$. O is a set of observations $O = \{o_1, o_2, ...\}$ and Γ is an observation function $\Gamma = \{\Gamma : S \times O \rightarrow [0, 1]\}$. Note that, POTD3 is stochastic, resulting in random transitions between states. The DRL agent aims to find the optimal policy $\pi^* : O \rightarrow A$, which is defined as the maximum expected

Algorithm 2: POTD3.

Initialize actor and its target network θ^{μ}, $\theta^{\mu'}$;
Initialize critics and their target networks θ^{Q_1}, θ^{Q_2}, $\theta^{Q'_1}$,$\theta^{Q'_2}$;
Initialize replay buffer $\mathcal{M}$;
for episode t in 1,2,3,... do
    Observe state s_t and obtain unobservable state u_{t-1};
    Update the observation $o_t = (o_{t-1}, s_t, u_{t-1})$;
    Store transition $(o_{t-1}, a_{t-1}, o_t, r_{t-1})$ into $\mathcal{M}$;
    Select action $a_t = \pi_{\theta^{\mu}}(o_t) + \epsilon$ and calculate reward r_t;
    Sample mini-batch of N transitions (o_n, a_n, o_{n+1}, r_n) from buffer $\mathcal{M}$;
    Smooth action a_n: $\widetilde{a_n} \leftarrow \pi_{\theta^{\mu'}}(o_n) + \epsilon$, $\epsilon \sim clip(\mathcal{N}(0, \widetilde{\sigma}), -c, c)$;
    Loss function for critics is formulated as:
    $y_n \leftarrow r_n + \gamma \cdot min_{i=1,2}\pi_{\theta^{Q'_i}}(o_{n+1}, \widetilde{a_n})$;
    Update critic networks: $\theta^{Q_i} \leftarrow \min_{\theta^{Q_i}} N^{-1} \sum (y_n - \pi_{\theta^{Q_i}}(o_n, a_n))$;
    if t *mod* δ then
        Update actor network:
        $\nabla_{\theta^{\mu}} J(\theta^{\mu}) = N \sum \nabla_{a_n} \pi_{\theta^{Q_1}}(o_n, a_n) \mid_{a_n = \pi_{\theta^{\mu}}(o_n)} \nabla_{\theta^{\mu}}(o_n)$;
        Update target networks of actor and critics:
        $\theta^{\mu'} \leftarrow \nu\theta^{\mu} + (1-\nu)\theta^{\mu'}$;
        $\theta^{Q'_i} \leftarrow \nu\theta^{Q_i} + (1-\nu)\theta^{Q'_i}$;
    end
    Continue till convergence;
end

discounted reward $\pi^* = \arg\max_\pi E[\sum_{t=0}^{\infty} \varrho R_t^\pi | b]$, where R_t^π denotes the reward given by the policy π at time slot t; ϱ is the discount rate, $0 < \varrho < 1$; b denotes the starting belief state [86]. In POMDP, the action a_t is selected according to the observation o_t as the state s_t is partially known. Namely, DRL agent cannot apply the incomplete states of the environment according to $\pi_{\theta^\mu}(s_t)$ directly. Instead, it selects an action based on $\pi_{\theta^\mu}(o_t)$.

The details of POTD3 are presented in Algorithm 2. In each iteration t, the agent first observes available environment state s_t and obtains the estimation of the unobservable state u_{t-1}. Then, the observation o_t is updated relying on the state s_t, unobservable state u_{t-1} and last observation o_{t-1}. Afterwards, the transition $(o_{t-1}, a_{t-1}, o_t, r_{t-1})$ is stored into the replay buffer $\mathcal{M}$ for the training of neural networks. According to the observation o_t, the agent chooses an appropriate action a_t with an exploration noise ϵ using the actor network π_{θ^μ}. After the interaction is done, the reward r_t is calculated accordingly. In order to prevent the overfitting to narrow peak, the action a_n issued by π_{θ^μ} is smoothed to a small area, which is described in Line 10. In Line 12, the pair of critic networks is updated based on the minimum target value of actions. Once every δ iterations, the actor network π_{θ^μ} is updated using $\pi_{\theta^{Q_1}}$ with deterministic policy gradient, which is described in Line 14. To apple POTD3 to the MEC offloading problem, we proposed a new DRL-based offloading scheme, PDMO, which is illustrated in the next subsection.

The computational complexity of Algorithm 2 involves two components: actor network update and critic network update. The actor network is updated by executing forward-propagation $N_{mb} + \delta$ times and backward-propagation N_{mb} times during each updating window, where N_{mb} denotes the size of the mini-batch and δ is the length of the updating window. In our research, we use N_a to denote the number of neurons in the actor network. The computational complexity of updating actor network during one iteration corresponds to the total number of neuron operation involved in the required forward and backward propagation: $O(((N_{mb} + \delta) + N_{mb}) * N_a)$ $= O((2 * N_{mb} + \delta) * N_a)$. The critic network update involves carrying out forward propagation $2 * N_{mb} * \delta$ times and backward propagation $2 * N_{mb} * \delta$ times during each updating window. In our research, we use N_c to denote the number of neurons in

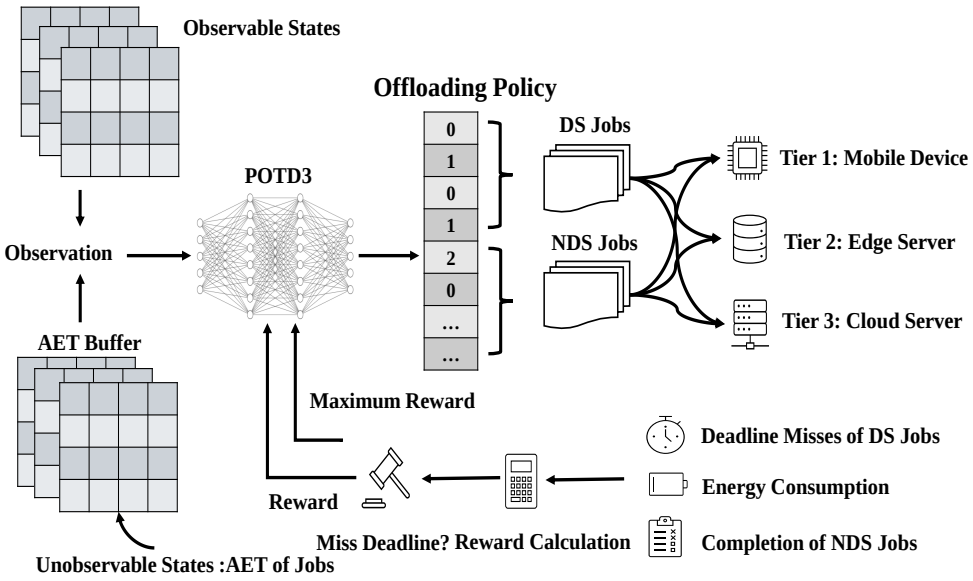


Figure 3.3: Scheme of PDMO.

the critic network. The computational complexity of updating critic network during one iteration corresponds to the total number of neuron operation involved in the required forward and backward propagation: $O((2 * N_{mb} * \delta + 2 * N_{mb} * \delta) * N_c) = O(4 * N_{mb} * \delta * N_c)$. Suppose that Algorithm 2 needs i iterations to converge, the computational complexity of Algorithm 2 is $O((2 * N_{mb} + \delta) * N_a + 4 * N_{mb} * \delta * N_c)$. Once Algorithm 2 converges, the actor network itself can be used to make offloading decisions. This decision-making process only involves one forward propagation of the actor network. Therefore, its computational complexity is $O(N_a)$.

3.2.2 Details of PDMO

As shown in Fig. 3.3, the PDMO scheme involves three tiers: mobile device, edge server, and cloud server. Initially, there are a set of DS tasks $\mathcal{T} = \{\tau_1, \tau_2, ...\}$ and a set of NDS tasks $\Phi = \{\varphi_1, \varphi_2, ...\}$ in the local device. As DS jobs repeat periodically, the number of DS jobs in each hyperperiod is unchanged. The number of NDS jobs in each hyperperiod varies due to their aperiodic nature. As described in Section 3.1.1, the total number of jobs in a hyperperiod x can be calculated using Eq. (3.2). To make proper offloading decisions for varying x, PDMO maintains a uniform offloading

profile for both DS and NDS jobs $\Lambda = (\lambda_1^1, ...\lambda_n^i, ..., \lambda_{N_{DS}}, \lambda_1, ..\lambda_{N_{NDS}}, 0, ..., 0)$. Note that, at the beginning of a hyperperiod, the number of DS jobs that will arrive during the current hyperperiod, N_{DS}, is fixed because DS tasks are periodic, while the number of NDS jobs that have arrived, N_{NDS}, varies from hyperperiod to hyperperiod. Obviously, $N_{NDS} = |\mathcal{Q}_{nr}|$. Since a learning algorithm prefers to deal with an offloading profile with a fixed number of elements, a uniform offloading profile with $N_{DS} + N_{NDS,Fixed}$ elements is maintained in PDMO. Namely, $N_{NDS,Fixed}$ positions in Λ are reserved for NDS jobs. If N_{NDS}=$N_{NDS,Fixed}$, all the reserved positions will be utilized. If $N_{NDS} < N_{NDS,Fixed}$, the unused positions at the end of Λ will be filled with zeros (i.e. placeholders). If $N_{NDS} > N_{NDS,Fixed}$, only the first N_{NDS} NDS jobs will be scheduled, other NDS jobs will be buffered for future processing.

The objective of POTD3 is to learn the best offloading policy so that the minimum amount of energy is consumed on the mobile device while the deadline of DS jobs can be met as well as the completion rate of NDS jobs is maximized. To achieve this objective, the multi-tier offloading optimization problem is transformed into to a POMDP. Then, a satisfactory offloading policy is generated at the beginning of each hyperperiod. The details of PDMO are summarized in Algorithm 3. The major modules of PDMO are presented as follows.

A. System Observation

At the beginning of the t-th hyperperiod, PDMO first observes the current system state s_t that involves network condition, workload of mobile device and workload edge server and cloud server. Meanwhile, the AET of DS jobs in the last hyperperiod is also observed. Thereafter, the average AET is calculated and stored in an AET buffer. Instead of using the system state s_t directly, the input of PDMO has switched to the observation o_t that consists of multiple elements. One of the elements is the observable system state s_t, which is denoted as a 5-tuple $(\mathcal{B}_t, \mathcal{L}_t, \mathcal{Q}_t, \mathcal{E}_t, \mathcal{C}_t)$. The elements involved in the 5-tuple are presented as follows:

(i) Network Condition ($\mathcal{B}_t$): Network condition $\mathcal{B}_t$ is mainly determined by the transmission rate R_e since the transmission rate from edge to cloud R_c is relatively stable.

(ii) Local Workload ($\mathcal{L}_t$): Since DS jobs are periodic, the total number of DS jobs in each hyperperiod is fixed, leading to an unchanged system utilization before jobs are placed into different queues. Therefore, the fluctuation of workload $\mathcal{L}_t$ results from the changing number of NDS jobs in $\mathcal{Q}_{nr}$.

(iii) Edge Queuing Information ($\mathcal{Q}_t$): In the MEC system, all associated mobile devices can compete for the limited computation resources at the edge server. Therefore, the edge queuing delay is a factor that should be taken into considerations. With PDMO, the queuing signal of eligible edge server $\mathcal{Q}_t$ should be sent back to the mobile device before it makes offloading decisions for the current hyperperiod.

(iv) Workload of Edge and Cloud Servers ($\mathcal{E}_t$ and $\mathcal{C}_t$): The workload of edge and cloud server, $\mathcal{E}_t$ and $\mathcal{C}_t$, affect the offloading decisions made on the mobile device. The two states are mainly based on the speedups ω_e and ω_c as well as the instantaneous workload ζ_e and ζ_c.

With PDMO, the unobservable state of the MEC offloading system corresponds to the average AET of DS jobs in the t-th hyperperiod, $\mathcal{A}_t$. Although $\mathcal{A}_t$ is unavailable, $\mathcal{A}_{t-1}$ is available because the DS jobs in the $(t-1)$-th hyperperiod have been completed. In our research, the system observation at the t-th hyperperiod, o_t, is set to $(o_{t-1}, o_{t-2}, ...o_{t-M}, s_t, \mathcal{A}_{t-1})$, where M is a predefined constant. Note that $o_t \in O$. In addition, the elements in the vector for o_t are generated with random values if $t < (M+1)$.

B. System Action

PDMO attempts to make a holistic decision for both DS and NDS jobs for each hyperperiod. Namely, at the beginning of a hyperperiod, PMDO needs to determine whether a job should be offloaded or not by configuring the offloading profile. Namely, the action at the t-th hyperperiod, a_t corresponds to $\Lambda_t, \forall t \in \mathbb{N}$. Formally, Λ_t can be generated by the actor network π_{θ^μ} according to the observation o_t using Eq. (3.26):

$$\Lambda_t = \pi_{\theta^\mu}(o_t) + \epsilon. \tag{3.26}$$

where ϵ corresponds to the exploration noise.

C. Observation Transition

After the mobile device performs the offloading actions, the observation o_t will switch to the observation for the next hyperperiod, o_{t+1}. Note that, the transition from s_t to s_{t+1} is a stochastic process, resulting in the entire transition being stochastic.

D. Reward Function

The objective of PMDO is to minimize the energy consumption of the mobile device, satisfy the deadline constraint of DS jobs, and maximize the completion rate of NDS jobs. To achieve the objective, PMDO initially determines whether a DS job τ_n^i misses its deadline at the end of each hyperperiod after all the DS and NDS jobs in the hyperperiod are processed according to the offloading policy Λ_t. If the job τ_n^i misses its deadline, its deadline-miss variable, l_n^i, is set to 1. Then, the reward of job τ_n^i is replaced with a predefined maximum reward, denoted as r^*, to informatively guide the DRL agent to move towards other secure offloading policies. Otherwise, PMDO calculates the total energy consumption of the mobile device E_l and the completion rate of NDS jobs $\mathcal{C}_{nr}$. Thus, the immediate reward of the job (r_t) for the current hyperperiod can be given by

$$r_t = \begin{cases} \mathcal{M}_r \cdot r^*, & if\ \exists\ l_n^i = 1, \\ \kappa \cdot E_l + \upsilon \cdot \mathcal{C}_{nr}, & otherwise, \end{cases} \tag{3.27}$$

where $\mathcal{M}_r$ denotes the number of DS jobs that miss their deadlines. Note that, κ and υ can be configured by mobile device users manually for different performance goals. For example, the setting of $\kappa > \upsilon$ indicates that the mobile user intends to save more energy rather than completing more NDS jobs.

With PDMO, the objective is to find an optimal policy so that the cumulative reward R that considers both the current hyperperiod and all past hyperperiods, instead of the immediate reward for the current hyperperiod, is maximized. Mathematically, R is the weighted average of $-r_t$ (where $t = 0, 1, ...$), which can be calculated using Eq. (3.28):

$$R = AVG(-r_t \cdot \varepsilon_t), \tag{3.28}$$

where $\varepsilon_t \in [0,1]$ denotes the discount rate and it goes up at time goes by.

The details of PMDO are summarized in Algorithm 3. Specifically, the system state s_t is first observed according to the conditions of the mobile device, edge server, and cloud server. With the unobservable state $\mathcal{A}_{t-1}$, the current observation o_t is obtained accordingly. Next, the transition sample $(o_{t-1}, \Lambda_{t-1}, o_t, r_{t-1})$ is stored into the replay buffer $\mathcal{M}$. Afterwards, PMDO chooses an offloading policy Λ_t based on the observation o_t. According to the generated offloading policy Λ_t, each job in the hyperperiod t is dispatched to one of the six job subsets and then scheduled aligned with Algorithm 1. At the end of each hyperperiod, the immediate reward r_t can be obtained using Eq. (3.26). On this basis, the cumulative reward, R, can be obtained, and the actor/critic network is updated using POTD3. PMDO continues until the stop condition is satisfied.

Algorithm 3: PDMO.

Input: DS task set $\mathcal{T}$ and NDS task set Φ

Output: Offloading policy $\Lambda_t, \Lambda_t \in \mathcal{O}$

1 for Hyperperiod $t, t = 1, 2, 3..$, do
2 Observe current state of mobile device, network and servers: s_t=$(EW_t, CW_t, LQ_t, BW_t, EQ_t)$;
3 Obtain the average AET of DS jobs in last hyperperiod to calculate $\mathcal{A}_{t-1}$;
4 Update current observation o_t with o_{t-1}, s_t and $\mathcal{A}_{t-1}$;
5 Store transition $(o_{t-1}, \Lambda_{t-1}, o_t, r_{t-1})$ into memory $\mathcal{M}$;
6 Choose an offloading policy Λ_t using actor network π_{θ^μ};
7 Schedule jobs using Algorithm 1;
8 Update actor and critics according to Lines 13-18 of Algorithm 2;
9 Continue till stop condition is satisfied;
10 end

3.3 Evaluation

In this section, we comprehensively evaluate the convergence performance of PDMO by varying the number of tasks and the ratio of cloud transmission rate to edge transmission rate. In addition, the performance of PDMO is compared to that of the existing offloading schemes in terms of energy consumption and deadline miss.

3.3.1 Evaluation Settings

In our simulations, we consider a set of DS tasks and a set of NDS tasks. The total number of jobs corresponding to these tasks can be calculated using Eq. (3.2). Note that mobile applications typically include a number of light-weight tasks. The period and data size of these tasks tend to be relatively small. In our simulations, a set of DS tasks are generated. Unless specified otherwise, the period of these DS tasks is set to a value in the range of $[4, 32]$ seconds. The hyperperiod of this set of tasks, $\mathcal{H}_p$, is 288 seconds, which is calculated using Eq. (3.1). The data size of these tasks varies from 0 Mb to 100 Mb. The WCET of the DS tasks is set to a value in the range of $[1, 6]$ seconds. To allow dynamic frequency scaling, the AET of a DS job is a random number in the range of [0.5 x WCET, WCET]. The transmission power of the transceiver in a mobile device is set to $1.0J/s$. To estimate the energy consumption of mobile devices, we adopt the power consumption model used in [14, 80]. The speedup of an edge server and a cloud server over a mobile device is set to $\omega_e = 4$ and $\omega_c = 12$ respectively. As mentioned in Section 3.1.3, we focus on one of the mobile devices connected to an edge server in our simulations. We assume that the jobs offloaded from other mobile devices to the edge server follow a Poisson distribution. The average arrival rate of these jobs, ϑ, is set to a value in the range of 0 job/sec to 1 job/sec. Unless specified otherwise, the parameters used in our simulations are summarized in Table 3.4.

Note that PDMO is essentially a trial-and-error scheme. Consequently, it is possible that too many DS jobs are assigned to $\mathcal{J}_{l,r}$ at the beginning of a hyperperiod. Obviously, some of these DS jobs cannot be executed at all duo the limited computation resources on a mobile device. To speed up the convergence of PDMO, when a

Table 3.4: Simulation Parameters in Chapter 3.

Parameter	Value
Number of DS tasks ($\|\mathcal{T}\|$)	[4, 10]
Number of NDS tasks ($\|\Phi\|$)	[0, 10]
Period (p_n)	[4, 32] seconds
WCET	[1, 6] seconds
Data (d_n^i, d_m)	[10, 100] Mb
Transmission power ($\mathcal{P}_{trans}$)	1.0 J/s
Server speedups (ω_e, ω_c)	4, 12
Transmission rates (R_e, R_c)	(0, 75] Mbps, [50, 150] Mbps
Propagation delay (L_c)	0.5 seconds
Average job arrival rate (ϑ)	(0, 1) job/sec
Performance preference (κ, υ)	0.8, 0.2
Length of hyperperiod ($\mathcal{H}_p$)	288 seconds
Mini-batch size	128
Learning rate for actor and critic network	0.0001, 0.0003
Discount rate	0.99
Updating episode (δ)	100 x $\mathcal{H}_p$

DS job assigned to $\mathcal{J}_{l,r}$ is not executed at all, the energy consumption of this job is set to $\max(E_l^c, E_l^t)$. As mentioned previously, E_l^c and E_l^t can be calculated using Eq. (3.15) and Eq. (3.18) respectively.

In our research, we compared PDMO with a set of baseline offloading methods that are described as follows:

(i) Local Computing (LC): With LC, given a job set, every DS and NDS job is processed on the local device. With this method, the DVFS technique is used to lower the CPU frequency whenever possible.

(ii) Server Computing (SC): With SC, all jobs are offloaded to either an edge or cloud server. When jobs are offloaded to servers, we employ the TransScheduling function in Algorithm 1.

(iii) Random Offloading (RO): With RO, the DS and NDS jobs are assigned to one of the six job subsets randomly. Thereafter, each job is processed at one of the computation tiers accordingly.

(iv) DDPG-based Multi-tier Offloading (DMO): DMO and PDMO are highly similar. The only difference between them is that DMO leverages the traditional DDPG algorithm, instead of POTD3, to learn the optimal offloading policy. Since DDPG is used as the learning algorithm, the unobservable state of the MEC system (i.e. the AET information) cannot be utilized.

(v) PDMO-WD (PDMO Without DVFS): Both PDMO and PDMO-WD use POTD3 to learn the optimal offloading policy. However, PDMO-WD does not include the DVFS mechanism. With PDMO-WD, a job processed on the local mobile device is always executed at the maximum CPU frequency.

To quantify the performance of the offloading schemes under investigation, three evaluation metrics, which are normalized with min-max normalization, are used in our research:

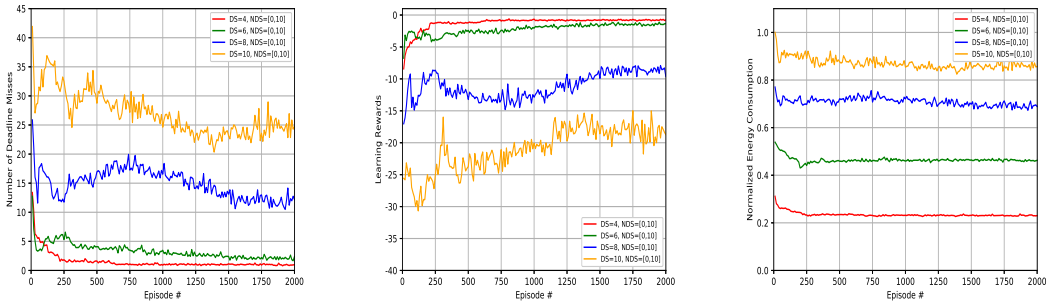
(i) Number of Deadline Misses: It indicates the total number of deadline misses for DS jobs. Reducing this number can ensure that more DS jobs are completed within their periods.

(ii) Energy Consumption: It consists of two components: energy cost of local computation and transmission energy consumption.

(iii) Completion of NDS jobs: It is the percentage of the completed NDS jobs in a hyperperiod.

3.3.2 Convergence

In this section, we present the convergence performance of PDMO using the results from two experiments. In the first experiment, we aim to study the impact of the number of tasks on the convergence of PDMO. In each simulation, the number of DS tasks stays the same for each hyperperiod. In our experiments, this number could be set to 4, 6, 8 or 10. The number of NDS tasks varies from hyperperiod to hyperperiod. In our experiments, this number is a random number in the range of 0 to 10. In the following sections, the number of NDS tasks is set in the same manner. The period of



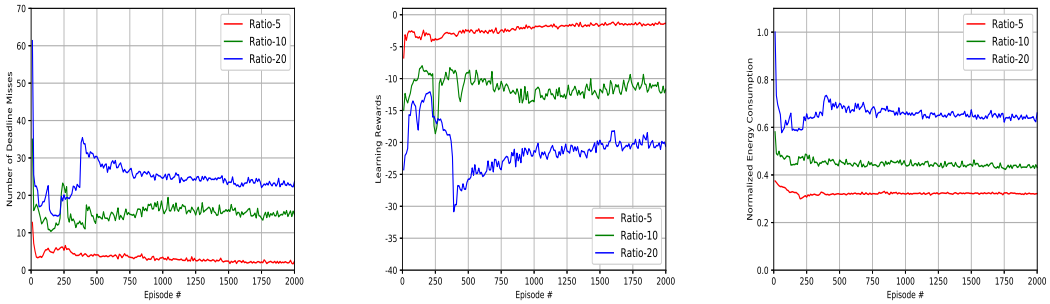
(a) Deadline Misses (b) Learning Rewards (c) Energy Consumption

Figure 3.4: Convergence vs. Number of Tasks

each RL task in one simulation is selected from the period set $\{4, 6, 12, 18, 32, 16\}$. In our experiments, the number of RL tasks and the periods selected for each simulation leads to a fixed hyperperiod 288 seconds. When the number of RL tasks is equal to 4, 6, 8 and 10, the total number of RL jobs are 115, 187, 217 and 289, respectively. The range of R_e in this experiment is set to [10 Mbps, 30 Mbps]. The average job arrival rate is set to 0.05 job/sec. Our simulation results are summarized in Fig. 3.4. Obviously, PDMO can quickly converge after a few gradient updates. For instance, in the simulation where there are 4 DS tasks, PDMO converges to its best performance within 250 episodes or so. That is because, in this case, most jobs are dispatched to different computation tiers properly, resulting in fewer incorrect decisions in the training process. It is worth noting that as the number of tasks increases, the it takes longer and longer for PDMO to converge. The reason behind this trend is that a higher number of tasks mean that more jobs need to compete for the limited computation and transmission resources, leading to a longer training time to arrive at the optimal arrangement. For the simulation where there are 10 DS tasks, PDMO converges within around 1750 episodes.

In the second experiment, we attempt to understand the impact of network condition on PDMO convergence. In these simulations, the number of DS tasks is set to 6. Our simulation results are summarized in Fig. 3.5. Since the transmission rate R_c between edge and cloud is relatively stable, we vary the transmission rate R_e and arrive at different R_c/R_e ratios. Specifically, R_c/R_e is set to 5, 10 and 20 in three

different simulations. Our simulation results show that PDMO can always converge to its optimum within 1650 episodes when R_c/R_e ratio varies.



(a) Deadline Misses (b) Learning Rewards (c) Energy Consumption

Figure 3.5: Convergence vs. Ratio of R_c to R_e

3.3.3 Deadline Misses and Energy Consumption

In this section, we compare PMDO with the existing methods under investigation in terms of energy consumption and deadline miss. In this set of simulations, the number of DS tasks within a hyperperiod is set to 6 and the R_c/R_e ratio is set to 5. Correspondingly, the number of RL jobs within each hyperperiod is 187. The range of R_e is [10 Mbps, 30 Mbps]. The average job arrival rate is 0.05 job/sec. Fig. 3.6 includes the detailed simulation results. Overall, four conclusions could be drawn from the simulations. First of all, blindly offloading all jobs to servers (i.e. the SC scheme) would consume much energy and force numerous jobs to miss their deadlines. That is because transmission energy consumption depends on the size of the data transferred between mobile devices and edge/cloud servers. When the size of the data is relatively large, the transmission energy consumption could easily be higher than local energy consumption. With our simulation configuration, transmission energy consumption tends to be higher than local energy consumption. Furthermore, when there are many jobs in TQ, the queuing time would spike, ultimately resulting in lengthened completion time. Compared with SC, Rand-O consumes less energy and leads to less deadline misses. This is due to the fact that, with Rand-O, all jobs are evenly dispatched to three computation tiers. Consequently, only 2/3 of the jobs are offloaded

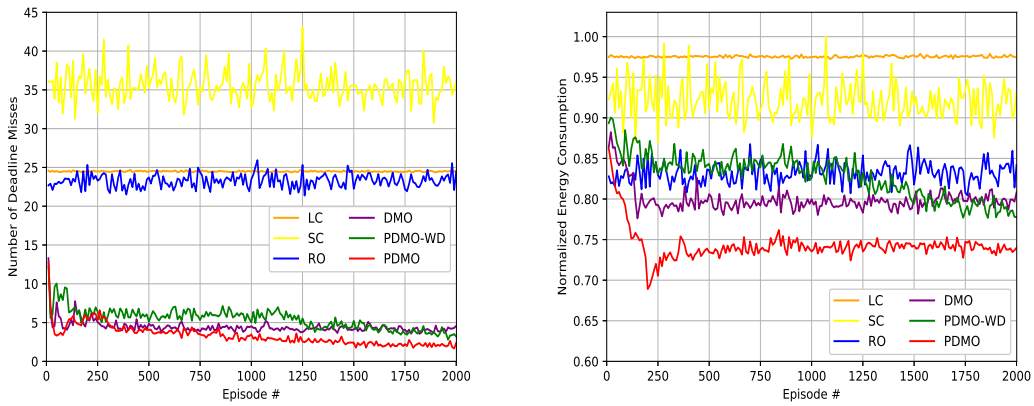
Table 3.5: Summary of Comparison Results.

Method	Miss Rate	Energy Consumption	NDS Completion Rate
LC	13.0 %	1	68%
SC	17.3%	91.7%	88%
RO	10.8 %	82.7%	93%
DMO	1.9 %	79.3%	95%
PDMO-WD	1.4 %	78.5%	94%
PDMO	0.8%	74.6%	97%

to either an edge server or a cloud server. Thirdly, all learning-based offloading schemes (including DMO, PDMO-WD, and PDMO) outperform non-learning-based offloading schemes in terms of both deadline miss and energy consumption. Note that, although DMO and PDMO-WD are close to PDMO in term of deadline miss (because all of them take deadline into consideration during the learning process), DMO and PDMO-WD lead to much higher energy consumption. Finally, PDMO results in the least amount of energy consumption and the fewest number of deadline misses. There are two main reasons why PDMO outperforms DMO. Firstly, PDMO leverages two critic networks to efficiently eliminate overestimated Q-values during the learning process. Additionally, unlike DMO, which is fed with incomplete information in partially observable MEC systems, PDMO approximates the missing system states by utilizing past experiences. With complete information about the system as inputs, PDMO is more likely to generate better offloading policies. Compared to PDMO-WD, PDMO can save more energy due to the utilization of DVFS techniques.

A summary of the simulation results in terms of normalized energy consumption, deadline miss rate and NDS completion rate is included in Table 3.5. The experimental results indicate the PDMO outperforms other offloading schemes under investigation in terms of all the metrics under investigation. In particular, PDMO can achieve an 0.6% and a 3.9% advantage over the second best offloading scheme in terms of deadline miss and energy consumption respectively. Furthermore, with PDMO, the completion rate of NDS jobs is 97%, which is very close to the perfect rate of 100%.

Overall, our experimental results indicate that PDMO can always converge within



(a) Deadline Misses (b) Energy Consumption

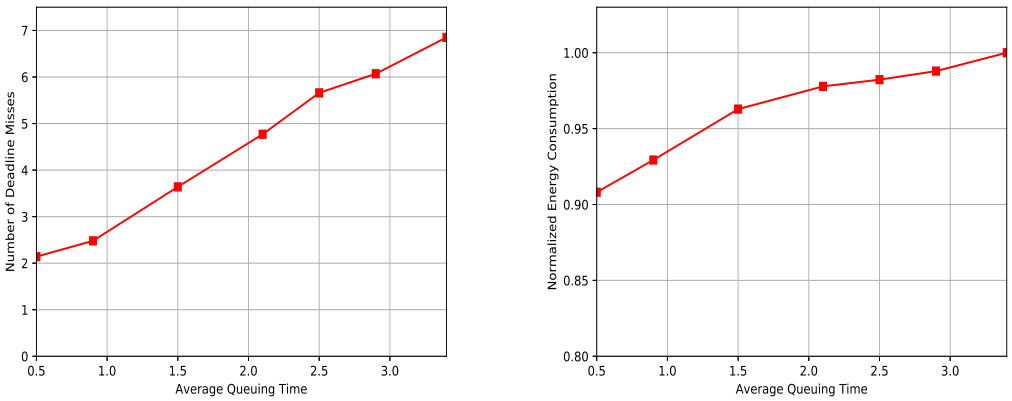
Figure 3.6: Deadline Misses and Energy Consumption

a short period. In addition, among the offloading methods under investigation, PDMO leads to the lowest number of deadline misses, the lowest energy consumption and the highest NDS completion rate. We believe that PDMO is a highly feasible offloading scheme for 3-tier MEC systems involving mobile devices, edger severs and cloud server.

3.3.4 Impact of Queuing Time at Edge Server

As mentioned in Section 3.1.3, the queuing time of an offloaded job on the edge server, T_e^q, is determined by the total number of offloaded jobs from connected mobile devices and the scheduling algorithm for the queue. In our research, we focus on one of the mobile devices connected to the edge server, and we assume that the jobs from other connected mobile devices arrive at the edge server according to a Poisson distribution. Furthermore, we assume that the edge server uses EDF to schedule the offloaded jobs in its queue. Consequently, the average arrival rate of the jobs from other connected mobile devices, ϑ, determines T_e^q of the job offloaded from the mobile device that we focus on. In this set of simulations, the parameters (such as period and AET) of the jobs from other mobile devices are generated according to Table 3.4, which ensures that the jobs from other mobile devices are similar to those from the mobile device that we focus on. The average arrival rate, ϑ, is set to 0.05, 1.0, 0.3, 0.5, 0.7, 0.8, and

0.9 job/sec respectively. As a result, in our simulations, the average queuing time of a job offloaded to the edge server is 0.5, 0.9, 1.5, 2.1, 2.5, 2.9, and 3.4 seconds respectively. The performance of PDMO under varied queuing time is included in Fig. 3.7. The experimental results indicate that both the number of deadline misses and the amount of energy consumption increase with the queuing time. Namely, the queuing time at the edge server has a negative impact on the performance of PDMO.



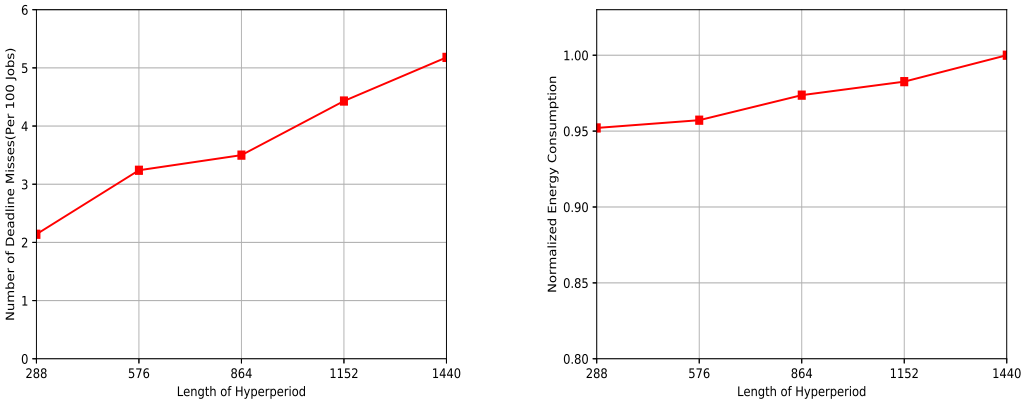
(a) Deadline Misses (b) Energy Consumption

Figure 3.7: Impact of Edge Server Queuing Time on PDMO

3.3.5 Impact of Hyperperiod Length

In this section, we study the performance of PDMO when one of the DS tasks rarely arrives. In this scenario, the resulting hyperperiod is lengthy. Specifically, we consider a set of 6 DS tasks. The periods of the first five DS tasks are 4, 6, 12, 18, and 32 seconds respectively. The period of the last DS task is set to 16, 576, 864, 1152, or 1440 seconds, which leads to a hyperperiod of 288, 576, 864, 1152, or 1440 seconds. Furthermore, since the total number of jobs in a hyperperiod is related to the length of the hyperperiod (typically, the longer the hyperperiod, the higher the number of jobs within a hyperperiod), we use the number of deadline misses per 100 jobs (instead of the number of deadline misses within each hyperperiod) to quantify the deadline-meeting performance of PDMO in this section. Finally, the average arrival rate, ϑ, is set to 0.05 job/sec. Fig. 3.8 includes the details of the impact of hyperperiod

length on PDMO. Note that as the length of the hyperperiod goes up, the number of jobs within a hyperperiod increases. Consequently, the decision-making environment becomes more complex and PDMO's offloading decision deteriorates, resulting in more deadline misses and more energy consumption. In summary, the longer the hyperperiod, the worse the performance of PDMO in terms of deadline miss and energy consumption.



(a) Deadline Misses (b) Energy Consumption

Figure 3.8: Impact of Hyperperiod Length on PDMO

3.4 Major Conclusions of PDMO

In our research, we propose a partially-observable DRL-based multi-tier offloading scheme, PDMO, for MEC systems without complete system information, which involves both DS and NDS tasks. Different from the existing offloading methods for MEC, PDMO employs an enhanced DDPG learning algorithm, POTD3, to learn the appropriate offloading policy so that the deadlines of DS tasks can be met, the total energy consumption is minimized, and the completion rate of NDS tasks is maximized. Technically, the offloading problem is formulated as a POMDP, which can be solved using the proposed learning algorithm, POTD3. In addition, the DVFS technique is used to further reduce the computation cost. To evaluate the effectiveness of PDMO, we investigate the convergence of PDMO by varying the number of

tasks and the transmission rate ratio. In addition, we compare the performance of PDMO to that of the existing offloading schemes in terms of energy consumption and deadline miss number. Finally, we study the impact of edge server queuing time and hyperperiod length on the performance of PDMO. Our experimental results indicate that PDMO outperforms the existing offloading schemes for MEC.

Note that, three key parameters are used to adjust the weight of the optimization objectives: minimizing the number of deadline misses, minimizing energy consumption of mobile devices and maximizing the completion rate of NDS tasks. Although this is a feasible approach, selecting the appropriate values for these key parameters is not a simple task. Our next-step research will focus on a more feasible method to integrate the optimization objectives. Furthermore, with PDMO, each mobile device can only offload its tasks to one edge server. If each mobile device can interact with multiple edge servers simultaneously, the resources on edge servers could be better utilized, potentially leading to less deadline misses and energy consumption. In the near future, we also plan to investigate the performance of PDMO in this multi-edge-server scenario.

Chapter 4

Edge-assisted DRL-based Task Offloading

In this chapter, we present the edge-assisted learning-based task offloading scheme for multiple-edge-server MEC systems, MELO, which is aimed at minimizing task completion time. Specifically, we first present the system models of MELO. Subsequently, we delve into the detailed description of this proposed mechanism. Finally, we conduct a comprehensive evaluation on the performance of MELO against various baseline schemes.

4.1 System Model

In this research, we attempt to solve the problem of finding the optimal offloading policy for periodic deadline-sensitive applications on mobile devices in the multiple-edge-server MEC systems. To formally formulate the problem, we adopt three independent models: task model, communication model, and completion time model. The details of these models are described in this section.

4.1.1 Overview

In this section, we present a 6G-empowered MEC architecture for offloading a set of periodic deadline-sensitive tasks. As shown in Fig. 4.1, the hierarchy of this architecture is $1/M/1$, consisting of one mobile device, a number of $\mathcal{E}$ homogeneous edge servers $ES = \{ES_1, ES_2, ..., ES_{\mathcal{E}}\}$ in close proximity, and one remote cloud server. Note that, the hardware and software settings, e.g., the maximum frequency of processors, are identical among $\mathcal{E}$ edge servers. In the architecture, mobile device communicates with each edge server $ES_e (e \in \mathcal{E})$ via the potential 6G wireless networks that are capable of supporting a more reliable and faster connectivity than current 4G/5G networks [44]. On the other hand, a task is allowed to be offloaded to the cloud server through wired optical cables if edge servers do not have enough computational resources.

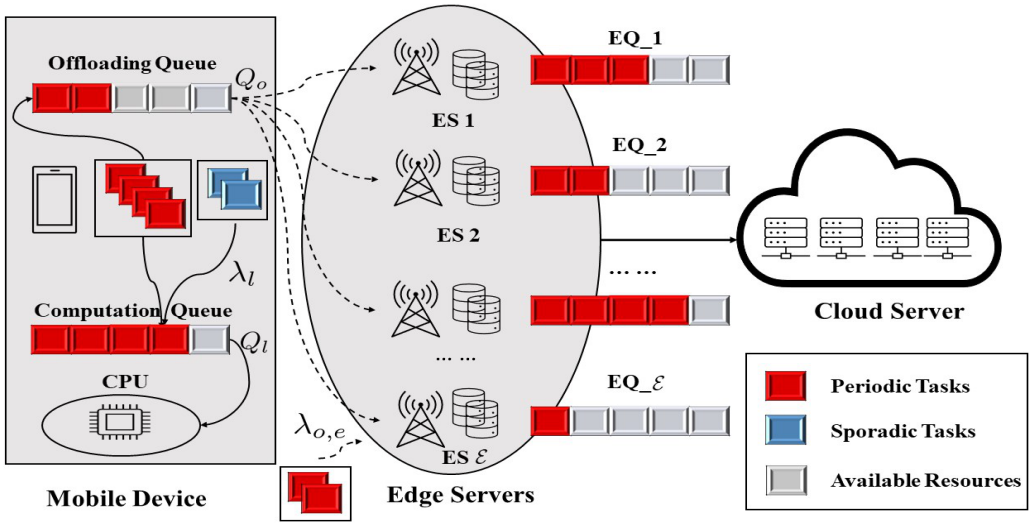


Figure 4.1: Architecture of the MEC System under Investigation for MELO.

In mobile device, we consider two sorts of tasks, including periodic deadline-sensitive tasks and sporadic deadline-sensitive tasks. The mobile devices, e.g., smartphone, laptop, and autonomous vehicle, execute the former tasks periodically while only scheduling the latter in a comprehensive manner after it arrives to system with a random interval. We do not consider how to offload the sporadic task to remote servers and simply dispatch all of them to the local device for two reasons. First, our model aims to make the offloading decision for a set of tasks within a slotted time period, the arrival of sporadic tasks cannot be captured at the decision points. More details will be illustrated in section 4.1.2. Second, sporadic tasks, e.g., airbag activation task in a vehicle and routing replanning in autopilot system, typically have relatively short deadlines. It is hard to guarantee the deadlines in highly dynamic networks [36, 107]. For a periodic task, each mobile device needs to decide not only whether to offload the task or not, but also where to offload the task. In addition, we assume that mobile device maintains two waiting queues, computation queue Q_l and offloading queue Q_o, to well schedule local tasks and the offloaded tasks, respectively. In other words, for each incoming task in mobile devices, it needs to compete for the limited computation or transmission resources with other ready

Table 4.1: Key Notations in Chapter 4

Symbol	Definition
$\mathcal{U}, \mathcal{V}$	Sets of periodic and sporadic tasks
μ_i, ν_k	i-th periodic task and k-th sporadic task
μ_i^j	j-th job of periodic task i in a hyperperiod
ν_k^n	n-th job of sporadic task i in a hyperperiod
$\mathcal{H}_p$	Hyperperiod of task set $\mathcal{U}$
$\hat{\mathcal{Z}}_p$	Total number of periodic jobs in a hyperperiod
$\Upsilon_l, \Upsilon_{o,e}$	Workloads of mobile device and $ES_e, ES_e \in ES$
R_u	Uplink rate from mobile device to edge server
f_l, f_e, f_c	Computation speedup of mobile, edge and cloud server
$\mathcal{J}_l$, $\mathcal{J}_o$	Job sets for local and edge processing periodic jobs
$\mathcal{J}_s$	Job set of sporadic task arriving in current hyperperiod
$\mathcal{J}_f$	Job set for unfinished sporadic task in last hyperperiod
$\mathcal{J}_{o,e}$	Job set to be offloaded to $ES_e, e \in \mathcal{E}$
$\mathcal{X}$	Offloading policy vector
$\mathcal{P}_m$	Transmission power
d_m	Distance from mobile device to edge servers
$\mathcal{I}_x$	Interference from other mobile devices
x_i^j	Offloading decision of periodic job $\mu_i^j, \mu_i^j \in \mathcal{U}$
$\mathcal{P}_m^e$	Transmission power of mobile device
$\lambda_{o,e}$	Arrival rate of periodic task at edge server ES_e
$\lambda_{l,s}$	Arrival rate of sporadic task at mobile device

tasks in queues. Alternatively, the computation resources at edge servers also is assumed to be shared with a set of other mobile devices. Thus, the MEC system should manage a set of waiting queues for each independent edge server that is defined as $EQ = \{EQ_1, EQ_2, ...\}$. For each edge server, the arrival pattern of the periodic deadline-sensitive tasks from other devices is portrayed as a Poisson distribution that follows the arrival rate $\lambda_{o,e}, e \in \{1, 2, ...\mathcal{E}\}$ [73]. Furthermore, we assume that the characteristics of the offloaded tasks from other devices, such as the required CPU cycles and data bits, are identical. Without loss of generosity, EDF is leveraged to ensure the deadlines of ready tasks in every waiting queue [105]. To keep consistency with practical scenarios, we assume all edge servers are resource-constrained. Thus, to prevent deadline miss of an offloaded task, it is allowed to push the computation to the cloud server.

4.1.2 Task Model

Each mobile device involves a set of periodic deadline-sensitive tasks $\mathcal{U} = \{\mu_1, \mu_2, \ldots\}$, each of which continuously repeats its operations with a fixed pattern, e.g. periodic traffic monitoring [48]. Particularly, each periodic task $\mu_i, \mu_i \in \mathcal{U}$, can be characterized by a three-tuple: (p_i, c_i, d_i). Note that, task μ_i is repeated periodically. Thus, p_i represents the period of task μ_i. During each p_i, task μ_i is executed once. c_i denotes the required computation resources, which are the CPU cycles required to run the task μ_i with the maximum speed of mobile device f_l (cycles per second) [21]. Since task μ_i is deadline-sensitive, the execution time of task μ_i at mobile device can be c_i/f_l, which is generally shorter than its period (deadline). Otherwise, task μ_i will pile up as time goes by. If periodic task μ_i cannot be completed within period p_i, it is claimed that task μ_i misses its deadline. The last element of the tuple d_i is corresponding to the size of data associated with task μ_i. Besides, we also consider a set of sporadic deadline-sensitive tasks in our model that can be denoted as $\mathcal{V} = \{\nu_1, \nu_2, \ldots\}$. As sporadic tasks are assumed to be processed solely at the mobile device, the data size can be omitted in its notations. Then, each sporadic task $\nu_k, \nu_k \in \mathcal{V}$, is represented as only a two-tuple (c_k, ω_k), where c_k and ω_k is the required CPU cycles and the deadline of task ν_k, respectively. We denote the arrival pattern of sporadic tasks that follow a Poisson Distribution with rate $\lambda_{l,s}$ [73]. For simplicity, the key notations used in this paper are summarized in Table 4.1.

In our model, it is assumed that a periodic task denoted as μ_i initiates its execution at the onset of each period p_i. Each execution instance of task μ_i is regarded as a job associated with task μ_i. Note that, in practice, tasks tend to have different periods. To simplify job scheduling, a term named "hyperperiod" is used in our research [106]. Mathematically, the hypeperiod, $\mathcal{H}_p$, is defined as the least common multiple of the periods of the periodic tasks in task set $\mathcal{U}$. This definition guarantees that, within each hyperperiod, each periodic task is executed at least once. Formally, the hyperperiod for task set $\mathcal{U}$ can be calculated using Eq. (4.1):

$$\mathcal{H}_p = LCM(p_1, p_2, ...), \tag{4.1}$$

where $LCM(\cdot)$ denotes the function that calculates the least common multiple of a series of numbers. For example, if task set $\mathcal{U}$ includes 3 periodic tasks and their periods are 2, 3, and 4 seconds respectively, then the corresponding hyperperiod is 12 seconds.

During each hyperperiod, task μ_i are executed $\mathcal{H}_p/p_i$ times. Thus, the total number of periodic tasks in a hyperperiod can be given by

$$\hat{\mathcal{Z}}_p = \sum_{\mu_i \in \mathcal{U}} (\mathcal{H}_p/p_i). \tag{4.2}$$

In our research, let μ_i^j denote the j-th job of task μ_i in a hyperperiod. Similarly, one sporadic task can be invoked multiple times in a hyperperiod, the n-th arrival of this task is denoted as ν_k^n. Unless stated otherwise, we use the terms job and task interchangeably. Furthermore, we define $\mathcal{J}_l$ and $\mathcal{J}_o$ as the job sets to store the periodic jobs for local processing and edge computing, respectively, while $\mathcal{J}_s$ denotes the set of sporadic jobs that arrive at mobile system following the arrival rate λ_l during one hyperperiod. It is noteworthy that $\mathcal{J}_l$ and $\mathcal{J}_o$ are cleared out after switching to a new hyperperiod as all periodic jobs have reached the deadlines. Let $\mathcal{J}_{o,e}$ denote the job set that contains the jobs being offloaded to ES_e, then we have $\mathcal{J}_o = \mathcal{J}_{o,1} \cup, ..., \mathcal{J}_{o,e} \cup, ..., \mathcal{J}_{o,\mathcal{E}}$. Let $\mathcal{J}_f$ denote the set of incomplete sporadic jobs in the last hyperperiod. Therefore, the total job set of local processing $\mathcal{J}_{to}$ is a combination of $\mathcal{J}_s$, $\mathcal{J}_f$, and $\mathcal{J}_l$. At the beginning of each hyperperiod, the mobile system only has information about $\mathcal{J}_l$, $\mathcal{J}_o$, and $\mathcal{J}_f$. Then, after the periodic jobs are assigned to either $\mathcal{J}_l$ or $\mathcal{J}_o$ according to a specific partitioning algorithm, the instant system workload of the local device at this time point can be calculated using Eq. (4.3):

$$\Upsilon_l = \frac{\sum_{\mu_i^j \in \mathcal{J}_l} c_i^j}{f_l \cdot \mathcal{H}_p} + \Upsilon_l^s, \tag{4.3}$$

where c_i^j is equal to c_i that represents the required CPU cycles of periodic job μ_i^j, and Υ_l^s denotes the workload generated by job set $\mathcal{J}_f$ that can be given by

$$\Upsilon_l^s = \sum_{\nu_k^n \in \mathcal{J}_f} c_k^n / f_l. \tag{4.4}$$

Note that, this workload will be re-accessed during a hyperperiod in the conditions that one periodic job is completed or one sporadic is invoked.

At the edge tier, we assume that the length of hyperperiods of other mobile devices associated to ES_i can be different. Then, the immediate workload of edge server ES_e at the beginning of each local hyperperiod, $\Upsilon_{o,e}(e \in \mathcal{E})$, can be modeled as

$$\Upsilon_{o,e} = \frac{\sum_{\mu_i^j \in \mathcal{J}_{o,e}} c_i^j}{f_e \cdot \mathcal{H}_p} + \Upsilon_{o,e}^p. \tag{4.5}$$

In E.q. (4.5), f_e is the maximum operating frequency at the edge server. $\Upsilon_{o,e}^p$ indicates the remaining workload of the last hyperperiod, which was offloaded by other associated mobile devices to ES_e. This workload can be formulated as follow:

$$\Upsilon_{o,e}^p = \sum_{\mu_i^j \in \hat{\mathcal{J}}_{o,e}^p} c_i^j / f_e, \tag{4.6}$$

where $\hat{\mathcal{J}}_{o,e}^p$ is a job set that includes the incomplete periodic jobs of ES_e in the last hyperperiod. Similar to the local device, the edge server is to re-estimate its instantaneous workload once there is either a job completion or arrival in system.

4.1.3 Communication Model

To offload a periodic job from mobile device to edge servers, transmission delay is generated in a 6G wireless channel with millimeter-wave communication [77]. In our offloading model, we assume the transmission rates during uplink and downlink follow Shannon-Hartley Theorem. Let $\mathcal{B}_u$ denote the wireless bandwidth. $\mathcal{P}_m$ and $\mathcal{G}_m$ are referred to the transmission power and channel gain. $\mathcal{C}$ is the noise power. I_x denotes the interference of other mobile devices and servers. Then, the uplink rate of mobile devices can be calculated as follows:

$$R_u = \mathcal{B}_u \cdot \log_2(1 + \frac{\mathcal{P}_m \cdot \mathcal{G}_m}{\mathcal{C} + I_x}). \tag{4.7}$$

In Eq. (4.7), the channel power gain $\mathcal{G}_m$ can be given by

$$\mathcal{G}_m = L_m^{e \cdot -\aleph}, \tag{4.8}$$

where L_m^e denotes the distance from mobile device to edge server, and $\aleph$ is the path loss exponent [12].

Since the size of the computation results from remote servers is much smaller than that of the uploading data, it is reasonable to assume that this communication is delay-free, especially when that amount of data is transferred in a 6G network, which is expected to be 100 times faster than the current wireless networks [103]. Thus, we do not consider the downlink rate for the transmission of computation results from edge server to mobile.

To generate an offloading policy for each periodic job, we define $\mathcal{X} = \{x_1^1, ..., x_i^j, ...\}$ as the $\hat{\mathcal{Z}}_p$ dimensional offloading decision vector in local device. Particularly, for each offloading decision x_i^j, we have

$$x_i^j = \begin{cases} e, & \text{if } \mu_i^j \text{ is offloaded to } ES_e, e \in \mathcal{E} \\ 0, & \text{otherwise.} \end{cases} \tag{4.9}$$

Relying on the decision vector, all periodic jobs in a hyperperiod are dispatched to the corresponding job sets for further processing.

4.1.4 Completion Time Model

The objective of the proposed scheme is to minimize the job completion time and the number of deadline violations. Targeting on this goal, the proper offloading decisions should be cautiously made by the local scheduler when considering the delay incurring the offloading of a job. For simplicity, we make two assumptions for edge and cloud computing. First, we assume both edge and cloud servers can provide faster computation services compared with mobile devices. In addition, both transmission rate R_u and the arrival rates $(\lambda_l, \lambda_{o,1}, ...)$ are invariable during a hyperperiod. Based on the assumptions, given an offloading vector $\mathcal{X}$, each periodic job can be executed either at mobile device or one of the edge servers. In this section, two completion time models are illustrated as follows.

Completion time at Mobile Device

Let $TC_i^{j\to l}$ be the completion time of a periodic job μ_i^j that is allocated to mobile device according to $\mathcal{X}$. Without loss of generality, we assume that only one processor with limited computation capability is embedded in mobile device. Therefore, one periodic job cannot be scheduled immediately at its arrival time if existing the deadline-sensitive jobs with higher priority in local computation queue Q_l. Due to this resource competition mechanism, $TC_i^{j\to l}$ includes two components: local queuing time $TQ_i^{j\to l}$ and local computation time $TP_i^{j\to l}$. Let $\Phi_{Q_l}(\mu_i^j)$ denote a job set that contains the deadline-sensitive jobs that have a higher priority than job $\mu_i^j(\mu_i^j \in \mathcal{J}_l)$ at its arrival time point. Given the local operating frequency, f_l, the local queuing time of μ_i^j is calculated using Eq. (4.10):

$$TQ_i^{j\to l} = \sum_{\mu_m^b \in \Phi_{Q_l}(\mu_i^j)} \sum_{\nu_z^v \in \Phi_{Q_l}(\mu_i^j)} \frac{c_m^b + c_z^v}{f_l}. \tag{4.10}$$

After job μ_i^j is scheduled successfully, the corresponding computation time $TP_i^{j\to l}$ is calculated using Eq. (4.11):

$$TP_i^{j\to l} = c_i^j / f_l. \tag{4.11}$$

To this end, the completion time of job $\mu_i^j(\mu_i^j \in \mathcal{J}_l)$ is the summation of its local queuing and processing time that is

$$TC_i^{j\to l} = TQ_i^{j\to l} + TP_i^{j\to l}. \tag{4.12}$$

Similarly, for a sporadic job, it also needs to wait for its scheduling window after arriving at systems. Therefore, the completion for a sporadic job TC_k^n at the local device can be given by

$$TC_k^n = \Big(\sum_{\mu_m^b \in \Phi_{Q_l}(\nu_k^n)} \sum_{\nu_z^b \in \Phi_{Q_l}(\nu_k^n)} \frac{c_m^b + c_z^v}{f_l} \Big) + \frac{c_k^n}{f_l}, \tag{4.13}$$

where $\Phi_{Q_l}(\nu_k^n)$ is the higher priority job set corresponding to job ν_k^n. For each local job, if it cannot be completed before the deadline, the deadline miss occurs. We define two variables, l_i^j and l_k^n, for the deadline-sensitive jobs that cannot meet the deadline requirements of local and edge processing using Eq. (4.14) and Eq. (4.15):

$$l_i^j = \begin{cases} 1, & \text{if } TC_i^j > p_i^j \\ 0, & \text{otherwise.} \end{cases} \tag{4.14}$$

$$l_k^n = \begin{cases} 1, & \text{if } TC_k^n > z_k^n \\ 0, & \text{otherwise.} \end{cases} \tag{4.15}$$

Completion time at Edge Server

Once a periodic job is offloaded to edge server, transfer delay, edge queuing delay, and edge computation delay should be considered. Specifically, after offloading decisions have been made by local scheduler, all the offloading jobs $\mu_i^j (\mu_i^j \in \mathcal{J}_{o,e}, e \in \mathcal{E})$ are executed at the associated edge server ES_e. That means, the necessary data of these jobs need to be transmitted to the designated edge server for further processing. Provided that the size of computation results is generally small, it is reasonable to ignore this amount of transfer delay. Once one offloaded job μ_i^j finishes its work at edge server, a signal is returned to the mobile device to indicate whether this job has missed the deadline. Due to the limited computation capacity of edge servers, the offloaded jobs are allowed to be forwarded to the cloud server if the associated edge server becomes overloaded. For the reasons, we have two communication scenarios that are discussed as follows.

In the first scenario, we assume that the servicing edge server ES_e has sufficient computation resources for the offloaded jobs. Let $TT_i^{j \to e}$ denote the transfer delay of an offloaded job μ_i^j from the local device to edge server ES_e that includes two parts (transmission delay and propagation delay). The former is incurred during the data transmission for a job, while the latter can be omitted at edge computing as mobile device and edge server are generally located close to each other. Then, the transfer delay of an offloaded job can be computed by

$$TT_i^{j \to e} = \frac{d_i^j}{R_u^e} + \sum_{e=1}^{\mathcal{E}} \sum_{\mu_m^b \in \Phi_{Q_o}(\mu_i^j)} \frac{d_m^b}{R_u^e} \tag{4.16}$$

where R_u^e indicates the instant transmission rate of edge server ES_e, and $\Phi_{Q_o}(\mu_i^j)$ is the priority job set referred to job μ_i^j in offloading queue Q_o.

As the edge servers are resource-constrained, the limited resources need to be competed by all the associated mobile devices. Then, an offloaded job μ_i^j cannot be scheduled immediately until all the predecessor jobs complete their work. Let $TQ_i^{j\to e}$ denote the queuing time of job μ_i^j at edge server ES_e. The length of $TQ_i^{j\to e}$ is determined by the realtime workload of the edge system that has been discussed in section 4.1.2, and the arrival rate of edge server ES_e, $\lambda_{o,e}$. Another delay is the computation delay that is denoted as $TP_i^{j\to e}$. The length of $TP_i^{j\to e}$ is strongly relevant to the processor speed of the edge server (f_e) that can be computed by Eq. (4.17).

$$TP_i^{j\to e} = \frac{c_i^j}{f_e}, \tag{4.17}$$

To this end, the total completion time of $\mu_i^{j\to e}$ at this scenario can be given by

$$TC_i^{j\to e} = TT_i^{j\to e} + TQ_i^{j\to e} + TP_i^{j\to e}, \tag{4.18}$$

Remark 1. The edge servers are generally configured with more CPU cores, consequently shortening the queuing time of the offloaded jobs by dispatching these jobs into different cores and then processing them in a parallel manner. In addition, the processor speed of edge server is much faster than local device, eventually resulting in less computation time ($TP_i^{j\to e} \ll TP_i^{j\to l}$).

The second scenario is that the resources of edge servers become deficient. In this case, inappropriate offloading decisions for the deadline-sensitive jobs can lead to deadline violation. To reduce the deadline misses, deadline-sensitive jobs are allowed to forward to cloud server. Thus, the completion time of the offloaded jobs consists of three components that are discussed as follows. The first component is the transfer delay from the mobile device to edge server ES_e, $TT_i^{j\to c}$, which can be calculated by Eq. (4.20). Note that, one offloaded job should be shifted to the corresponding edge server first before forwarding to the cloud server. Second, the round-trip propagation delay between edge server ES_e and cloud server is considered because these two kinds of servers are physically deployed far away from each other that leads to a long propagation delay. We assume the round-trip delay is a constant in our model that is represented as $2 \cdot \mathcal{L}_c$. The last component is the computation time at cloud server,

which is denoted as $TP_i^{j\to c}$. Thus, we can compute the total completion time of a periodic job $\mu_i^j(\mu_i^j \in \mathcal{J}_{o,e})$ in this scenario by using Eq. (4.19), Eq. (4.20), and Eq. (4.21).

$$TC_i^{j\to c} = TT_i^{j\to c} + TP_i^{j\to c} + 2\cdot\mathcal{L}_c, \tag{4.19}$$

where $TT_i^{j\to c}$, $TP_i^{j\to c}$ are calculated as follows:

$$TT_i^{j\to c} = 2\cdot\mathcal{L}_c + TT_i^{j\to e}, \tag{4.20}$$

$$TP_i^{j\to c} = \frac{d_i^j}{f_c}, \tag{4.21}$$

where f_c denotes the maximum speed of cloud server, and $TT_i^{j\to e}$ can be obtained by using Eq. (4.16). Similar to the local computation, the deadline variable of the offloaded jobs is determined by Eq. (4.14).

Remark 2. As the computation capability of cloud server is sufficient because it has more CPU cores and faster processor speed ($f_c \gg f_e$), none of the offloaded jobs need to wait for computation resources. Therefore, the queuing time in cloud server $TQ_i^{j\to c}$ is negligible.

4.1.5 Problem Formulation

To appropriately support computation-intensive and deadline-sensitive applications in 6G-based MEC systems, we integrate the models mentioned previously to find the appropriate offloading strategies so that the minimum completion time of a set of periodic deadline-sensitive tasks is achieved.

Definition 1. Penalty Completion Time (PCT). Let $\hat{\mathcal{J}}$ be the set of periodic jobs that violate the deadline constraints in a hyperperiod. That is $l_i^j = 1, \forall\mu_i^j \in \hat{\mathcal{J}}$. The overall PCT is defined as the sum of the worst-case completion time of these jobs,

$$\Pi_n = \sum_{\mu_i^j\in\hat{\mathcal{J}}} p_i^j. \tag{4.22}$$

Definition 2. Total Completion Time (TCT). Given the PCT and job set $\hat{\mathcal{J}}$ in a hyperperiod, the TCT for a set of periodic jobs can be given by

$$\Pi = \Pi_n + \sum_{\mu_i^j \in \{\mathcal{J}_l, \mathcal{J}_o / \hat{\mathcal{J}}\}} TC_i^j. \tag{4.23}$$

To this end, the computation offloading problem in our model, subjecting to a set of constraints, can be formulated as follows:

$$\begin{aligned} \text{OP}_{\text{Mobile}} : \min_{\mathcal{X}} \quad & \Pi \\ \text{subject to} \quad & C1 : x_i^j \in \{0, 1, ..., \mathcal{E}\}, \forall \mu_i^j \in \mathcal{U}, \\ & C2 : f_l \ll f_e \ll f_c, \\ & C3 : \Upsilon_l, \Upsilon_{o,e} < 1, \forall e \in \mathcal{E}, \\ & C4 : TC_i^{j \to l}, TC_i^{j \to e}, TC_i^{j \to c} < p_i^j, \forall \mu_i^j \in \mathcal{J}_l, \mathcal{J}_o, \\ & C4 : TC_k^n < z_k^n, \forall \nu_k^n \in \mathcal{J}_s, \mathcal{J}_f. \end{aligned} \tag{4.24}$$

The constraints in the formulation are illustrated as follows: $C1$ indicates that each periodic job can only choose to be executed either locally or offloaded to one of the edge servers. $C2$ guarantees that the CPU speed of cloud server is much faster than the speed at edge servers, as well as the speed of edge servers is faster than that of mobile device. $C3$ makes sure the instant workloads of mobile device and edge servers cannot exceed their computation capacity. $C4$ and $C5$ guarantee that the completion time of each job of mobile device cannot surpass its deadline. This optimization problem is computationally hard to solve. In our research, we attempted to employ a deep reinforcement learning method, TD3, to find a satisfactory solution.

4.2 Edge-assisted DRL-based Offloading Scheme

In our model, the offloading policy is closely related to the communication cost between mobile device and edge server, the available computation resources of mobile device and edge servers, the arrival rate of sporadic jobs of mobile device as well as the job arrival rates of each edge server. These system properties tend to fluctuate due to the dynamic nature of the MEC environment. Therefore, it is hard to find

an appropriate solution for the offloading problem $\mathrm{OP}_{\mathrm{Mobile}}$ in a polynomial time with the traditional optimization methods. Furthermore, the existing computation offloading schemes are impractical as they do not consider the resources consumption at mobile tier for the training of complicated deep learning model [17]. To address these challenges, we first present a novel architecture for the edge-assisted learning, called EALA, which attempts to reclaim more mobile computation resources for the execution of local deadline-sensitive jobs through deploying the learning components at edge tiers and executing these components with the delegated core of edge servers. Then, we propose a TD3-based computation offloading scheme, MELO, to optimize the completion time of a set of deadline-sensitive jobs by leveraging the proposed EALA to train the near-optimal offloading decision model.

4.2.1 Edge-Assisted Learning

EALA aims to leverage the computation resources of edge servers to alleviate the burden of mobile device. Instead of learning the offloading model at model device, EALA decouples its inference and training. In other words, the inference is carried out at mobile device, while the training process is conducted at edge server. In EALA, mobile devices store one policy actor $\theta_m^{\emptyset}$ that is responsible to issue the offloading decisions referred to the immediate states s^t. Alternatively, edge servers are configured with one actor network $\theta_e^{\emptyset}$, one actor target $\theta_e^{\emptyset'}$, two critic networks (θ_e^{Q1} and θ_e^{Q2}), and two related target networks ($\theta_e^{Q1'}$ and $\theta_e^{Q2'}$). To train TD3 model, the interaction (s^t, a^t, r^t, s^{t+1}) from mobile device is delivered to edge server and then stored into a replay buffer $\mathcal{M}_e$. For each episode δ_e, TD3 agent of edge servers will randomly choose a mini-batch of interaction samples from $\mathcal{M}_e$ for the model updating. After completing this update, the parameters of the edge actor $\theta_e^{\emptyset}$ is returned to the mobile device ($\theta_m^{\emptyset} \leftarrow \theta_e^{\emptyset}$). The framework of EALA is shown in Fig. 4.2.

With EALA, the entire learning process is conducted at edge server, which leads to a significant complexity reduction at mobile device. Furthermore, with a faster processor at edge servers, especially the servers configured with TPU, the training can be greatly accelerated. Thus, the time complexity for a training episode δ_e at mobile device comprises two parts. The first one is communication time complexity,

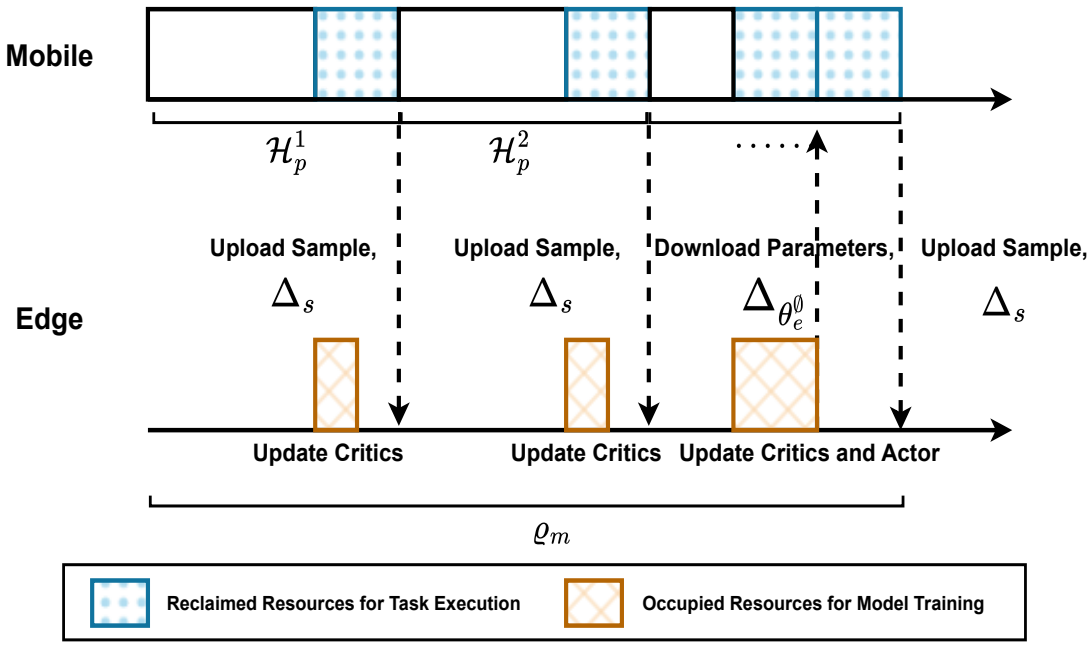


Figure 4.2: EALA: Training TD3 on Edge Server.

which is determined by transmission time of samples being offloaded to edge server and the time the parameters of $\theta_e^\emptyset$ being downloaded to the mobile device. Let Δ_s and $\Delta_{\theta_e^\emptyset}$ denote the data size of sample (s^t, a^t, r^t, s^{t+1}) and the parameters of edge actor $\theta_e^\emptyset$, respectively. N_d denotes the number of forward-propagation of $\theta_m^\emptyset$ during a training episode δ_e. Note that, we assume the uplink rate is equal to the downlink rate, $R_d = R_u$. Then, the time complexity for communication depends on the number of the samples being offloaded to the edge server is $O(\frac{\Delta_s \cdot N_d}{R_u} + \frac{\Delta_{\theta_e^\emptyset}}{R_d})$. The second part is the computation time complexity that depends on the number of forward-propagation of mobile actor denoted as N_f and the number of neurons of mobile actor network N^*. Then, the computation time complexity of mobile device for EALA is $O(N_f \cdot N^*)$. Suppose the model can converge to the best policy after learning for N_i episodes. The total time complexity of learning at mobile device is $O_m = O(\frac{\Delta_s \cdot N_d \cdot N_i}{R_u} + \frac{\Delta_{\theta_e^\emptyset} \cdot N_i}{R_d} + N_f \cdot N^* \cdot N_i)$. In the next generation of 6G-based MEC system, we can assume $\Delta_s, \Delta_{\theta_e^\emptyset} \ll R_d$ [76, 96]. It is reasonable to ignore the time complexity for data transmission of samples and parameters. For example, one actor policy network consists of one input layer, two hidden layers, and one

output layers. We also assume that the corresponding number of neurons of layers are 20, 1024, 1024, 20, respectively. The size of each parameter is 32 bits. Then, the total size of parameters of this neural network is 34.85 Mb. We assume that the downlink transmission rate is 100 Mbps. The transmission time is $34.85/100 = 0.34$ seconds. Hence, it is reasonable to ignore the time complexity for the data transmission of samples and parameters as only a small amount of data need to be tranmitted with a tiny cost in the high-speed 5G/beyong 5G networks [76, 96]. Additionally, it is noteworthy that the partially-trained parameters of neural networks can be transferred to the mobile device upon completion of each hyperperiod update. This allows sufficient time for the transmission of policy network parameters to the mobile device before it starts subsequent decision-making processes. Therefore, it can be concluded that the time complexity of EALA for the mobile device can be lowered down to $O(N_f \cdot N^* \cdot N_i)$.

4.2.2 Modelling Task Offloading as an MDP

To adopt DRL to solve our computation offloading problem, we first need to model the offloading problem as an Markov Decision Process (MDP). As discussed in section 4.2.1, the MELO scheme comprises a local policy network situated at the mobile device and a global learning model deployed at the edge tier. The basic elements at t-th heperperiod of TD3 are defined as follows:

- State: To decide whether offloading a set periodic jobs of a hyperperiod, we should consider a set of environment factors at each decision point, including realtime workload of mobile and edge server $\mathcal{W}^t = \{\Upsilon_l^t, \Upsilon_{o,1}^t, \Upsilon_{o,2}^t, ..., \Upsilon_{o,e}^t, ...\}$, the arrival rates of jobs at the local device and edge servers $\Theta^t = \{\lambda_{l,s}^t, \lambda_{o,1}^t, ..., \lambda_{o,e}^t, ...\}$, the transmission rates of edge servers $\mathcal{R}^t = \{\mathcal{R}_{u,1}^t, ..., \mathcal{R}_{u,e}^t, ...\}$, and the tasks profile $\mathcal{I}^t = \{\mathcal{I}_l^t, \mathcal{I}_{o,1}^t, ..., \mathcal{I}_{o,e}^t\}$, $e \in \mathcal{E}$. Particularly, $\mathcal{W}^t$ can be obtained by using Eq. (4.3) and Eq. (4.5). The status of arrival rates, Θ^t, are captured periodically by the mobile device. The instant transmission rates of edge servers, $\mathcal{R}^t$, are calculated as Eq. (4.7). Then, the system state of t-th hyperperiod can be

defined as:

$$\mathcal{S} : s^t = \{\mathcal{W}^t, \Theta^t, \mathcal{R}^t, \mathcal{I}^t\}, t \in \mathbb{N}. \tag{4.25}$$

- Action: For each of periodic job μ_i^j that is allowed to being processed locally or at either of edge servers, $1 + \mathcal{E}$ choices are available for its execution. At the start of each hyperperiod, we have the number of $\hat{\mathcal{Z}}_p$ periodic jobs. Then, the action is a decision vector that can be defined as

$$\mathcal{A} : \mathcal{X}^t = \{x_1^{1,t}, ..., x_i^{j,t}, ...\}. \tag{4.26}$$

- Reward Function: As the target of our offloading model is to minimize the completion time of mobile jobs with hard deadlines, the minimum reward should be granted along with the deadline guarantee of these jobs. If a job (μ_i^j or ν_k^n) can be completed successfully before its deadline, the reward is defined as its actual completion time. Otherwise, the reward is assigned to a larger value. Thus, the reward function $\mathcal{R}^t$ is formulated as

$$\mathcal{R}^t : r^t = \begin{cases} \Pi^t, & \text{if no deadline misses,} \\ r^{*,t}, & \text{Otherwise,} \end{cases} \tag{4.27}$$

where $r^{*,t}$ is the reward for the inappropriate offloading policy that leads to deadline misses. To punish the improper offloading policies, a larger value is assigned to the reward $r^{*,t} = (\Pi^t)^{(1+\kappa\cdot\varsigma^t)}$, where κ is a coefficient, and ς^t denotes the miss rate for both periodic and sporadic deadline-sensitive jobs in t-th hyperperiod that can be formulated as follows:

$$\varsigma^t = \frac{\sum_{\mu_i^j \in \mathcal{J}_l^t, \mathcal{J}_o^t} l_i^j + \sum_{\nu_k^n \in \mathcal{J}_s^t, \mathcal{J}_f^t} l_n^k}{\hat{\mathcal{Z}}_p + |\mathcal{J}_s^t| + |\mathcal{J}_f^t|}. \tag{4.28}$$

We define the computation offloading policy as $\pi_{\theta_m^\emptyset}(s^t|\mathcal{X}^t)$. Then, the near-optimal policy $\pi^*_{\theta_m^\emptyset}(s^t|\mathcal{X}^t)$ can be learned through maximizing the negative discounted long-term cumulative reward $L(\pi_{\theta_m^\emptyset})$ that can be defined as follows:

$$\pi^*_{\theta_m^\emptyset}(s^t|\mathcal{X}^t) = \arg\max_{\pi} -L(\pi_{\theta_m^\emptyset}), \tag{4.29}$$

where $L(\pi_{\theta_m^\emptyset}) = E(s^t, \mathcal{X}^t)_{\sim\rho_{\pi_{\theta_m^\emptyset}}}[\sum_{i=t}^{\infty} \kappa^{i-t} \cdot r^i]$. $\rho_{\pi_{\theta_m^\emptyset}}$ denotes the state-action mapping inducing by the policy $\pi_{\theta_m^\emptyset}$ and $\kappa \in [0, 1)$ represents the discounted factor.

4.2.3 Details of MELO

In this section, we present the implementation of the proposed MELO scheme. Given a task set $\mathcal{U} = \{\mu_1, \mu_2, ...\}$, this study attempts to select the best offloading policy $\mathcal{X} = \{x_1^1, x_2^1, ...\}$ for the number of $\hat{\mathcal{Z}}_p$ periodic jobs in a hyperperiod. The aim of this study is to minimize the completion time of these periodic jobs at mobile device while the corresponding time constraints can be met. To achieve this goal, an TD3 based offloading scheme, MELO, is devised to make the optimal offloading decisions by using neural networks. In addition, a new learning architecture, which takes advantage of the powerful edge servers and the advanced 6G networks, is developed to enhance the scalability of MELO. The data flows of MELO are illustrated in Fig. 4.3. From the figure, it indicates that our offloading model involves two key tiers that are described as follows:

- At the mobile tier, the offloading scheduler is committed to map environment states s^t to a policy $\mathcal{X}^t$ via mobile actor network $\theta_m^{\emptyset}$ at the start of t-th hyperperiod. Aligned with the policy $\mathcal{X}_t$, each periodic job is either allocated to mobile device or one of the edge servers. Then, the periodic jobs ($\mu_i^j \in \mathcal{J}_l$) and the detained sporadic jobs in the $(t-1)$th hyperperiod ($\nu_k^n \in \mathcal{J}_f^t$) are co-scheduled by the local processor based on the EDF algorithm. When new sporadic jobs are coming during the hyperperiod, the total jobs in the local processor are re-scheduled accordingly. At the end of each hyperperiod, a training sample is forwarded to replay buffer at edge server.

- At the edge tier, each edge server is responsible for managing the jobs from all associated mobile devices properly, including offering comprehensive scheduling and allocating sufficient computation resources, and forwarding extra jobs to cloud server. More importantly, edge servers are also committed to offer training services for mobile devices. Namely, for each episode δ_e, the learning agent of edge server needs to randomly select a batch of training samples from replay buffer $\mathcal{M}$, and then update the policy actor $\theta_e^{\emptyset}$ by using TD3. Once $\theta_e^{\emptyset}$ is renewed, a copy of its parameters is sent back to the mobile device.

Notice that, once the mobile actor $\theta_m^{\emptyset}$ is updated in light of $\theta_e^{\emptyset}$, the offloading decision can be made through mapping the state to the relevant offloading policy.

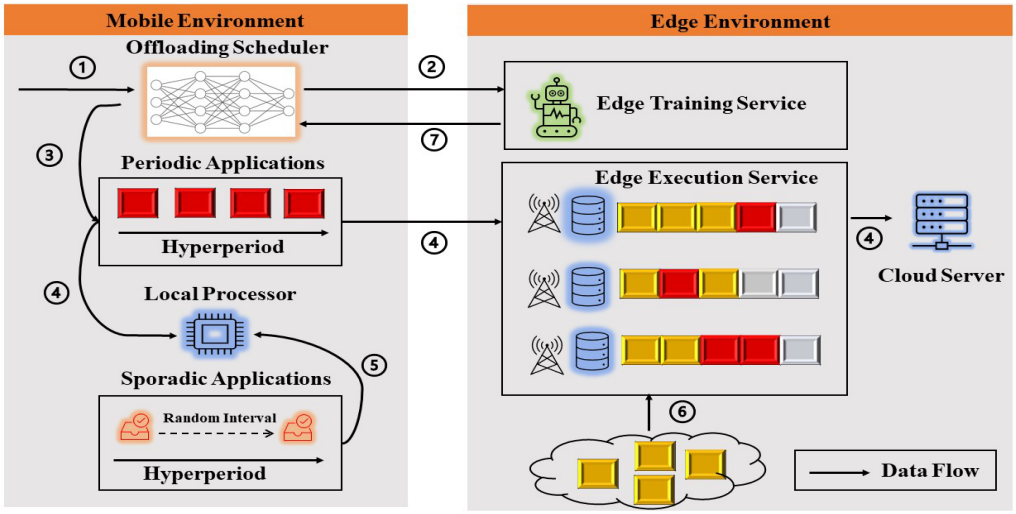


Figure 4.3: Architecture of MELO. The Data that Flow in the MEC System Include: 1) Environment States; 2) Training Samples; 3) Offloading Policy; 4) Periodic Jobs; 5) Sporadic Jobs; 6) Periodic Jobs from Other Mobile Devices; 7) Parameters of Edge Actor Network.

Alg. 4 includes the detailed steps in MELO. Specifically, MELO takes a specified periodic task set $\mathcal{U}$, a sporadic task set $\mathcal{V}$, and information of edge environment as the input and outcomes with the near-optimal offloading policy in $\mathcal{X}$. Before heading into the training phase, MELO agent first needs to initialize the neural networks at both mobile and edge tier ($\theta_e^{\emptyset}$, $\theta_e^{\emptyset'}$, θ_e^{Q1}, θ_e^{Q2}, $\theta_e^{Q1'}$), and the Replay Buffer ($\mathcal{M}$) at edge tier (Line 1). Meanwhile, mobile device downloads a copy of the parameters of $\theta_e^{\emptyset}$ and then assigns the downloaded parameters to its own actor network $\theta_m^{\emptyset}$. At the beginning of t-th hyperperiod, the reward in the last hyperperiod r^{t-1} is calculated by using Eq. (4.26) (Line 6). Afterward, the mobile agent observes the environment state s^t and then uploads the transaction sample $(s^{t-1}, \mathcal{X}^{t-1}, r^{t-1}, s^t)$ to the Replay Buffer $\mathcal{M}$(Line 7-8). Relying on the mobile policy network $\theta_m^{\emptyset}$, the offloading policy $\mathcal{X}^t$ can be obtained as follow:

$$\mathcal{X}^t = \pi_{\theta_m^{\emptyset}}(s^t) + \epsilon, \tag{4.30}$$

where ϵ denotes the noise for exploration purpose. According to $\mathcal{X}^t$, each job μ_i^j is allocated to two job subsets $\mathcal{J}_l$ and $\mathcal{J}_o$ (Line 10). For every time episode δ_e, MELO will take several steps to update its neural networks (Line 13-19). As EALA is one re-organized version of TD3, it is applied to the same update mechanism that is detailed in the related work [29]. First, it samples a minibatch of N transitions $(s^n, \mathcal{X}^n, r^n, s^{n+1})$ arbitrarily from Replay Buffer $\mathcal{M}$ to train the neural networks. To avoid the overfitting to local optimum, the policy actions $\mathcal{X}^n$ issued by $\pi_{\theta_e^{\emptyset}}$ are smoothing to a small area, which is defined as follows:

$$\mathcal{X}^n : \widetilde{\mathcal{X}^n} \leftarrow \pi_{\theta_e^{\emptyset'}}(s^n) + \epsilon, \epsilon \sim clip(\mathcal{N}(0, \widetilde{\sigma}), -c, c). \tag{4.31}$$

Then, the loss function for the critic networks is formulated as

$$y^n \leftarrow r^n + \gamma \cdot \min(\pi_{\theta_e^{Q'_i}}(s^{n+1})), i = 1, 2. \tag{4.32}$$

Based on the loss function, the critic networks $\theta_e^{Q_1}$, $\theta_e^{Q_2}$ are updated according to the minimum target value of actions that is defined as Eq. (4.33) and Eq. (4.34).

$$\theta_e^{Q1} \leftarrow \theta_e^{Q1} - \alpha \sum_{n=1}^{N} \frac{\mathrm{d}\pi_{\theta_e^{Q1}}(s^n, \mathcal{X}^n)}{\mathrm{d}\theta_e^{Q1}} \cdot (\pi_{\theta_e^{Q1}}(s^n, \mathcal{X}^n) - y^n) \tag{4.33}$$

$$\theta_e^{Q2} \leftarrow \theta_e^{Q2} - \alpha \sum_{n=1}^{N} \frac{\mathrm{d}\pi_{\theta_e^{Q2}}(s^n, \mathcal{X}^n)}{\mathrm{d}\theta_e^{Q2}} \cdot (\pi_{\theta_e^{Q2}}(s^n, \mathcal{X}^n) - y^n) \tag{4.34}$$

After each δ_e hyperperiods, actor network $\theta_e^{\emptyset}$ is updated by using $\pi_{\theta_e^{Q1}}$ with deterministic policy gradient that is given by Eq. (4.35).

$$\theta_e^{\emptyset} \leftarrow \theta_e^{\emptyset} + \beta \cdot \sum_{n=1}^{N} \frac{\mathrm{d}\pi_{\emptyset_e^{Q1}}(s^n, \mathcal{X})}{\mathrm{d}\mathcal{X}} \cdot \frac{\pi_{\theta_e^{\emptyset}}(s^n)}{\mathrm{d}\theta_e^{\emptyset}}. \tag{4.35}$$

Next, the target networks of actor and critics are updated as follows:

$$\theta_e^{\emptyset'} \leftarrow \tau\theta_e^{\emptyset} + (1 - \tau)\theta_e^{\emptyset'}. \tag{4.36}$$

$$\theta_e^{Q'_i} \leftarrow \tau\theta_e^{Q_i} + (1 - \tau)\theta_e^{Q'_i}, i = 1, 2. \tag{4.37}$$

Lastly, the updated $\theta_e^{\emptyset}$ will be sent back to mobile device. The training process is completed until the predefined conditions are satisfied.

Algorithm 4: MELO Offloading.

Input: Periodic task set $\mathcal{U}$, sporadic task set $\mathcal{V}$, edge environment information

Output: Offloading policy $\mathcal{X}^t, \mathcal{X}^t \in \mathcal{X}$

Initialize parameters of neural networks $\theta_e^{\emptyset}$, $\theta_e^{\emptyset'}$, θ_e^{Q1}, θ_e^{Q2}, $\theta_e^{Q1'}$, $\theta_e^{Q2'}$, and Replay Buffer $\mathcal{M}$;

Download a copy of $\theta_e^{\emptyset}$ from the edge server to the mobile device, $\theta_m^{\emptyset} \leftarrow \theta_e^{\emptyset}$;

Calculate the length of $\mathcal{H}_p$ by using Eq. (4.1);

for each $\mathcal{H}_p^t, t = 1, 2, 3...,$ do

  Update the reward r^{t-1} based on Eq. (4.26);

  Observe environmental state $s^t\{\mathcal{W}^t, \Theta^t, \mathcal{R}^t, \mathcal{I}^t\}$;

  Upload the transition $(s^{t-1}, a^{t-1}, r^{t-1}, s^t)$ to edge server and store it into $\mathcal{M}$;

  Select an action $\mathcal{X}^t$ by Eq. (4.30);

  Allocate job $\mu_i^j, \mu_i^j \in \mathcal{U}$ into $\mathcal{J}_l$, $\mathcal{J}_o$ upon $\mathcal{X}^t$;

  Schedule jobs in $\mathcal{J}_l$, $\mathcal{J}_f$, and $\mathcal{J}_s$ with frequency f_l;

  Offload jobs in $\mathcal{J}_o$ to either of edge server ES_i in light of $\mathcal{X}^t$;

  Sample a mini-batch of N transitions $(s^n, \mathcal{X}^n, r^n, s^{n+1})$ from $\mathcal{M}$;

  Smooth action $\mathcal{X}^n$ adopting Eq. (4.31);

  Calculate the loss function of critic networks by using Eq. (4.32);

  Update critic networks θ_e^{Q1} and θ_e^{Q1} by using Eq. (4.33) and Eq. (4.34) ;

  if (t mod $\delta_e = 0$) then

    Update actor network $\theta_e^{\emptyset}$ using Eq. (4.35);

    Update target networks $\theta_e^{\emptyset'}$, $\theta_e^{Q1'}$, and $\theta_e^{Q2'}$ with Eq. (4.36) and Eq. (4.37) ;

    Download a copy of $\theta_e^{\emptyset}$ to mobile device, $\theta_m^{\emptyset} \leftarrow \theta_e^{\emptyset}$;

  end

  Until reach stop conditions;

end

4.3 Evaluation

In this section, we describe the configuration of the experiments and the details of our experimental results.

4.3.1 Evaluation Settings

In our research, we simulate the task model, communication model, completion time model, and EALA design with Pytorch. Specifically, in our simulations, we consider a circular region whose area is $\pi \times 200m \times 200m$. Three to five edge servers are evenly distributed on the edge of the experimental region. Each edge server is connected to one cloud server. The distance from mobile devices to edge servers is set to $L_m^e = 200m$. The corresponding path loss exponent $\aleph$ is set to 2 [12]. We consider a set of periodic deadline-sensitive tasks. Each task consists of its own required computation cycles, period, and size of data. The required computation cycles of a task is randomly selected from the interval $[1, 10]$ G cycles, and the size of data is in the range from 100 Mb to 2Gb. As mobile devices typically tend to execute small and light applications due to lack of sufficient computation resources, the periods of these tasks are set to be small values. Thus, the periods of tasks are randomly generated from the range $[4, 32]$ seconds. The length of hyperperiod for the selected periodic jobs is set to 32 seconds according to Eq. (4.1). In addition, a 500 Mhz wireless bandwidth network is considered in our model. The computation speeds of mobile device, edge servers, and cloud server (f_l, f_e, and f_c) are set to 1G, 8G, and 20G cycles/s, respectively. The arrival rate $\lambda_{l,s}$ and the arrival rate of each edge server $\lambda_{o,e}$ are set to the interval $[0.01, 1.0]$ job/s. To obtain the transmission rates of edge servers via Eq. (4.7), the interference $\mathcal{I}_x$ is set to the interval $[10^{-6.95}, 10^{-8.78}]$, which leads the transmission rates into the range $[50, 1000]$ Mbps. In the model, the transmission power $\mathcal{P}_m$ is set to 64 mWatt. The implementation of the learning module is similar to that of our previous study [40]. Specifically, both the actor and critic networks consist of 2 hidden layers, where the related number of neurons is 256. The learning rate and discount rate are set to 0.0003 and 0.99, respectively. Besides, the mini-batch of training samples is set to 128. The length of episode δ_e is set to 100 x $\mathcal{H}_p$.

We compare MELO with the following baseline methods:

(i) Local Computing (LC):Both periodic and sporadic jobs are executed on mobile devices during the entire period. Namely, no job is offloaded to edge or cloud servers.

(ii) Random Offloading (RO): Part of periodic jobs are offloaded to edge servers in a random fashion, while other jobs are performed on mobile devices.

(iii) Server Computation Learning-based Offloading (SCLO): SCLO is similar to MELO. It also learns the TD3-based decision model at edge servers. The difference between these two offloading schemes is that SCLO attempts to offload all the jobs to the edge servers. Namely, there are no periodic jobs being executed at the local device. Consequently, each job should pick either of the edge servers for its execution.

(iv) Multi-access Mobile-assisted Learning-based Offloading (MMLO): Similar to MELO, MMLO generates the optimal offloading decisions with the TD3-based offloading model. However, instead of training at the edge tier, MMLO places the training operations at the mobile device. For this reason, mobile device needs to allocate a portion of computation resources to the update of neural networks. In our simulations, mobile device reserves a predefined amount of resources for network training at the end of each hyperperiod. The corresponding required cycles for the training are set to $[5, 10]$ G cycles.

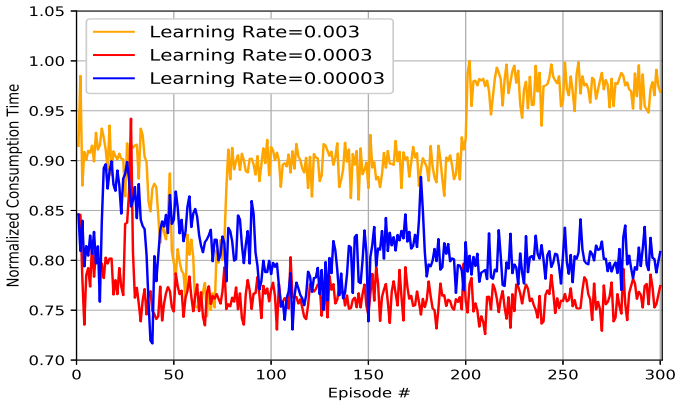
In our research, two metrics are used to measure the performance of MELO and other baseline methods. Specifically, the first metric is the total completion time of periodic jobs in a hyperperiod, which is normalized using min-max normalization and can be obtained using Eq. (4.23). The other metric is the number of deadline misses during a hyperperiod that can be calculated using Eq. (4.28).

4.3.2 Convergence

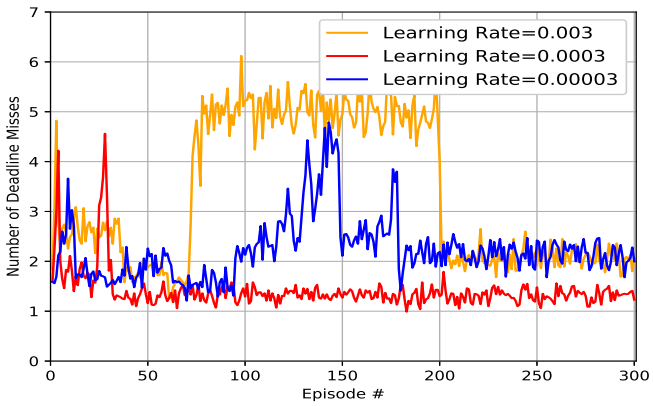
In this section, we illustrate the convergence of MELO under different learning rates. In the experiments, the learning rate is set to 0.003, 0.0003, and 0.00003, respectively. Furthermore, there are 10 periodic tasks (33 jobs) being executed in the MEC system. The wireless transmission rate is set to a value in the range of 400 Mbps to 600 Mbps. The local arrival rate $\lambda_{l,s}$ is in the range $[0.01, 0, 05]$ job/s, and the arrival rate $\lambda_{e,o}$ for each edge server is set to 0.05 job/s. Fig.4.4 shows the curve in terms of the normalized completion time and the number of deadline misses for the specified learning rates. Apparently, when learning rate is equal to 0.003 and 0.00003, MELO consumes considerable training to reach its convergence. More importantly, the learning model cannot learn the most proper offloading policy. Alternatively, when the learning rate=0.0003, MELO can converge to the near-optimal policy after being trained for roughly 50 episodes. Therefore, learning rate = 0.0003 that has better training efficiency, is leveraged in our following experiments.

4.3.3 Impact of Transmission Rates

In this section, we first investigate the impact of transmission rates on MELO and the baseline methods. In the experiments, two scenarios are considered: a relatively light workload of edge servers,$\lambda_{o,e} = 0.5$ as well as a near-overloaded workload of edge servers $\lambda_{o,e} = 1.0$. In addition, 10 periodic tasks (33 jobs) are involved. The transmission rate of sporadic tasks at mobile device is set to 0.05 s/job, and the coefficient κ is set to 1. Fig. 4.5 and Fig. 4.6 illustrate the effectiveness of MELO under different average transmission speeds in both scenarios. From Fig. 4.5, we can observe that the total completion time declines dramatically when the average transmission rates are speeded up from 100 Mbps to 500 Mbps, but only a slight improvement can be achieved after the transmission rate is greater than 700 Mbps. Moreover, the number of deadline misses also drops sharply as the transmission rate becomes faster. When the transmission rate is faster than 700 Mbps, the miss rates of the offloaded jobs for two learning-based counterparts lie in a satisfactory range from 6.1% (MELO)



(a) Completion Time vs. Learning Rates

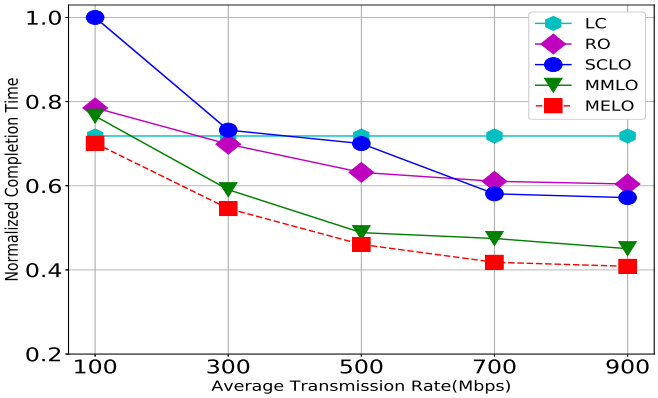


(b) Deadline Miss vs. Learning Rates

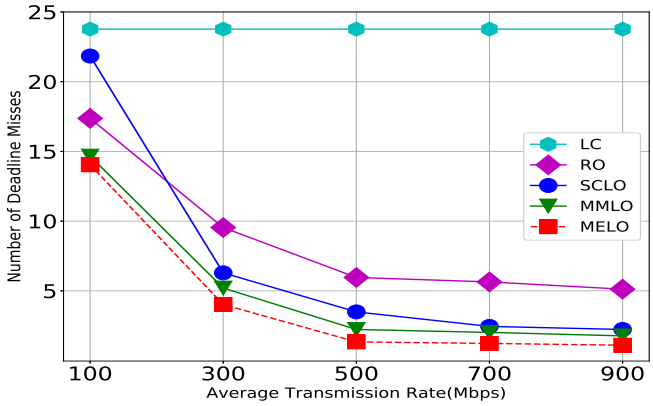
Figure 4.4: Convergence of MELO

to 7.4%(SCLO). This observation indicates that the system performance can be enhanced via increasing the transmission speed.

However, when the workload of edge servers nearly reach its maximum capacity, e.g., $\lambda_{o,e} = 1$, as shown in Fig. 4.6, SCLO has a limited improvement in terms of the completion time compared with MMLO and MELO, even performs much worse than the other two non-learning-based methods (LC and RO). That is because SCLO is sensitive to the growth of edge workload since the increasing workload. If there are too many jobs from other devices that have a shorter deadline, the queuing time of the offloaded jobs can be extensive. Another observation is that MMLO performs as



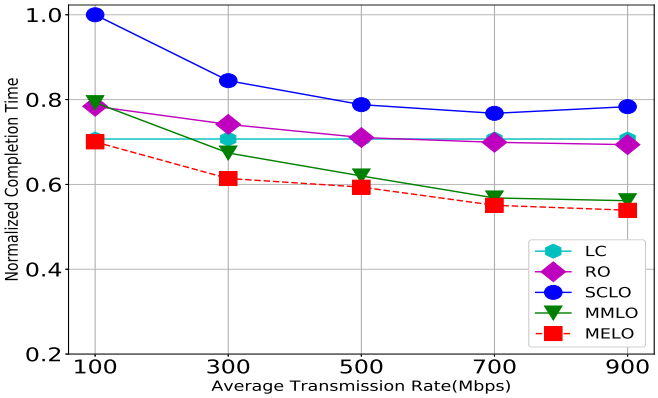
(a) Completion Time vs. Transmission Rate



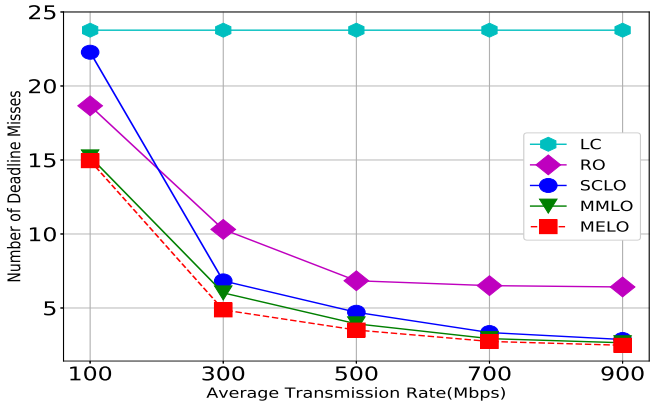
(b) Deadline Miss vs. Transmission Rate

Figure 4.5: Impact of Transmission Rate ($\lambda_{o,e}$= 0.5)

better as MELO in terms of the reduction of deadline misses when the transmission speed is faster than 500 Mbps. However, in the cases that MEC system only can support a lower transmission rate, e.g., less than 500 Mbps, MMLO sees a rise at miss rate, which can reach up to 18.2%. It is because MMLO should allocate extra local computation resources for model training at the end of each hyperperiod. Consequently, most jobs arrived at these time slots need to be offloaded to edge servers to catch the deadlines. For the circumstances that the transmission rate and resources of edge servers are deficient, more jobs being offloaded will lead to more deadline violations. Clearly, MELO outperforms these other offloading methods in terms of



(a) Completion Time vs. Transmission Rate



(b) Deadline Miss vs. Transmission Rate

Figure 4.6: Impact of Transmission Rate ($\lambda_{o,e} = 1.0$)

both two measurements. That is because MELO not only can intelligently schedule more jobs to mobile device if there is serious congestion at edge tier, but also offload more jobs to edge servers if they have sufficient computation resources and stable communication.

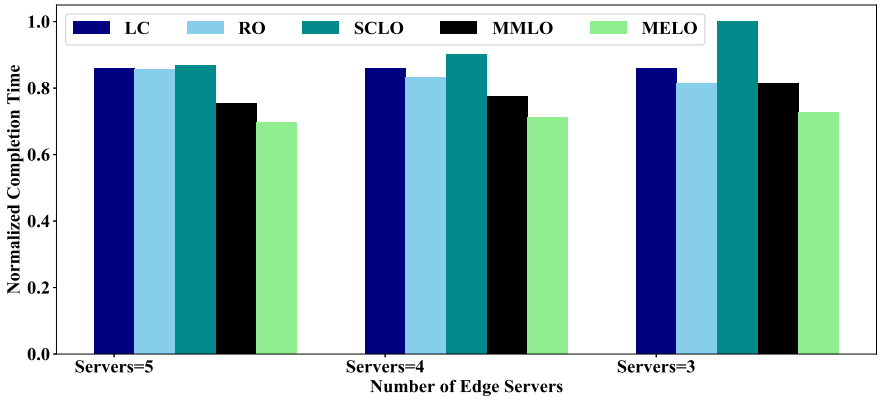
4.3.4 Impact of Number of Edge Servers

To further demonstrate the effectiveness of our offloading model in the dynamic MEC systems, we study the performance of the schemes under investigation when the number of edge servers vary. In the simulations, 10 periodic tasks are considered

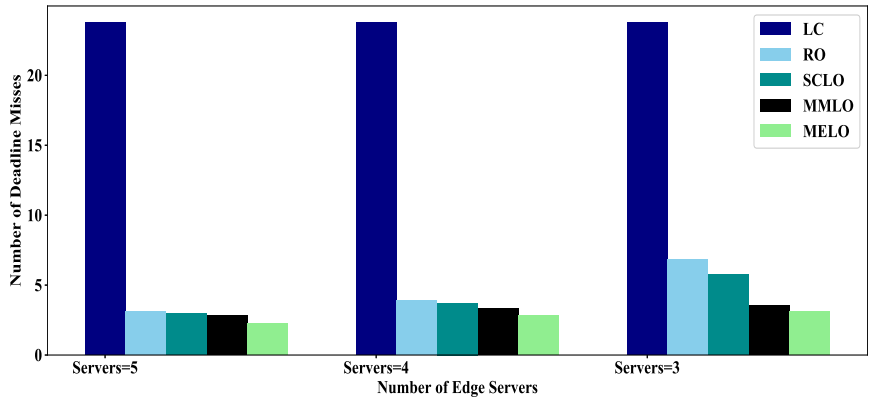
in a hyperperiod, and the corresponding amount of jobs is 33. In addition, the transmission rate of each edge server is set to $[400, 600]$ Mbps. $\lambda_{l,s}$ is set from 0.01 job/s to 0.05 job/s. On the other hand, the arrival rates at edge server $\lambda_{o,e}$ are set to the range $[0.5, 1.0]$ job/s. As shown in Fig 4.7, it can be seen that both the completion time and the number of deadline misses increase slowly when the number of edge servers decreases from 5 to 4, while having a significant performance degradation when the number of edge servers is lowered down to 3. The observation demonstrates that, when MEC system contains a limited number of edge servers, mobile device has a smaller flexibility on its decision-making for the offloaded jobs. Then, many of the offloaded jobs are inevitably executed at the busy servers. Furthermore, to prevent more deadline misses, more jobs having a short deadline intend to be processed locally with a slower processor rather than being offloaded to those busy servers, consequently resulting in a larger completion time. For these reasons, increasing the number of edge servers can effectively improve the performance of MEC systems. However, even more servers are involved, both SCLO and MMLO methods are still stuck in severe deadline violation and a large completion time. This is because SCLO is only capable of processing jobs at edge servers. Once edge servers become overloaded, the offloaded jobs should cost a large amount of time to wait for its scheduling window, explicitly enlarging the total completion time and deadline misses. On the other hand, with a smaller number of servers, MMLO needs to process more jobs at local processor with a lower operating frequency. Thus, the deadline misses and completion time of MMLO grow rapidly along with the reducing number of servers. Compared to baseline methods, our proposed offloading method can reduce the completion time and deadline misses by at least 24% and 31% with respect to the second-best algorithm.

4.4 Major Conclusions of MELO

In this chapter, we propose an edge-assisted learning based offloading scheme, MELO, to generate the appropriate offloading decisions for deadline-sensitive tasks in a 6G-empowered MEC system. Specifically, we consider a 3-tier system that involves mobile devices, edge servers, and cloud servers. With MELO, depending on the



(a) Completion Time vs. No. of Edge Servers



(b) Deadline Miss vs. No. of Edge Servers

Figure 4.7: Impact of Number of Edge Servers

workload of mobile devices and edge servers, deadline-sensitive tasks are completed locally or offloaded to edge/cloud server in order to minimize their completion time. Note that MELO is based on a novel edge-assisted learning architecture, EALA. The advantage of EALA over the existing learning process in MEC systems is that the inference and training operation of the learning algorithm are decoupled. The training operation is carried out on edge servers while the inference operation is performed on mobile devices. In this manner, more computational resources could be used to handle deadline-sensitive tasks on mobile devices. Our experimental results

indicate that MELO outperforms the existing offloading schemes in terms of completion time and deadline miss. We notice that, in our experiments, the completion time decreases as the transmission rate between mobile devices and edge servers goes up. However, when the transmission rate approaches 700 Mbps, the completion time starts to decrease at a lower rate. Note that the impact of mobility on MELO has not been thoroughly investigated. In the future, we will attempt to understand the performance of MELO when mobility is taken into consideration.

Chapter 5

Safe Task Offloading with Constrained Reinforcement Learning

In this chapter, we introduce the proposed safe task offloading scheme (CRLO) to minimize task completion time. First of all, we outline the system model utilized in CRLO. Subsequently, we systematically present the details of the proposed scheme. Finally, we analyze the performance of CRLO in terms of convergence, completion time, deadline miss, and scalability.

5.1 System Model

In this section, we first give the overview of our offloading MEC system. Afterwards, three essential models, including task model, communication model, and completion time model, are presented systematically. At the end, the offloading problem with a set of constraints is formally formulated. For clarity, the key notations used in Chapter 5 are outlined in Table 5.1.

5.1.1 Overview

In the research, we consider a multi-tier 5G-enabled edge computing system, which consists of a mobile layer, edge layer, and cloud layer. The structure of this system is described in Fig. 5.1. In the mobile device layer, there is a group of homogeneous mobile devices, $\mathcal{M} = \{1, ..., m, ...\}$, that seamlessly communicate with a cluster of homogeneous computing edge servers, $\Omega = \{\Omega_1, ..., \Omega_e, ...\}$, and one learning server that is deployed at edge, Ω_M, via wireless communication. Unless stated otherwise, in the subsequent sections, the term "edge server" refers to the computing edge server. Each mobile device m performs its own periodic task set, $\mathcal{U}_m$, where each task in $\mathcal{U}_m$ is repeated according to its period untill it is explicitly terminated.

In our research, we focus on the periodic task offloading behavior regarding a specific mobile device m in the edge computing system, which is illustrated in Fig. 5.1. Hence, the tasks offloaded from other mobile devices $\widetilde{\mathcal{M}}$ ($\widetilde{\mathcal{M}} = \mathcal{M}/m$) to the

Table 5.1: Key Notations in Chapter 5

Symbol	Definition
$\mathcal{M}$	Set of mobile devices
m	m-th mobile device
$\widetilde{\mathcal{M}}$	Set of mobile devices excluded m
Ω, Ω_e	Set of edge servers and e-th edge server, $\Omega_e \in \Omega$
$\mathcal{T}_m$	Time line at mobile device m
T_m	Hyperperiod of periodic task set $\mathcal{U}_m$, $m \in \mathcal{M}$
t_m	Time slot of a hyperperiod at mobile device m
$\mathcal{U}_m$	Periodic task set of mobile device m
$\mu_{i,m}^k$	k-th periodic task i in a hyperperiod, $m \in \mathcal{M}$
v_m^i	Aperiodic task i, $m \in \mathcal{M}$
$\mathbb{U}_m$	Task set that includes all tasks of a hyperperiod, $m \in \mathcal{M}$
$\mathfrak{R}_{i,m}^k$	Deadline indicator of task $\mu_{i,m}^k$
$\mathcal{Q}_m^{co}$	Computation queue of mobile device m
$\mathcal{Q}_m^{tr}$	Transmission queue of mobile device m
$\mathcal{Q}_m^a$	Aperiodic task queue of mobile device m
$\mathcal{Q}_e$	Task queue of edge server Ω_e
λ_z^e	Task arrival rate of edge server Ω_e
$\lambda_m^{e,ch}$	Rate parameter of channel gain
λ_m^a	Task arrival rate of aperiodic task of mobile device m
$f_{\overline{m}}$	CPU frequency of mobile device
$f_{\overline{e}}$	CPU frequency of edge server
$f_{\overline{c}}$	CPU frequency of cloud server
R_m^e	Transmission rate of mobile device m to edge server Ω_e
$\mathcal{U}_m^{\overline{m}}$	Sub task set of mobile device m processed locally
$\mathcal{U}_m^{e,\overline{e}}$	Sub task set of mobile device m processed at edge server Ω_e
$\mathcal{J}_{i,m}^k$	Offloading decision of task $\mu_{i,m}^k$
$\mathcal{D}_e^{p,\overline{c}}$	Round-trip propagation time from edge server to cloud server
$\mathcal{L}_m^e$	Distance from mobile device m to edge server Ω_e
$\Re_{i,m}^k$	Deadline indicator of task $\mu_{i,m}^k$
π_m	Orginal policy network of mobile device m
π_m^f	the policy network with forecasting module in the mobile device m
π_m^{safe}	Safe policy network of mobile device m

edge server Ω_e is assumed to be followed a Poisson distribution with an expected rate of λ_z^e. Additionally, a set of aperiodic tasks arrives at the mobile device m similarly following a Poisson distribution with an expected rate of λ_m^a. The ready aperiodic tasks are store in a FIFO queue $\mathcal{Q}_m^a$. As aperiodic tasks generally have a soft deadline, they are only processed on the mobile device m if there are no ready periodic tasks in the local task queue. We only consider the offloading of periodic tasks, as our offloading model focuses on MTS task offloading, where scheduling aperiodic tasks that have an uncertain arrival pattern is impractical [40, 41]. At each decision point, periodic tasks are possibly processed locally or offloaded to one of the edge servers according to an offloading policy. To efficiently handle local-processing tasks and offloaded tasks efficiently, we maintain two priority queues on the mobile device: the computation queue $\mathcal{Q}_m^{co}$ and the transmission queue $\mathcal{Q}_m^{tr}$. These queues prioritize the in-queue tasks based on EDF [105]. More specifically, the computation queue manages locally-precessed tasks, e.g., ordering the arrived local tasks and dispatching the highest-priority task to the processor. Similarly, the transmission queue manage the offloaded tasks in a same manner. At the beginning of each MTS period, the mobile scheduler initially determines the proper offloading decisions for all periodic tasks in the period. If one task is designated to be processed on the mobile device, it will be delivered to computation queue $\mathcal{Q}_m^{co}$ once it arrives in the operating system. Otherwise, the task is placed in transmission queue $\mathcal{Q}_m^{tr}$ for further task transmission. Note that, each offloaded task is labeled to explicitly indicate to which edge server the task is being offloaded. For clarity, the key notations of this report are outlined in Table 5.1

In the edge layer, all edge servers are deployed evenly in proximity to mobile devices to support computational services. In our offloading model, we assume that all edge servers are configured with identical hardware and software settings, such as the same processor speed and memory capacity. To comprehensively schedule the offloaded tasks from all associated mobile devices, each edge server Ω_e maintains a priority queue $\mathcal{Q}_e$ based on EDF as well. Due to limited computation resources, not all offloaded tasks can be processed properly when edge servers are overloaded. If the excessive offloaded tasks cannot be scheduled promptly, considerable deadline

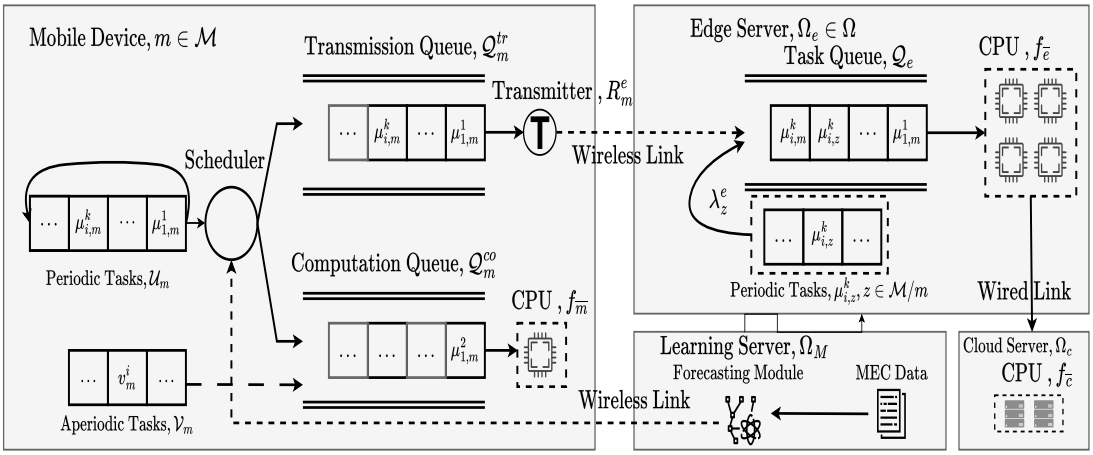


Figure 5.1: Architecture of the MEC System under Investigation for CRLO.

misses may unexpectedly occur. To expand computational capability of our MEC system, the surplus offloaded tasks can potentially be delivered to a complementary computing resource, the cloud server, which is assumed to be equipped with sufficient computation resources to execute all offloaded tasks in parallel fashion. In addition, our offloading model involves a forecasting module deployed at the learning server. Specifically, the forecasting module can be regarded as a virtual version of the physical MEC system. The virtual avatar of the MEC system is capable of capturing realtime information of the physical MEC system, such as task arrival rates of edge servers and network channel dynamics. Additionally, it can process computation-intensive analysis tasks for the associated mobile devices. With the forecasting module, it implies that standalone "myopic" mobile devices are no longer treated as isolated information islands. Instead, the global optimality of decision-making on mobile devices can potentially be achieved by utilizing the acquired "far-sighted" information and profitable analysis from the forecasting module.

5.1.2 Task Model

In practice, many mobile devices support a variety of different computation-intensive applications, such as face recognition and navigation. Computation-intensive applications often involve periodic tasks. In our research, we focus on the offloading scheme for a set of periodic tasks on the mobile device m, denoted as $\mathcal{U}_m = (\mu_{1,m}, ..., \mu_{i,m}, ...), m \in \mathcal{M}$. Each periodic task $\mu_{i,m}$ is characterized as a three-tuple, $\mu_{i,m} = (d_{i,m}, c_{i,m}, p_{i,m})$, where $d_{i,m}$, $c_{i,m}$, and $p_{i,m}$ denote the data size, the required computation cycles, and task period, respectively. Additionally, the aperiodic task of mobile device m is denoted as $\mathcal{V}_m = (v_m^1, ..., v_m^i, ...), m \in \mathcal{M}$. Each aperiodic task has one attribute, $v_m^i = (\chi_m^i)$, where χ_m^i denotes the required CPU cycles of task v_m^i.

Due to the nature of periodic tasks, all of them arrive to system with an invariant pattern, allowing the mobile scheduler to schedule multiple tasks simultaneously for a fixed number of time slots [105]. Namely, scheduling periodic tasks can be considered as an MTS optimization problem. For this reason, we divide the timeline of mobile device m into a series of equally-divided time intervals $\mathcal{T}_m = \{1, ..., T_m, ...\}$, each of which contains multiple time slots. Also, we denote t_m as a single time slot in each interval T_m, $t_m \in \{1, ..., |T_m|\}$, where $|T_m|$ represents the length of a time interval. Without loss of generosity, we refer to the equally-divided time interval T_m as a "hyperperiod" consistent with the previous schemes [40, 41, 43]. The hyperperiod can be calculated using the least common multiple of the task periods, as shown in Eq. (5.1). The reason for introducing the hyperperiod is to ensure that each periodic task in the set $\mathcal{U}_m$ can be executed at least once within a given time interval. Additionally, by using a one-time policy generation for multiple tasks, the computation overhead of the mobile scheduler can be significantly reduced.

$$|T_m| = LCM(p_{1,m}, ..., p_{i,m}, ...). \tag{5.1}$$

During the T_m-th hyperperiod, each periodic task $\mu_{i,m}(T_m)$ may be executed multiple times. To differentiate each repetition of a task during a hyperperiod, we define $\mu_{i,m}^k(T_m)$ as the k-th executable iteration of task $\mu_{i,m}(T_m)$ in the hyperperiod. Hence,

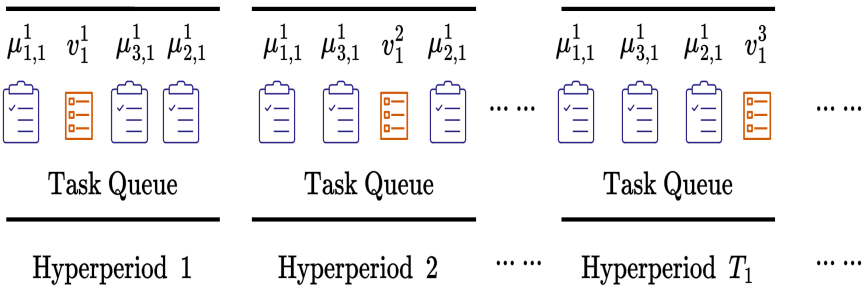
the total number of periodic tasks in a hyperperiod can be given by

$$|\mathbb{U}_m| = \sum_{\mu_{i,m} \in \mathcal{U}_m} \frac{|T_m|}{p_{i,m}}. \tag{5.2}$$

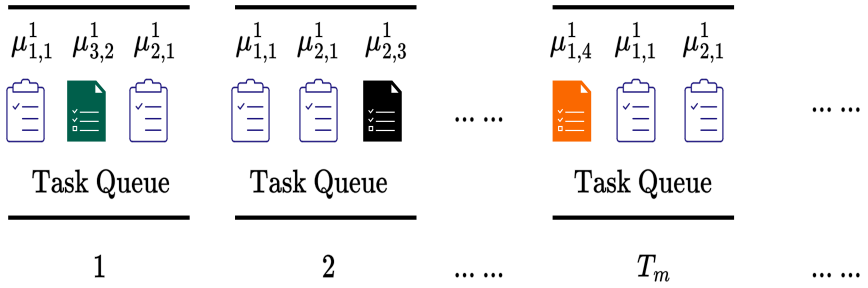
As mentioned in Section 5.1.1, there are two priority queues ($\mathcal{Q}_m^{co}$ and $\mathcal{Q}_m^{tr}$) that operate in the mobile device m based on EDF scheduling algorithm. Given a periodic task set $\mathcal{U}_m$ being scheduled in the task queues $\mathcal{Q}_m^{co}$ and $\mathcal{Q}_m^{tr}$ in the hypereriod T_m, the execution pattern of set $\mathcal{U}_m$ can be predetermined to align with the deadlines of tasks in the queues. Similarly, the edge server Ω_e also handles a task queue $\mathcal{Q}_e$ to manipulate offloaded tasks orderly. The discrepancy of schedulings between mobile devices and edge servers is that mobile scheduling only needs to address the allocated local tasks, while edge scheduling not only must consider the offloaded tasks of mobile device m, but also the tasks arrived arbitrarily from other coexisting mobile devices. This means that the execution priorities of offloaded tasks in task queue $\mathcal{Q}_e$ are constantly reprioritized to adapt to the system dynamics, e.g., the arrival of a new task at the edge server. The mechanism of task scheduling are demonstrated in Fig. 5.2. Due to the flexible completion deadlines of aperiodic tasks, it is important to note that aperiodic tasks certainly have lower scheduling priorities than periodic tasks in the queue $\mathcal{Q}_m^{co}$. Since all periodic tasks have stringent deadlines, any periodic task failed to be accomplished on time will be discarded before the start of the next hyperperiod. Let $\Re_{i,m}^k(T_m)$ denote the deadline-miss indicator of task $\mu_{i,m}^k$ at T_m-th hyperperiod. Specifically, if task $\mu_{i,m}^k$ violates the deadline constraints, the indicator is set to 1. Otherwise, it is set to 0.

5.1.3 Communication Model

In this paper, we adopt a wireless communication model in which part of the periodic tasks are offloaded to edge servers via orthogonal channels, and hence there is no interference among mobile devices during the data transmission procedure [68]. To remain consistent with existing work [35, 87], we assume that wireless transmission suffers from path loss and small-scale fading. Let $|\mathcal{G}_m^e|^2$ denote the channel gain from mobile device m to edge server Ω_e, which follows an exponential distribution with rate



(a) Mobile Task Scheduling: The execution pattern of periodic tasks during different hyperperiods on mobile device 1 is static, e.g., $\forall T_m \in \mathcal{T}_m, \mu^1_{1,1} > \mu^1_{3,1} > \mu^1_{2,1}$. Aperiodic tasks v^1_1, v^2_1, and v^3_1, are allowed to execute when there are no ready periodic tasks in the task queue $\mathcal{Q}^{co}_m$.



(b) Edge Task Scheduling: Suppose $\mu^1_{1,1}$ and $\mu^1_{2,1}$ are offloaded to an edge server sequentially. The execution pattern of these two tasks depends on when the tasks from other mobile devices that have a higher priority arrive. In this example, the sequence in hyperperiod 1 is $\mu^1_{1,1} > \mu^1_{3,2} > \mu^1_{2,1}$, while it has been changed to $\mu^1_{1,1} > \mu^1_{2,1} > \mu^1_{2,3}$ in hyperperiod 2. Note that, the execution sequence of the offloaded tasks within a hyperperiod are constantly updated as new tasks arrive and in-queue tasks are completed.

Figure 5.2: Task Scheduling at Mobile Device and Edge Server

parameter $\lambda^{e,ch}_m$ [110]. we also use $\mathcal{P}_{m,e}$ to denote the transmission power of mobile device m. Therefore, the instantaneous transmission rate from mobile device m to edge server Ω_e can be expressed as

$$R^e_m = \mathbb{B}^{up}_m \log_2(1 + \frac{|\mathcal{G}^e_m|^2 \cdot \mathcal{P}^e_m}{(\mathcal{L}^e_m)^\ell \cdot \varrho^2}), \tag{5.3}$$

where $\mathbb{B}^{up}_m$ denotes the allocated bandwidth of mobile device m, and $\mathcal{L}^e_m$ is the distance between mobile device m and edge server Ω_e. Here, ℓ denotes the path loss exponent, and ϱ^2 indicates the power of Gaussian noise at mobile device m. We assume that the results of periodic tasks in edge computing are negligible, and therefore we do not consider downlink transmission rate in this paper. For example, when performing

periodic object recognition, edge servers only need to return the object label, which involves a minimal amount of data that can be easily transmitted to mobile devices within a short time.

Mobile devices typicall have limited processing power, which means that not all periodic tasks can be executed successfully on the local processor if the workload of mobile devices have been exceeded their computation capacity. To address this issue, a portion of periodic tasks can be offloaded to resource-rich edge servers. Due to the dynamic nature of the MEC system, mobile devices need to make careful offloading decisions when deciding which tasks to be offloaded. For instance, data-driven tasks may opt for local computing to avoid potential network congestion, while computation-driven tasks prefer remote computing at edge servers because of the limited computation resources of mobile devices. Let $\mu_{i,m}^k(T_m)$ denote task $\mu_{i,m}^k$ at hyperperiod T_m. The offloading decision for this task can be represented as $\mathcal{J}_{i,m}^k(T_m)$, where $\mathcal{J}_{i,m}^k(T_m) = 0$ indicates that task $\mu_{i,m}^k(T_m)$ is assigned to mobile device m for local processing. Alternatively, if task $\mu_{i,m}^k(T_m)$ is offloaded to the e-th edge server, $\mathcal{J}_{i,m}^k(T_m)$ is set to e. It is worth noting that each periodic task $\mu_{i,m}^k(T_m)$ is indivisible and can only be processed either on mobile device m or on one of the edge servers during the hyperperiod T_m.

5.1.4 Completion Time Model

As mentioned earlier, each periodic task can be processed on a mobile device, an edge server, or a cloud server. In this subsection, we explain how the total completion time of a periodic task $\mu_{i,m}^k(T_m)$ is calculated in various computing sceanrios. Furthermore, we present the deadline constraints that are considered in our research.

Completion Time of Local Processing Tasks

Given the offloading decision of task $\mu_{i,m}^k(T_m)$, e.g., $\mathcal{J}_{i,m}^k(T_m) = 0$, it indicates that task $\mu_{i,m}^k(T_m)$ needs to be placed in the computation queue $\mathcal{Q}_m^{comp}$ for local computing. It is assumed that mobile devices are equipped with a single CPU that operates a frequency of $f_{\overline{m}}$, which is measured in CPU cycles per second. Hence, due to the

limited computation resources at mobile device m, task $\mu_{i,m}^{k}(T_m)$ possibly postpones its execution at the time slot of arrival, which is denoted as $(k-1)\cdot p_{i,m}^{k}$. The task will be revoked until all of its in-queue predecessors are processed. We define $\mathcal{D}_{i,m}^{k,q,\overline{m}}(T_m)$ as the queuing time of task $\mu_{i,m}^{k}(T_m)$ in queue $\mathcal{Q}_m^{co}$, which can be calculated using the following equation:

$$\mathcal{D}_{i,m}^{k,q,\overline{m}}(T_m) = \sum_{\Psi_m^c(\mu_{i,m}^{k}(T_m))} \frac{c_{x,m}^{y}(T_m)}{f_{\overline{m}}}, \tag{5.4}$$

where $\Psi_m^c(\cdot)$ denotes a task set that includes all of the predecessors of task $\mu_{i,m}^{k}(T_m)$ in queue $\mathcal{Q}_m^{co}$, and $x \neq i, y \neq k$.

After all the preceding tasks have been executed, task $\mu_{i,m}^{k}(T_m)$ is immediately reactivated and then placed on the mobile processor for local processing. Therefore, the local computation time of task $\mu_{i,m}^{k}(T_m)$ can be calculated as

$$\mathcal{D}_{i,m}^{k,c,\overline{m}}(T_m) = \frac{c_{i,m}^{k}(T_m)}{f_{\overline{m}}}. \tag{5.5}$$

Let $\mathcal{D}_{i,m}^{k,\overline{m}}(T_m)$ denote the completion time of task $\mu_{i,m}^{k}(T_m)$ being processed locally. Given the local queueing time $\mathcal{D}_{i,m}^{k,q,\overline{m}}(T_m)$ and local computation time $\mathcal{D}_{i,m}^{k,c,\overline{m}}(T_m)$ of task $\mu_{i,m}^{k}(T_m)$, we have

$$\mathcal{D}_{i,m}^{k,\overline{m}}(T_m) = \mathcal{D}_{i,m}^{k,q,\overline{m}}(T_m) + \mathcal{D}_{i,m}^{k,c,\overline{m}}(T_m). \tag{5.6}$$

For example, suppose in the 2-th hyperperiod of 2-th mobile device ($m = 2, T_2 = 2, |T_2| = 12,$), where periodic task $\mu_{3,2}^{1}(2)$ arrives at time slot 4. However, it cannot be executed immediately as there are two predecessor tasks $\mu_{1,2}^{2}(2)$ and $\mu_{2,1}^{1}(2)$ in set $\Psi_m^c(\mu_{3,2}^{1}(2))$. Let's assume that $\mathcal{D}_{1,2}^{2,\overline{m}}(2) = 1$ and $\mathcal{D}_{2,1}^{1,\overline{m}}(2) = 2$. Therefore, $\mu_{3,2}^{1}(2)$ queues for a total of 3 time slots ($\mathcal{D}_{3,2}^{1,q,\overline{m}}(2) = 3$) until it is executed at time slot 7. Assuming that $\mathcal{D}_{3,2}^{1,c,\overline{m}}(2) = 1$, task $\mu_{3,2}^{1}(2)$ is completed at time slot 8. Thus, the total completion time of task $\mu_{3,2}^{1}(2)$ for local processing is $3 + 1 = 4$ slots.

Completion Time of Offloaded Tasks

In our model, an offloaded periodic task $\mu_{i,m}^{k}(T_m)$ has to be allocated to the e-th edge server according to the offloading decision, e.g., $\mathcal{J}_{i,m}^{k}(T_m) = e$. At the beginning of hyperperiod T_m, the mobile scheduler initially places the task $\mu_{i,m}^{k}(T_m)$ into queue $\mathcal{Q}_m^{tr}$ and then compares its priority with other in-queue tasks. If there exist incomplete predecessors of tasks $\mu_{i,m}^{k}(T_m)$ at its arrival time, the task $\mu_{i,m}^{k}(T_m)$ cannot be transmitted until all its predecessors complete the data transmission. Let $\Psi_m^t(\mu_{i,m}^{k}(T_m))$ denote the predecessor set of task $\mu_{i,m}^{k}(T_m)$ being assigned to edge computation, the waiting time of this task in mobile transmitter can be obtained as follows:

$$\mathcal{D}_{i,m}^{k,e,w,\overline{e}}(T_m) = \sum_{\Psi_m^t(\mu_{i,m}^{y}(T_m)),x\neq i,y\neq k} \frac{d_{x,m}^{y}(T_m)}{R_m^e(t_m)}, \tag{5.7}$$

where $R_m^e(t_m)$ is the instantaneous transmission rate between mobile device m and edge server Ω_e at time slot $t_m, t_m \in T_m$. Once all the predecessors are completed, task $\mu_{i,m}^{k}(T_m)$ is invoked and begins transmitting its data. Hence, the total time consumed by data transmission of task $\mu_{i,m}^{k}(T_m)$ can be given by

$$\mathcal{D}_{i,m}^{k,e,d,\overline{e}}(T_m) = \frac{d_{i,m}^{k}(T_m)}{R_m^e(t_m)}. \tag{5.8}$$

Additionally, the total time consumed by the offloaded task in mobile device, denoted by $\mathcal{D}_{i,m}^{k,e,\widetilde{m},\overline{e}}(T_m)$, is the sum of the local waiting time in the mobile transmitter and the data transmission time, which can be computed as follows:

$$\mathcal{D}_{i,m}^{k,e,\widetilde{m},\overline{e}}(T_m) = \mathcal{D}_{i,m}^{k,e,w,\overline{e}}(T_m) + \mathcal{D}_{i,m}^{k,e,d,\overline{e}}(T_m). \tag{5.9}$$

We have made the assumption that the limited computation resources at edge servers are fairly competed by all the associated mobile devices. For this reason, instead of scheduling a task $\mu_{i,m}^{k}(T_m)$ instantly, it needs to join the queue $\mathcal{Q}_e$ upon arrival at the edge server Ω_e. Namely, this task is potentially postponed its execution until the completion of all its predecessors. Thus, the edge queuing time of task $\mu_{i,m}^{k}(T_m)$ is the total completion time of its predecessors that can be given by

$$\mathcal{D}_{i,m}^{k,e,q,\overline{e}}(T_m) = \sum_{\Psi_e(\mu_{i,m}^k(T_m))} \frac{c_{x,m}^y(T_m) + c_{i,z}^k(T_m)}{f_{\overline{e}}}. \tag{5.10}$$

In the equation, $\Psi_e(\mu_{i,m}^k(T_m))$ represents the predecessors set of the offloaded task $\mu_{i,m}^k(T_m)$ when it is offloaded to the edge server Ω_e. $c_{x,m}^y(T_m)$ and have not yet completed their computation. The variable $c_{x,m}^y(T_m)$ denotes the required CPU cycles of the predecessors offloaded from mobile device m, where $x \neq i$ and $y \neq k$. On the other hand, $c_{i,z}^k(T_m)$ denotes the cycles of the predecessors from other associated mobile devices, where $z \in \widetilde{\mathcal{M}}$. The notation $f_{\overline{e}}$ denotes the operating frequency of edge servers. It is worth noting that the predecessor set $\Psi_e(\mu_{i,m}^k(T_m))$ is constantly refreshed because of the arbitrary arrival of tasks from mobile set $\widetilde{\mathcal{M}}$. This refresh happens when a new task arrives or an old task is accomplished. Once task $\mu_{i,m}^k(T_m)$ is executed on the edge server Ω_e, it is possible to calculate the corresponding edge computation time as

$$\mathcal{D}_{i,m}^{k,e,c,\overline{e}}(T_m) = \frac{c_{i,m}^k(T_m)}{f_{\overline{e}}}. \tag{5.11}$$

Let $\mathcal{D}_{i,m}^{k,e,\widetilde{e},\overline{e}}(T_m)$ denote the total time cost of task $\mu_{i,m}^k(T_m)$ at the edge server Ω_e, which is the sum of edge queuing time and edge computation time that can be given by

$$\mathcal{D}_{i,m}^{k,e,\widetilde{e},\overline{e}}(T_m) = \mathcal{D}_{i,m}^{k,e,q,\overline{e}}(T_m) + \mathcal{D}_{i,m}^{k,e,c,\overline{e}}(T_m). \tag{5.12}$$

To this end, the completion time of an offloaded task $\mu_{i,m}^k(T)$ for edge computing on the edge server Ω_e can be calculated as follows:

$$\mathcal{D}_{i,m}^{k,e,\overline{e}}(T_m) = \mathcal{D}_{i,m}^{k,e,\widetilde{m},\overline{e}}(T_m) + \mathcal{D}_{i,m}^{k,e,\widetilde{e},\overline{e}}(T_m). \tag{5.13}$$

On the other hand, if the task $\mu_{i,m}^k(T_m)$ is incorrectly offloaded to an overloaded edge server, it must be uploaded to the remote cloud server to mitigate the impact of the inappropriate offloading decisions. Specifically, the cloud server acts as complementary computing resources to expand the computation capability of the MEC system in handling the excessive offloaded tasks of edge servers. If the total processing time of a task at the edge server, comprising both the queueing time and computation

time, leads to a deadline violation, and if the sum of the round-trip propagation time from the edge server to the cloud server and its computation time does not result in a deadline violation, the task is shifted to the cloud server. It should be noted that even with the cloud server as a backup computation resource, it does not imply that edge servers have infinite computation resources to process all offloaded tasks. In addition, we also assume that the resource-sufficient cloud server enables parallelism in task execution, which results in no extra queuing time in cloud computation. Let $\mathcal{D}_e^{p,\bar{c}}$ denote the round-trip propagation time of the task $\mu_{i,m}^k(T_m)$ between the edge server Ω_e and the cloud server. Then, the time cost corresponding to the cloud computing can be given by

$$\mathcal{D}_{i,m}^{k,e,\bar{c}}(T_m) = \mathcal{D}_e^{p,\bar{c}} + \frac{c_{i,m}^k(T_m)}{f_{\bar{c}}}, \tag{5.14}$$

where $f_{\bar{c}}$ denotes the CPU speed of cloud server. The second component of Eq. (5.14) is the processing time of task $\mu_{i,m}^k(T_m)$ at the cloud server. Thus, combined with transmission time from the mobile device m to the edge server Ω_e, the completion time of task $\mu_{i,m}^k(T)$ being processed at the cloud server can be computed by

$$\mathcal{D}_{i,m}^{k,\bar{c}}(T_m) = \mathcal{D}_{i,m}^{k,e,\widetilde{m},\bar{e}}(T_m) + \mathcal{D}_{i,m}^{k,e,\bar{c}}(T_m). \tag{5.15}$$

To this end, we have the completion time of task $\mu_{i,m}^k(T_m)$ being processed among various computing layers,

$$\mathcal{D}_{i,m}^k(T_m) = \begin{cases} \mathcal{D}_{i,m}^{k,e,\bar{e}}(T_m) | \mathcal{D}_{i,m}^{k,\bar{c}}(T_m), & \mathcal{J}_{i,m}^k(T_m) = e \\ \mathcal{D}_{i,m}^{k,\overline{m}}(T_m), & \mathcal{J}_{i,m}^k(T_m) = 0. \end{cases} \tag{5.16}$$

Deadline Constraints

In this section, we illustrate the task completion and deadline satisfaction on the mobile device m. Let $\mathcal{U}_m^{\overline{m}}(T_m)$ denote the allocated task set for local-processing in T_m-th hyperperiod. The completion time of a local-processing task $\mu_{i,m}^k(T_m) \in \mathcal{U}_m^{\overline{m}}(T_m)$ involves both local queuing time and local execution time. Additionally, we define another task set $\mathcal{U}_m^{e,\bar{e}}(T_m)$ to include the tasks offloaded to the edge server Ω_e. As shown

in Fig. 5.2, task scheduling on mobile devices relies on EDF, where task priorities can be predetermined at decision points. On this basis, the relevent deadline indicators of local-processing tasks, $\Re_m^{\overline{m}}(T_m) = \Re_{1,m}^1(T_m) \cap, ..., \cap \Re_{i,m}^k(T_m) = 0, \mu_{i,m}^k(T_m) \in \mathcal{U}_m^{\overline{m}}(T_m)$, should satisfy

$$\sum_{\mu_{i,m}^k(T_m) \in \mathcal{U}_m^{\overline{m}}(T_m)} \frac{c_{i,m}^k(T_m)}{f_{\overline{m}} \cdot |T_m|} < 1, \tag{5.17}$$

where $\frac{c_{i,m}^k(T_m)}{f_{\overline{m}}|T_m|}$ represents the portion of workload imposed by task $\mu_{i,m}^k(T_m)$ on the mobile device m in hyperperiod T_m [43, 105]. If the accumulated total workload of mobile device m in a hyperperiod is less than 1, it indicates that all deadlines of local-processing tasks can be strictly met under EDF scheduling. Alternatively, if a task $\mu_{i,m}^k(T_m)$, $\mu_{i,m}^k(T_m) \in \mathcal{U}_m^{e,\overline{e}}(T_m)$, is accomplished successfully on either the edge server Ω_e or the cloud server, the relevant deadline satisfaction can be guaranteed if

$$\aleph_{i,m}^k(T_m) + \mathcal{D}_{i,m}^{k,e,\overline{e}}(T_m) < k \cdot p_{i,m}^k(T_m), \tag{5.18}$$

$$\aleph_{i,m}^k(T_m) + \mathcal{D}_{i,m}^{k,\overline{c}}(T_m) < k \cdot p_{i,m}^k(T_m), \tag{5.19}$$

where $\aleph_{i,m}^k(T_m) = (k-1) \cdot p_{i,m}^k(T_m)$ denotes the arrival time of task $\mu_{i,m}^k(T_m)$. Inequations (5.18) and (5.19) demonstrate that the completion time of task $\mu_{i,m}^k(T_m)$ being processed on the edge server Ω_e or the cloud server cannot exceed the corresponding deadline. Notably, both task sets $\mathcal{U}_m^{\overline{m}}(T_m)$ and $\mathcal{U}_m^{e,\overline{e}}(T_m)$ will be empty by the end of a hyperperiod, as all in-set task deadlines will have expired by then. To assess deadline satisfaction, the mobile scheduler compares the completion time of each task with its respective deadlines. If the completion time of task $\mu_{i,m}^k(T_m)$ suffices to its deadline constraint, the deadline indicator $\Re_{i,m}^k(T_m)$ is set to 0. Otherwise, this indicator is assigned a value of 1.

Problem Formulation

In the research work, we focus on the task offloading problem in a 5G-enabled MEC system. In this system, the mobile device acts as an intelligent agent to automatically generate appropriate offloading policies for a set of computation-intensive periodic tasks. Edge servers provide computation services for the offloaded tasks of multiple mobile devices. In general, the MEC system is a highly dynamic working environment characterized by frequent variations in network channels and uncertain workload of edge servers. That implies that making offloading decisions for tasks in a nonstationary environment is an exceedingly challenging task. The objective of our work is to find the optimal offloading decisions that minimize the overall completion time while ensuring task deadline constraints. We define the task set that contains all offloaded tasks of mobile device m in hyperperiod T_m as $\mathcal{U}_m^{\bar{e}}(T_m)$. This set is formed by taking the union of all task sets for each edge server Ω_e in the system, such that $\mathcal{U}_m^{\bar{e}}(T_m) = \mathcal{U}_m^{1,\bar{e}}(T_m) \cup, ..., \cup \mathcal{U}_m^{e,\bar{e}}(T_m), ..., \Omega_e \in \Omega$. The total completion time of local-processing tasks and offloaded tasks in a hyperperiod, $\overline{\mathcal{D}}_m^{\overline{m}}(T_m)$ and $\overline{\mathcal{D}}_m^{\bar{e}}(T_m)$, can be calculated independently as follows:

$$\overline{\mathcal{D}}_m^{\overline{m}}(T_m) = \sum_{\mu_{i,m}^k(T_m) \in \mathcal{U}_m^{\overline{m}}(T_m)} \mathcal{D}_{i,m}^{k,\overline{m}}(T_m), \tag{5.20}$$

$$\overline{\mathcal{D}}_m^{\bar{e}}(T_m) = \sum_{\mu_{i,m}^k(T_m) \in \mathcal{U}_m^{\bar{e}}(T_m)} \mathcal{D}_{i,m}^{k,e,\bar{e}}(T_m) + \mathcal{D}_{i,m}^{k,\bar{c}}(T_m). \tag{5.21}$$

Let $\mathcal{J}_m(T_m) = \{\mathcal{J}_{1,m}^1(T_m), ..., \mathcal{J}_{i,m}^k(T_m), ...\}$ denote the profile of offloading decisions at hyperperiod T_m. Since the minimum completion time of tasks can be achieved by properly adjusting the offloading decisions $\mathcal{J}_m(T_m)$, our task offloading problem can be formulated as follows:

$$\min_{\mathcal{J}_m(T_m)} \quad \overline{\mathcal{D}}_m^{\overline{m}}(T_m) + \overline{\mathcal{D}}_m^{\bar{e}}(T_m) \tag{5.22}$$

$$\text{subject to} \quad C1 : \mathcal{J}_{i,m}^k(T_m) \in \{0, 1, ..., e, ...\}, \tag{5.23}$$

$$Constraints : (5.17) - (5.19). \tag{5.24}$$

The aforementioned constraints are described as follows: $C1$ indicates that each periodic task must be allocated to either mobile device m or an edge servers. Constraint (5.17) to (5.19) ensure that all periodic tasks on mobile device m satisfy their deadline constraints. However, due to constraint $C1$, the resulting offloading problem (5.22) becomes a computationally-hard MINLP problem. Furthermore, solely focusing on minimizing completion time in a greedy manner may result in significant deadline violations. Therefore, it is crucial to strike a balance between optimizing completion time and ensuring deadline satisfaction. Moreover, due to privacy concerns, standalone mobile devices often face challenges in accessing global system information, e.g., the offloading policies of other coexisting mobile devices. When making offloading decisions merely based on local information, it can lead to substantial suboptimal decisions. Therefore, we propose a constrained reinforcement learning-based offloading approach to address the offloading problem. Our approach adopts constrained reinforcement learning and a long-sequence forecasting model to improve the accuracy of decision-making in a highly-vibrating MEC system. Practical MEC systems typically involve other costs, such as energy consumption and fees charged by resource renting on servers. These additional costs can be integrated into Eq. (5.22) and designated with appropriate coefficients, which can then be optimized by our model-free reinforcement learning model.

5.2 Safety-critical Learning-based Task Offloading

In this section, we first introduce three basic elements used in CRLO, including long-sequence forecasting model and constrained reinforcement learning. Furthermore, we propose a new policy network that seamlessly integrates the conventional policy network of actor-critic based algorithms with the long-sequence forecasting model and constrained reinforcement learning. Aiming to address the offloading problem while satisfying a set of constraints, we reformulate the problem as a Constrained Markov Decision Process (CMDP). To solve the reformulated problem, we propose a safety-critical learning-based offloading scheme, CRLO, to intelligently generate near-optimal offloading policies while satisfying safety constraints. Finally, we conclude

this section with the complexity analysis of the proposed offloading scheme.

5.2.1 Long-sequence Forecasting Model

The long-sequence forecasting model takes a multivariate state matrix consisting of a series of historical long-sequences of environment states as inputs and outputs predictions of the states [92, 111]. Particularly, the multivariate input matrix is defined as $\mathcal{L}_m^{hist}(T_m) = \{\mathcal{L}_m^{hist}(T_m - 1), ..., \mathcal{L}_m^{hist}(T_m - h), ...\}$, where $\mathcal{L}_m^{hist}(T_m - h)$ denotes the previous experience of environment states at $(T_m - h)$-th hyperperiod. Alternatively, the output matrix is also constituted with a set of predicted long-sequences, defined as $\mathcal{L}_m^{r}(T_m)$. Noticeably, each long sequence in $\mathcal{L}_m^{hist}(T_m)$ or $\mathcal{L}_m^{r}(T_m)$ is referred to a certain input state, e.g., the transmission rate R_m^e and task arrival rate λ_z^e in our offloading model. Instead of using RNN and LSTM for the state predictions, one computationally-efficient and predictively-accurate long-sequence forecasting model, Informer, is utilized to tackle our MTS long-sequence offloading problem [111]. To be more specific, Informer consists of an encoder and a decoder. The encoder uses a combination of self-attention and convolutional operation to process the input matrix $\mathcal{L}_m^{hist}(T_m)$ and generates a compressed representation of the input. The decoder contains two multi-head attention layers, each of which has its own feed-forward network. The decoder takes the compressed representation of encoder and the second half of matrix $\mathcal{L}_m^{hist}(T_m)$ as inputs to predict the future values of the long sequences. Eventually, the decoder outputs the final results of long-sequence prediction, $\mathcal{L}_m^{r}(T_m)$.

5.2.2 Constrained Reinforcement Learning

As most existing DRL-based offloading approaches in the Chapter 4 have not sufficiently considered the safety of model learning, directly applying RL-based offloading approaches to safety-critical systems, such as aircraft control and temperature monitoring systems in nuclear plant, is infeasible [43, 62]. This concern has prompted RL agents to incorporate safety considerations as keeping constraints as "on average" can inevitably incur catastrophic consequences in the systems. One emerging subfield of reinforcement learning, called Constrained Reinforcement Learning (CRL),

ensures that the RL agent learns to perform tasks in a safe and reliable fashion. In our next work, we will leverage CRL to regulate the unsafe offloading policy during the learning process. Basically, CRL is specifically designed to address optimization problems with a set of constraints. It is normally defined as a tuple with six elements: $< \mathcal{S}, \mathcal{A}, \Gamma, \mathcal{R}, \kappa, \mathcal{C} >$ [2, 23, 62]. In the tuple, $\mathcal{S}$ is the state space, $s_m(T_m) \in \mathcal{S}$. $\mathcal{A}$ denotes the action space, $a_m(T_m) \in \mathcal{A}$. In addition, $\Gamma(s_m(T_m+1)|s_m(T_m), a_m(T_m))$ is the probability of transitioning from state $s_m(T_m)$ to state $s_m(T_m+1)$ after performing the action $a_m(T_m)$: $\mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0,1]$. $\mathcal{R}$ is the reward function, which can be characterized as $\mathcal{R} : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$. κ denotes the discount factor of learning, $\kappa \in (0,1)$, and $\mathcal{C} = \{\mathcal{C}_{i,m} : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}, i \in \{1, ..., |\mathbb{U}_m|\}, m \in \mathcal{M}\}$ is a set of constraint functions. One representative constrained reinforcement learning is leveraging a safety layer for safe exploration, in which the risky actions can be reduced significantly [23]. To learn the constraint functions of the safety layer, a set of associated safety signals that indicates per-state observations of constrained values is defined as $\overline{\mathcal{C}} = \{\overline{\mathcal{C}}_{i,m} : \mathcal{S} \rightarrow \mathbb{R}, i \in \{1, ..., |\mathbb{U}_m|\}, m \in \mathcal{M}\}$, where $\overline{\mathcal{C}}_{i,m}(s_m(T_m+1))$ is the safety signal of state $s_m(T_m+1)$ after performing state-action pair $(s_m(T_m), a_m(T_m))$. That is $\overline{\mathcal{C}}_{i,m}(s_m(T_m)+1) \triangleq \mathcal{C}_{i,m}(s_m(T_m), a_m(T_m))$. It is important to note that the safety signal $\overline{\mathcal{C}}_{i,m}(s_m(T_m)+1)$ represents the safe distance against the upper bounded constants $\widetilde{C}_{i,m}$. After the policy network π_m generates an action $a_m(T_m)$, it is internally calibrated within the safety layer through a closed-form calculation to obtain the optimal safe action $a_m^*(T_m)$.

5.2.3 A Novel Policy Netowk

In the MEC system, mobile devices are inherently equipped with a parameterized policy network π_m that aims to establish the appropriate mappings between the system states and the offloading policies by constantly interacting with the environment. Nevertheless, the policy network used in existing reinforcement learning methods, such as DQN and DDPG, exclusively struggles to issue safe and far-sighted policies for solving the MTS long-sequence problem due to the lack of a constraint-guaranteed and state forecasting mechanism, as noted in [44, 93]. To overcome these drawbacks, we redesigned the policy network by incorporating a forecasting model and a safety

layer to enhance its efficiency. Specifically, instead of directly feeding immediate states into the policy network, a forecasting model is first introduced to proactively predict the fluctuations of the states in the next hyperperiod. By having more accurate knowledge about the variations in the state, a potentially better offloading policy can be achieved. Furthermore, to prevent risky offloading decisions that result in high costs of failure, we developed an advanced safety layer that can ensure the satisfactions of multiple active constraints simultaneously, in contrast to the conventional safety layer in [23], which only permits one active constraint during each policy generation. The advanced safety layer is integrated at the end of the policy network π_m. An overview of the redesigned policy network, denoted as π_m^{safe}, is shown in Fig. 5.3.

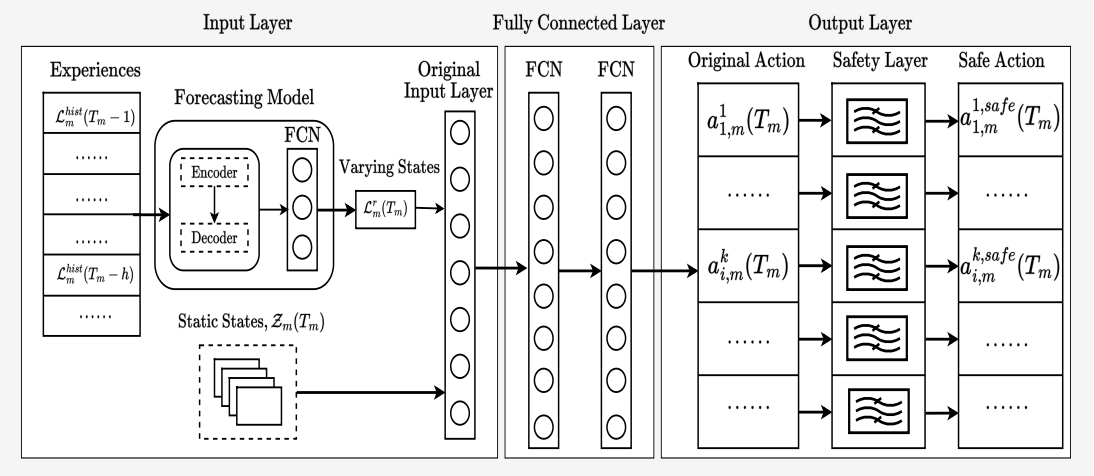


Figure 5.3: Structure of Policy Network.

The first layer of the policy network π_m^{safe} is an advanced input layer that incorporates a forecasting model concatenated ahead of the original input layer to improve the input states. Specifically, the forecasting model predicts state variations for the next hyperperiod based on a series of historical records, while the original input layer accepts all states of the MEC system, including static states and varying states. To be specific, static states, e.g., the remaining workload of edge servers in the last hyperperiod, are assumed to remain unchanged within a hyperperiod but vary among different hyeperperiods. On the other hand, varying states constantly change over

time and can be predicted using the forecasting model. In each iteration, the forecasting model initially takes experience as input and outputs the predictions of varying states. These predictions, combined with observations of static states, create a hybrid of states that are fed into the original input layer before being passed to the subsequent layers. Notice that we use the Informer forecasting model, which has an extended prediction capacity and is particularly effective in solving long-sequence forecasting problems, to achieve the varying states in our offloading model. During each forward propagation of the policy network π_m^{safe}, the Informer forecasting model uses the historical experience $\mathcal{L}_m^{hist}(T_m)$ to predict the varying states $\mathcal{L}_m^r(T_m)$. Let $\mathcal{Z}_m(T_m) = \{\mathcal{Z}_1(T_m), ..., \mathcal{Z}_{h_z}(T_m), ...\}$ denote the multivariate matrix of static states whose values can be observed internally in the mobile device at decision points. The hybrid inputs $s_m(T_m) = \{\mathcal{L}_m^r(T_m), \mathcal{Z}_m(T_m)\}$ is fed into the original input layer at the end of forecasting preprocessing phase. The fully connected layer is responsible for connecting the input layer with the output layer. It is constituted of several hidden layers, each of which contains a batch of neurons that are activated using Rectified Linear Unit (ReLU). Technically, the outcome of the input layer is processed by passing it through these four hidden layers. The output of the last hidden layer is then passed to the output layer. The output layer is responsible for generating safe actions that comply with a set of constraints. To ensure compliance with these constraints, a safety layer is additively attached to the end of the original output layer, which corrects any improper outputs of the original policy network. This assures that the ultimate issued actions not only lead to near-optimal performance but also strictly obey the predefined constraints. Unlike [23], where only one active constraint is allowed in each iteration, the constraint satisfaction in our offloading model is more complicated since multiple active constraints need to be secured simultaneously. Namely, the constraint set $\mathcal{C}_m(T_m) = \{\mathcal{C}_{1,m}^1(T_m), ..., \mathcal{C}_{i,m}^k(T_m), ...\}$ must be satisfied in each iteration of decision-making. Therefore, the safety layer is essentially accountable to ensuring the constraints of all tasks in a hyperperiod. Let $\mathbb{L}_m^{safe}$ denote a safety layer referred to all safe actions $\{a_{1,m}^1(T_m), ..., a_{i,m}^k(T_m), ...\}$.

Technically, the deployment of the safety layer $\mathbb{L}_m^{safe}$ consists of two steps: the pretraining of the linear safety signal model and the generation of safe actions. In

the first step, a safety signal model is trained using a set of pre-collected data. To create pretraining dataset, the MEC system is randomly executed for a number of hyperperiods, and transition samples $(s_m^p(T_m), a_m^p(T_m), s_m^p(T_m+1))$ are collected into a pretraining buffer $\mathcal{B}_m^p$. Note that each execution is terminated instantly if there is a constraint violation or the predefined time limit is reach. Without loss of generosity, let θ_m^{safe} denote the parameters of the Neural Network (NN) used to approximate the associated constraints $\mathcal{C}_m(s_m(T_m), a_m(T_m))$, and $f_m(s_m(T_m); \theta_m^{safe})$ denote a neural network that takes $s_m(T_m)$ as input and outputs a vector with the same dimension as the action $a_m(T_m)$. To approximate the constraint $\mathcal{C}_m(s_m(T_m), a_m(T_m))$, the following linearization is performed as

$$\overline{\mathcal{C}}_m(s_m(T_m)+1) \triangleq \mathcal{C}_m(s_(T_m)) \leftarrow \overline{\mathcal{C}}_m(s_m(T_m)) + f_m(s_m(T_m); \theta_m^{safe}) \cdot a_m(T_m). \quad (5.25)$$

In pretraining phase, a random batch of transitions $\mathcal{I}_m^p = \{s_{n,m}^p(T_m), a_{n,m}^p(T_m), s_{n,m}^p(T_m+1)\}$ are selected arbitrarily from the pretrain buffer $\mathcal{B}_m^p$ to train the neural network of signal model. Note that each sub-level $\mathbb{L}_{i,m}^{k,safe}$ corresponds to one sub-action $a_{i,n,m}^{k,p}(T_m)$, which is an element of $a_{n,m}^p(T_m)$. When entering the action-calibration process, all sub-levels are operated as one overall safety layer $\mathbb{L}_m^{safe}$ to guarantee all the constraints simultaneously. Based on the collected pretrain set $\mathcal{B}_m^p$, the neural network $f_m(s_m(T_m); \theta_m^{safe})$ can be trained by solving the following optimization problem:

$$\begin{aligned} \arg\min_{\theta_m^{safe}} \sum_{\mathcal{I}_m^p} & (\overline{\mathcal{C}}_m(s_{n,m}^p(T_m)+1) - (\overline{\mathcal{C}}_m(s_{n,m}^p(T_m)) \\ & + f_m(s_{n,m}(T_m), \theta_m^{safe})^\intercal \cdot a_{n,m}^p(T_m)))^2. \end{aligned} \quad (5.26)$$

After the neural network of the safety layer is fully trained, the pre-trained safety layer is appended to the end of the original output layer to calibrate unsafe actions. During the phase of action generation, the original action $a_m(T_m)$ is fed into the safety layer $\mathbb{L}_m^{safe}$. In light of state $s_m(T_m)$, the optimal safe action $a_m^{safe}(T_m)$ can be achieved by solving

$$\arg\min_{a} \parallel a - a_m(T_m) \parallel^2 \tag{5.27}$$

$$\begin{aligned} s.t. \quad & \mathcal{C}_m(s_m(T_m), a_m(T_m)) \triangleq \overline{\mathcal{C}}_m(s_m(T_m)) \\ & + f_m(s_m(T_m); \theta_m^{safe}) \cdot a_m(T_m) \leqslant \widetilde{\mathcal{C}}_m. \end{aligned} \tag{5.28}$$

The optimization above aims to find a safe action that is closest to the original action $a_m(T_m)$ in terms of Euclidean distance. To solve this problem, a closed-form solution is used to calculate the most appropriate safe sub-action $a_{i,m}^{k,safe}(T_m)$ independently, as presented in [23]. It is worth noting that the reliable outputs of the policy network π_m^{safe} is the overall safe action $a_m^{safe}(T_m)$, which combines all safe sub-actions and can be represented as $a_m^{safe}(T_m) = \{a_{1,m}^{1,safe}(T_m), ..., a_{i,m}^{k,safe}(T_m), ...\}$.

5.2.4 Reformulating the Offloading Problem as a CMDP

In each hyperperiod, mobile device m observes not only static states, such as task features and remaining workload, but also a series of varying states, including network dynamics and workload variations in the near future. By taking both static states and varying states as inputs, the policy network π_m^{safe} aims to generate appropriate offloading policies to specify the subsequent task allocation. To evaluate performance of offloading policy, two metrics are utilized, including task completion time and deadline miss. The main objective of our offloading model is to minimize both of the measurements via continually optimizing the offloading policies using constraint reinforcement learning.

System State

As mentioned, each state set $s_m(T_m)$ consists of static state set $\mathcal{Z}_m(T_m)$ and varying state set $\mathcal{L}_m^r(T_m)$, that is, $s_m(T_m) = \{\mathcal{Z}_m(T_m), \mathcal{L}_m^r(T_m)\}$. Specifically, static state set includes three elements. The initial element pertains to the workload of mobile device m, denoted as $\mathcal{W}_m^a(T_m)$, which arises from the presence of prior-arrived aperiodic tasks within the local environment. The second element is the rest workload of edge servers in the last hyperperiod, which is denoted as $\mathcal{W}^{\overline{e}}(T_m) = \{\mathcal{W}^1(T_m), ...\mathcal{W}^e(T_m), ...\}$. The last element is the profile of task set $\Theta_m(T_m)$, e.g., the required CPU cycles

and data size of periodic tasks. The varying state set $\mathcal{L}_m^r(T_m)$ is constitued of two components: the transmission rates from mobile device m to edge servers, denoted as $R_m(T_m) = \{R_m^1(T_m), ..., R_m^e(T_m), ...\}$, and the task arrival rates of edge servers $\lambda_z^{\overline{e}}(T_m) = \{\lambda_z^1(T_m), ...\lambda_z^e(T_m), ...\}$, which implicitly reflect the runtime workload of edge servers in a sequential manner. Noticeably, the states in set $Z_m(T_m)$ are assumed to be constant during a hyperperiod but vary among different hyperperiods. In contrast, due to the uncertainties in the MEC system, each state in set $L_m^r(T_m)$ constantly changes among different timeslots. For this reason, the states in set $\mathcal{Z}_m(T_m)$ can be observed instantaneously with a minor overhead at decision points. On the other hand, $R_m^e(T_m)$ and $\lambda_z^e(T_m)$, $R_m^e(T_m) \in R_m(T_m)$, $\lambda_z^e(T_m) \in \lambda_z^{\overline{e}}(T_m)$, represent the variations of the upcoming hyperperiod that can be predicted using the forecasting model. It is aware of that each $R_m^e(T_m)$ or $\lambda_z^e(T_m)$ is a long MTS sequence that can be denoted specifically as $R_m^e(T_m) = \{R_m^e(1), ..., R_m^e(t_m), ...\}$ and $\lambda_z^e(T_m) = \{\lambda_z^e(1), ..., \lambda_z^e(t_m), ...\}$. Additionally, the immediate workload of edge server Ω_e, $\mathcal{W}^e(T_m)$, arises because of the incomplete tasks of the last hyperperiod. It can be calculated as follows:

$$\mathcal{W}^e(T_m) = \sum_{\mu_{i,m}^k(T_m) \in \Psi_m^{e,rest}(T_m)} \frac{c_{i,m}^k(T_m)}{f_{\overline{e}} \cdot |T_m|}, \tag{5.29}$$

where $\Psi_m^{e,rest}(T_m)$ denotes the task set that embraces all incomplete tasks from the previous hyperperiod at the edge server Ω_e. In addition, the mobile device m internally monitors the system conditions and stores the information in the profile $\Theta_m(T_m)$ before entering the next hyeperperiod. Due to continuous variations in the transmission rates and task arrival rates of the edge servers, it is essential to use a forecasting model to foreseeably identify the potential changes in the states so as to the actual variations in near future can be properly estimated. To achieve this goal, we leverage the Informer forecasting model, which is described in Section 5.2.1, to predict the future variations of states $R_m(T_m)$ and $\lambda_z^{\overline{e}}(T_m)$ proactively. Particularly, we estimate each transmission rate between the mobile device m and the edge server Ω_e for different timeslots using Eq. (5.3). Moreover, the task arrival rate of the edge server Ω_e follows a Poisson distribution. As a result, the input states of policy network π_m^{safe}

in the hyperperiod T_m can be characterized as follows:

$$s(T_m) = \{\mathcal{W}_m^a(T_m), \mathcal{W}^{\overline{e}}(T_m), \Theta_m(T_m), R_m^{\overline{e}}(T_m), \lambda_z^{\overline{e}}(T_m)\}. \tag{5.30}$$

Action

At the beginning of each hyperperiod T_m, the RL agent of mobile device m chooses an appropriate offloading policy for a set of periodic tasks $\mathcal{U}_m(T_m)$. This policy not only decides whether to offload the tasks, but also explicitly specify where to offload them. Therefore, the offloading policy of task set $\mathcal{U}_m(T_m)$ in each hyperperiod is regarded as the action of learning model. As the original policy networks of DDPG and TD3 are primarily designed to solve problems with continuous action space, they face difficulties in efficiently deadling with problems that have a large discrete action space. For instance, the action space in our offloading problem is equivalent to $|\mathbb{U}_m^{|\Omega|}|$. Apparently, when the number of edge servers increases slightly, the action space grows exponentially, making it a challenging task to handle. Similar to the work of Christodoulou *et al.* [18], the original "raw" action $\mathcal{J}_m(T_m)$ generated by policy network π_m is coded using a softmax function for vectorization to tackle the large discrete action space. However, this "raw" action does not consider the constraints of the MEC system. To ensure these constraints are met, the "raw" action $\mathcal{J}_m(T_m) = \pi_m(s_m(T_m))$ is rectified to its safe neighboring action by passing through the safety layer of policy network π_m^{safe}, where $\mathcal{J}_m^{safe}(T_m) = \mathbb{L}_m^{safe}(\mathcal{J}_m(T_m))$. Therefore, the final action of mobile device m at hyperperiod T_m can be represented as a vector of safe offloading decisions corresponding to the task set $\mathcal{U}_m(T_m)$, which are defined as follows:

$$\mathcal{J}_m^{safe}(T_m) = (\mathcal{J}_{1,m}^{1,safe}(T_m), ..., \mathcal{J}_{i,m}^{k,safe}(T_m), ...), \tag{5.31}$$

where $\mathcal{J}_{i,m}^{k,safe}(T_m)$ is the safe offloading decision associated with the task $\mu_{i,m}^k(T_m)$.

Reward

Since the objective of our offloading model is to minimize the completion time of periodic tasks from mobile device m, the periodic tasks need to be processed either at

the mobile device or at the remote servers. If the offloading decision of task $\mathcal{J}_{i,m}^{k}(T_m)$ is equal to 0, this task should be processed locally. Otherwise, it needs to be offloaded to the edge server Ω_e aligned with the offloading decision $\mathcal{J}_{i,m}^{k}(T_m) = e$. To upgrade the offloading decisions, the RL agent calculates the total reward based on the overall task completion time and the total number of deadline violations. More specifically, if a task $\mu_{i,m}^{k}(T_m)$ violates the deadline constraint (e.g., $\mathfrak{R}_{i,m}^{k}(T_m) = 1$), the task reward is assigned to a constant penalty of r^* to discourage the incorrect decision. Otherwise, the task reward is computed based on its completion time. In the context of no deadline is missed in a hyperperiod, the total reward for task set $\mathcal{U}_m(T_m)$ is simply the sum of the completion times of all tasks, that is $\overline{\mathcal{D}}_m(T_m) = \overline{\mathcal{D}}_m^{\overline{m}}(T_m) + \overline{\mathcal{D}}_m^{\overline{e}}(T_m)$. However, in the case of missed deadline in a hyperperiod, the reward calculation should be carried out separately. Let $\phi_m^{miss}(T_m)$ denote the set of tasks that miss their deadlines in hyperperiod T_m, and $\phi_m^{done}(T_m)$ denote the task set of tasks without deadline violations. Therefore, the total reward for hyperperiod T_m can be calclulated as follows:

$$\mathcal{R}_m(T_m) = \begin{cases} \overline{\mathcal{D}}_m(T_m), & \phi_m^{miss}(T_m) = \varnothing, \\ \mathcal{R}_m^{miss}(T_m) + \mathcal{R}_m^{done}(T_m), & otherwise, \end{cases} \tag{5.32}$$

where $\mathcal{R}_m^{miss}(T_m)$ denotes the total reward for the tasks violated the deadlines, which can be given by

$$\mathcal{R}_m^{miss}(T_m) = \sum_{\mu_{i,m}^{k}(T_m) \in \phi_m^{miss}(T_m)} r^* \cdot |\phi_m^{miss}(T_m)|. \tag{5.33}$$

Another reward $\mathcal{R}_m^{done}(T_m)$ denotes the total completion time of the tasks being completed before deadlines can be calculated as

$$\mathcal{R}_m^{done}(T_m) = \sum_{\mu_{i,m}^{k}(T_m) \in \phi_m^{done}(T_m)} (D_{i,m}^{k,\overline{m}}(T_m) + D_{i,m}^{k,e,\overline{e}}(T_m) + D_{i,m}^{k,\overline{c}}(T_m)). \tag{5.34}$$

Noticeably, the number of deadline misses $\mathcal{R}_m^{miss}(T_m)$ will be drastically reduced with the application of the safety layer, particularly in cases where the length of hyperperiod is relatively short. The relevant evaluations will be presented in Section 5.3.

Constraints

In deadline-sensitive MEC systems, constraint satisfaction is critical when performing deadline-sensitive task offloading. For safety concerns, the completion time of a task $\mu_{i,m}^{k}(T_m)$ must not exceed its deadline. Hence, the upper bound constant $\widetilde{\mathcal{C}}_{i,m}^{k}$ in CMDP is mapped to the deadline of task $\mu_{i,m}^{k}(T_m)$, $\widetilde{\mathcal{C}}_{i,m}^{k} = p_{i,m}^{k}$. Considering the periodicity nature of tasks, the upper bound of tasks remains constant across different hyperperiods. Therefore, the constraint of task $\mu_{i,m}^{k}(T_m)$ can be formulated as

$$\mathcal{C}_{i,m}^{k}(T_m) : \mathcal{D}_{i,m}^{k}(T_m) < p_{i,m}^{k},\ \mu_{i,m}^{k(T_m)} \in \mathcal{U}_m(T_m), \tag{5.35}$$

where $\mathcal{C}_{i,m}^{k}(T_m)$ denotes constraint of task $\mu_{i,m}^{k}(T_m)$ in hyperperiod T_m. On this basis, the constraint set $\mathcal{C}_m(T_m)$ referred to task set $\mathcal{U}_m(T_m)$ in hyperperiod T_m can be given by

$$\mathcal{C}_m(T_m) = \{\mathcal{C}_{1,m}^{1}(T_m), ..., \mathcal{C}_{i,m}^{k}(T_m), ...\}. \tag{5.36}$$

Problem Reformulation

In our work, an offloading decision is made via using the policy network π_m^{safe} to generate an appropriate offloading policy $\mathcal{J}_m^{safe}(T_m)$ regarding state $s_m(T_m)$, $\mathcal{J}_m^{safe}(T_m) = \pi_m^{safe}(s_m(T_m))$. Aiming to this goal, we strive to discover the best offloading policy $\pi_m^{*,safe}$ for task set $\mathcal{U}_m(T_m)$ that ensures minimum completion time and meeting deadlines. As our offloading problem entails minimizing the positive total reward, which is equivalent to maximizing the negative of the total award in TD3 learning, we can formulate the problem as a CMDP that can be expressed as follows:

$$\pi_m^{*,safe} = \arg\max_{\pi_m^{safe}} \sum_{T_m \in \mathcal{T}_m} \kappa^{(T_m - 1)} \cdot -\mathcal{R}_m(T_m) | \pi_m^{safe}, \tag{5.37}$$

$$\text{subject to} \quad \textit{Constraint } (5.36), \tag{5.38}$$

where κ is the discount factor, $\kappa \in (0, 1]$, that is used to calculate the discounted reward in the future.

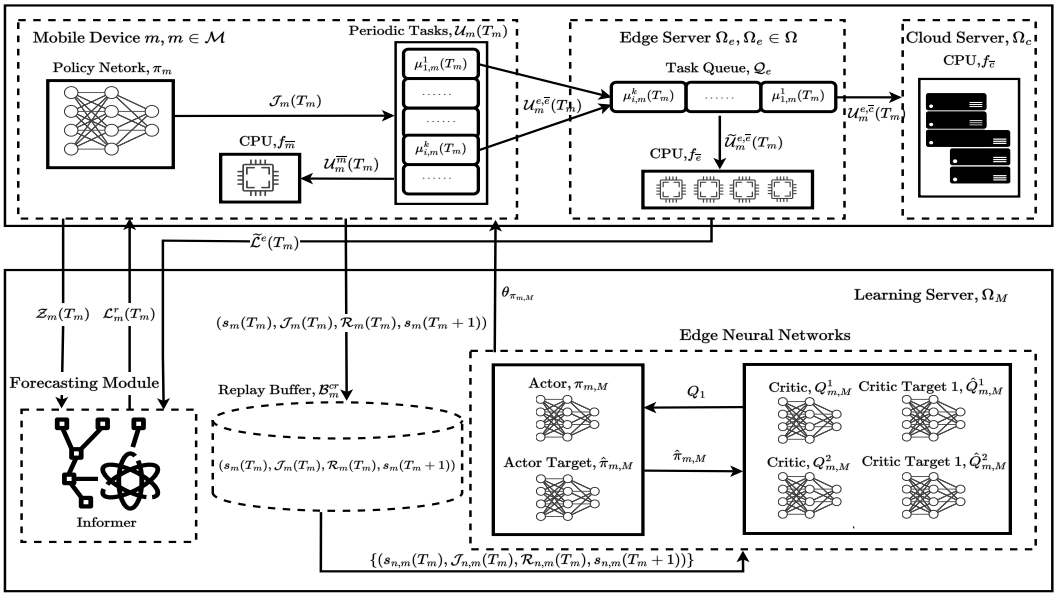


Figure 5.4: Framework of CRLO.

5.2.5 Details of CRLO

To alleviate the computation burden of mobile devices, we have developed a computationally-efficient learning-based offloading architecture. This architecture automates the optimization of offloading policies and explicitly mitigates the computation burden on the mobile device by separating model trainning from policy inference. Specifically, the module of model training that consumes massive computation resources is relocated to the resource-sufficient learning server. Alternatively, the computationally-light policy inference is regularly operated on the resource-limited mobile device. Furthermore, the forecasting module, another computation-intensive module, is also moved to the learning server. The fundamental changes in mode organization brings the benefit of freeing up substantial computation resources on mobile devices and edge servers to handle computation-intensive tasks. The framework of CRLO is demonstrated in Fig. 5.4. In this framework, mobile device m is configured with one processor that has a frequency of $f_{\overline{m}}$. In addition, there is one policy network π_m^{safe} installed on the mobile device m for periodic policy generation. It is worthy noting that the policy network π_m^{safe} only includes the orginal policy network π_m and the safety layer since the

forecasting module has been detached and relocated to the learning server. To train policy network π_m^{safe}, the replay buffer $\mathcal{B}_{m,M}^{cr}$ and pretrain buffer $\mathcal{B}_{m,M}^{p}$ are constructed accordingly at the learning server. The former regularly collects transition samples $\{(s_m(T_m), \mathcal{J}_m^{safe}(T_m), \mathcal{R}_m(T_m), s_m(T_m+1))\}$ from mobile device m, while the latter acquires one collection of pretraining samples $\{(s_m^p(T_m), \mathcal{J}_m^{p,safe}(T_m), s_m^p(T_m+1))\}$ ahead of starting the learning phase. Noticeably, both the pretraining of the safety layer and the model learning have been moved to the learning server. Hence, we use distinct notations for the neural networks at the learning server. To differentiate them from their counterparts on the mobile device m, we include an additional "M" in the subscripts. For example, π_m^{safe} represents the policy network on the mobile device m, while $\pi_{m,M}^{safe}$ is its avatar on the learning server. Furthermore, the forecasting module periodically observes the state variations of the MEC system and updates its internal information accordingly. Let $\widetilde{L}^{\overline{e}}(T_m)$ denote the varying states of the MEC system with respect to the mobile device m in the hyperperiod T_m. Notably, unlike the varying states $\mathcal{L}^r(T_m)$ being constituted with the state predictions returned from the forecasting module, state set $\widetilde{\mathcal{L}}^{\overline{e}}(T_m)$ consists of the ground truth values of the varying states in the hyperperiod T_m. With adequate global information, the forecasting module is capable of providing a variety of functionalities for all associated mobile devices. For instance, to reduce the cost of state predictions on mobile devices, the Informer forecasting model is migrated to the learning server and operated on the forecasting module. As the entire learning process is carried out remotely, mobile device m only stores one replication of the policy network π_m^{safe} locally. Moreover, to support the two aforementioned computation-intensive modules, one assumption has been made that the computation resources of the learning server are sufficient. We denote the replications of neural networks at the learning server as $\pi_{m,M}^{safe}$, $Q_{m,M}^1$, $Q_{m,M}^2$, $\hat{\pi}_{m,M}^{safe}$, $\hat{Q}_{m,M}^1$, and $\hat{Q}_{m,M}^2$, $\pi_{m,M}^{safe}$, and $\hat{\pi}_{m,M}^{safe}$, respectively. Specifically, the first three notations represent one pair of actor-critic networks and the last three notations denote the relevant target networks.

Algorithm 5: CRLO Module at Mobile Device m

Initialize the policy network π_m;
for each hyperperiod, $T_m = 1, 2, 3...$, do
    if receive new $\pi_{m,M}^{safe}$ then
        Update the policy network: $\pi_m^{safe} \leftarrow \pi_{m,M}^{safe}$;
    end
    Observe the stable states of local system $\mathcal{Z}_m(T_m)$;
    Fetch the varying states $\mathcal{L}_m^r(T_m)$ from the learning server;
    Feed the state $s_m(T_m) = \{\mathcal{Z}_m(T_m), \mathcal{L}_m^r(T_m)\}$ into policy network π_m;
    Generate the safe offloading policy $\mathcal{J}_m^{safe}(T_m)$ by using Eq. (5.39);
    Allocate the task set $\mathcal{U}_m^{\overline{m}}(T_m)$ to local processor;
    Execute the task $\mu_{i,m}^k(T_m) \in \mathcal{U}_m^{\overline{m}}(T_m)$ with frequency $f_{\overline{m}}$;
    Offload the task set $\mathcal{U}_m^{e,\overline{e}}(T_m)$ to the edge server $\Omega_e, \Omega_e \in \Omega$;
    for $\mu_{i,m}^k(T_m) \in \mathcal{U}_m^{e,\overline{c}}(T_m)$ arrives to the edge server Ω_e do
        if ($\mathcal{Q}_e$ is full) then
            Forward the task $\mu_{i,m}^k(T_m) \in \mathcal{U}_m^{e,\overline{c}}(T_m)$ to the cloud server Ω_c;
            Execute task $\mu_{i,m}^k(T_m)$ with frequency $f_{\overline{c}}$;
        end
        Execute the task $\mu_{i,m}^k(T_m) \in \widetilde{\mathcal{U}}_m^{e,\overline{e}}(T_m)$ with frequency $f_{\overline{e}}$;
    end
    Calculate the total reward $\mathcal{R}_m(T_m)$ using Eq. (5.32) ;
    Observe the next state $s_m(T_m + 1)$;
    Upload transition sample $(s_m(T_m), \mathcal{J}_m^{safe}(T_m), \mathcal{R}_m(T_m), s_m(T_m + 1))$ up to the learning server Ω_M;
end

The Module of CRLO at Mobile Device

The primary responsibility of mobile devices is to generate the most suitable offloading policy automatically, which can adapt to the variations of the MEC system and achieve the best performance. At each decision point, mobile device m regularly assesses whether to update the policy network π_m^{safe}. If the replication of the neural network $\pi_{m,M}^{safe}$ completes one-round of training with a batch of transition samples, the temporally-trained parameters are returned to the mobile device m, and the local policy network π_m^{safe} is updated accordingly, $\pi_m^{safe} \leftarrow \pi_{m,M}^{safe}$ (Line 3-5). Afterward, mobile device m observes the internal states $\Theta_m(T_m)$ and acquires the workload conditions of edge servers $\mathcal{W}^{\overline{e}}(T_m)$(Line 6). At the same time, it sends a request to the forecasting module on the learning server to fetch the varying states of the MEC system $\mathcal{L}_m^r(T_m)$, which are proactively forecasted with the Informer forecasting model (Line 7). The hybrid state $s_m(T_m) = \{\mathcal{Z}_m(T_m), \mathcal{L}_m^r(T_m)\}$ is fed into the policy network π_m^{safe} to generate the "raw" offloading policy $\mathcal{J}_m(T_m)$ (Line 8). Then, the safety layer $\mathbb{L}_m^{safe}$ modifies the "raw" offloading policy to achieve the best safe offloading policy $\mathcal{J}_m^{safe}(T_m)$. The process of safe policy generation can be expressed as follows:

$$\begin{aligned} \mathcal{J}_m^{safe}(T_m) &= \pi_m^{safe}(s_m(T_m)) = \mathbb{L}_m^{safe}(\mathcal{J}_m(T_m)) \\ &= \mathbb{L}_m^{safe}(\pi_m(s_m(T_m)) + \varepsilon), \end{aligned} \tag{5.39}$$

where ε denotes the exploration noise (Line 9). $\mathbb{L}_m^{safe}(\cdot)$ is the correction function of the safety layer.

Aligned with safe offloading policy $\mathcal{J}_m^{safe}(T_m)$, all tasks in the set $\mathcal{U}_m(T_m)$ are dispatched to specified computing resources accordingly. Namely, the task set $\mathcal{U}_m^{\overline{m}}$, which consists of all local-processing tasks, is forwarded to the mobile computation queue $\mathcal{Q}_m^{co}$ and then executed on the local processor with the speed of $f_{\overline{m}}$ (Line 10-11). Alternatively, the offloaded tasks in the set $\mathcal{U}_m^{e,\overline{e}}(T_m)$ are uploaded to the edge server Ω_e. Once scheduled to the processor of the edge server, the tasks are executed with the CPU frequency of $f_{\overline{e}}$ (Line 12). As the computation resources at edge servers are fairly competed with all other coexisting mobile devices of the MEC system, partial offloaded tasks in the set $\mathcal{U}_m^{e,\overline{e}}(T_m)$ must be forwarded to the cloud server if the edge

server Ω_e becomes overwhelmed. Hence, the task set $\mathcal{U}_m^{e,\bar{e}}(T_m)$ is divided into two subsets, $\widetilde{\mathcal{U}}_m^{e,\bar{e}}$ and $\mathcal{U}_m^{e,\bar{c}}(T_m)$, where $\widetilde{\mathcal{U}}_m^{e,\bar{e}}$ denotes the subset being executed at the edge server Ω_e, while $\mathcal{U}_m^{e,\bar{c}}(T_m)$ is the subset being offloaded to the cloud server. The details of edge computation are illustrated in Lines 12-19. At the end of hyperperiod T_m, mobile device m accumulates the total completion time of all periodic tasks and the number of deadline misses. It then calculates the total learning reward $\mathcal{R}_m(T_m)$ using Eq. (5.32) (Line 20). Additionally, mobile device m observes the next system state $s_m(T_m+1)$ and uploads the transition $(s_m(T_m), \mathcal{J}_m^{safe}(T_m), \mathcal{R}_m(T_m), s_m(T_m+1))$ to the replay buffer $\mathcal{B}_m^{cr}$ at the learning server (Line 21-22).

The Module of CRLO at Learning Server

The learning server is the most critical component in our offloading model as it is capable not only of performing model training, but also of operating a forecasting module to gather information of the MEC system globally, and then make predictions on varying states of the policy network π_m^{safe}. To be specific, the learning server must pretrain the safety layer and forecasting model independently ahead of entering the training phase of policy network π_m^{safe}. Consequenctly, during training process of policy network π_m^{safe}, the learning server can offer two fundamental services for the mobile device. Firstly, it is periodically in response to the requests of mobile device m by providing the desired varying states of the MEC system. Secondly, it constantly trains the TD3 agent with transition samples collected from the MEC system and sends partially trained parameters of the policy network to the mobile device m. The working mechanism is articulated as follows. Initially, the learning server collects a batch of randomly-executed pretrain samples from the mobile device m, denoted as $\mathcal{I}_m^p = \{s_m^p(T_m), \mathcal{J}_m^{p,safe}(T_m), s_m^p(T_m+1)\}$, where $m \in \mathcal{M}$. These samples are subsequently stored in the pretrain buffer $\mathcal{B}_{m,M}^p$ (Line 1-2). With the collection of pretrain samples, the safety signal model of the safety layer $\mathbb{L}_{m,M}^{safe}$ is trained using Eq. (5.26) (Line 3). Once pretraining is finished, the fully-trained safety layer $\mathbb{L}_{m,M}^{safe}$ is concatenated with the policy network $\pi_{m,M}^{safe}$ and its target network $\hat{\pi}_{m,M}^{safe}$ (Line 4). The forecasting module maintained by the learning server includes the informer forecasting model, which can predict the future state fluctuations of the MEC system.

When mobile device m sends a request to acquire the predictions of varying states for the upcoming hyperperiod, the learning server retrieves the disired predictions from the forecasting model and returns them to the mobile device m immediately. Let $\mathbb{F}_{m,M}$ be the Informer forecasting model used in the forecasting module. At the end of each hyperperiod T_m, the forecasting module updates its information based on a collection of data captured by the MEC system that can be represented as

$$\mathbb{S}_{m,M}(T_m) = \{\mathcal{Z}_m(T_m), \widetilde{\mathcal{L}}^e_m(T_m)\}. \tag{5.40}$$

The forecasting module leverages its predictive capabilities to anticipate the necessary varying states $\mathcal{L}^r_m(T_m)$ by utilizing the historical experience of forecasting module $\mathcal{L}^{hist}_m(T_m)$, $\mathcal{L}^r_m(T_m) = \mathbb{F}_{m,M}(\mathcal{L}^{hist}_m(T_m))$. The predicted states are subsequently returned to the mobile device m (Line 7-9).

The learning server plays important role in training the neural networks of TD3 algorithm (Line 5) and (Line 10-22). At the beginning of the training process, a set of model components at the learning server. e.g., the replay buffer B^{cr}_m and neural networks $\pi^{safe}_{m,M}$, $\hat{\pi}^{safe}_{m,M}$, $Q^1_{m,M}$, $Q^2_{m,M}$, $\hat{Q}^1_{m,M}$, $\hat{Q}^2_{m,M}$, need to be initialized before the learning process can begin systematically (Line 5). When entering the training phase, the forecasting module first updates itself regularly with the latest system information $\mathbb{S}_M(T_m)$ that is uploaded from all associated mobile devices and edge servers (Line 7). When a request for state prediction is received from the mobile device m, the corresponding varying states $\mathcal{L}^r_m(T_m)$ are predicted using the Informer $\mathbb{F}_{m,M}$ based on a series of historical records (Line 8). The forecasted results $\mathcal{L}^r_m(T_m)$ are immediately returned to the mobile device m (Line 9). At the end of a hyperperiod T_m, the replay buffer $\mathcal{B}^{cr}_{m,M}$ receives a transition sample $(s_m(T_m), \mathcal{J}^{safe}_m(T_m), \mathcal{R}_m(T_m), s_m(T_m+1))$ from the mobile device m (Line 10). To train the critic networks, a mini-batch of transition samples $\mathcal{I}^{tr}_m = \{s_{n,m}(T_m), \mathcal{J}^{safe}_{n,m}(T_m), \mathcal{R}_{n,m}(T_m), s_{n,m}(T_m+1)\}$ previously recorded in the replay buffer $\mathcal{B}^{cr}_{m,M}$ is arbitrarily selected (Line 11). In order to reduce the variance of stochastic training and prevent overfitting to a local optimum, the smoothing regularization is utilized to mitigate the impact of the variance on policy generation (Line 12). As a result, the generated safe offloading policies referred to the selected samples are revised slightly by adding a small amount of clipped noise as

$$\begin{aligned}\mathcal{J}_{n,m}^{safe}(T_m+1) :&\widetilde{\mathcal{J}}_{n,m}^{safe}(T_m+1) \leftarrow \hat{\pi}_{m,M}^{safe}(s_{n,m}(T_m+1)) + \varepsilon,\\ &\varepsilon \sim clip(\mathcal{N}(0,\widetilde{\sigma}), -c, c).\end{aligned} \tag{5.41}$$

Unlike DDPG that can induce substantial overestimation by using a single estimated Q-value to derive the value target, TD3 utilizes two estimated values to minimize deviation from the ground-truth value target. In each iteration, two estimated Q-values are derived according to Eq. (5.42) and Eq. (5.43)

$$\hat{Q}_{m,M}^{1,target} = \hat{Q}_{m,M}^{1}(s_{n,m}(T_m+1), \mathcal{J}_{n,m}^{safe}(T_m+1)), \tag{5.42}$$

$$\hat{Q}_{m,M}^{2,target} = \hat{Q}_{m,M}^{2}(s_{n,m}(T_m+1), \mathcal{J}_{n,m}^{safe}(T_m+1)). \tag{5.43}$$

Next, the minimum estimated Q-value is used to calculate the value target $\mathbb{Y}_{n,m}$ that can be given by

$$\mathbb{Y}_{n,m} \leftarrow \mathcal{R}_{n,m}(T_m) + \kappa \cdot \min_{i=1,2}(\hat{Q}_{m,M}^{1,target}, \hat{Q}_{m,M}^{2,target}). \tag{5.44}$$

Let $\theta_{m,M}^{1,Q}$ and $\theta_{m,M}^{2,Q}$ denote the parameters of the critic networks $Q_{m,M}^{1}$ and $Q_{m,M}^{2}$, respectively. The parameters of critic networks $\theta_{m,M}^{1,Q}$ and $\theta_{m,M}^{2,Q}$ can be updated based on the value target by minimizing the loss function as follows:

$$\begin{aligned}\theta_{m,M}^{1,Q} \leftarrow &\theta_{m,M}^{1,Q} - \alpha \sum_{\mathcal{I}_m^{tr}} \frac{\mathrm{d}Q_{m,M}^{1}(s_{n,m}(T_m), \mathcal{J}_{n,m}^{safe}(T_m))}{\mathrm{d}\theta_{m,M}^{1}}\\ &\cdot (Q_{m,M}^{1}(s_{n,m}(T_m), \mathcal{J}_{n,m}^{safe}) - \mathbb{Y}_{n,m})\end{aligned} \tag{5.45}$$

$$\begin{aligned}\theta_{m,M}^{2,Q} \leftarrow &\theta_{m,M}^{2,Q} - \alpha \sum_{\mathcal{I}_m^{tr}} \frac{\mathrm{d}Q_{m,M}^{2}(s_{n,m}(T_m), \mathcal{J}_{n,m}^{safe}(T_m))}{\mathrm{d}\theta_{m,M}^{2}}\\ &\cdot (Q_{m,M}^{2}(s_{n,m}(T_m), \mathcal{J}_{n,m}^{safe}) - \mathbb{Y}_{n,m})\end{aligned} \tag{5.46}$$

After a certain number of ϱ hyperperiods, the policy network $\pi_{m,M}^{safe}$ is updated using the deterministic policy gradient with the assistance of critic network $Q_{m,M}^{1}$ (Line 16). Let $\theta_{m,M}^{safe}$ denote the parameters of the policy network $\pi_{m,M}^{safe}$. The update of $\theta_{m,M}^{safe}$ can be expressed as follows:

$$\begin{aligned}\theta_{m,M}^{safe} \leftarrow \theta_{m,M}^{safe} + &\beta \cdot \sum_{\mathcal{I}_m^{tr}} \frac{\mathrm{d}Q_{m,M}^{1}(s_{n,m}(T_m), \mathcal{J}_{m}^{safe}(T_m))}{\mathrm{d}\mathcal{J}_{m}^{safe}(T_m)}\\ &\cdot \frac{\pi_{m,M}^{safe}(s_{n,m}(T_m))}{\mathrm{d}\theta_{m,M}^{safe}}.\end{aligned} \tag{5.47}$$

When the parameters of safe actor $\theta_{m,M}^{safe}$ are updated, the latest version of the updated parameters is downloaded to the mobile device m (Line 17). Let $\hat{\theta}_{m,M}^{safe}$, $\hat{\theta}_{m,M}^{1,Q}$, and $\hat{\theta}_{m,M}^{2,Q}$ denote the parameters of the actor target and critic targets. The last step of the learning process in each iteration is to update the parameters of the target networks (Lines 19-20). The learning process will continue until it meets the predefined stop conditions.

Complexity Analysis

The total computation complexity of handling task set $\mathcal{U}_m$ on the mobile device m consists of two parts: the complexity of the multi-layer task offloading and the complexity of the multiple functionalities of the learning server. In this subsection, we will analyze the complexity of these two parts independently. The first part of complexity involves the policy generation complexity on the mobile device m and the task execution complexity at different computing tiers. The other part of complexity includes the complexity of the safety layer pretraining, the complexity of maintaining the forecasting module, and the complexity of TD3 trainning. Since the update period of the policy network $\pi_{m,M}^{safe}$ is set to one episode ($\varrho \times |T_m|$), we will conduct the complexity analysis for this length of time. For each episode, the mobile device m needs to carry out both policy generation and local task execution ϱ times. Let Υ_m^f denote the number of multiplication operations required for one forward propagation of the policy network, and Υ_m^l denote the operations required for processing local tasks in a hyper-period aligned with the chosen offloading policy. Hence, the computation complexity of mobile device m is $\mathcal{O}(\varrho^2 \Upsilon_m^l \Upsilon_m^f)$. Accordingly, the unique role of edge servers and cloud server is to process the offloaded tasks from the mobile device m. Let $\Upsilon_m^{\overline{e}}$ and $\Upsilon_m^{\overline{e},c}$ denote the multiplication operations required to process the offloaded tasks uploaded from the mobile device m to the edge servers and cloud server, respectively. The corresponding computation complexities of the edge servers and cloud server are $\mathcal{O}(\varrho \Upsilon_m^{\overline{e}})$ and $\mathcal{O}(\varrho \Upsilon_m^c)$. Therefore, the overall computation complexity of multi-tier task offloading referred to the mobile device m is $\mathcal{O}(\varrho^2 \Upsilon_m^l \Upsilon_m^f + \varrho \Upsilon_m^{\overline{e}} + \varrho \Upsilon_m^c)$. It should be noted that the complexity is identical both during training and after training since the learning modules have been entirely moved to the learning server.

Algorithm 6: CRLO Module at Learning Server

Collect a set of safe training samples from the mobile device m: $\mathcal{I}_m^p = \{s_m^p(T_m), \mathcal{J}_m^{p,safe}(T_m, s_m^p(T_m+1))\}$;
Store the collected pretrain set into the pretrain buffer $\mathcal{B}_{m,M}^p$;
Pretrain the safety signal model of safety layer for the mobile device m using Eq. (5.26);
Concatenate the pretrained safety layer $\mathbb{L}_m^{safe}$ with the policy netwok $\pi_{m,M}^{safe}$ and the target network $\hat{\pi}_{m,M}^{safe}$;
Initialize the neural networks at the learning server $\pi_{m,M}^{safe}$, $\hat{\pi}_{m,M}^{safe}$, $Q_{m,M}^1$, $Q_{m,M}^2$, $\hat{Q}_{m,M}^1$, $\hat{Q}_{m,M}^2$;
for each hyperperiod, $T_m = 1, 2, 3...$, do
    Update the forecasting module based on the global information of the MEC system $\mathbb{S}_{m,M}(T_m)$;
    Generate the predictions of varying states using forecasting model: $\mathcal{L}_m^r(T_m) = \mathbb{F}_{m,M}(\mathcal{L}_m^{hist}(T_m))$;
    Return the forecasted varying states $\mathcal{L}_m^r(T_m)$ to the mobile device m;
    Store the transition sample of mobile device m $(s_m(T_m), \mathcal{J}_m^{safe}(T_m), \mathcal{R}_m(T_m), s_m(T_m+1))$ into the replay buffer $\mathcal{B}, \mathcal{M}_m^{cr}$;
    Sample a mini-batch of transitions $\mathcal{I}_m^{tr}$ from $\mathcal{B}_{m,M}^{cr}$;
    Smooth the offloading policy $\mathcal{J}_{n,m}^{safe}(T_m)$ in the set $\mathcal{I}_m^{tr}$ using Eq. (5.41);
    Calculate the value target $\mathbb{Y}_{n,m}$ with Eq. (5.44);
    Update the parameters of critic networks $\theta_{m,M}^{Q,1}$ and $\theta_{m,M}^{Q,2}$ using Eq. (5.45) and Eq. (5.46);
    if $(T_m = \varrho)$ then
        Update actor $\pi_{m,M}^{safe}$ using Eq. (5.47);
        Send the parameters of the policy network $\pi_{m,M}^{safe}$ to the mobile device m;
        Update the target networks $\hat{\pi}_{m,M}^{safe}$, $\hat{Q}_{m,M}^1$, $\hat{Q}_{m,M}^2$ using the equations as follows: $\hat{\theta}_{m,M}^{safe} \leftarrow \tau \cdot \theta_{m,M}^{safe} + (1-\tau) \cdot \hat{\theta}_{m,M}^{safe}$;
        $\hat{\theta}_{m,M}^{i,Q} \leftarrow \tau \cdot \theta_{m,M}^{i,Q} + (1-\tau) \cdot \hat{\theta}_{m,M}^{i,Q}, i = 1, 2$;
    end
    Until reach the predefined stop conditions;
end

In terms of the complexity of the learning server, it primarily consists of three components. The first component is due to the pretraining of safety layer. Let $\Upsilon^{s}_{m,M}$ denote the multiplication operations required for pretraining the safety layer $\mathbb{L}^{safe}_{m}$ at the learning server. Then, the complexity of the first component is $\mathcal{O}(|\mathcal{I}^{p}_{m}|\Upsilon^{s}_{m,M})$. The second component is produced because of the maintenance of the forecasting module. Let $\Upsilon^{d,U}_{m,M}$ denote the cost for the updating the forecasting module, and $\Upsilon^{d,I}_{m,M}$ denote the forecasting cost of system states. Thus, the complexity of forecasting module in an episode is $\mathcal{O}(\varrho(\Upsilon^{d,U}_{m,M} + \Upsilon^{d,I}_{m,M}))$. The last component is the complexity of training neural networks of TD3. Let $\Upsilon^{b,ac}_{m,M}$ and $\Upsilon^{b,cr}_{m,M}$ denote the required multiplication operations for backward propagation of actor and critic network, respectively. In each episode, TD3 updates actor network once and two critic networks ϱ times. Therefore, the computational complexity for the training of neural networks is $\mathcal{O}(|I^{tr}_{m}|(2\varrho\Upsilon^{b,cr}_{m,M} + \Upsilon^{b,ac}_{m,M}))$. After model training, the computation complexity of learning server, which only consists of the first component, is reduced significantly to $\mathcal{O}(\varrho(\Upsilon^{d,U}_{m,M} + \Upsilon^{d,I}_{m,M}))$.

5.3 Evaluation

In this section, we first describe the evalution settings of our experiments. Next, the convergence of our offloading scheme is presented. Afterwards, we compare the proposed offloading scheme with a few baseline methods in the scenarios of varied task arrival rates and hyperperiod lengths. Furthermore, the impact of the forecasting model (i.e., Informer) and the safety layer on CRLO are evaluated separately. Finally, we demonstrate the scalability of our offloading scheme by varying the number of edge servers.

5.3.1 Evaluation Settings

We consider a non-stationary MEC system that covers a $\pi \times 200m \times 200m$ circular region, evenly encompassing 1 to 5 edge servers. Each edge server is connected to the same cloud server. Each mobile device m communicates with all available edge servers in the region simultaneously, where $\mathcal{L}^{e}_{m} = 200$. The experimental settings are described in Table 5.2. To simulate a time-dependent environment and reveal the

Table 5.2: Simulation Parameters in Chapter 5.

Parameter	Value	Description
$\|\mathcal{U}_m\|$	[4, 10]	Number of periodic tasks in $\mathcal{U}_m$
$p_{i,m}^k$	[4, 36]	Period of tasks
$d_{i,m}^k$	[10, 100] Mb	Data size of tasks
$c_{i,m}^k$	$[1, 10] \times 10^9$ cycles	Required CPU cycles of tasks
$f_{\overline{m}}$	1 GHz	CPU speed of mobile device
$f_{\overline{e}}$	4 GHz	CPU speed of edge server
$f_{\overline{c}}$	8 GHz	CPU speed of cloud server
$\mathcal{P}_{trans}$	64 mWatt	Transmission power
R_e	(0, 300] Mbps	Transmission rates
ℓ	2	Path loss exponent
λ_m^a	0.1	Arrival rate of aperiodic task
$\mathcal{D}_e^{p,\overline{c}}$	0.6 seconds	Roundtrip cloud propagation delay
$\mathcal{I}_m^{tr}$	512	Mini-batch size of training samples
κ	0.99	Discount rate
ϱ	100 x $\|T_m\|$	Updating episode of $\pi_{m,M}^{safe}$

impact of forecasting model on decision-making, we model the task arrival process of edge server λ_z^e as a Markov chain with four transition states. The states are defined with "low=$\lambda_{z,1}^e$", "medium=$\lambda_{z,2}^e$", "high=$\lambda_{z,3}^e$", and "very high=$\lambda_{z,4}^e$". Each of these states is the hyperparameter of a Poisson distribution. Another feature that needs to be predicted using the forecasting model is the transmission rate between a mobile device and an edge server. As discussed in Section 5.1.3, the transmission rate is explicitly determined by the channel gain $|\mathcal{G}_m^e|^2$, which follows an exponential distribution. The rate parameter of the exponential distribution also consists of four transition states in our model, denoted as $\lambda_{m,1}^{e,ch}$, $\lambda_{m,2}^{e,ch}$, $\lambda_{m,3}^{e,ch}$, and $\lambda_{m,4}^{e,ch}$. To verify the effectiveness of our proposed model under different time-dependent transition states, we create a four-state Markov chains similar to [109]. In this Markov chain, the transition possibility matrix $\mathbb{P}_{Mar}$ is formulated as follows:

$$\mathbb{P}_{Mar} = \begin{pmatrix} 0 & 1.0 & 0 & 0 \\ 0.25 & 0 & 0.30 & 0.45 \\ 0 & 1.0 & 0 & 0 \\ 0 & 1.0 & 0 & 0 \end{pmatrix}. \tag{5.48}$$

Notice that the transition frequency of Markov states is set to 20 time slots.

As shown in Fig. 5.4, our proposed offloading method adopts a decoupled actor-critic architecture that separates the training modules and policy inference into different computing tiers. More concisely, the policy network and critic networks are each composed of three fully connected hidden layers, with 1024 neurons in each layer. The size of the replay buffer $\mathcal{B}_m^{cr}$ is set to 10^6, and the mini-batch size for training in each iteration is set to 512. In addition, we set the learning rate to 10^{-7} and the discounted rate to 0.99. It is worth noting that the degradation of additive noise in policy generation is utilized during the training process. Hence, the coefficient of additive noise ε is gradually decreased from 0.5 to 0.1 during the training process. In the safety layer, the neural network consists of three hidden layers, and each layer contains 2048 neurons. The learning rate for the safety layer is set to 10^{-6}. To sufficiently pretrain the safety layer, we utilize a total number of 10240 samples. The hyperparameters for "Informer" forecasting model are configured consistently with the work [111].

We compare the performance of our proposed scheme with the following baseline offloading methods:

(i) All-SERVER: All tasks are purely offloaded to edge servers.

(ii) All-RANDOM: All tasks are kept local or offloaded to an edge server randomly.

(iii) LSTM-O: All tasks are kept local or offloaded to an edge server using the TD3 learning algorithm enhanced by the LSTM forecasting model. LSTM-O is based on [84], where Double Deep Q-Network (DDQN) is used as the learning algorithm and LSTM is used to provide predicted system states for DDQN. For the purpose of fair comparison, the offloading scheme in [84] is modified by replacing DDQN with TD3, which finally leads to this baseline method, LSTM-O.

(iv) MATCH-O: In accordance with matching theory and under a set of constraints, all tasks are processed either on the mobile device or one of the edge servers [34]. Initially, each task is randomly allocated to one of the processing units (mobile device or edge servers). Subsequently, the tasks are swapped continuously until the total task utility is minimized.

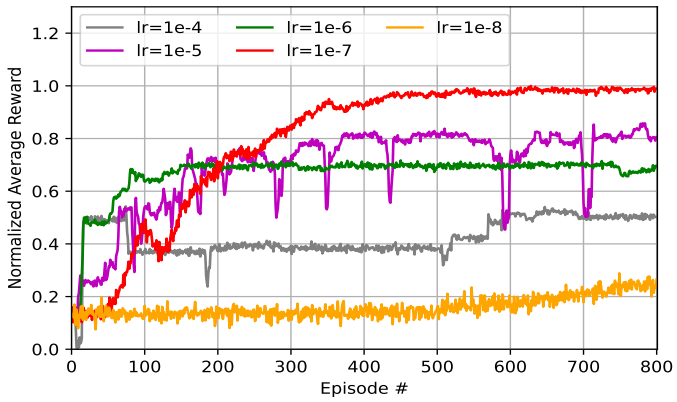
(v) PPO-O: All tasks are executed either on a mobile device or an edge server utilizing the PPO learning algorithm, wherein reward shaping is employed to penalize deadline violations [90].

(vi) TD3-O: All tasks are are kept local or offloaded to an edge server according the TD3 learning algorithm, in which reward shaping is similarly used to penalize deadline misses [41].

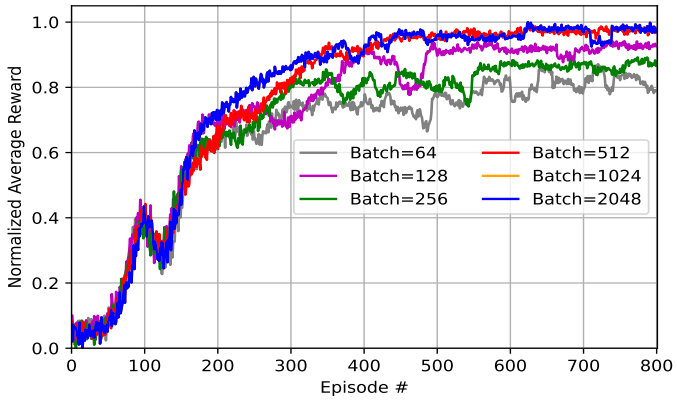
MATCH-O is selected because it addresses the constrained task offloading problem using matching theory in multi-edge-server MEC systems. LSTM-O is similar to our approach, focusing on offloading non-divisible tasks among multiple edge servers and aiming to enhance the offloading policy with more representative inputs to the policy network. PPO-O and TD3-O represent existing works dedicated to addressing similar offloading problems using on-policy and off-policy reinforcement learning techniques, respectively. In our research, we evaluate the performance of the offloading models using two metrics: 1) min-max normalized completion time and 2) number of deadline misses

5.3.2 Convergence

To demonstrate validation of the proposed offloading model in terms of convergence, we perform the model in the context of different hyperparameters of neural networks. In the experiments, we investigate the average reward of the proposed algorithm across 600 episodes, each of which consists of 100 hyperperiods. Each hyperperiod represents one interaction of the mobile device with the MEC system, taking the action generated by current policy network π_m^{safe}. Four transition states of transmission rate and task arrival rate are set to $[0.5, 1.0, 1.5, 2.0]$. We set $|\mathcal{U}_m|$ to 10, resulting in a total number



(a) Impact of Learning Rate on Convergence



(b) Impact of Batch Size on Convergence

Figure 5.5: Convergence of CRLO

of tasks in a hyperperiod $|\mathbb{U}_m| = 88$, and a length of a hyperperiod $|T_m| = 80$ time slots. The simulation results are presented in Fig. 5.5, where the x-axis indicates the episode of learning, and the y-axis shows the normalized average reward of episodes. Fig. 5.5(a) depicts the convergence of our proposed approach under various learning rate settings, denoted as "lr" in the figure. Note that in our experiments, the learning rate refers to the step size of the policy network in each episode towards the optimal value of the loss function. As shown in the figure, when the learnig rate is set to lr= 10^{-4}, it leads to unstable convergence and suboptimal average reward since a large step size implies a comparatively fast optimization process, which can potentially

miss the best value inadvertently and converge to the local optimum. However, as the learning rate steadily increases to lr= 10^{-5} and lr= 10^{-6}, both the learning rates still correspond to a relatively small average reward due to an inappropriate step size in terms of learning exploration. When the learning rate set to a small value, e.g., lr= 10^{-8}, the model training is too slow and hardly converges to the best average reward. Although a relatively small learning rate, e.g., lr= 10^{-7}, induces a slightly slower convergence rate, it allows the RL agent to gradually learn until converging to the optimal average reward. Fig. 5.5(b) illustrates the convergence of the proposed offloading model with different batch sizes of training samples $\mathcal{B}^{cr}_{m,M}$. As seen in the figure, when the batch size is increased from 64 to 256, the improvement in convergence speed and average reward is insignificant. When we further increase the batch size to 512, the model can learn smoothly and converge to the best reward after being trained for fewer than 500 epochs. However, with a larger batch size, e.g., 1024 and 2048, the performance remains identical with the bath size of 1024. It indicates that when the size of training samples reaches a threshold, e.g., 512, expanding the batch size would no longer improve the average total reward.

5.3.3 Completion Time and Deadline Misses

In this section, we evaluate the proposed offloading method by comparing its performance with other counterparts described in Section 5.3.1. In the experiments, the average task arrival rate $\lambda^{e}_{z,avg}$ is calculated as the average of the rates of the four state transitions. For instance, when the four state transitions are set to $[0.2, 0.4, 0.6, 0.8]$, the average task arrival rate $\lambda^{e}_{z,avg}$ is equal to 0.5. In Fig. 5.6 shows that MATCH-O outperforms ALL-RANDOM and ALL-SERVER. However, it is less effective than all learning-based algorithms. Furthermore, it is observed that learning-based algorithms achieve lower completion times and fewer deadline misses as the average task arrival rate increases gradually. Particularly, the performance PPO-O is similar to that of TD3-O, but worse than that of CRLO. The reason is that CRLO not only reduces overestimation with its double-critic structure but also achieves safe policy generation with its forecasting model and safety layer. Furthermore, LSTM-O slightly outperforms TD3-O, implying that the LSTM forecasting model is effective in predicting

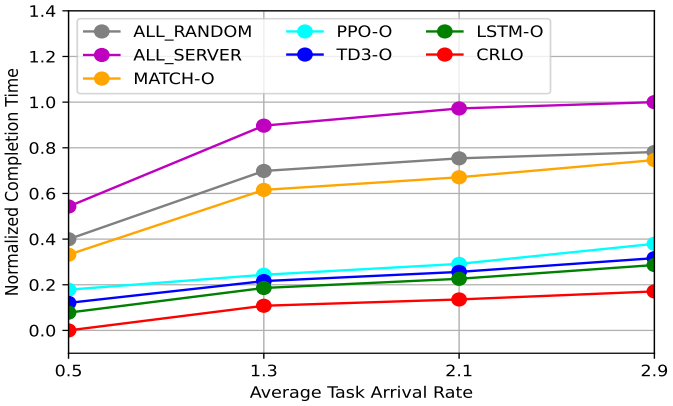
MEC states. On the other hand, CRLO maintains the shortest completion time and the fewest number of deadline misses when compared to other baseline methods as the task arrival rate increases. In particular, when the average task arrival rates are small, for instance, $\lambda^{e}_{z,avg} = 0.5$, there is enormous optimized space for learning-based offloading algorithms to learn how to effectively distribute computation-intensive tasks to the most appropriate computing tiers, resulting in optimized completion time and reduced number of deadline misses. As the task arrival rate increases from 0.5 to 1.3, CRLO improves the performance of completion time and miss rate to 9.8% and 2.5% compared to the second-best algorithm (LSTM-O). As the average task arrival rate increases to 2.1 and 2.9, the edge servers become overwhelmed due to the massive tasks offloaded from other mobile devices. In contrast to other baseline methods, which result in significant increases in both completion time and deadline misses as the task arrival rate goes up, the completion time of CRLO experiences only slight increases of 15.8%, 4.4%, and 6.4%, respectively. Additionally, the associated deadline miss rate does not exceed 5.5%.

We also investigate the correlation between the length of hyperperiod and the performance of the offloading algorithms. In the experiments, the hyperperiod lengths are configured with 40, 48, 64, and 80 time slots, which correspond to 29, 34, 45, and 88 tasks, respectively. The workloads of the hyperperiods are 60%, 98%, 115%, and 260%, respectively. In Fig. 5.7(a), we can observe that the CRLO algorithm consistently achieves the lowest average completion time as the length of hyperperiod is enlarged, particularly when the length of hyperperiod is greater than 48 time slots. Note that when the length of hyperperiod is relatively small, such as $|T_m| = 40$ and $|T_m| = 48$, MATCH-O demonstrates comparable performance with other learning-based algorithms. Additionally, all learning-based offloading algorithms achieve the same level of performance. However, as the length gradually increases up to 80 time slots, CRLO outperforms other baseline methods significantly. For instance, the average completion time of CRLO is shorter than that of PPO-O, TD3-O, and LSTM-O by 22.8%, 15.8%, and 11.9%, respectively. The reason behind this observation is that as the hyperperiod length increases, the impact of state variations on decision-making is aggravated accordingly. Without proper measures to improve the fidelity

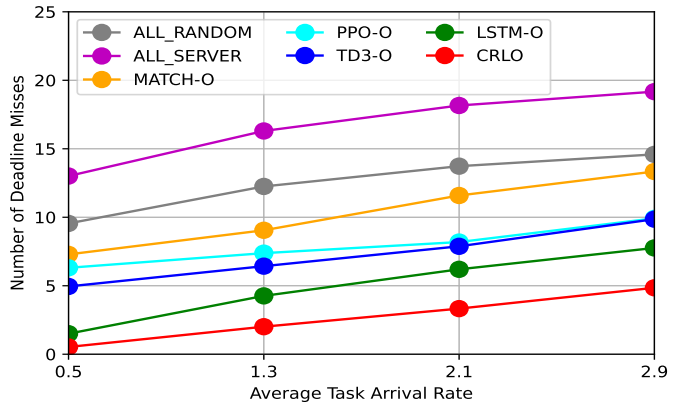
of the long-sequence state inputs of the policy network, both TD3-O and LSTM-O struggle to generate optimal offloading decisions. Fig. 5.7(b) illustrates that as the hyperperiod length increases, the number of deadline misses also goes up. It can be seen that when the length of the hyeperperiod is shorter than a threshold, e.g., $|T_m| = 48$ in our case, all learning-based algorithms can strictly guarantee the deadline satisfaction. However, when the length is expanded to larger than 48, only CRLO can maintain practical rates of deadline miss, which are 0.3% and 2.1% for $|T_m| = 64$ and $|T_m| = 80$, respectively. This is because when the length of the hyperperiod becomes larger, more workload of the mobile device needs to be uploaded to edge servers. Due to the uncertainties in the MEC system, more deadline misses are incurred. Additionally, longer hyperperiods undermine the performance of LSTM-O, whose miss rate dramatically raises to 6.19% when the hyeperperiod length is equal to 80. However, under the same configuration, CRLO achieves only a 2.1% miss rate.

5.3.4 Scalability

To prove the robustness of CRLO, we gradually increase the number of edge servers to investigate how this configuration affects the scalability and overall performance of our offloading method. As the number of edge servers increases, the All-Server and All-Random offloading methods exhibit limited enhancements in terms of average completion time and deadline misses. This can be attributed to the limited ability of these methods to effectively utilize the augmented computational capacity offered by the edge servers, thereby resulting in unsatisfactory performance. Furthermore, the performance of these two offloading methods is inferior compared to the other baseline algorithms. Thus, in Fig. 5.8, we particularly compare the average completion time of MATCH-O and other three learning-based offloading algorithms. We observe that the performance of MATCH-O degrades significantly as the number of edge servers decreases since it is hard to yield accurate matching with insufficient information in a resource-limited MEC system. Additionally, it is also noteworthy that both average completion time and the number of deadline misses of all learning-based algorithms decline gradually as the number of edge servers increases. It should be noted that CRLO constantly outperforms the other algorithms in all the experiments. This is



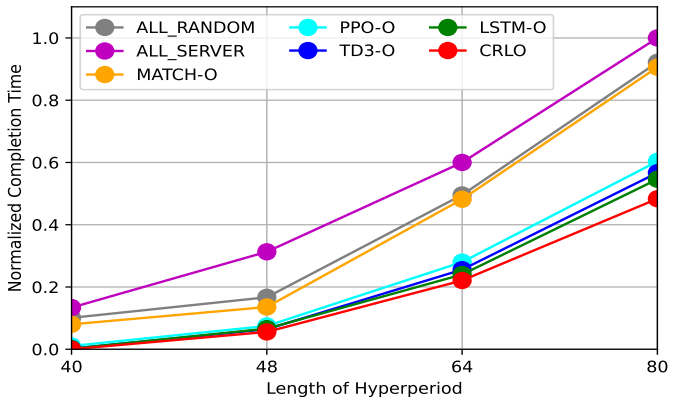
(a)



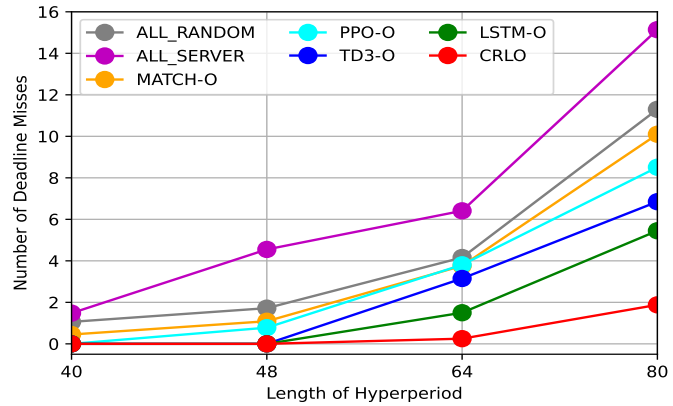
(b)

Figure 5.6: Performance of CRLO: Varied Average Task Arrival Rates

because CRLO not only considers the promotion on the input states but also regulates the number of deadline misses. As shown in Fig. 5.8, when the MEC system consists of a small number of edge servers, e.g., $|\Omega| = 1$, PPO-O, TD3-O, and LSTM-O lead to worse results than CRLO, with an average completion time at least 12.2% slower and a deadline miss rate 16.3% higher. The reason for this observation is that when the capacity of the MEC system is limited, these three learning-based counterparts struggle to make optimal offloading decisions as they have difficulty in balancing the optimization of average completion and deadline misses. As the number of edge servers increases from 2 to 5, it indicates that the capacity of the



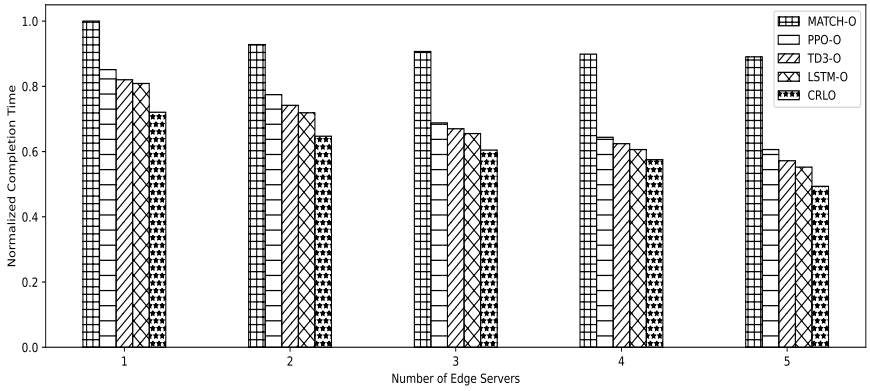
(a)



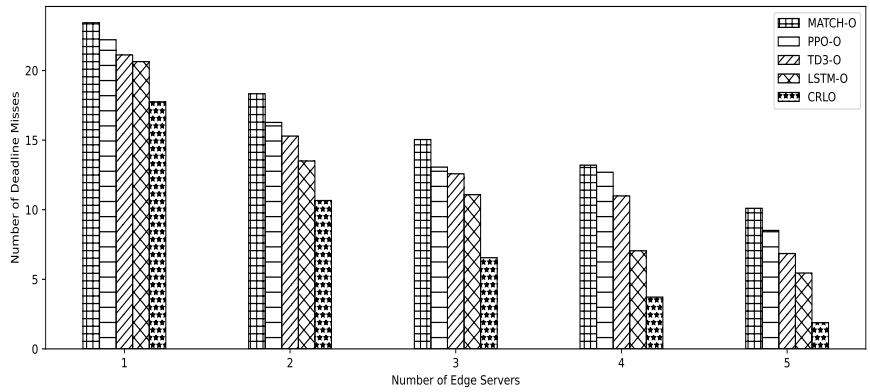
(b)

Figure 5.7: Performance of CRLO: Varied Lengths of Hyperperiod

MEC system has been gradually grown. Correspondingly, the performance of CRLO improves steadily, especially when the number of edge servers increases to 5, for which the average completion time of CRLO decreases significantly by 46% and the miss rate is also largely reduced to 2.1% compared to the case when $|\Omega| = 1$. The experiments support our speculation that the performance of the MEC system can be effectively enhanced by increasing the number of edge servers.



(a)



(b)

Figure 5.8: Scalability of CRLO

5.3.5 Impact of Informer and Safety Layer

To investigate the impact of informer and safety layer on CRLO, we carry out a series of experiments in the context of different average task arrival rates and hyperperiod lengths. To assess the impact, we developed two additional learning-based offloading models, TD3-IO and TD3-SO, which are uniquely designed to include either the Informer or safety layer. It is worth noting that, similar to TD3-O, TD3-IO also leverages reward shaping as a safety measure during model learning. Moreover, our analysis includes two boundary methods, TD3-O and CRLO, that have been described in Section 5.3.1.

As shown in Table 5.3, CRLO perpetually achieves lower task completion time and miss rate than TD3-IO and TD3-SO, with the figures always being better than those for TD3-O. Notably, when edge servers have massive idle computation resources, e.g., $\lambda^e_{z,avg} = 0.5$, the impact of Informer and safety layer on average completion time and deadline misses is similar. There are two main reasons for this observation. First, since only a small number of tasks from other mobile devices are offloaded to edge servers, the RL agent has sufficient space to improve offloading decisions. Second, even if the chosen offloading decisions are suboptimal, the performance of the MEC system remains acceptable and close to that of CRLO. That is because the average completion time and miss rate remain low when the MEC system is not busy, leaving little room for optimization with both Informer and safety layer. As the average task rate increases from 1.3 to 2.9, TD3-SO shows slightly worse performance than TD3-IO, with an average completion time that is roughly 3% longer and miss rate that is 1% higher. This observation suggests that when more tasks arrive at edge servers within a hyperperiod, the MEC environment becomes more dynamic. In such circumstances, the impact of Informer is more significant than safety layer in terms of decision-making. In particular, when $\lambda^e_{z,avg} = 2.9$, Informer can improve average completion time and miss rate by 9% and 4.5%, respectively, compared to TD3-O. These improvements are slightly higher than those of TD3-SO, which improve TD3-O by 5% and 3.8%, respectively.

To further investigate the impact of the Informer and safety layer on CRLO, we vary the lengths of the hyperperiod and consider different numbers of tasks in a hyperperiod as discussed in Section 5.3.3. As shown in Table 5.4, the impact of both Informer and safety layer graudually agrravate as the length of hyperperiod increases. When the hyperperiod is short, e.g., $|T_m| = 40$, both of technologies achieve similar improvements compared to TD3-O. This implies that if the mobile device is not busy and only executes a small number of tasks, reward shaping is sufficient to ensure deadline satisfaction, as in TD3-IO. Alternatively, since the hyperperiod is relatively short, the impact of state forecasting is insignificant, resulting in decent performance of TD3-SO becasue TD3-SO makes offloading decisions based only on immediate and historical states without taking into account the intrinsic patterns of

Table 5.3: Impact of Informer and Safety Layer: Varied Average Task Arrival Rates

	Normalized Completion Time			
Parameter	TD3-O	TD3-IO	TD3-SO	CRLO
$\lambda^e_{z,avg}$=0.5	1	0.86	0.87	0.85
$\lambda^e_{z,avg}$=1.3	1	0.91	0.94	0.88
$\lambda^e_{z,avg}$=2.1	1	0.92	0.95	0.87
$\lambda^e_{z,avg}$=2.9	1	0.91	0.95	0.87
	Deadline Miss Rate			
Parameter	TD3-O	TD3-IO	TD3-SO	CRLO
$\lambda^e_{z,avg}$=0.5	5.6%	0.7%	0.6%	0.6%
$\lambda^e_{z,avg}$=1.3	7.3%	2.8%	3.8%	2.3%
$\lambda^e_{z,avg}$=2.1	9.0%	4.7%	5.9%	4.0%
$\lambda^e_{z,avg}$=2.9	11.2%	6.7%	7.4%	5.5%

state transition. As the length of the hyperperiod increases from 48 to 80, it is seen that Informer has a greater impact on performance optimization, achieving 7%, 10%, and 12% improvements in average compeltion time, and 0.9%, 3.7%, and 4.0% in the rate of deadline miss with respect to TD3-O. Nevertheless, the effect of safety layer increases more slowly along with the expansion of the hyperperiod. This is because as the length of the hyeperperiod increases, it becomes more difficult for realtime and historical states to accurately uncover the upcoming state variations in a hyperperiod. As a result, the effectiveness of TD3-SO is significantly undermined.

5.4 Major Conclusions of CRLO

In this chapter, we propose a safety-critical RL-based offloading scheme, CRLO, which automatically generates the appropriate offloading policies for computation-intensive tasks on mobile devices in MEC systems. Specifically, we devise a novel offloading policy network that adopts a forecasting model and a safety layer to safely generate efficient offloading policies. With the forecasting model, long-term system states could be utilized to help arrive at effective offloading decisions. The safety layer regulates the output of the RL agent so that unsafe offloading decisions are calibrated before

Table 5.4: Impact of Informer and Safety Layer in the Scenarios of Varied Hyperperiod Lengths

	Normalized Completion Time			
Parameter	TD3-O	TD3-IO	TD3-SO	CRLO
$\|T_m\| = 40$	1	0.98	1	0.96
$\|T_m\| = 48$	1	0.93	0.98	0.91
$\|T_m\| = 64$	1	0.90	0.96	0.88
$\|T_m\| = 80$	1	0.88	0.94	0.86
	Deadline Miss Rate			
Parameter	TD3-O	TD3-IO	TD3-SO	CRLO
$\|T_m\| = 40$	0	0	0	0
$\|T_m\| = 48$	1.1%	0.2%	0.3%	0
$\|T_m\| = 64$	4.9%	1.2%	1.4%	0.4%
$\|T_m\| = 80$	7.8%	3.8%	4.3%	2.1%

they are executed. Furthermore, we develop a multi-layer task offloading structure, where all resource-consuming learning modules are relocated to a learning server at the network edge. In this manner, most of the computation resources on mobile devices and edge servers could be dedicated to the execution of computation-intensive tasks. Our experimental results indicate that the proposed offloading scheme outperforms the baseline offloading methods in terms of task completion time and deadline satisfaction. Our potential future work is described as follows. First, as stated in Section 5.1.1, we model the arrival of other concurrent tasks at edge servers using the Poisson distribution. In the future, we will consider a more realistic scenario where the task arrival corresponds to the offloading policies of other coexisting mobile devices. We will then attempt to solve the related offloading problem using multi-agent reinforcement learning. Additionally, we have not yet considered the impact of mobile device mobility on our proposed approach. Our future work will evaluate the effectiveness of our offloading approach by taking into consideration the mobility of mobile devices.

Chapter 6

Conclusions and Future work

In this chapter, we outline the conclusions drawn from our research, followed by the potential research directions for our future work.

6.1 Conclusions

In this thesis, we present three DRL-based offloading approaches aimed at enhancing system performance by generating more accurate offloading strategies for periodic deadline-sensitive tasks in the multi-tier MEC systems. Specifically, we focus on improving system performance from two perspectives, energy consumption minimization and completion time minimization.

Firstly, we propose a novel DRL-based offloading scheme called PDMO, which incorporates DVFS and partially-observable deep reinforcement learning to minimize energy consumption for a set of periodic deadline-sensitive tasks. To effectively schedule both periodic deadline-sensitive tasks and aperiodic non-deadline-sensitive tasks in mobile devices, an efficient task scheduling algorithm based on the cycle conserving DVFS technique is developed. Furthermore, we address the issue of unobservable states in the MEC system by proposing a partially observable DRL algorithm known as PDTD3, which can effectively handle incomplete observations using a series of historical records. Moreover, reward shaping is leveraged to guide the RL agent in generating secure policies. The experimental results indicate that PDMO achieves convergence well across different numbers of tasks and various transmission rates, outperforms baseline methods in terms of energy minimization while maintaining a low number of deadline violations.

Secondly, we propose another DRL-based offloading approach, MELO, which leverages edge-assised DRL learning to minimize task completion time. To be specific, we introduce a decoupled learning architecture called EALA, which relocates the computational modules of the learning algorithm to a dedicated learning server.

This novel learning structure not only significantly facilitates the learning process by taking advantage of superior computing capability of the learning server, but also preserves a vast amount of computation resources on mobile devices and computing edge servers for task execution. As a result, the overall task completion time is shortened. Furthermore, we integrate a double-critics DRL algorithm (TD3) to eliminate overestimation during model training, resulting in more accurate offloading policies.

Finally, we propose a safety-critical DRL-based learning approach, CRLO, which utilizes safe DRL to address offloading problems with a set of constraints. Specifically, by integrating a pretrained safety layer into the original policy network, risky offloading policies can be transformed into safe policies. Moreover, to tackle the challenges of task offloading in a highly dynamic and temporally dependent MEC system, we incorporate a time-series long-sequence forecasting module to predict the state variations of the subsequent hyperperiod proactively. Consequently, the policy network can produce improved offloading policies based on more accurate states.

6.2 Future Work

In this thesis, we primarily focus on DRL-based deadline-sensitive task offloading in multi-tier MEC systems. Along this path, there are serveral potential directions for our future work. The details of these research directions are presented as follows:

(i) As mentioned in Section 5.1.1, our current model characterizes the arrival of computational tasks at edge servers using the Poisson distribution. In the future, we will attempt to improve the fidelity of our simulations by establishing explicit connections between task arrivals at edge servers and the offloading decisions made by varied mobile devices. This implies that each mobile device will be equipped with its own decision-making agent, and the policies generated by these agents will directly influence the decision of other agents within the system. To achieve this goal, we intend to delve into the realm of multi-agent reinforcement learning to tackle the multi-agent offloading problem. Instead of depending on conventional methods such as game theory, we will introduce a novel approach termed Centralized Multi-Agent Transformer Offloading (CMATO).

This approach leverages the capabilities of Multi-Agent Transformer (MAT) to derive holistic offloading decisions for all participating agents. With MAT, the multi-agent offloading task can be converted into a sequential decision-making problem, which is thereafter solved by multi-agent reinforcement learning. This approach is expected to enhance both the efficiency and efficacy of decision-making processes in multi-agent scenarios, enabling all participating agents to dynamically adapt to changing environments and optimize offloading policies in real-time.

(ii) The existing offloading approaches have overlooked the mobility feature of modern mobile devices, a pivotal aspect influencing task offloading decisions. Any movement of mobile devices alters the environment that they are involved in, including factors such as network connectivity and the availability of reachable edge servers. Thus, incorporating mobility into offloading strategies is crucial for accurately capturing the dynamics of MEC environments. It becomes even more pronounced in MEC systems, where the intricate coordination of mobile devices in three-dimensional space is considered. Our future work will focus on evaluating the efficacy of our proposed offloading approaches in the scenarios where mobility is considered, taking into account the dynamic nature of their spatial relationships and movement patterns.

(iii) Existing DRL-based offloading approaches are proposed for specific MEC configurations distinguished by fixed parameters, such as the computational capabilities of edge servers and task arrival patterns. Although some parameters, such as task sizes and channel conditions, may vary over time, others remain constant. Consequently, when these DRL-based methods are deployed in varied MEC environments, they may experience performance decline due to their inability to adapt to unforeseen or novel scenarios. In light of this limitation, we are motivated to explore novel transfer DRL-based approches that enable rapid adaptation to unseen MEC environments. Our objective is to develop a fast-adaptive DRL-based framework for task offloading, characterized by its

ability to seamlessly adjust to new environments with minimal fine-tuning requirements.

Bibliography

[1] L. Ale, N. Zhang, X. J. Fang, X. F. Chen, S. H. Wu, and L. Z. Li. Delay-aware and energy-efficient computation offloading in mobile-edge computing using deep reinforcement learning. IEEE Transactions on Cognitive Communications and Networking, 7(3):881–892, 2021.

[2] S. Amani, C. Thrampoulidis, and L. Yang. Safe reinforcement learning with linear function approximation. In International Conference on Machine Learning, pages 243–253, 2021.

[3] M. Ansari, K. A. Yeganeh, S. Safari, and A. Ejlali. Peak-power-aware energy management for periodic real-time applications. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 39(4):779–788, 2019.

[4] P. A. Apostolopoulos, G. Fragkos, E. E. Tsiropoulou, and S. Papavassiliou. Data offloading in uav-assisted multi-access edge computing systems under resource uncertainty. IEEE Transactions on Mobile Computing, 22(1):175–190, 2023.

[5] H. Aydin, R. Melhem, D.Mosse, and P. Mejia-Alvarez. Power-aware scheduling for periodic real-time tasks. IEEE Transactions on Computers, 53(5):584–600, 2004.

[6] H. Aydin, R. Melhem, D. Mossé, and P. Mejía-Alvarez. Power-aware scheduling for periodic real-time tasks. IEEE Transactions on computers, 53(5):584–600, 2004.

[7] M. Bambagini, M. Marinoni, H. Aydin, and G. Giorgio. Energy-aware scheduling for real-time systems: a survey. ACM Transactions on Embedded Computing Systems (TECS), 15(7):7:1–7:34, 2016.

[8] C. Berner, G. Brockman, B. Chan, V. Cheung, P. Dkebiak, C. Dennison, D. Farhi, Q. Fischer, S. Hashme, and C. Heess. Dota 2 with large scale deep reinforcement learning. arXiv preprint arXiv:1912.06680, 2019.

[9] Z. Cai and T. Shi. Distributed query processing in the edge-assisted IoT data monitoring system. IEEE Internet of Things Journal, 8(16):12679–12693, 2021.

[10] Z. Cai and X Zheng. A private and efficient mechanism for data uploading in smart cyber-physical systems. IEEE Transactions on Network Science and Engineering, 7(2):766–775, 2020.

[11] H. J. Cao and J. Cai. Distributed multiuser computation offloading for cloudlet-based mobile cloud computing: a game-theoretic machine learning approach. IEEE Transactions on Vehicular Technology, 67(1):752–764, 2017.

[12] T. F. Cao, C. Q. Xu, J. P. Du, Y. W. Li, H. Xiao, C. H. Gong, L. J. Zhong, and D. Niyato. Reliable and efficient multimedia service optimization for edge computing-based 5G networks: game theoretic approaches. IEEE Transactions on Network and Service Management, 17(3):1610–1625, 2020.

[13] J. Y. Chen, S. B. Li, and M. Tomizuka. Interpretable end-to-end urban autonomous driving with latent deep reinforcement learning. IEEE Transactions on Intelligent Transportation Systems, 23(6):5068–5078, 2021.

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

[15] W. W. Chen, D. Wang, and K. Q. Lin. Multi-user multi-task computation offloading in green mobile edge cloud computing. IEEE Transactions on Services Computing, 12(5):726–738, April 2018.

[16] X. Chen, J. S. Zhang, B. Lin, Z. Y. Chen, K. Wolter, and G. Y. Min. Energy-efficient offloading for DNN-based smart IoT systems in cloud-edge environments. IEEE Transactions on Parallel and Distributed Systems, 33(3):683–697, 2021.

[17] X. Chen, J. S. Zhang, B. Lin, Z. Y. Chen, K. Wolter, and G. Y. Min. Energy-efficient offloading for DNN-based smart IoT systems in cloud-edge environments. IEEE Transactions on Parallel and Distributed Systems, 33(3):683–697, 2021.

[18] P. Christodoulou. Soft actor-critic for discrete action settings. arXiv preprint arXiv:1910.07207, 2019.

[19] R. G. Cirstea, B. Yang, C. J. Guo, T. Kieu, and S. R. Pan. Towards spatio-temporal aware traffic time series forecasting. In Proceedings of IEEE International Conference on Data Engineering (ICDE), pages 2900–2913, 2022.

[20] M. Dai, Z. Su, Q. Xu, and N. Zhang. Vehicle assisted computing offloading for unmanned aerial vehicles in smart city. IEEE Transactions on Intelligent Transportation Systems, 22(3):1932–1944, 2021.

[21] P. L. Dai, K. W. Hu, X. Wu, H. L. Xing, and Z. F. Yu. Asynchronous deep reinforcement learning for data-driven task offloading in MEC-empowered vehicular networks. In Proceedings of IEEE Conference on Computer Communications (INFOCOM), pages 1–10. IEEE, 2021.

[22] Y. Y. Dai, K. Zhang, S. Maharjan, Y. Zhang, S. Pushp, and X. Z. Liu. Edge intelligence for energy-efficient computation offloading and resource allocation in 5G beyond. IEEE Transactions on Vehicular Technology, 69(10):12175–12186, 2020.

[23] G. Dalal, K. Dvijotham, M. Vecerik, T. Hester, C. Paduraru, and Y. Tassa. Safe exploration in continuous action spaces. arXiv preprint arXiv:1801.08757, 2018.

[24] T. Dinh, Q. D. La, T. Q. Quek, and H. Shin. Learning for computation offloading in mobile edge computing. IEEE Transactions on Communications, 66(12):6353–6367, 2018.

[25] J. B. Du, L. Q. Zhao, J. Feng Q. Li, and X. L. Chu. Computation offloading and resource allocation in mixed fog/cloud computing systems with min-max fairness guarantee. IEEE Transactions on Communications, 66(4):1594–1608, December 2018.

[26] M. Z. Du, Y. Wang, K. J. Ye, and C. Z. Xu. Algorithmics of cost-driven computation offloading in the edge-cloud environment. IEEE Transactions on Computers, 69(10):1519–1532, 2020.

[27] K. Elgazzar, P. Martin, and H. S. Hassanein. Cloud-assisted computation offloading to support mobile services. IEEE Transactions on Cloud Computing, 4(3):279–292, 2014.

[28] L. Espeholt, R. Marinier, P. Stanczyk, K. Wang, and M. Michalski. Seed rl: Scalable and efficient deep-rl with accelerated central inference. Proceedings of the IEEE International Conference on Learning Representations (ICLR), 2020.

[29] S. Fujimoto, H. Van Hoof, and D. Merger. Addressing function approximation error in actor-critic methods. arXiv:1509.02971v6, 2018.

[30] S. Fujimoto, D. Meger, and D. Precup. Off-policy deep reinforcement learning without exploration. In International Conference on Machine Learning (ICML), pages 2052–2062, 2019.

[31] H. H. Gao, X. J. Wang, W. Wei, A. Al-Dulaimi, and Y. S. Xu. Com-ddpg: task offloading based on multiagent reinforcement learning for information-communication-enhanced mobile edge computing in the internet of vehicles. IEEE Transactions on Vehicular Technology, 73(1):348–361, 2024.

[32] Z. Gao, L. Yang, and Y. Dai. Large-scale computation offloading using a multi-agent reinforcement learning in heterogeneous multi-access edge computing. IEEE Transactions on Mobile Computing, January 2022. DOI: 10.1109/TMC.2022.3141080.

[33] Y. L. Geng, Y. Yang, and G. H. Cao. Energy-efficient computation offloading for multicore-based mobile devices. In Proceedings of IEEE Computer Communications Conference(INFOCOM), pages 46–54, April 2018.

[34] B. Gu, Z. Y. Zhou, S. Mumtaz, V. Frascolla, and A. K. Bashir. Context-aware task offloading for multi-access edge computing: matching with externalities. In Proceedings of IEEE global communications conference (GLOBECOM), pages 1–6, 2018.

[35] S. T. Guo, J. D. Liu, Y. Y. Yang, B. Xiao, and Z. T. Li. Energy-efficient dynamic computation offloading and cooperative task scheduling in mobile cloud computing. IEEE Transactions on Mobile Computing, 18(2):319–333, 2019.

[36] M. Hamad, Z. A. Hammadeh, S. Saidi, V. Prevelakis, and R. Ernst. Prediction of abnormal temporal behavior in real-time systems. In Proceedings of the 33rd Annual ACM Symposium on Applied Computing, pages 359–367, 2018.

[37] Y. X. Hao, Y. Q. Jiang, M. S. Hossain, M. F. Alhamid, and S. U. Amin. Learning for smart edge: cognitive learning-based computation offloading. Mobile Networks and Applications, 25(3):1016–1022, 2020.

[38] H. Van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning. In Proceedings of the AAAI conference on Artificial Intelligence, pages 2094–2100, 2016.

[39] H. Huang, M. Lin, and Q. C. Zhang. Double-q learning-based dvfs for multicore real-time systems. In Proceedings of International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), pages 522–529, 2017.

[40] H. Huang, Q. Ye, and Y. T. Zhou. Deadline-aware task offloading with partially-observable deep reinforcement learning for multi-access edge computing. IEEE Transactions on Network Science and Engineering, 9(6):3870–3885, 2021.

[41] H. Huang, Q. Ye, and Y. T. Zhou. 6g-empowered offloading for realtime applications in multi-access edge computing. IEEE Transactions on Network Science and Engineering, 10(3):1311–1325, 2022.

[42] Y. Hui, Z. Su, and T. H. Luan. Unmanned era: a service response framework in smart city. IEEE Transactions on Intelligent Transportation Systems, 23(6):5791–5805, 2021.

[43] F. Islam and M. Lin. Hybrid DVFS scheduling for real-time systems based on reinforcement learning. IEEE Systems Journal, 11(2):931–940, 2017.

[44] B. F. Ji, Y. N. Wang, K. Song, C. G. Li, H. Wen, V. G. Menon, and S. Mumtaz. A survey of computational intelligence for 6G: key technologies, applications and trends. IEEE Transactions on Industrial Informatics, 17(10):7145–7154, 2021.

[45] T. X. Ji, C. Q. Luo, L. X. Yu, Q. L. Wang, S. H. Chen, A. Thapa, and P. Li. Energy-efficient computation offloading in mobile edge computing systems with uncertainties. IEEE Transactions on Wireless Communications, 21(8):5717–5729, 2022.

[46] H. B. Jiang, X. x. Dai, Z. Xiao, and A. K. Iyengar. Joint task offloading and resource allocation for energy-constrained mobile edge computing. IEEE Transactions on Mobile Computing, January 2022. DOI: 10.1109/TMC.2022.3150432.

[47] Y. X. Jiang and D. H. Tsang Tsang. Delay-aware task offloading in shared fog networks. IEEE Internet of Things Journal, 5(6):4945–4956, 2018.

[48] S. Jošilo and G. Dán. Computation offloading scheduling for periodic tasks in mobile edge computing. IEEE/ACM Transactions on Networking, 28(2):667–680, 2020.

[49] H. Kalantarian, C. Sideris, B. Mortazavi, N. Alshurafa, and M. Sarrafzadeh. Dynamic computation offloading for low-power wearable health monitoring systems. IEEE Transactions on Biomedical Engineering, 64(3):621–628, 2016.

[50] S. Khanderwal and D. Mohanty. Stock price prediction using arima model. International Journal of Marketing & Human Resource Research, 2(2):98–107, 2021.

[51] X. J. Kong, G. H. Duan, M. L. Hou, G. J. Shen, H. Wang, X. R. Yan, and M. Collotta. Deep reinforcement learning based energy efficient edge computing for Internet of Vehicles. IEEE Transactions on Industrial Informatics, 18(9):6308–6316, 2022.

[52] G. K. Lai, W. C. Chang, Y. M. Yang, and H. X. Liu. Modeling long-and short-term temporal patterns with deep neural networks. In Proceedings of International ACM SIGIR Conference on Research & Development in Information Retrieval, pages 95–104, 2018.

[53] J. Li, H. Gao, T. J. Lv, and Y. M. Lu. Deep reinforcement learning based computation offloading and resource allocation for MEC. In Proceedings of International Conference on IEEE Wireless Communications and Networking Conference, pages 1–6, June 2018.

[54] M. S. Li, J. G, L. Zhao, and X. M. Shen. Deep reinforcement learning for collaborative edge computing in vehicular networks. IEEE Transactions on Cognitive Communications and Networking, 6(4):1122–1135, 2020.

[55] S. Y. Li, X. Y. Jin, Y. Xuan, X. Y. Zhou, W. H. Chen, Y. X. Wang, and X. F. Yan. Enhancing the locality and breaking the memory bottleneck of transformer on time series forecasting. In Proceedings of IEEE International Conference on Neural information processing systems(NeurIPS), volume 32, 2019.

[56] Z. Z. Liang, Y. Liu, T. M. Lok, and K. B. Huang. Multiuser computation offloading and downloading for edge computing with virtualization. IEEE Transactions on Wireless Communications, 18(9):235–250, 2019.

[57] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, Y. Tassa T. Erez, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. In Proceedings of International Conference Learning Representations(ICLR), 2016.

[58] M. H. Liu, A. L. Zeng, M. X. Chen, Z. J. Xu, Q. X. Lai, L. N. Ma, and Q. Xu. Scinet: Time series modeling and forecasting with sample convolution and interaction. In Proceedings of the Advances in Neural Information Processing Systems, pages 5816–5828, 2022.

[59] S. H. Liu, B. Wang, X. J. Deng, and L. T. Yang. Self-attentive graph convolution network with latent group mining and collaborative filtering for personalized recommendation. IEEE Transactions on Network Science and Engineering, 9(5):3212–3221, 2021.

[60] S. Q. Liu, K. C. See, K. Y. Ngiam, L. A. Celi, X. Z. Sun, and M. L. Feng. Reinforcement learning for clinical decision support in critical care: comprehensive review. Journal of Medical Internet Research, 22(7):e18477, 2020.

[61] Z. K. Liu, P. L. Dai, H. L. Xing, Z. F. Y, and W. Zhang. A distributed algorithm for task offloading in vehicular networks with hybrid fog/cloud computing. IEEE Transactions on Systems, Man, and Cybernetics: Systems, 52(7):4388–4401, 2021.

[62] Z. X. Liu, Z. P. Cen, V. Isenbaev, W. Liu, S. Wu, B. Li, and D. Zhao. Constrained variational policy optimization for safe reinforcement learning. In Proceedings of IEEE Conference on Machine Learning (ICML), pages 13644–13668, 2022.

[63] H. D. Lu, X. M. He, M. Du, X. K. Ruan, Y. F. Sun, and K. Wang. Edge QoE: computation offloading with deep reinforcement learning for Internet of Things. IEEE Internet of Things Journal, 7(10):9255–9265, 2020.

[64] M. H. Min, L. Xiao, Y. Chen, P. Cheng, D. Wu, and W. H. Zhuang. Learning-based computation offloading for IoT devices with energy harvesting. IEEE Transactions on Vehicular Technology, 68(2):1930–1941, 2019.

[65] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In Proceedings of International conference on machine learning (ICML), pages 1928–1937, 2016.

[66] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602, 2013.

[67] A. Naouri, H. X. Wu, N. A. Nouri, S. Dhelim, and H. S. Ning. A novel framework for mobile-edge computing by optimizing task offloading. IEEE Internet of Things Journal, 8(16):13065–13076, 2021.

[68] A. Ndikumana, N. H. Tran, T. M. Tai, Z. Han, W. Saad, D. Niyato, and S. C. Hong. Joint communication, computation, caching, and control in big data multi-access edge computing. IEEE Transactions on Mobile Computing, 19(6):1359–1374, 2020.

[69] P. Pallai and K. G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In Proceedings of International Conference on ACM SIGOPS Operating Systems Review, pages 89–102, 2001.

[70] S. K. Panda, M. Lin, and T. Zhou. Energy efficient computation offloading with dvfs using deep reinforcement learning for time-critical iot applications in edge computing. IEEE Internet of Things Journal, 10(8):6611–6621, 2022.

[71] C. Pradhan, A. Li, C. Y. She, Y. H. Li, and B. Vucetic. Computation offloading for IoT in C-RAN: optimization and deep learning. IEEE Transactions on Communications, 68(7):4565–4579, 2020.

[72] L. P. Qian, Y. Wu, F. L. Jiang, N. N. Yu, W. D. Lu, and B. Lin. NOMA assisted multi-task multi-access mobile edge computing via deep reinforcement learning for industrial Internet of Things. IEEE Transactions on Industrial Informatics, 17(8):5688–5698, 2020.

[73] X. D. Qin, B. Li, and L. Ying. Distributed threshold-based offloading for large-scale mobile cloud computing. In Proceedings of IEEE Conference on Computer Communications, pages 1–10, 2021.

[74] X. Y. Qiu, L. B. Liu, W. H. Chen, Z. C. Hong, and Z. B. Zheng. Online deep reinforcement learning for computation offloading in blockchain-empowered mobile edge computing. IEEE Transactions on Vehicular Technology, 68(8):8050–8062, 2019.

[75] T. Ren, Z. Y. Hu, H. He, J. W. Niu, and X. F. Liu. Feat: towards fast environment-adaptive task offloading and power allocation in mec. In Proceedings of the IEEE Conference on Computer Communications, pages 1–10, 2023.

[76] T. K. Rodrigues, J. J. Liu, and N. Kato. Offloading decision for mobile multi-access edge computing in a multi-tiered 6G network. IEEE Transactions on Emerging Topics in Computing, 10(3):1414–1427, 2021.

[77] T. K. Rodrigues, K. Suto, and N. Kato. Edge cloud server deployment with transmission power control through machine learning for 6G Internet of Things. IEEE Transactions on Emerging Topics in Computing, 9(4):2099–2108, 2019.

[78] s. f. Wu, X. Xiao, Q. G. Ding, P. L. Zhao, Y. Wei, and J. Z. Huang. Adversarial sparse transformer for time series forecasting. In Proceedings of the Advances in Neural Information Processing Systems, pages 17105–17115, 2020.

[79] E. Seo, J. Jeong, S. Park, and J. Lee. Energy efficient scheduling of real-time tasks on multicore processors. IEEE Transactions on Parallel and Distributed Systems, 19(11):1540–1552, 2008.

[80] S. Z. Sheikh and M. A. Pasha. Energy-efficient real-time scheduling on multicores: a novel approach to model cache contention. ACM Transactions on Embedded Computing Systems (TECS), 19(4):1–25, 2020.

[81] S. Sundar and B. Liang. Offloading dependent tasks with communication delay and deadline constraint. In Proceedings of IEEE Conference on Computer Communications (INFOCOM), pages 37–45, 2018.

[82] J. Tan, R. Khalili, H. Karl, and A. Hecker. Multi-agent distributed reinforcement learning for making decentralized offloading decisions. In Proceedings of the IEEE International Conference on Computer Communications (Infocom), pages 2098–2107, 2022.

[83] H. R. Tang, R. Houthooft, D. Foote, A. Stooke, X. Chen, Y. Duan, J. Schulman, F. De Turck, and P. Abbeel. A study of count-based exploration for deep reinforcement learning. In Proceedings of IEEE International Conference on Neural Information Processing Systems(NeurIPS), pages 4–9, 2017.

[84] M. Tang and V. W. S. Vincent. Deep reinforcement learning for task offloading in mobile edge computing systems. IEEE Transactions on Mobile Computing, 21(6):1985–1997, 2022.

[85] U. U. Tariq, H. Ali, L. Liu, J. Hardy, M. Kazim, and W. Ahmed. Energy-aware scheduling of streaming applications on edge-devices in iot-based healthcare. IEEE Transactions on Green Communications and Networking, 5(2):803–815, 2021.

[86] E. Walraven and M. T. Spaan. Accelerated vector pruning for optimal POMDP solvers. In AAAI, pages 3672–3678, 2017.

[87] F. Wang, J. Xu, and S. G. Cui. Energy-efficient dynamic computation offloading and cooperative task scheduling in mobile cloud computing. IEEE Transactions on Wireless Communications, 19(4):2443–2459, 2020.

[88] F. Wang, J. Xu, and S. G. Cui. Optimal energy allocation and task offloading policy for wireless powered mobile edge computing systems. IEEE Transactions on Wireless Communications, 19(4):2443–2459, 2020.

[89] J. Wang, J. Hu, G. Y. Min, A. Y. Zomaya, and N. Georgalas. Fast adaptive task offloading in edge computing based on meta reinforcement learning. IEEE Transactions on Parallel and Distributed Systems, 32(1):242–253, 2021.

[90] T. Wang, Y. X. Deng, Z. Yang, Y. Wang, and H. B. Cai. Parameterized deep reinforcement learning with hybrid action space for edge task offloading. IEEE Internet of Things Journal, March 2023. DOI: 10.1109/JIOT.2023.3327121.

[91] X. F. Wang, Y. W. Han, V. C. Leung, D. Niyato, X. Q. Yan, and X. Chen. Convergence of edge computing and deep learning: a comprehensive survey. IEEE Communications Surveys & Tutorials, 22(2):869–904, 2020.

[92] H. X. Wu, J. H. Xu, J. M. Wang, and M. S. Long. Autoformer: Decomposition transformers with auto-correlation for long-term series forecasting. In Proceedings of the Advances in Neural Information Processing Systems, volume 34, pages 22419–22430, 2021.

[93] Y. Z. Xia, X. J. Deng, L. Z. Yi, L. T. Yang, X. Xiao, C. L. Zhu, and Z. P. Tian. AI-driven and MEC-empowered confident information coverage hole recovery in 6G-enabled IoT. IEEE Transactions on Network Science and Engineering, 10(3):1256–1269, 2022.

[94] M. W. Xu, Q. Feng, M. Z. Zhu, F. F. Huang, S. Pushp, and X. Z. Liu. Deepwear: adaptive local offloading for on-wearable deep learning. IEEE Transactions on Mobile Computing, 19(2):314–330, 2019.

[95] B. Yang, X. L. Cao, J. Bassey, X. F. Li, and L. J. Qian. Computation offloading in multi-access edge computing: A multi-task learning approach. IEEE Transactions on Mobile Computing, 20(9):2745–2762, 2021.

[96] Y. You, Z. Zhang, C. J. Hsieh, J. Demmel, and K. Keutzer. Fast deep neural network training on distributed systems and cloud TPUs. IEEE Transactions on Parallel and Distributed Systems, 30(11):2449–2462, 2019.

[97] F. Yu, V. Koltun, and T. Funkhouser. Dilated residual networks. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 472–480, 2017.

[98] A. Zappone, M. Di Renzo, and M. Debbah. Wireless networks design in the era of deep learning: model-based, AI-based, or both? IEEE Transactions on Communications, 67(10):7331–7376, 2019.

[99] A. L. Zeng, M. X. Chen, L. Zhang, and Q. xu. Are transformers effective for time series forecasting? In Proceedings of the AAAI conference on artificial intelligence (AAAI), pages 11121–11128, 2023.

[100] Y. F. Zhan, S. Guo, P. Li, and J. Zhang. A deep reinforcement learning based offloading game in edge computing. IEEE Transactions on Computers, 69(6):883–893, 2020.

[101] D. Y. Zhang, L. Tan, J. Ren, M. K. Awad, S. Zhang, Y. X. Zhang, and P. J. Wan. Near-optimal and truthful online auction for computation offloading in green edge-computing systems. IEEE Transactions on Mobile Computing, 19(4):880–893, 2020.

[102] D. Y. Zhang, L. Tan, J. Ren, M. K. Awad, S. Zhang, Y. X. Zhang, and P. J. Wan. A survey on mobile augmented reality with 5G mobile edge computing: architectures, applications, and technical aspects. IEEE Communications Surveys & Tutorials, 23(2):1160–1192, 2021.

[103] J. Zhang, G. J. Han, J. F. Sha, Y. J. Qian, and J. Liu. AUV-assisted subsea exploration method in 6G enabled deep ocean based on a cooperative pac-men mechanism. IEEE Transactions on Intelligent Transportation Systems, 23(2):1649–1660, 2021.

[104] J. Zhang, X. P. Hu, Z. L. Ning, E. C-H. Ngai, L. Zhou, J. B. Wei, J. Cheng, B. Hu, and V. CM. Leung. Joint resource allocation for latency-sensitive services over mobile edge computing networks with caching. IEEE Internet of Things Journal, 6(3):4283–4294, 2019.

[105] Q. C. Zhang, M. Lin, L. T. Yang, Z. K. Chen, S. U. Khan, and P. Li. A double deep Q-learning model for energy-efficient edge scheduling. IEEE Transactions on Services Computing, 12(5):739–749, 2019.

[106] Q. C. Zhang, M. Lin, L. T. Yang, Z. K. Chen, and P. Li. Energy-efficient scheduling for real-time systems based on deep Q-learning model. IEEE Transactions on Sustainable Computing, 4(1):132–141, 2017.

[107] W. Y. Zhang, Z. Z. He, L. Y. Liu, Z. H. Jia, Y. X. Liu, M. Gruteser, D. Raychaudhuri, and Y. Y. Zhang. Elf: accelerate high-resolution mobile deep vision with content-aware parallel offloading. In Proceedings of the 27th Annual International Conference on Mobile Computing and Networking, pages 201–214, 2021.

[108] G. M. Zhao, H. L. Xu, Y. M. Zhao, C. M. Qiao, and L. S. Huang. Offloading dependent tasks in mobile edge computing with service caching. In Proceedings of IEEE Computer Communications Conference(INFOCOM), pages 1997–2006, July 2020.

[109] J. K. Zheng, T. H. Luan, L. X. Gao, Y. Zhang, and Y. Wu. Learning based task offloading in digital twin empowered internet of vehicles. arXiv preprint arXiv:2201.09076, 2021.

[110] C. J. Zhong, H. A. Suraweera, G. Zheng, L. Krikidis, and Z. Y. Zhang. Wireless information and power transfer with full duplex relaying. IEEE Transactions on Communications, 62(10):3447–3461, 2014.

[111] H. Y. Zhou, S. H. Zhang, J. Q. Peng, S. Zhang, J. X. Li, H. Xiong, and W. C. Zhang. Informer: Beyond efficient transformer for long sequence time-series forecasting. In Proceedings of the AAAI Conference on Artificial Intelligence, pages 11106–11115, 2021.

[112] T. Zhu, T. Shi, J. Li, Z. Cai, and X. Zhou. Task scheduling in deadline-aware mobile edge computing systems. IEEE Internet of Things Journal, 6(3):4854–4866, 2019.