Task Offloading and Resource Allocation using Deep Reinforcement Learning

by

Kaiyi Zhang

Thesis submitted to the University of Ottawa
In partial fulfillment of the requirements
For the M.Sc. degree in
Systems Science

School of Electrical Engineering and Computer Science
Faculty of Engineering
University of Ottawa

Abstract

Rapid urbanization poses huge challenges to people's daily lives, such as traffic congestion, environmental pollution, and public safety. Mobile Internet of things (MIoT) applications serving smart cities bring the promise of innovative and enhanced public services such as air pollution monitoring, enhanced road safety and city resources metering and management. These applications rely on a number of energy constrained MIoT units (MUs) (e.g., robots and drones) to continuously sense, capture and process data and images from their environments to produce immediate adaptive actions (e.g., triggering alarms, controlling machinery and communicating with citizens). In this thesis, we consider a scenario where a battery constrained MU executes a number of time-sensitive data processing tasks whose arrival times and sizes are stochastic in nature. These tasks can be executed locally on the device, offloaded to one of the nearby edge servers or to a cloud data center within a mobile edge computing (MEC) infrastructure. We first formulate the problem of making optimal offloading decisions that minimize the cost of current and future tasks as a constrained Markov decision process (CMDP) that accounts for the constraints of the MU battery and the limited reserved resources on the MEC infrastructure by the application providers. Then, we relax the CMDP problem into regular Markov decision process (MDP) using Lagrangian primal-dual optimization. We then develop advantage actor-critic (A2C) algorithm, one of the model-free deep reinforcement learning (DRL) method to train the MU to solve the relaxed problem. The training of the MU can be carried-out once to learn optimal offloading policies that are repeatedly employed as long as there are no large changes in the MU environment. Simulation results are presented to show that the proposed algorithm can achieve performance improvement over offloading decisions schemes that aim at optimizing instantaneous costs.

Acknowledgements

First of all, I would like to express my very great appreciation to my parents for their endless support and love in my life.

Next, I would like to express my deep gratitude to my research supervisor, Professor Nancy Samaan, for her patient guidance, enthusiastic encouragement and helpful suggestion throughout my graduate studies.

Finally, I would also like to express my gratefulness to my friends for their friendship and encouragement. Thankful for all that encounter in life, whether happy or sad, it's all a treasure.

Table of Contents

List of Tables

List of Figures

Nomenclature

Abbreviations

A2C Advantage actor-critic

ADP Approximate Dynamic Programming

AI Artificial Intelligence

AP Access Point

BS Base Station

CMDP Constrained Markov Decision Process

CPU Central Processing Unit

DDPG Deep Deterministic Policy Gradient

DL Deep Learning

DNN Deep Neural Network

DQN Deep Q-Network

DRL Deep Reinforcement Learning

DVFS Dynamic Frequency and Voltage Scaling

ICN Information-Centric Network

IoT Internet of Thing

LSTM Long Short-term Memory

MC Monte Carlo

MCC Mobile Cloud Computing

MDP Markov Decision Process

MEC Mobile Edge Computing

MLP Multi-layer Perceptron

NFV Network Functions Virtualization

NN Neural Network

PI Policy Iteration

QoE Quality of Experience

QoS Quality of Service

RAN Radio Access Network

RL Reinforcement Learning

RNN Recurrent Neural Network

SDN Software-Defined Network

TD Temporal Difference

VI Value Iteration

VM Virtual Machine

WAN Wide Area Network

Chapter 1

Introduction

In the last few years, we have been witnessing the explosive growth of mobile user equipment. Based on the latest Ericsson mobility report, the number of mobile subscriptions totaled 7.9 billion globally up to the fourth quarter of 2019 [3]. The trend boosts the proliferation of some newly-developed applications such as face recognition, automatic speech recognition (ASR), virtual reality (VR) and augmented reality (AR). The services that the traditional mobile network architecture satisfies are mainly social networking and web browsing, and there are deficiencies such as small wireless bandwidth and large network delay. For example, In order to let users experience the feeling of being immersive in VR services, it is necessary to increase the existing 4k/90 frame rendering processing capacity by 50 to 100,000 times, and the VR interactive delay must be within 20ms [4]. In addition to the intensive computing process of video rendering, AR also needs to track user positioning and establish an environment model [5]. The above applications all need to complete a large amount of calculation processing in a short time. At the same time, due to the significant increase in energy consumption, the extremely high requirements are placed on mobile devices' computing power and energy efficiency.

On the other hand, the prosperous Internet of Things (IoT) technology has brought a emerging concept smart city. With IoT-based smart cities, a large number of different IoT devices run miscellaneous advanced services in all areas of city life. Recent advancements in manufacturing smart devices (e.g., smart wearable devices, camera mounted drones and

autonomous environment sensing robots) have paved the way for developing a large number of diverse mobile Internet of things (MIoT) applications in several domains including but not limited to tourism, health care, public services and road safety [6]. Examples of these applications are smart tourism, where moving robots in attraction sites can identify and interact with tourists and their devices, and air pollution monitoring, where drones can collect various air samples to test the air quality in a given area. Another example is smart transportation, which can utilize traffic networks more efficiently and safety, where sensors embedded to the vehicles, or cameras installed in the crossroad can help to offer optimized route suggestions, economic street lighting, accident prevention, and autonomous driving [7]. Other applications also include smart parking [8] and automated gas and electricity metering and maintenance. These applications rely on accurate sensing of the environment and on the timely execution of a continuous stream of tasks on the collected data (e.g., object recognition, automatic speech recognition, and meter data analysis). Although advanced IoT devices are becoming more powerful in terms of the number and speed of their central processing units (cpus) and their memory and battery capacities, they may still be incapable of concurrently processing a large number of computationally-intensive tasks over a short period of time [9].

In response to the above contradiction, a feasible strategy is to offload these computationally intensive tasks to a remote cloud, because the cloud can provide intensive computing resources, which is the basic idea of Mobile Cloud Computing (MCC). However, MCC suffers from long communication delay and limited backhaul bandwidth due to its centralized architecture. Concerning such challenges of MCC, Mobile Edge Computing (MEC) shifts the services to the edge of the mobile network and use the servers deployed at the edge of the network. With the aid of MEC, IoT devices are enabled to offload computation tasks to MEC servers at the edge of pervasive radio access networks in close proximity to end-devices, rather than using the servers in the core network [10]. Due to the proximity of the edge servers to the MIoT devices, they can better satisfy the applications needs in terms of fast and interactive responses due to their low-service latency when compared to distant cloud data centers.

1.1 Motivation

IoT-based smart cities are characterized by numerous services running over a large number of end IoT devices as well as applications hosted in remote servers [11]. The massive amounts of data generated by the multiple services running on numerous IoT devices need to be further processed by applications hosted on remote servers to enable rich intelligent functionality. In this regard, computation offloading can effectively reduce the task execution delay and extend the battery life of the terminal. However, although offloading tasks to the servers can effectively utilize the computing resources on the server-side, the quality of service (QoS) of the smart city services differs under different task offloading and computation resource allocation strategies significantly due to the limited computing resources on the server-side. The computation offloading strategy is not always effective, and a bad computation offloading decision will result in higher processing delay and energy consumption. Given the large number of smart city services as well as their different QoS requirements, the critical challenge for computing offloading in the MEC system is rationally designing the task offloading strategy. It is also a big challenge for servers to optimally allocate limited computation resources to all hosted applications. This concern becomes even more stringent with the expansion of smart city scale.

The limited computing resources of the MEC system is a huge challenge facing increasing computing needs. Over the last few years, the problem of mobile task offloading to MEC infrastructures has received a considerable attention in the Literature [9, 12, 13, 14]. The research literature on this issue can be roughly divided into two categories. The first category is to adopt reasonable allocation of computing resources to make full use of limited resources. The second category is to introduce auxiliary nodes to expand the computing power of the MEC system. The introduction of auxiliary nodes can not only distribute the computational load of the MEC server, but also make use of the resources that are idle. These auxiliary nodes may be D2D (Device to Device) devices [15], remote clouds [14], and MEC servers in adjacent areas. The above two ideas are not completely split. In the MEC system with auxiliary nodes, the computation offloading and resources allocation strategy also needs to be considered. However, the majority of existing offloading solutions work

under several limiting assumptions. The first is that the resources on the edge servers are always available while the second is that the battery of the MIoT device may have sufficient battery charge to serve or offload all incoming tasks. Furthermore, a large number of this solution do not consider the effects of current offloading decisions on the performance on the MIoT device's remaining power or resource availability for tasks arriving in the future. However, resources at the edge servers may still be limited compared to core data centers and they must be shared by a large number of running smart city applications, in addition, they are expected to be more expensive.

Furthermore, an application operator may want to ensure that a certain number of tasks, or tasks arriving during a given time interval (e.g., working hours within a day) must be served before the mobile device battery reaches the minimum charge needed to reach a recharging station. The battery consumption of the MIoT devices determines whether they will work reliably and independently for a long time. The high battery consumption may result in added labor cost. For example, if a sensing robot runs out of power before reaching the next charging station, the staff will have to manually move the robot. Therefore, controlling the battery consumption of MIoT devices plays a vital role on the long-term performance.

We envision that in order to reduce their task offloading costs and to guarantee the performance of their services, MIoT application providers may opt to lease pre-configured virtual machines (VMs) that can host offloaded tasks from a large number of their MIoT devices. Each device's tasks may run in isolation using containers [16]. Furthermore, to ensure the execution of these tasks, additional VMs may also be reserved at the core network. In view of this, the objective of the thesis is to improve the performance of offloading and resource allocation decisions on the basis of the long-term performance of a MEC system.

1.2 Contribution

In this thesis, we focus on the operation of a single MIoT unit (MU) that is executing a number of tasks (e.g., image capture for object recognition, air sampling for pollution control or meters reading for utility services) for a given period of time. The tasks arrive at random intervals and have varying processing requirements both following stochastic distributions. These tasks must be executed either at the MU or offloaded to containers hosted on preassigned virtual machines on a multi-tier MEC. The first tier in the MEC represents a number of edge servers adjacent to the base stations (BS) of the cellular networks and the second tier corresponds to a cloud data center connected to WiFi access points (APs), located in the vicinity of the MU, through the backbone. We design a novel advantage-actor-critic (A2C) learning scheme to generate near-optimal offloading decisions and resources allocation strategies for each arriving task. The developed scheme not only learns how to balance a desired tradeoff between the current cost, in terms of the incurred delay and consumed device energy, and costs of future tasks, but also controls the battery consumption to prolong the operational period before a battery recharge is required. More precisely, the main contributions of this thesis can be summarized as follows [17],

- We describe a novel and more realistic scenario of MIoT roaming a predefined area to sense their environments and execute their tasks. The MU has a limited battery and must recharge if it reaches a minimum threshold. In addition, application provider purchase limited resources and preassigns VMs within the multi-tier MEC infrastructure to the MU.

- We formulate the execution and offloading problem of the tasks generated by the MU using a multi-tier MEC infrastructure as a constrained Markov decision process (CMDP). In this model, the offloading action for each arriving task does not only depend on the available resources at the time of the decision but also takes into consideration concurrently running tasks (i.e., with the past decisions) and the uncertainties in the execution costs of future tasks. Since these previously executed actions have consumed part of the available battery since the beginning of the con-

sidered time period and for some concurrently running tasks some of the resources on the MU and the VMs on the edge servers and the cloud might be occupied.

- To learn an optimal long-term offloading policy for the repeated operation of the MU, we relax the CMDP problem using Lagrangian primal-dual optimization. We then develop a novel advantage actor-critic (A2C) learning scheme where two deep neural networks are designed to evaluate the actions and approximate the value function for each state, respectively. We demonstrate through performance evaluation results the efficiency of the proposed scheme when compared to instantaneous optimization solutions.

1.2.1 Publication

Kaiyi Zhang and Nancy Samaan. Optimized look-ahead offloading decisions using deep-reinforcement learning for battery constrained mobile iot devices. In *2020 IEEE International Conference on Smart Cloud (SmartCloud)*, pages 181–186, 2020.

1.3 Thesis Outline

This reminder of the thesis is organized as follows.

- Chapter 2 gives a background of mobile edge computing and discusses the computation offloading problem with some relevant solutions. Then, A fundamentals of deep learning and deep reinforcement learning is also introduced. Some common models is also presented.

- Chapter 3 presents the system model and formulates the offloading problem as a Constrained Markov Decision Process problem.

- Chapter 4 describes the proposed constrained deep reinforcement learning (CDRL) algorithm to find near optimal offloading decisions that reduce the offloading cost for current and future tasks.

- Chapter 5 discusses the performance evaluation results.
- Chapter 6 gives the conclusion of this thesis and discusses a few possible future research directions.

Chapter 2

Background and Literature Review

This chapter gives a literature review on the development and computation offloading technologies on Mobile Edge Computing. Section 2.1 gives the background and explains some fundamental concepts of Mobile Edge Computing area. Section 2.2 introduces the computation offloading and some factors that may affect the offloading decision. Section 2.3 introduces the fundamentals of deep learning and some classic models. Section 2.4 introduces the fundamentals of deep reinforcement learning and some standard algorithms. Section 2.5 gives some categories about the existing approaches that solve the computation offloading problem.

2.1 Background

Over the last decade, with the rapid development of Internet of things (IoT), a tremendous number of ubiquitous smart devices and objects (e.g., sensors, actuators, mobile phones and laptop) can be built into every fabric of urban environment and connected with each other [18]. The IoT-based smart city has received considerable attention and is becoming a promising technology that integrates ubiquitous sensing, universal networking, intelligent information processing and real-time control. The main goal of the IoT-based smart city is to efficiently use public resources, thus providing a wide range of intelligent services, including smart building, smart manufacturing, smart energy, and smart healthcare. In

the background of the IoT-based smart city, ubiquitous IoT devices are applied to monitor the surroundings in people's daily lives in real time by collecting and processing the local sensed contents such as images, videos, and textual data [19]. In essence, the development of smart cities inevitably will lead to a significant increase in the multimedia applications and services [20]. However, the limited computation resources and battery capacity of IoT devices have become huge constraints for guaranteeing the efficient computing in smart cities. Therefore, the idea that transferring the computing tasks from IoT devices to computationally powerful remote servers is proposed, which is called by 'cloud computing'.

2.1.1 From Centralized Clouds to Mobile Edges

To provide IoT devices with abundant computation resources, mobile cloud computing (MCC) is proposed as a promising approach, which combines cloud computing, mobile computing, and wireless communication networks [21]. By offloading the computation-intensive task to remote resource-rich commercially cloud infrastructures such as Amazon EC2, Alibaba Cloud and Google Cloud, the computation load on the device is transferred to the cloud. Thus the energy consumption of IoT devices is reduced significantly, and delay is decreased to some degree. Over the years, the cloud elasticity model has been widely successful and an added value for both enterprises and cloud providers [12]. However, some delay-sensitive applications are challenging the scalability and resiliency models of the traditional mobile cloud computing with more rigorous demands in response latency and data storage. One of the critical challenges is due to the long distance from IoT devices to remote cloud platforms, and it may result in a huge transmission delay for applications. Because users are very sensitive to delay, it would have a negative effect on user experience. Additionally, if too many IoT devices choose to offload the tasks to the cloud simultaneously, it is highly like to generate severe interference and thus decrease the overall network performance [22]. Thus, it is hardly to satisfy the acceptable delays of these applications.

To overcome this constraint, Satyanarayanan et al.[23] proposed the cloudlet based MCC as a promising solution, which is illustrated in Figure 2.1. Instead of depending on

remote cloud servers, the cloudlet decreases delay by offloading the computation task to the nearby computing server/cluster via a one-hop WiFi wireless access. The approach allows mobile users to obtain the benefits of the cloud while avoiding the latency inherent to access the cloud in a remote location. Nevertheless, the local cloud must be connected to a more powerful distant cloud that provides additional resources. Cloudlets are envisioned as clusters of multi-core computers with high Gigabit internal connectivity that are self-managed in order to require little to no administration. However, there are three shortcomings that cannot be ignored for the cloudlet: 1) due to limited coverage of WiFi networks (typically available for indoor environments), cloudlet can not guarantee ubiquitous service provision everywhere; 2)because cloudlets are supposed to be mostly accessed by IoT devices through WiFi connection, IoT devices have to switch between the mobile network and WiFi whenever the cloudlet services are exploited [9]; 3) due to space constraints, cloudlet usually utilizes a computation sever/cluster with small/medium computation resources, which may not satisfy QoS of a large number of mobile users [24].

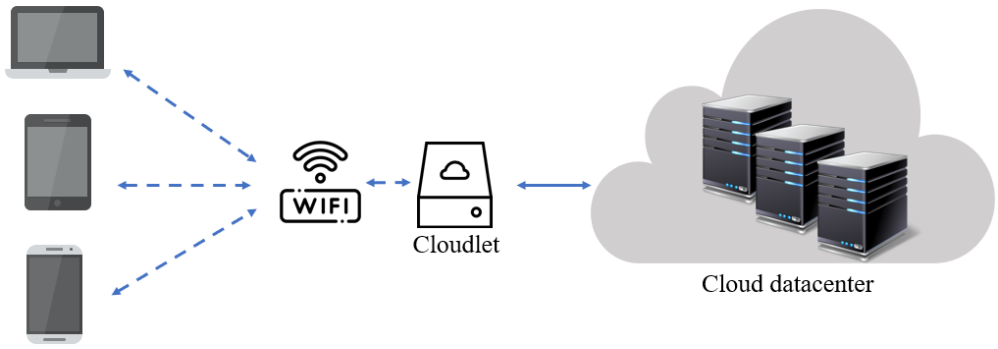

Figure 2.1: The general concept of a Cloudlet

In recent years, to address the above-mentioned challenge and improve the shortcomings of the cloudlet, a novel concept, known as Mobile edge computing(MEC), has been developed.

2.1.2 Mobile Edge Computing

Mobile Edge Computing (MEC) has received extraordinary attention due to the increasing number of emerging applications in recent years. Unlike the conventional mobile cloud computing (MCC) where the computation resources are usually located at tens of kilometers away, MEC is beneficial in terms of service delay due to its proximity to the end devices. However, its computational resources are relatively limited compared to those of the public cloud data centers due to the physical size constraint, which would significantly limit the service capacity of the edge [25]. The concept of MEC was firstly proposed by the European Telecommunications Standard Institute (ETSI) in 2014. It was defined as a new platform that "*provides IT service environment and cloud-computing capabilities at the edge of the mobile network, within the Radio Access Network(RAN) in close proximity to mobile subscribers*" [26]. By deploying MEC servers at the macro or micro base stations, MEC can improve the user experience by processing the user request at the network edge with reduced latency and location-awareness, as well as alleviate the load over the core network, as shown in Figure 2.2. MEC architecture can also be seen as the middle layer between the cloud and IoT devices. Therefore, the infrastructure is derived as a three-layer hierarchy—the cloud, MEC, and IoT devices. Together with network function visualization (NFV) and software-defined network (SDN), MEC has been deemed as a key enabling technology toward the 5G era [27]. In order to expand the influence of MEC with heterogeneous access technologies, e.g., 4G, 5G, WiFi, and fixed connection, ETSI officially extends the terminology of MEC from mobile edge computing to multi-access edge computing in 2017 [28].

According to the ETSI introductory technical white paper [29], MEC can be characterized by:

- *On-premises*: MEC is able to operate in isolated environments. In other words, MEC can perform isolated from the other part of the network and has the right to utilize the local resources.
- *Proximity*: MEC servers are usually positioned in close proximity to mobile users.

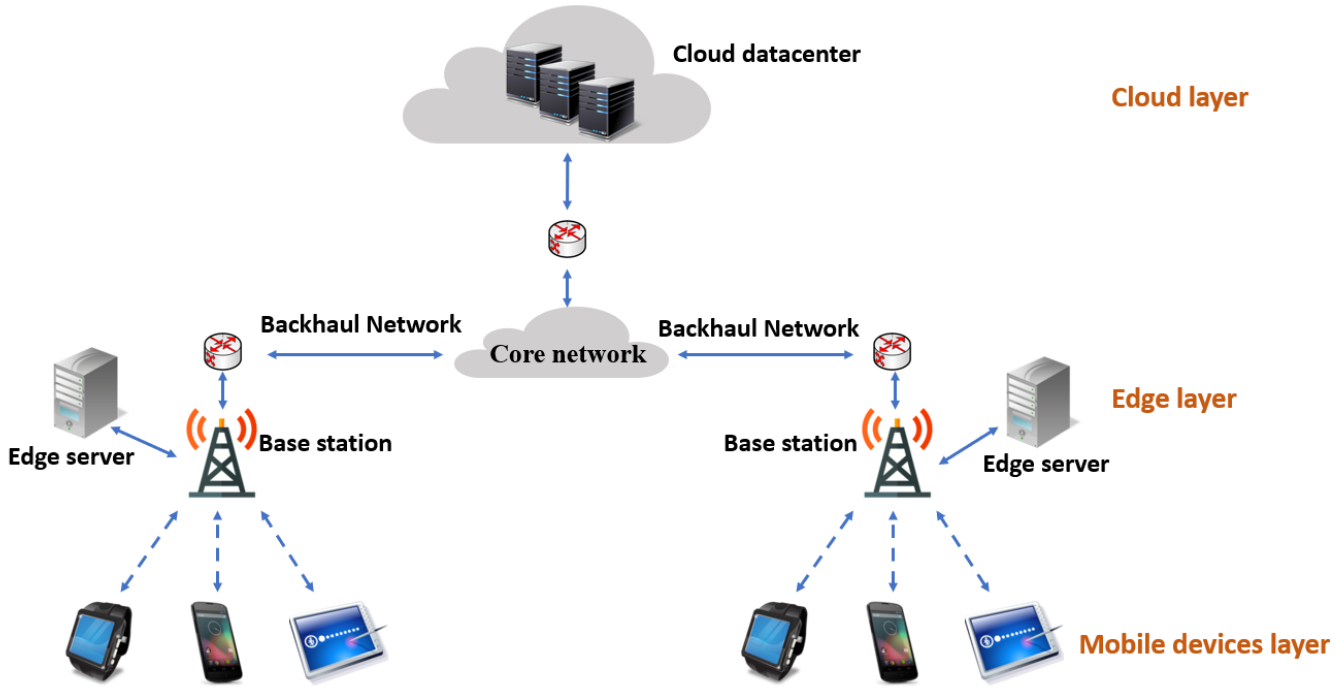

Figure 2.2: Architecture of Mobile Edge Computing

Therefore, MEC can leverage readily useful information from mobile users for further purposes such as big data analysis and processing.

- *Lower latency*: Service providers deploy MEC services at the nearest location to user devices that avoids network data flow from the core network. Therefore, user experience is assured of high quality with ultra-low latency and high bandwidth.

- *Location awareness*: Since the MEC server is near the mobile users, MEC can utilize low-level signaling information received from end-users to estimate their precise locations, which created several applications about location-based services, e.g., the recommendation for nearby restaurants.

- *Network contextual information*: Due to the property of proximity, MEC can utilize the real-time radio network data and local contextual information to optimize the network and QoS. For example, real-time and contextual information can be used to improve user experience by offering personalized services.

MEC is composed of servers or virtual servers distributed in different locations with built-in

smart services. These servers are implemented locally at mobile subscriber premises such as parks, parking lots, and shopping centers. MEC may utilize cellular network elements, such as the base station (BS), WiFi AP, or femto AP (i.e., low power cellular base station). MEC may be deployed at a fixed location, for example, in an office space or on a mobile device located in any moving object, e.g., car or bus. The network operators can deploy MEC servers at various locations within RAN and collocated with different elements of the network edge, such as BSs (aka eNB in 4G and gNB in 5G), optical network units, radio network controller sites, and WiFi access points [21]. To drive intelligence at the base station and to effectively optimize RAN services, MEC technology develops a dynamic ecosystem and a new value chain that allows intelligent and smart services at nearby locations to the mobile subscribers [30].

To summarize, the key value proposition of MEC is that it offers cloud computing by driving cloud resources (e.g., computation, network and storage) to the edge of the mobile network in order to fulfill application requirements that are compute-hungry (e.g., multi-player game applications), latency-sensitive (e.g., AR applications) and high-bandwidth demanding (e.g., mobile big data analysis).

2.2 Task Offloading

Task offloading is a critical component of MEC to improve application performance, response time, and energy consumption. Generally speaking, a vital part regarding computational offloading is to determine the location of execution of the mobile application, such that the execution of the application leads to a lower cost. The offloading location generally refers to the MIoT device itself, the edge server, and the cloud server. However, since multiple different types of servers or combinations exist, a diverse of offloading scenario needs to be considered. The cost includes delay, energy consumption, monetary cost, or a weighted sum of them typically. The cost has different metrics in different conditions according to the type of application, the battery status, and the network environment. For example, if the application is delay-sensitive, shorten the delay is primary concerns. If the

remaining battery capacity is low, the primary concern of the problem should be to extend the battery life by minimizing energy consumption. If the MIoT device is charging, the energy cost will not be a worth-noting problem.

In practice, according to the specific scenario considered and the assumptions made, there are several definitions of the offloading problem. For instance, the number of mobile users and servers may vary in different scenarios. What's more, even if we consider the same scenario, the objective of the offloading problem may vary. On the one hand, delay and energy consumption are the less the better. On the other hand, robustness and privacy protection need to be maximized.

In the MEC framework, the energy consumption and delay of task offloading include three parts. The first part is the transmission energy consumption and delay when MIoT devices upload the task data to remote servers. Due to the dynamic wireless channel states and the different sizes of the computation tasks, the transmission energy consumption may vary among the MIoT devices. Furthermore, in the scenario where the radio resources are shared by MIoT devices, severe interference may occur from time to time. The interference will decrease the data transmission rates for the tasks, and hence lower the energy efficiency of the MEC offloading [10]. The second part is the processing energy consumption and delay caused by processing data. This amount depends on the computing capability of the servers or the MIoT devices. The third part is generated when the remote servers return the results to MIoT devices.

2.2.1 Generic Offloading Process

When modeling the offloading problem, two offloading modes can be considered, i.e., full offloading and partial offloading. For full offloading, the whole computation is offloaded and executed by MEC servers. Whereas for partial offloading, a part of the offloading is offloaded while the rest is executed locally. A MEC-based offloading process consists of three relatively independent sub-tasks, i.e., profiling, partitioning, and decision-making.

Profiling

For the IoT devices, after receiving an offloading request, the profiling module needs to collect three aspects of information. The first one is the current task size and the requirement of delay. The second one is the computing status of IoT devices and offloading resources, i.e., the number of available VMs and the corresponding CPU speed. The last one is the networking condition, which includes network bandwidth and wireless connection status. Due to the nature of mobility, the offloading environment can change dramatically during the offloading process. To maximize the benefits of offloading, it is necessary to update the execution environment information during a specific interval, such that making the real-time offloading decision.

Partitioning

This step is just for partial offloading. Application partitioning is to divide the task into offloadable and non-offloadable components based on different information. For the non-offloadable part, it must be executed locally on the IoT device because this part may involve some equipment that is unavailable in the offloading location, such as camera, GPS, and accelerometer. In addition, as noted by [31], considering some security issues, tasks involve confidential information cannot be executed by third-party cloud services.

Decision-making

In order to make the offloading decision, the following four aspects need to be considered. Firstly, for fine-grained tasks, we should decide which part of the task should be offloaded and which part should be executed locally. Secondly, we should find the right location to offload. Thirdly, we should find the right time to offload, e.g., when the network environment is good, when the amount of communication data is small or when the amount of computation is large. Finally, we need to decide the transmission technique, e.g., using WiFi or cellular network. We summarize some factors that may have an influence on offloading decisions in the next section.

2.2.2 Factors Affecting Offloading Decisions

Table 2.1 illustrates some factors that impact the offloading decision. In the following, some details about the factors are discussed.

Factor	Value
Application type	delay-sensitive, delay-tolerant
Wireless environments	4G;5G;WiFi
User preference	Data confidentiality
Mobile device specification	CPU speed, available memory, battery level
Server specifications	CPU speed, available memory, load
Application specifications	Execution time constraints

Table 2.1: Factors affecting the offloading decision [1]

Classification of Applications

The relative importance of different types of applications to the two factors of delay and energy consumption is different. The offloading decision need to consider the different conditions to achieve the best performance. We present the two type of application.

- *Delay-Tolerant Applications*: for some mobile applications that process machine learning model training and personal health analytics which are tolerable to maximum delays ranging from minutes to hours. Thus, a viable approach is to offload these workloads to the powerful public cloud servers (e.g., Amazon EC2 and Microsoft Azure) for processing when the edge resources are limited or expensive [25]. Novel participatory sensing applications are an excellent illustration of data-intensive yet delay-tolerant applications. With the help of the collected information from sensors embedded in the user's mobile phone, participants can monitor and document health-related issues, such as diet behaviors, depression, and stress conditions [32]. Through the cellular network or any available WiFi, the collected data is uploaded from a user's phone to a back-end server. The submission to the servers may be postponed until there is an energy-efficient network environment since the collected information is not time-critical. Therefore, for delay-tolerant applications, response time is not the primary concern, and reducing energy consumption is more relevant.

- *Delay-Sensitive Applications*: for some delay-sensitive applications (e.g., natural language processing, face recognition, virtual reality, vehicular communications), mobile subscribers look forward to a fast response which is comparable to their cognitive capabilities. In some particular scenarios, minor time errors can also cause severe security problems, e.g., the applications of driverless cars. Thus, for better user experience, the response time of these applications should be as much as possible low. Therefore, for delay-sensitive applications, rapid response is the primary concern.

Heterogeneous Wireless Environments

Mobile devices often have multiple wireless interfaces with varying availability, delay, and energy costs, such as 3G, 4G, 5G, and WiFi for data transfer. The difference between WiFi and cellular networks is shown in Table 2.2.

- *Data Rate*: the achievable data rates for different access technology depends on the environment and can vary significantly. For the 4G, long term evolution advanced (LTE-A) can achieve 3Gbps in DL and 1.5Gbps in UL. For the 5G, using beam division multiple access (BDMA) and non-and quasi-orthogonal or filter bank multi-carrier (FBMC) access, the data rate can achieve 10-50 Gbps [33]. For the 802.11n wireless local area network standard, the WiFi can reach 600 Mbps. With the introduction of the 802.11ax standard, the maximal data rate can achieve almost 9.6Gbps [34].

- *Availability*: cellular networks such as 3G and 4G, usually have much higher availability than WiFi, in particular, 4G has very high coverage in the present society. There are generally WiFi hotspots at home, office, and public indoor places like campus and airports. However, the coverage of WiFi is limited. It is worth noting that 5G networks can implement a much faster transmission rate than conventional WiFi but are still not widespread.

- *Stability*: In general, the 4G has a better stability than the WiFi. The number of people connected to the same WiFi hotpot can make the transmission rate fluctuate.

By contrast, the cellular has the more stable data transmission rate.

- *Energy-Efficiency*: the energy consumption for transferring a fixed amount of data can differ by order of magnitude or more [35]. In general, the WiFi interface is more energy-efficient than the cellular interface. 5G consumes the most energy since it utilizes ultra-high-frequency bands (e.g., millimeter-wave), which have narrower beams and weaker diffraction capability [36].

	Cellular	**WiFi**
Delay	High	Low
Availability	High	Low
Stability	High	Low
Energy-efficiency	Low	High

Table 2.2: Comparison of WiFi and cellular network[2]

Not only the availability and quality of access points (APs) may vary from place to place, but also the uplink and downlink bandwidths fluctuate frequently due to multiple factors such as weather, mobility and building shield. Data transmission in good connectivity consumes much less energy than that under bad conditions.

2.2.3 Resources Allocation

When a offloading decision is made, some offloading related actions also need to be considered. For example, a suitable amount of communication and computation resources have to be allocated for the transmission and processing of each task.

Wireless Resources Allocation

Zhang et al. [10] investigate a multi-users computation offloading and radio resource allocation for MEC in 5G heterogeneous networks, where a Macro Base Station (MBS) equipped with a MEC server, and besides the MBS, there is a Small Base Station which operates in the same frequency band with MBS. The spectrum is divided into K channels of equal bandwidth, and there is a backhaul between the MBS and the SBS. If two or more devices

share the same channel, there will be interference with the transmission speed. Each user chooses offloading decisions, i.e., the computation task is executed locally, transmitted through the MBS or the SBS, respectively. At the same time, each user just selects one channel for transmission. They propose an energy-efficient computation offloading (EECO) scheme consisting of three stages, i.e., mobile device classification, priority determination, and radio resource allocation. The simulation results show the EECO scheme can achieve energy-efficiency improvement under latency constraints.

Computation Resources Scheduling/Allocation

In general, the computation performance of computing devices is controlled by the CPU-cycle frequency f_m (also known as the CPU clock speed). The state-of-the-art mobile CPU architecture has adopted the advanced dynamic frequency and voltage scaling (DVFS) technique, which allows adjusting the CPU-cycle frequency (or voltage), resulting in growing and reducing energy consumption. To leverage the DVFS capability, Dinh et al. [37] propose a semidefinite relaxation-based approach to computation offloading with coupled DVFS with task offloading by dealing with irregular granularity in offloadable data. Simulation results show the proposed scheme can greatly reduce the energy consumption and execution delay when considering multiple edge devices and elastic CPU frequency. Mao et al. [38] consider a scenario where an energy harvesting device is served by a MEC server. DVFS and power control are applied to optimize local execution process and data transmission, respectively. They proposed an online algorithm based on Lyapunov optimization to decide the offloading decision, the CPU-cycle frequencies for local execution, and the transmit power for offloading.

When multiple users offload their tasks to the same MEC server, they share limited computation resources. The principle is that the total amount of computing resources allocated to each user cannot exceed the total capacity, i.e., $\sum_{u \in \mathcal{U}} f_u \leq f_c$, where f_c denotes the total computing capacity (in terms of the number of CPU cycles/s), and f_u [cycles/s] denotes the amount of computing resources that the MEC server allocates to user u. This method of computation resource allocation is investigated in [39], where they consider

a multi-users multi-servers MEC enabled wireless network. They formulate the problem of joint task offloading and resource allocation as a mixed integer nonlinear program that involves jointly optimizing the task offloading decision, uplink transmission power of mobile users, and computing resource allocation at the MEC servers. The proposed approach decomposes the original problem into a Resource Allocation (RA) problem with fixed task offloading decision and a Task Offloading (TO) problem that optimizes the optimal-value function corresponding to the RA problem. Then, the RA problem is solved by convex and quasi-convex optimization techniques, and a heuristic algorithm is proposed to solve the TO problem. The same allocation method is also applied in [40], where they developed an iteration based algorithm to study the task offloading and computation resource allocation problem in a software-defined ultra-dense network.

A comprehensive study on jointly optimizing offloading decisions, power control, and communication and computation resources allocation is given in [41], where the high interference, multi-access property, and limited resources of small cell base stations are taken into account. They propose a sub-optimal algorithm combined with the advantages of genetic algorithm (GA) and particle swarm optimization (PSO). The experimental results show that the proposed algorithm obtains better performance than GA algorithm, PSO algorithm, and random method both in the large-scale and small-scale scenarios.

2.3 Deep Learning

The development of smart cities is closely related to intelligent IoT devices. Artificial intelligence (AI) is the domain that introduce intelligence into the computation system. For a dynamic MEC offloading environment, to make the IoT device become intelligent, we rely on a smaller subset of AI methods, referred as Machine Learning. Machine learning-based offloading frameworks have been studied in these literatures [42, 43, 44]. These models achieve a success addressing smaller scale problems. Nevertheless, applying them in larger scale problem is not easy. As the front runner of machine learning, deep learning (DL) has achieved remarkable performance in handling large amounts of data. These

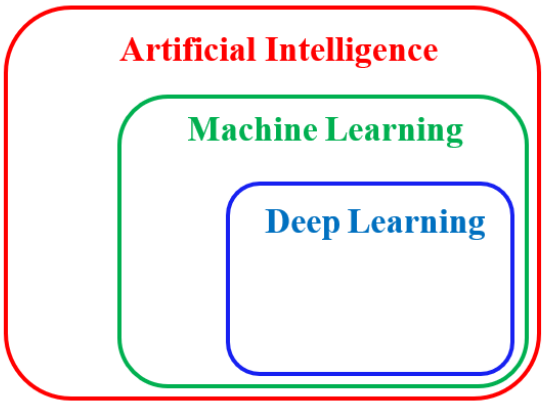

Figure 2.3: Relationship between Artificial Intelligence, Machine Learning and Deep Learning

achievements not only depend on the evolution of DL but also inextricably associated with growing data and computing power. In this section, we introduce several DL models.

2.3.1 Multilayer Perceptron

Multi-layer Perceptron (MLP) is a supervised learning algorithm that learns a function $f(\cdot) : R^m \rightarrow R^n$ by training on a dataset, where m is the number of dimensions for input, and n is the number of dimensions for output. A MLP with two hidden layers is shown in Figure 2.4. MLP can be consider as a subset of neural networks (NNs), and only NNs with a sufficient number of hidden layers (usually more than one) can be regarded as 'deep' models, i.e., deep neural network (DNN).

For a given input vector $\mathbf{x}$, a standard operation of a MLP layer is given as follows:

$$\mathbf{y} = \sigma(\mathbf{W}\mathbf{x} + b) \tag{2.1}$$

Where $\mathbf{W}$ and b denotes the weights and bias. $\mathbf{y}$ is the output of the perceptron and $\sigma(\cdot)$ is the activation function, which is proposed to improve the nonlinearity of the model. Here we list some commonly used activation function [45]:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.2}$$

$$\text{sigmoid}(x) = \frac{1}{1 + e^{-x}} \tag{2.3}$$

Rectified Linear Unit (ReLU) [46],

$$\text{ReLU}(x) = \max(x, 0) \tag{2.4}$$

Additionally, the softmax function is always employed in the last layer when outputting the probabilities.

$$\text{softmax}\left(x_i\right) = \frac{e^{x_i}}{\sum_{j=0}^{n-1} e^{x_j}} \tag{2.5}$$

where n is the number of dimensions for output.

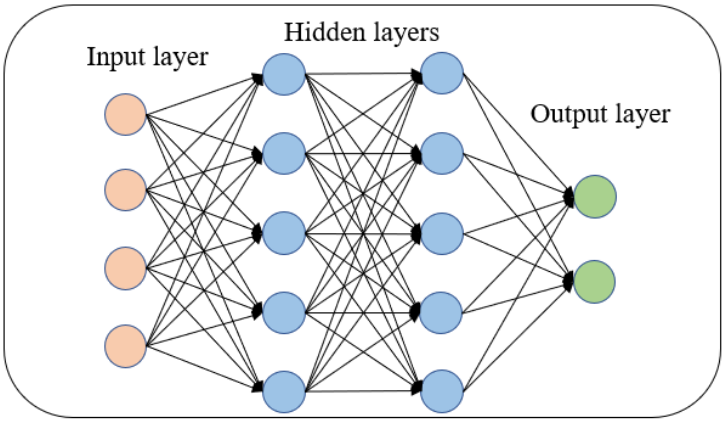

Figure 2.4: Structure of an MLP with 2 hidden layers

2.3.2 Recurrent Neural Network

Recurrent Neural Networks (RNN) are devised to deal with sequential data processing, where sequential correlations exist between samples. Such network can memorize information stored in previous nodes, and then apply to the current output process. Figure 2.5 shows the unfolded structure of the RNN structure.

Backpropagation Through Time (BPTT) can be used to train the traditional RNN. However, gradient vanishing problem is frequently reported in traditional RNN, which make them hard to learn. To deal with the issue, Long Short-Term Memory (LSTM) networks

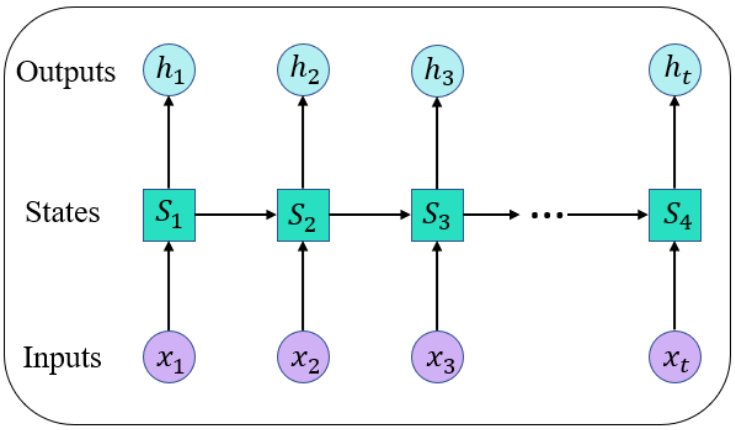

Figure 2.5: Recurrent Neural Network structure. x_t denotes the input sequence, and S_t denotes the state vector. h_t denotes the hidden outputs.

are introduced in [47], which increases the ability of model long-term dependencies by introducing a cell and three gates, e.g., forget gate, input gate, and output gate. The cell aims at remembering values over arbitrary time intervals; hence the word memory in LSTM. The forget gate controls how much information from previous cell state should be forgotten by the current cell state. The input gate handles how much information from the current input layer flows into the current cell state. The output gate controls how much information from the current cell state would be conveyed into the current output layer. Each of the three gates can be implemented by deep neural networks. The structure of an LSTM is shown in Figure 2.6.

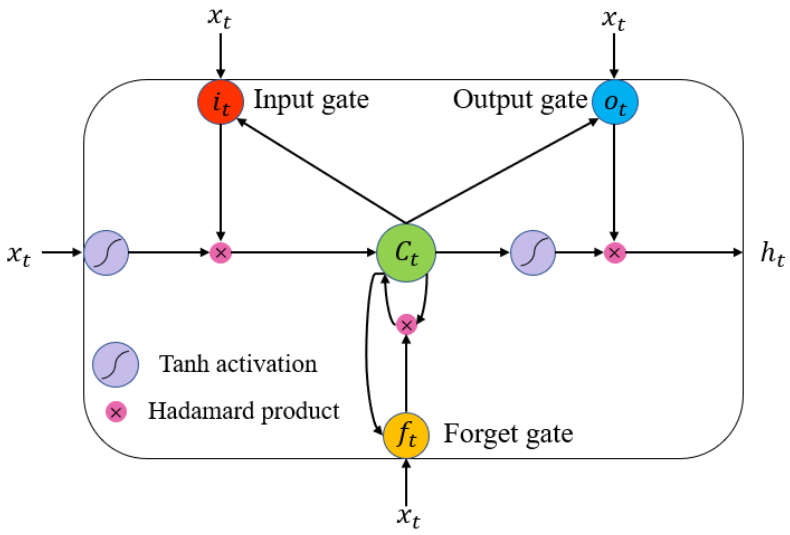

Figure 2.6: The inner structure of an LSTM layer. C_t denotes the cell outputs.

2.4 Deep Reinforcement Learning

Deep Reinforcement Learning (DRL) is the combination of deep learning (DL) and reinforcement learning (RL), but it focuses more on RL and aims to solve sequence decision-making problems. DNN plays the part of the agent and uses its powerful representation ability to fit the value function or the direct strategy to solve the explosion of state-action space or continuous state-action space problem (i.e., state space and action space are high-dimensional). The training goal of the DNN is to optimize its parameters, such that it can select actions that potentially lead to the best future return.

The goal of reinforcement learning(RL) is to enable an agent in the environment to take the best action in the current state to maximize long-term rewards, where the interaction between the agent's action and state through the environment normally is modeled as a Markov Decision Process (MDP). Figure 2.7 shows the structure of Deep Reinforcement Learning. In this section, we present fundamental knowledge of Markov decision process in the first place, and then introduce and discuss approximate dynamic programming (ADP) and some common DRL algorithms.

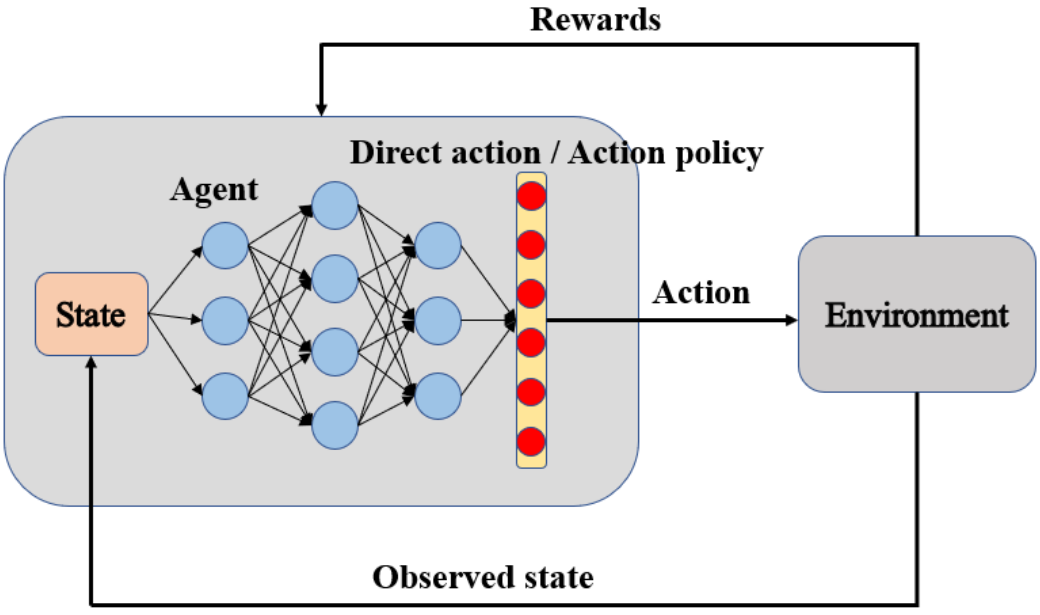

Figure 2.7: Architecture of deep reinforcement learning. The agent is a neural network model that approximates the required function.

2.4.1 Unconstrained Markov Decision Process

Markov decision process (MDP) [48] is useful for studying optimization problems which can be solved by dynamic programming and reinforcement learning techniques. A MDP consists of six necessary elements, a set of decision epochs, a set of possible states $\mathcal{S}$, a set of allowable actions $\mathcal{A}$, a reward function r, the transition probability $\mathcal{P}$ and discount factor γ. By observing the current state $s_t \in \mathcal{S}$, an agent makes action $a_t \in \mathcal{A}$ according to a policy π at each decision epoch. Policy π is a mapping from a state to an action. After that, the agent will receive a reward (cost if we are minimizing) $r_t = r(s_t, a_t)$, then the state evolve to the next state according to the transition probability $Pr(s_{t+1}|s_t, a_t)$. For simplicity, the transition probability is defined as $P^a_{ss'} = Pr\{s_{t+1} = s' \mid s_t = s, a_t = a\}$.

For a finite horizon MDP, the objective of the agent is to find a optimal π to maximize the cumulative reward R_t, i.e.,

$$R_t = r_{t+1} + r_{t+2} + r_{t+3} + \cdots + r_{t+T+1} = \sum_{k=0}^{T} r_{t+k+1} \tag{2.6}$$

where T is the total number of decision epochs.

In the cumulative reward function, a discounting concept can be used to favor immediate rewards over future rewards, i.e.,

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots \gamma^T r_{t+T+1} = \sum_{k=0}^{T} \gamma^k r_{t+k+1} \tag{2.7}$$

where $\gamma \in (0, 1]$ denotes the discount rate. When γ is close to 0, it means the agent is more interested in immediate profits. When γ is close to 1, it means the agent cares more about the future rewards.

A fundamental property which is frequently used in MDP called Bellman equation represents the recursive relationship of the value function. The state-value function $\mathcal{V}^\pi(s)$ (the value of a state under a policy π) is defined as the expected reward starting in state

s and following policy π:

$$\begin{aligned}
\mathcal{V}^{\pi}(s) &= \mathbb{E}_{\pi}\left\{R_t \mid s_t = s\right\} \\
&= \mathbb{E}_{\pi}\left\{\sum_{k=0}^{\infty}\gamma^k r_{t+k+1} \mid s_t = s\right\} \\
&= \mathbb{E}_{\pi}\left\{r_{t+1} + \gamma\sum_{k=0}^{\infty}\gamma^k r_{t+k+2} \mid s_t = s\right\} \\
&= \sum_{a\in\mathcal{A}}\pi(s,a)\sum_{s'\in\mathcal{S}}P_{ss'}^{a}\left[r(s,a) + \gamma\mathbb{E}_{\pi}\left\{\sum_{k=0}^{\infty}\gamma^k r_{t+k+2} \mid s_{t+1} = s'\right\}\right] \\
&= \sum_{a\in\mathcal{A}}\pi(s,a)\sum_{s'\in\mathcal{S}}P_{ss'}^{a}\left[r(s,a) + \gamma\mathcal{V}^{\pi}\left(s'\right)\right]
\end{aligned} \tag{2.8}$$

where $\mathbb{E}_{\pi}$ denotes taking expectation according to the law induced by the policy π, and $\pi(s,a)$ denotes the probability of selecting action a in state s based on policy π. A policy is called an optimal policy π^* when it maximizes the long-term reward. The optimal value function is one which yields maximum value compared to all other value function, and is expressed as:

$$\mathcal{V}^{*}(s) = \max_{\pi}\mathcal{V}^{\pi}(s) \tag{2.9}$$

2.4.2 Constrained Markov Decision Processes

In constrained MDPs (CMDPs), the system incurs an additional immediate cost function $\mathcal{C}$. Specifically, we augment the unconstrained MDP with m cost functions $C_1, \cdots, C_m$, where each cost function C_i: $\mathcal{S}\times\mathcal{A}\to\mathbb{R}$. Therefore, the infinite-horizon discounted-cost of under policy π is defined as,

$$C_i(\pi) = \mathbb{E}_{\pi}\left[\sum_{t=0}^{\infty}\gamma^t C_i\left(s_t, a_t\right)\right] \tag{2.10}$$

Let $b_i \in \mathbb{R}$ denotes a given constant which represents the corresponding constraint upper-bound. In CMDP, the objective is to obtain a policy π that maximizes the long-term reward while guaranteeing the constraints on the long-term costs $C_i(\pi) \leq b_i, \forall i \in \{1,\ldots,m\}$.

Therefore, we have the following problem,

$$\begin{aligned} \max_{\pi} \; R(\pi) =& \mathbb{E}_{\pi}\left[\sum_{t=0}^{\infty}\gamma^t r\left(s_t, a_t\right)\right] \\ & \text{s.t. } C_i(\pi) \leq b_i, \forall i \in \{1, \ldots, m\} \end{aligned} \tag{2.11}$$

Compared to unconstrained MDPs, it is more challenging to obtain a feasible and optimal policy in CMDPs. Normally it requires extra mathematical efforts. By defining an appropriate occupation measure and constructing a linear program over this measure, CMDPs can be solved. Another approach is to use a Lagrangian relaxation technique in which the CMDP can be converted into a corresponding unconstrained problem. Specifically, the Lagrangian function for the CMDP problem is:

$$\mathcal{L}(\pi, \boldsymbol{\lambda}) = R(\pi) - \sum_i \lambda_i \left(C_i(\pi) - d_i\right) \tag{2.12}$$

where $\boldsymbol{\lambda} = (\lambda_1, \cdots, \lambda_m)$ is the Lagrangian multiplier.

$$\max_{\pi} \min_{\boldsymbol{\lambda} \geq 0} \mathcal{L}(\pi, \boldsymbol{\lambda}) = \max_{\pi} \min_{\boldsymbol{\lambda} \geq 0} R(\pi) - \sum_i \lambda_i \left(C_i(\pi) - d_i\right) \tag{2.13}$$

and invoking the minimax theorem,

$$\max_{\pi} \min_{\boldsymbol{\lambda} \geq 0} \mathcal{L}(\pi, \boldsymbol{\lambda}) = \min_{\boldsymbol{\lambda} \geq 0} \max_{\pi} \mathcal{L}(\pi, \boldsymbol{\lambda}) \tag{2.14}$$

Then the constrained problem (2.11) can be converted to the following unconstrained problem:

$$(\pi^*, \boldsymbol{\lambda}^*) = \arg \min_{\boldsymbol{\lambda} \geq 0} \max_{\pi \in \Pi_\theta} \mathcal{L}(\pi, \boldsymbol{\lambda}) \tag{2.15}$$

The optimal policy π^* and Lagrange multiplier $\boldsymbol{\lambda}^*$ admits the following *saddle point* property,

$$\mathcal{L}\left(\boldsymbol{\lambda}^*, \pi\right) \geq \mathcal{L}\left(\boldsymbol{\lambda}^*, \pi^*\right) \geq \mathcal{L}\left(\boldsymbol{\lambda}, \pi^*\right), \quad \forall \pi, \boldsymbol{\lambda} \geq 0 \tag{2.16}$$

The right hand side of (2.14) can be solved on two-time scales: on a faster time scale dynamic programming (model-based) or gradient-ascent (model-free) is performed on state

values to find the optimal policy for a given set of Lagrangian variables, and on a slower time scale, gradient-descent is performed on the dual variables [49, 50].

2.4.3 Approximate Dynamic Programming

Approximate dynamic programming (ADP) refers to a class of computational methods that can be used to solve problems that are sometimes large and complex, and are usually (but not always) stochastic when given a perfect dynamical model of MDP. It is always introduced as a way to overcome the classic curse of dimensionality that is well-known to plague the use of Bellman's equation [51]. The first feature of ADP is to replace the true value function with some sort of statistical approximation, which can be a multilevel aggregation or the use of neural networks. The second feature is that instead of working backward through time to calculate the value function, ADP steps forward in time. In the following, two basic model-based methods are introduced.

Value Iteration

Value Iteration (VI) is the most widely used algorithm for solving Markov decision process problems. VI needs perfect knowledge of the state transition probabilities and the rewards for each transition. The update rule of VI is expressed as follows:

$$\mathcal{V}_{k+1}(s) = \max_{a} \sum_{s'} P^{a}_{ss'} \left[r(s,a) + \gamma \mathcal{V}_k \left(s' \right) \right] \tag{2.17}$$

for all $s \in \mathcal{S}$, where subscript k denotes k-th iteration.

This technique requires a large number of iterations to converge to the optimal value function until the value function calculated on two successive steps are close enough, i.e.,

$$\max_{s \in \mathcal{S}} |\mathcal{V}_{k+1}(s) - \mathcal{V}_k(s)| < \epsilon \tag{2.18}$$

where ϵ is a predefined threshold parameter. The smaller the threshold, the higher the accuracy of the algorithm. The steps of value iteration are shown in Algorithm 1.

Algorithm 1 Value Iteration Algorithm

Initialization:

$\mathcal{S}$: Set of possible states.
$\mathcal{A}$: Set of possible actions.
$\mathcal{P}$: The transition probability.
r: Reward Function.
ϵ: A small positive threshold

Output:

Optimal value function array $\mathcal{V}$

Arbitrarily initialize $\mathcal{V}(s) \in \mathbb{R}$ for every state $s \in S$.
repeat
  $\Delta \longleftarrow 0$
  for each state $s \in \mathcal{S}$ **do**
    $v \longleftarrow \mathcal{V}(s)$
    $\mathcal{V}(s) \longleftarrow \max_a \sum_{s'} P^a_{ss'} \left[r(s,a) + \gamma \mathcal{V}\left(s'\right)\right]$
    $\Delta \longleftarrow \max(\Delta, |v - \mathcal{V}(s)|)$
  end for
until $\Delta < \theta$
return Optimal value function array $\mathcal{V}$

Policy Iteration

There is another way of finding optimal policies, is known as policy iteration (PI). Policy iteration initializes a policy arbitrarily and once a policy π is given, it is feasible to examine whether it is the best policy. If a new policy π' is better than a old policy π, it means the value function $\mathcal{V}^{\pi'}$ is greater than or equal to $\mathcal{V}^{\pi}$. Therefore, policy iteration performs two steps: value evaluation and policy improvement. Value evaluation calculates the value function of each state given the current policy. Policy improvement updates the current policy if any improvement is possible. The algorithm terminates when the policy converges [52]. The steps of policy iteration are shown in Algorithm 2.

2.4.4 Value-based Method

When applying policy iteration or value iteration to solve MDP problem, we must have a perfect knowledge of the MDP, i.e., the transition probability of known. However, in real life, the transition probability of a system is hard to know or not easy to calculate. Therefore, some model-free models is developed.

Algorithm 2 Policy Iteration Algorithm

Initialization:

$\mathcal{S}$: Set of possible states.

$\mathcal{A}$: Set of possible actions.

$\mathcal{P}$: The transition probability.

r: Reward Function.

ϵ: a small positive threshold

Output:

Optimal value function array $\mathcal{V}$

Arbitrarily initialize $V(s) \in \mathbb{R}$ and a policy $\pi(s)$ for all state $s \in S$.

Step 1: Value Evaluation

repeat

  $\Delta \longleftarrow 0$

  for each state $s \in \mathcal{S}$ **do**

    $v \longleftarrow \mathcal{V}(s)$

    $\mathcal{V}(s) \longleftarrow \max_a \sum_{s'} P^a_{ss'} \left[r(s,a) + \gamma \mathcal{V}\left(s'\right)\right]$

    $\Delta \longleftarrow \max(\Delta, |v - \mathcal{V}(s)|)$

  end for

until $\Delta < \theta$

Step 2: Policy Improvement

for each state $s \in \mathcal{S}$ **do**

  $J \longleftarrow \pi(s)$

  $\pi(s) \longleftarrow \arg\max_a \sum_{s'} P^a_{ss'} \left[r(s,a) + \gamma \mathcal{V}^{\pi}\left(s'\right)\right]$

end for

if π does not converge (i.e., $J \neq \pi(s)$) **then**

  Return to Step 1.

end if

return optimal policy π and optimal value function array $\mathcal{V}$

Q-Learning

Q-learning [53] is a powerful algorithm to solve reinforcement learning problem, which uses Q-value (also called action-value function) to iteratively improve the behavior of the learning agent. Traditionally, the Q-value can be stored in a table, whose size is equal to the dimensionality of the action space times the dimensionality of the state space.

Similar to (2.8), the action-value function $\mathcal{Q}^{\pi}(s,a)$ can be defined as:

$$\begin{aligned}\mathcal{Q}^{\pi}(s,a) &= \mathbb{E}_{\pi}\left\{R_t \mid s_t = s, a_t = a\right\} \\ &= \mathbb{E}_{\pi}\left\{\sum_{k=0}^{\infty}\gamma^k r_{t+k+1} \mid s_t = s, a_t = a\right\} \\ &= \mathbb{E}_{\pi}\left\{r_{t+1} + \gamma\sum_{k=0}^{\infty}\gamma^k r_{t+k+2} \mid s_t = s, a_t = a\right\} \\ &= r(s,a) + \gamma\sum_{s'\in\mathcal{S}} P^{a}_{ss'}\left[\sum_{a'\in\mathcal{A}}\pi(s',a')\mathcal{Q}^{\pi}\left(s',a'\right)\right]\end{aligned} \tag{2.19}$$

If we have the optimal action value function $\mathcal{Q}^{*}(s,a)$ for all state-action pairs, we can get the optimal policy π^{*} by:

$$\pi^{*}(s,a) = \begin{cases} 1 & \text{if } a = \underset{a\in\mathcal{A}}{\operatorname{argmax}}\mathcal{Q}^{*}(s,a) \\ 0 & \text{otherwise} \end{cases} \tag{2.20}$$

Now, the problem is converted to find optimal values of Q-value, i.e., $\mathcal{Q}^{*}(s,a)$, for every possible state-action pairs, and this can be achieved by means of iteration processes. Particularly, the Q-value is updated according to the following rule:

$$\mathcal{Q}_{t+1}(s,a) = \mathcal{Q}_t(s,a) + \alpha_t\left[r_t(s,a) + \gamma\max_{a'}\mathcal{Q}_t(s',a') - \mathcal{Q}_t(s,a)\right] \tag{2.21}$$

The essence behind this update rule is to calculate the Temporal Difference (TD) between the predicted Q-value, i.e., $r_t(s,a) + \gamma\max_{a'}\mathcal{Q}_t(s',a')$ and its current value, i.e., $\mathcal{Q}_t(s,a)$. As shown in equation (2.21), the objective of the learning rate α_t is to determine the impact of new information to the current Q-value. The learning rate can be consider as a

constant, or it can also decrease with the increase the number of iteration.

Deep Q-Network

An optimal policy can be efficiently obtained by using the Q-learning algorithm when the problem with a small state space and action space. However, in real life, the system models are more complex, and these spaces are usually large. As a consequence, the Q-learning algorithm may not be able to find the optimal policy because it is infeasible to store the value of every state-action pair in a table. Thus, Deep Q-Network (DQN) algorithm is proposed to deal with this problem. Intuitively, instead of using a Q-table, a neural network is implemented to approximate the value of $\mathcal{Q}^*(s, a)$.

As stated in [54], when a nonlinear function approximator is used, the average reward obtained by reinforcement learning algorithms may not be stable or even diverge. This originates from the fact that small adjustment of Q-values may hugely influence the policy. Therefore, the data distribution and the correlations between the Q-values and the target values $r_t(s, a) + \gamma \max_{a'} \mathcal{Q}_t(s', a')$ are diverse. Two mechanisms, i.e., experience replay and target Q-network, are used to solve the problem.

- *Experience reply mechanism*: The algorithm first initializes a replay memory $\mathcal{D}$, i.e., replay buffer, which is used to store transitions (s_t, a_t, r_t, s_{t+1}), i.e., experiences generated randomly through using ϵ-greedy policy. Then, instead of feeding successive transitions into a mini-batch, the algorithm randomly picks transitions from $\mathcal{D}$, which consists of a mini-batch to train the DNN. The Q-values obtained by the trained DNN will be used to obtain new experiences. When the memory is full, new transitions will replace the old ones. This technique makes the DNN trained more efficiently by using both old and new experiences. Additionally, by making use of the experience replay memory, the transitions are more independent and identically distributed, and hence the correlations between observations can be solved.

- *Fixed target Q-network*: During the training, the action-value function will be shifted. Therefore, the action-value function estimations have a high probability of being out

of control if a constantly shifting set of values is used to update the deep Q-network. This may result in the destabilization of the algorithm. To address this concern, the target Q-network is used to update infrequently (every hundreds of iteration or so). In this way, the target network stays constant for a period of time, thereby stabilizing the algorithm. The DQN algorithm is summarized in Algorithm 3.

Algorithm 3 Deep Q-Network

Initialization:

action-value function Q with random weights θ
target action-value function $\hat{Q}$ with random weights θ'
relay memory $\mathcal{D}$

for *episode* in $\{1, 2, \ldots, M\}$ **do**
  Initialize initial state s_0
  for t in $\{0, 1, \ldots, T\}$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \arg\max_a Q(s_t, a; \theta)$
    Execute action a_t and observe reward r_t and next state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in $\mathcal{D}$
    Sample random mini-batch of transitions (s_j, a_j, r_j, s_{j+1}) from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j, & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(s_{t+1}, a'; \theta'), & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(s_j, a_j; \theta))^2$ with respect to the network parameters θ
    Every C step rest $\hat{Q} = Q$
  end for
end for

Double DQN

The shortcoming of DQN is that they extremely slow down training and increase the sample complexity. In addition, DQN still has stability issues. This is because the DQN algorithm always performs badly by reason of the over-estimations of true action values. These over-estimations result from the introduction of positive bias since Q-learning uses the maximum action value as an approximation of the maximum expected action value [55] as shown in Eq.(2.21). In particular, at the beginning of learning when Q-values are far from the correct value, if an action is over-estimated and picked by the target network as the next greedy action, the learned Q-value will also become over-estimated. Therefore, to

address the over-estimation issue, the authors in [55] propose a solution using two Q-value functions, i.e., $\mathcal{Q}_1$ and $\mathcal{Q}_2$, to select and evaluate action values at the same time through the loss function as follows:

$$r_j + \gamma \mathcal{Q}_2\left(s_{j+1}, \arg\max_{a_{j+1}} \mathcal{Q}_1(s_{j+1}, a_{j+1}; \theta_1); \theta_2\right) - \mathcal{Q}_1(s_j, a_j; \theta_1) \tag{2.22}$$

It is worth noting that the selection of an action, in the arg max, is still due to online weight θ_1. This indicates that, as in Q-learning, we are still giving the greedy action according to the current values, as defined by θ_1. However, the second set of weights θ_2 is used to give the Q-value of this policy. The second set of weights can be updated symmetrically by switching the roles of θ_1 and θ_2. Motivated by this thought, the authors in [55] then propose Double DQN model [56] with the loss function update rule as follows,

$$r_j + \gamma \hat{\mathcal{Q}}\left(s_{j+1}, \arg\max_{a_{j+1}} \mathcal{Q}(s_{j+1}, a_{j+1}; \theta); \theta'\right) - \mathcal{Q}(s_j, a_j; \theta) \tag{2.23}$$

where the weights of the second network θ_2 are replaced with the weights of the target networks θ'.

2.4.5 Policy-based Method

This subsection focuses on a particular family of reinforcement learning algorithms that use policy gradient methods. Different from the previously introduced value-based method, a map from states to actions is defined as policy π_θ, which is characterized by parameter θ. By using a parameterized function estimator, policy search methods directly learn to estimate the policy π_θ. The objective of the neural network is to maximize an objective function representing the *return* $R(\tau)$ (sum of rewards) of the trajectories $\tau = (s_0, a_0, s_1, a_1, \ldots, s_T, a_T)$ selected by the policy π_θ:

$$J(\theta) = \mathbb{E}_{\tau\sim\rho_\theta}[R(\tau)] = \mathbb{E}_{\tau\sim\rho_\theta}\left[\sum_{t=0}^{T} \gamma^t r\left(s_t, a_t, s_{t+1}\right)\right] \tag{2.24}$$

where $\mathbb{E}_{\tau\sim\rho_\theta}$ represents taking the mathematical expectation of the return over all

possible trajectories. The likelihood that a trajectory is generated by the policy π_θ is noted $\rho_\theta(\tau)$ and given by:

$$\rho_\theta(\tau) = p_\theta\left(s_0, a_0, \ldots, s_T, a_T\right) = p_0\left(s_0\right) \prod_{t=0}^{T} \pi_\theta\left(s_t, a_t\right) p\left(s_{t+1} \mid s_t, a_t\right) \tag{2.25}$$

where $p_0(s_0)$ denotes the probability that the initial state is s_0, and $p(s_{t+1}|s_t, a_t)$ represents the transition probability. After having the probability distribution of the trajectories, we can rewrite the objective function (2.24) in the form of integration:

$$J(\theta) = \int_\tau \rho_\theta(\tau) R(\tau) d\tau \tag{2.26}$$

Monte-Carlo (MC) sampling can be used to give an estimation of the objective function. The core idea is to sample multiple trajectories $\{\tau_i\}$ and average the obtained returns:

$$J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} R\left(\tau_i\right) \tag{2.27}$$

The gradient ascent method can be applied on the weights θ to maximize $J(\theta)$, what we only need is the gradient $\nabla_\theta J(\theta) = \frac{\partial J(\theta)}{\partial \theta}$. Once a proper estimation of the policy gradient is obtained, gradient ascent method is applied straightforward: $\theta \leftarrow \theta + \eta \nabla_\theta J(\theta)$, where η denotes the learning rate.

Estimating the policy gradient

Introduced in [57], an effective estimation of the policy gradient is proposed. Considering that the obtained return $R(\tau)$ of one trajectory is not dependent on the parameters θ, the

policy gradient can be calculated by the following way:

$$\begin{aligned}\nabla_\theta J(\theta) =& \nabla_\theta \int_\tau \rho_\theta(\tau) R(\tau) d\tau \\ =& \int_\tau \left(\nabla_\theta \rho_\theta(\tau)\right) R(\tau) d\tau \\ =& \int_\tau \rho_\theta(\tau) \nabla_\theta \log \rho_\theta(\tau) R(\tau) d\tau\end{aligned} \tag{2.28}$$

Here log-trick is used $\rho_\theta(\tau)\nabla_\theta \log \rho_\theta(\tau) = \rho_\theta(\tau)\frac{\nabla_\theta \rho_\theta(\tau)}{\rho_\theta(\tau)} = \nabla_\theta \rho_\theta(\tau)$. Therefore, we have the following form of a mathematical expectation:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \rho_\theta}\left[\nabla_\theta \log \rho_\theta(\tau) R(\tau)\right] \tag{2.29}$$

By equation (2.25), the log-likelihood of a trajectory is:

$$\log \rho_\theta(\tau) = \log p_0\left(s_0\right) + \sum_{t=0}^{T} \log \pi_\theta\left(s_t, a_t\right) + \sum_{t=0}^{T} \log p\left(s_{t+1} \mid s_t, a_t\right) \tag{2.30}$$

Because $\log p_0$ and $\log p\left(s_{t+1} \mid s_t, a_t\right)$ do not depend on the parameters θ, the gradient of the log-likelihood is simply:

$$\nabla_\theta \log \rho_\theta(\tau) = \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta\left(s_t, a_t\right) \tag{2.31}$$

The policy gradient is then given by:

$$\begin{aligned}\nabla_\theta J(\theta) =& \mathbb{E}_{\tau \sim \rho_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta\left(s_t, a_t\right) R(\tau)\right] \\ =& \mathbb{E}_{\tau \sim \rho_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta\left(s_t, a_t\right)\left(\sum_{t=0}^{T} \gamma^t r_{t+1}\right)\right]\end{aligned} \tag{2.32}$$

Now, it is straightforward to estimate the policy gradient using Monte-Carlo (MC) sampling, which produces the REINFORCE algorithm, and the procedure of the algorithm is summarized in Algorithm 4.

REINFORCE is a Monte-Carlo variant of policy gradients. The agent collects several

trajectories $\{\tau_i\}$ using its current policy, and uses them to update the policy parameter.

Algorithm 4 REINFORCE:A Monte-Carlo Policy-Gradient Method

Initialization:

a differentiable policy parameterization $\pi(a|s,\theta)$ and step size α

1: **loop**

2: Using the current policy π_θ to generate N episodes $\{\tau_i\}$ and observe the returns $\{R(\tau_i)\}$.

3: Estimate the policy gradient as an average over the episodes $\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta\left(s_t,a_t\right) R\left(\tau_i\right)$

4: Update the policy using gradient ascent. $\theta \leftarrow \theta + \alpha\nabla_\theta J(\theta)$

5: **end loop**

Policy Gradient theorem

The basic idea behind policy-based algorithms is to adjust the parameters θ of the policy in the direction of the performance gradient $\nabla_\theta J(\pi_\theta)$. Sutton et al. [58] proposed that the policy gradient can be estimated by substituting the return of the sampled trajectory with the Q-function of the state-action pair, which results in the policy gradient theorem:

$$\begin{aligned}\nabla_\theta J(\pi_\theta) &= \int_{\mathcal{S}} \rho^\pi(s) \int_{\mathcal{A}} \nabla_\theta \pi_\theta(a|s)\mathcal{Q}^\pi(s,a) da ds \\ &= \mathbb{E}_{s\sim\rho^\pi, a\sim\pi_\theta}\Big[\nabla_\theta \log \pi_\theta(a|s)\mathcal{Q}^\pi(s,a)\Big]\end{aligned} \tag{2.33}$$

where ρ^π is the distribution of states reachable under the policy π. Since the actual return $R(\tau)$ is substituted with $\mathcal{Q}^\pi(s,a)$, the policy gradient can be calculated by taking expectation over single transitions instead of complete trajectories, which allows bootstrapping as in temporal difference (TD) methods [59].

Actor-Critic Methods

The major concern with the above REINFORCE algorithm is the high variance of the policy gradient. In order to solve this limitation, the policy gradient theorem provides an actor-critic architecture able to learn parameterized policies. If we can estimate the action-value function $\mathcal{Q}(s,a)$ accurately and use it to guide the policy to update, and then there

will be better results. Actor-critic architecture is composed of two parts: an actor and a critic. The actor is responsible for giving the appropriate action based on the current state, and the critic is responsible for estimating the Q-value function. In DRL, both the actor and the critic can be represented by non-linear neural network function approximators. The actor uses gradients derived from the policy gradient theorem to adjust the policy parameters. Figure 2.8 shows the actor-critic architecture.

In order to reduce the variance further, subtracting a baseline function from the Q-value function is a feasible approach. The value function $\mathcal{V}^{\pi}(s)$ can be regarded as the baseline function. Therefore, we define the following advantage function which indicates whether things get better or worse than expected.

$$A^{\pi}(s,a) = \mathcal{Q}^{\pi}(s,a) - \mathcal{V}^{\pi}(s) \tag{2.34}$$

Now the problem is that we have to estimate two functions : $\mathcal{Q}^{\pi}(s,a)$ and $\mathcal{V}^{\pi}(s)$. The policy gradient can be rewritten as:

$$\nabla_{\theta} J(\pi_{\theta}) = \mathbb{E}_{s\sim\rho^{\pi}, a\sim\pi_{\theta}}\Big[\nabla_{\theta} \log \pi_{\theta}(a|s) A^{\pi}(s,a)\Big] \tag{2.35}$$

There are different methods can be used to estimate the advantage function:

- $A(s,a) = r(s,a) + \gamma\mathcal{V}(s') - \mathcal{V}(s)$ is the **TD advantage estimate or TD error**.
- $A(s,a) = \sum_{k=0}^{n-1} \gamma^{k} r_{t+k+1} + \gamma^{n}\mathcal{V}(s_{t+n+1}) - \mathcal{V}(s_t)$ is the **n-step advantage estimate**.
- $A(s,a) = G(s,a) - \mathcal{V}(s)$ is the **MC advantage estimate**, the action-value function is replaced by the actual return.

Deep Deterministic Policy Gradient

Although DQN algorithm can solve problems with high-dimensional state spaces, it can only handle problems with discrete and low-dimensional action space. However, the action spaces are continuous in many applications. The DQN algorithm is not able to be

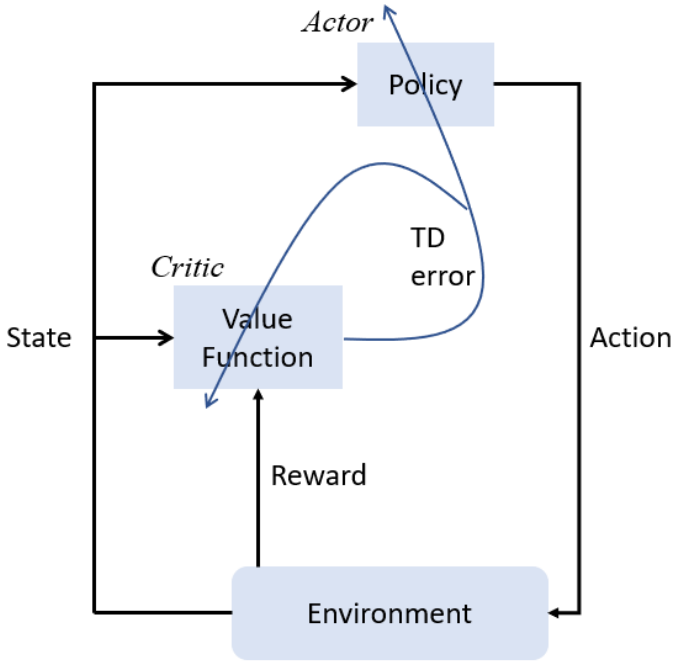

Figure 2.8: Actor-Critic architecture

straightforwardly applied to the system with continuous actions because they depend on choosing the best action that maximizes the Q-value function. Especially, a full search in a continuous action space to find the optimal action is often infeasible. In [60], the authors introduce a model-free off-policy actor-critic algorithm that can learn policies directly in continuous action spaces, which is based on the deterministic policy gradient (DPG) algorithm proposed in [61]. The DPG algorithm utilizes a deterministic policy $\mu_\theta(s)$ with parameter θ which deterministically map a state to a specific action. In the continuous case, we naturally give that the gradient of the objective function is the same as the gradient of the Q-value. Therefore, a unbiased estimate $\mathcal{Q}^\mu(s,a)$ can change the policy $\mu_\theta(s)$ in the direction of $\nabla_\theta \mathcal{Q}^\mu(s,a)$, which results in an action with a higher associated return:

$$\nabla_\theta J(\theta) = \mathbb{E}_{s\sim\rho_\mu}\left[\nabla_\theta \mathcal{Q}^\mu(s,a)|_{a=\mu_\theta(s)}\right] \tag{2.36}$$

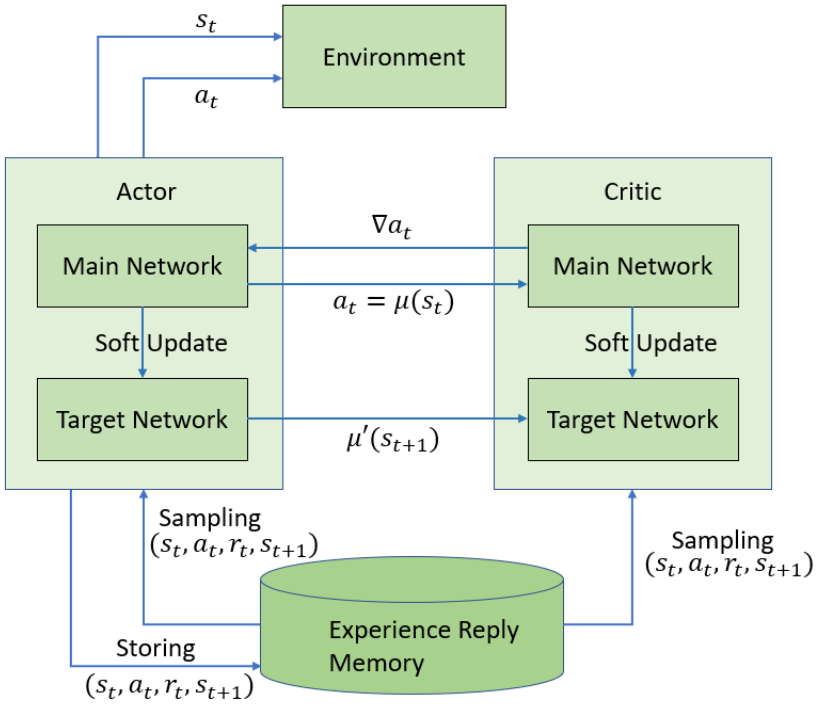

Figure 2.9: The principle diagram of the DDPG algorithm

where ρ_μ denotes the distribution of states reachable by the policy. The chain rule is used to expand the above gradient and we have:

$$\nabla_\theta J(\theta) = \mathbb{E}_{s\sim\rho_\mu}\left[\nabla_\theta \mu_\theta(s) \times \nabla_a \mathcal{Q}^\mu(s,a)|_{a=\mu_\theta(s)}\right] \tag{2.37}$$

According to the update rule, the authors then propose Deep DPG (DDPG) algorithm, which extended the DPG method with non-linear function approximators, i.e., DNN. DDPG combines the idea of DQN and DPG. The critic ideas borrowed from DQN are using an experience replay memory and target networks. The authors also found it is better to update the target networks with slowly tracking on the learned networks, i.e.,

$$\theta \leftarrow \tau\theta + (1-\tau)\theta' \tag{2.38}$$

where $\tau \ll 1$. This means that the target values are always late with respect to the trained networks. This technique enormously enhance the stability of learning process. It

is worth noting that a major challenge of learning in continuous action spaces is exploration. Therefore, an exploration policy μ' is constructed by adding noise samples from a noise process $\mathcal{N}$ to the actor policy. The architecture of the DDPG method is outlined in Figure 2.9.

2.5 Existing Approaches

As introduced in the sections above, many state-of-art DRL research can be used to deal with computation offloading problem. In the rest part of this chapter, we will review some related works according to the technique used.

2.5.1 Use Cases of MLPs

Yu et al. [62] propose a Deep Supervised Learning model in fine-grained computation offloading MEC framework to minimize the computation and offloading overhead. Their problem is formulated as a multi-label classification problem and the offloading actions taken by a mobile user consider the local execution overhead as well as varying network conditions. They obtain an optimal solution by the exhaustive strategy and the optimal solution can be used to train a DNN. The composite state of the MEC network is the input of DNN, and the output is the offloading decision. Their method provides a pre-calculated offloading solution which is employed when a certain level of knowledge about the application and network conditions is known.

Huang et al. [63] study MEC networks where multiple wireless devices choose to offload their computation tasks to an edge server by one wireless access point. They propose a distributed deep learning-based offloading (DDLO) algorithm for MEC networks, where multiple parallel DNNs with identical network structures are used to generate offloading decisions, then, the offloading decision with the lowest system utility is chosen as the output. They adopt a shared replay memory to store newly generated offloading decisions. Specifically, all DNNs share the same memory, and each DNN randomly extracts a batch of data samples from the memory to train and improve all DNNs. In [64], they apply

DDLO to MEC networks with multiple edge servers, one cloud servers and multiple wireless devices and further enhance the performance of DDLO by using heterogeneous DNN structures. The simulation shows that the heterogeneous DDLO achieves better performance and consumes several less computation time than a linear programming Relaxation-Based algorithm.

2.5.2 Use Cases of RNN

Miao et al. [65] propose a computation offloading and task migration algorithm based on task prediction. They utilize LSTM to predict the feature of computation tasks and to assist the estimation of delay. Based on the task prediction and multiple features of edge computation nodes, a comprehensive evaluation is conducted to obtain optimal offloading strategy.

Cui et al. [66] propose an LSTM-based availability prediction of base station to improve the offloading performance. Empirical Mode Decomposition (EMD) is utilized to decompose time series into sub-components, and the decomposition data is used to build the corresponding LSTM prediction model. Finally, the sum of each sub-predicted data is used to as the output of the whole model.

Zhao et al. [67] propose a multi-LSTM model to predict the real-time traffic of a small base station (SBS). By using a Usage Detail Records data-set from one network operator, they train the deep learning model using the traffic sequence of SBS and the number of users as the input. Through the multi-LSTM model, the real-time traffic load can be obtained, which can help controller better monitor the network environment and make the offloading decision more intelligent.

2.5.3 Use Cases of Dynamic Programming

Shahzad et al. [68] present a novel heuristic offloading algorithm called 'Dynamic Programming with Hamming Distance Termination' (DPH), which uses dynamic programming combined with randomization. It also uses a hamming distance as a termination

criterion.

Wu et al. [69] design a value iteration algorithm to maximize the total long-term reward in a vehicular fog and cloud computing system, where the departure of occupied vehicles is taken into account. The task offloading problem is formulated as an semi-Markov decision process. The results demonstrate the proposed offloading scheme can achieve higher gains than the common greedy method.

Lei et al. [6] study a computation offloading and multi-user scheduling problem in narrowband (NB) IoT MEC system, where the special characteristics of the NB-IoT technology is considered, and stochastic task arrival model is applied. The dynamic optimization problem is formulated as an infinite-horizon average-reward continuous-time Markov decision process (CTMDP) model. They utilize the linear function approximation, Temporal-Difference learning with post-decision state and semi-gradient descent algorithm, and design a semi-distributed algorithm. To reduce the complexity, uniformization is applied to decompose the global value function into the sum of local value functions. Simulation results show that the proposed algorithm achieve a better performance than random method and queue-aware method.

2.5.4 Use Cases of DRL

Value-based Method

Li et al. [70] investigate a multi-user and one edge server MEC scenario. They use a deep Q-learning algorithm to jointly optimize the offloading decision and computation resource allocation, so as to minimize the weighted sum cost of delay and energy consumption of all user equipment. The simulation result shows that the proposed method can achieve a significant reduction on the total cost of the system, as compared to fully-local, fully-offloading, and vanilla Q-learning baseline methods.

Similarly, a multiple MEC servers scenario and one IoT device scenario is considered in [71]. The authors propose a DRL-based offloading scheme for an IoT device with energy harvesting (EH). The computation offloading process can be regarded as an MDP.

The IoT device selects the offloading location and the offloading rate based on the system states composed by the present battery status, the previous radio transmission rate, and the predicted amount of harvested energy. The DRL offloading scheme utilizes the transfer learning method [72] to save the random exploration time at the initial state of the offloading process. They also use a convolutional neural network (CNN) to compress the state space and accelerates the learning speed.

Chen et al. [73] consider a MEC scenario where a mobile user in an ultra-dense sliced RAN with multiple base stations. They propose a Double DQN-based online strategic computation offloading algorithm with Q-function decomposition technique to maximize the long-term utility performance. The problem of finding an optimal offloading policy is modeled as a MDP, and the state includes the task queue length, the energy queue length and channel qualities. The proposed algorithm survive the curse of high dimensionality in state space and does not require a priori information of dynamics statistics.

All of the above work is based on the assumption that the perfect channel state information is available. However, in practice, the offloading agent may not have the perfect knowledge of channel condition because of limited sensing capabilities of IoT sensors and information loss [74]. Xie et al. [75] consider an IoT fog system with imperfect channel information, i.e., only the estimated channel state information can be observed in each block fading. The IoT device estimates the channel state information for the sub-channels and contend a subset to access. According to the information of access channels, the IoT device decides the number of tasks that are executed locally and the number of tasks offloaded to fog server. They formulate a partially observable Markov decision process (POMDP) problem with the objective of minimizing the average energy consumption while guaranteeing the requirement on task processing delay. An offline algorithm based on deep recurrent Q-network (DRQN) is developed to find the optimal offloading solution, which is a combination of an LSTM and a DQN.

Policy-based Method

To deal with the challenge of unavailable edge/cloud infrastructure in remote areas, cheng et al. [76] use a deep Actor-Critic offloading approach to learn the optimal offloading policy in a space-air-ground integrated network (SAGIN), where flying unmanned aerial vehicles (UAVs) and satellites provide access to edge computing and cloud computing respectively.

Feng et al. [77] propose a computation offloading and resource allocation algorithm based on Asynchronous advantage actor–critic (A3C) in a blockchain-enabled MEC environment, where multiple actor-learners running in parallel makes the exploration process more diverse. The blockchain technology ensures the reliability and irreversibility of data in MEC systems.

A decentralized dynamic computing offloading strategy proposed in [78], which minimizes the long-term average computing cost in a multi-user MEC system. Each user only can obtain state information about its local observation, and make actions independently from other users. The local observation of state includes the queue length of data buffer, the feedback from BS conveying the last receiving signal-to-interference-plus-noise (SINR), and channel vector. The framework is constructed with Deep Deterministic Policy Gradient algorithm (DDPG), which is used to allocate the power of local execution and offloading, and control the CPU speed due to DVFS techniques. Similarly, Ke et al. [79] apply DDPG in a heterogeneous vehicular network where vehicles and roadside units can offload computation tasks to MEC server. The Ornstein-Uhlenbeck (OU) noise vector is added to the action space for more effective stochastic exploration.

However, the DDPG-based method has the instability and the slow convergence issue in the offloading process because the actor network is highly correlated with the critic network. To improve the DDPG algorithm, a algorithm named the Double-Dueling-Deterministic Policy Gradients (D^3PG) is proposed in [80], where a QoE model considering service delay, energy consumption, and task success rate is proposed for edge-enabled IoT. For the critic entity, they combine Dueling network with Double DQN to modify the manner of estimating Q-value. The simulation results shows that the proposed algorithm achieve a better QoE than other DRL algorithms.

2.6 Summary

This chapter provided the background of Mobile Edge Computing and a brief introduction of deep reinforcement learning (DRL). We first illustrated the development of MEC and discussed the main issues of computation offloading. Then we introduced the common used models in deep learning field, which includes MLP and RNN. Next, following the basic idea of Markov decision process (MDP), two categories of methods are introduced, i.e., the model-based methods and model-free ones. For the class of model-based methods, it consists of value iteration and policy iteration. For the model-free methods, several well-known algorithms are introduced, including DQN, double DQN, actor-critic, and DDPG. Finally, based on the techniques used, the existing approaches for computation offloading are summarized. Considering the advantages and disadvantages of previous works, we will design a novel computation offloading strategy for a single MIoT to reduce the long-term cost in the following chapters.

Chapter 3

Proposed System Model

In this chapter, section 3.1 presents the overall system model. Section 3.2 formulates the problem as a Constrained Markov Decision Process (CMDP).

3.1 System Model

In this section, we present the system model. MEC Network and Task Model describe the details of the MEC environment and the features of tasks of MIoT devices. Communication Model, Computation Model and Monetary Cost Model show how delay, energy consumption and monetary cost are calculated.

3.1.1 MEC Network and Task Model

As shown in Figure 3.1, we consider an MIoT unit (MU) m that is roaming within a specific area (e.g., a city block) to gather and analyze data as part of a smart neighbourhood application (e.g., air pollution monitoring, gas and electricity maintenance and metering or for road safety). As the MU moves, it collects and processes the required data for the application (e.g., taken pictures for object recognition, environment sensing data or meter readings). We model each data processing task to be executed by the MU at time t_k by $U_k = (u_k, v_k)$, where u_k is the size of the data to be processed and v_k is the required cpu

cycles to complete the execution of U_k. The data is collected and processed at random arrival times, $t_0, t_1, \cdots$, that follow a Poisson distribution. In a given time period T (e.g., an hour or a day), the MU is expected to process an average of K tasks arriving at different time epochs $t_0, t_1, \cdots, t_{K-1} < T$. We assume that U_k is known only at t_k but may not change during offloading or execution. All tasks are critical and must be processed within an acceptable time.

Each task U_k can be executed locally or offloaded to one of I edge servers $\{e_1, \ldots, e_I\}$ via a cellular network where each edge server is allocated adjacent to a base station (BS). The task can also be offloaded to a cloud data-center c through a cheaper WiFi connection to access points (AP) connected to the cloud via a backbone. We use $\mathcal{L}=\{m, c\} \cup \{e_1, \cdots, e_I\}$ to denote the set of all possible execution locations, where m refers to the MU itself. To guarantee the efficient operation of the MU, the IoT application provider assigns a specialized, service optimized, dedicated or shared virtual machine (VM) on each edge server and on the cloud to processes the MU task. Denote by $q_0^{e_i}$, $e_i \in \{e_1, \ldots, e_I\}$ and q_0^c, the number of logical cpus dedicated to the MU on the VM, respectively, on e_i and on c. For simplicity of illustration, we also use q_0^m to denote the number of logical cpus on MU that are dedicated to data processing. We also use f^m, f^{e_i} and f^c to refer to the cpu processing frequency (i.e., cpu cycles per second), respectively, for m, the VM on e_i and on c. The MU can process several tasks concurrently, hence, portions of the local and VM resources become occupied and freed continuously as new tasks are executed and older ones are terminated.

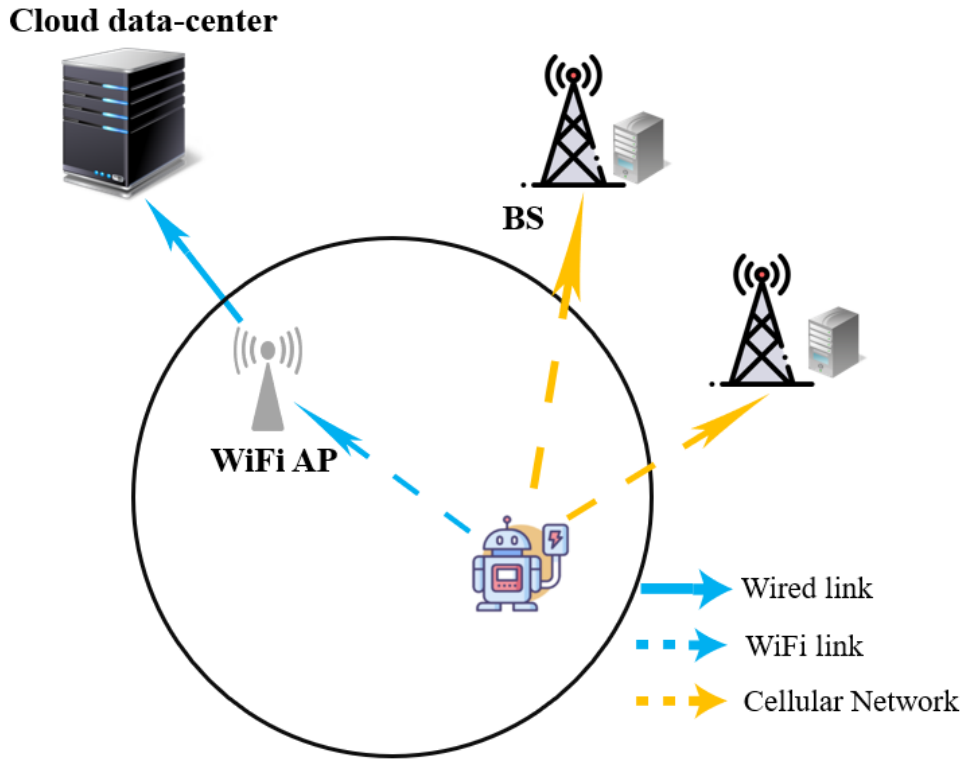

Figure 3.1: System Model

Let $q_k^\ell \leq q_0^\ell$, $\ell \in \mathcal{L}$, be the number of free cpus on location ℓ at each time t_k, when task U_k is to be executed. At each t_k, the MU must make a parameterized offloading decision (action) which is represented by a tuple $a_k = (\ell_k, x_k)$. Here, $\ell_k \in \mathcal{L}$ represents the decided execution location of U_k. The second parameter in the decision x_k represents the number of logical processors assigned on the MU or to a container hosted on the assigned edge or cloud VM. For $\ell_k \in \{m\} \cup \{e_1, e_2, \cdots, e_I\}$, x_k cannot exceed the available resources $q_k^{\ell_k}$. However, since tasks must be executed immediately as they arrive, they can always be offloaded to c. However, if the number of assigned and free logical cpus is not sufficient, i.e., $x_k > q_k^c$, the MU can offload U_k to c but the application provider must pay for a new VM with x_k cpus.

The MU use a rechargeable battery and we measure its current status using its normalized state of charge (SoC) b_k at t_k [81]. More precisely, $b_k \in [0, 1]$ represents ratio of current battery capacity to its nominal capacity B_n. The nominal capacity is given by the manufacturer and represents the maximum amount of charge that can be stored in the

battery. Our main objective is to train the MU to make offloading decisions, a_k, at each time epoch t_k, with the objective of minimizing a weighted sum of the overall delay, energy consumption and monetary costs at t_k as well as for all future tasks $k+1, \cdots, K-1$. The MU must operate while not exceeding the computational resources available locally or on the edge servers. Offloading decisions must also satisfy the condition that the battery state of charge b_k at any time t_k may not decrease below a minimum threshold $\underline{b}$ that is required for the MU to reach its battery recharging station. Finally, we define s_k $=(U_k, b_k, q_k^m, q^{e_1}, \cdots, q_k^{e_I}, q_k^c)$ to represent the arriving task and the state of available local and remote resources (battery and cpu) at t_k. In the following, we first derive the offloading costs of a task k in terms of the transmission and execution delays, consumed energy and monetary cost.

3.1.2 Communication Model

Let $\delta_{(tr)k}^{\ell}$ and $\xi_{(tr)k}^{\ell}$ be the transmission delay and consumed energy, respectively, when task k is executed at location $\ell \in \mathcal{L}$. The offloading transmission rate of a device depends on the wireless channel condition and the allocated spectrum. We assume that the channels' condition remain unchanged during each offloading period. The achievable transmission rates, $r_k^{e_i}$ and r_k^c, between the MU and e_i and c, respectively, at time t_k can be calculated as follows [82].

$$r_k^{e_i} = B_1 \log_2(1 + \frac{P(b_k) \times g_1(d_i)}{\sigma_1^2}) \tag{3.1}$$

$$r_k^c = B_2 \log_2(1 + \frac{P(b_k) \times g_2}{\sigma_2^2}) \tag{3.2}$$

Where B_1 and B_2 represent the bandwidth of the BS connected to e_i and that of the nearest AP, respectively. Similarly, $g_1(d_i)$ and g_2 denote the channel gain between the MU and, respectively, the BS for e_i at distance d_i from the MU and the nearest AP. The noise at the BS and the AP, respectively, are denoted by σ_1^2 and σ_2^2. $P(b_k)$ is the allowable transmission power of the MU, which depends on the remaining SoC, b_k. Following [83], we set $P(b_k) = b_k \times P_{max}$, where P_{max} is the maximum transmit power when the battery

is full, i.e., when $b_k = 1$.

For a typical terrestrial wireless channel, the channel gain normally include the following three effects. The first effect is the path loss, which is the power loss in the signal owing to the distance it travels. The second effect is shadowing duo to attenuation or blocking of the signal by large objects like buildings. The last effect is small-scale fading. This is because constructive or destructive combining of signals arrive at the receiver at the similar times, but taking different paths from the transmitter, and hence undergoing different attenuations and phase shifts.

Let $\delta_{(tr)}(s_k, a_k)$ be the transmission delay resulting from a decision $a_k = (\ell_k, x_k)$ on U_k, at s_k then we have

$$\delta_{(tr)}(s_k, a_k) = \begin{cases} \frac{u_k}{r_k^{e_i}}, & \ell_k = e_i, \quad i = 1, \cdots, I \\ \frac{u_k}{r_k^c} + \frac{u_k}{R}, & \ell_k = c \\ 0 & \ell_k = m \end{cases} \tag{3.3}$$

The above formulation states that when U_k is transmitted to e_i, it incurs only the uplink transmission delay to transmit u_k. On the other hand, if U_k is offloaded to c, the device first transmits the data to a nearby AP which then forwards it to the cloud data center via a backbone network, where R denotes the wired transmission rate between the AP and the cloud data center. And finally, no transmission delay is experienced for a local execution. In a similar manner, let $\xi_{(tr)}(s_k, a_k)$ be the consumed energy for data transmission, then

$$\xi_{(tr)}(s_k, a_k) = \begin{cases} P(b_k) \times \delta_{(tr)}(s_k, a_k), & \ell_k = e_i \\ P(b_k) \times \frac{u_k}{r_k^c}, & \ell_k = c \\ 0 & \ell_k = m \end{cases} \tag{3.4}$$

Furthermore, we do not account for the cost of the energy consumed during the communication between the WiFi AP and the cloud data center since our focus is to minimize the cost incurred by the device.

3.1.3 Computation Model

To calculate the task computational cost, let f^m, f^{e_i} and f^c, be the processing rate, e.g., CPU cycles per second for one logical processor, respectively for the device, a VM hosted on the edge e_i and on c. Then the processing delay for an offloading decision $a_k = (\ell_k, x_k)$ with U_k requiring v_k cpu cycles is

$$\delta_{(prs)}(s_k, a_k) = \frac{v_k}{x_k \times f^{\ell_k}}, \quad \ell_k \in \mathcal{L} \tag{3.5}$$

The processing energy $\xi_{(prs)}(s_k, a_k)$ is also given by:

$$\xi_{(prs)}(s_k, a_k) = \begin{cases} P_{loc} \times \delta_{(prs)}(s_k, a_k), & \ell_k = m \\ 0, & \ell_k \neq m \end{cases} \tag{3.6}$$

where P_{loc} denotes computational power of the MU.

3.1.4 Monetary Cost Model

As indicated before, if a task is offloaded to c but there were no available cpus on the cloud VM, an additional monetary cost is paid by the service provider to allocate the needed resources.

Different cloud providers may offer different types of pay-per-use metered method. For example, cloud providers can charge users based on the number of CPU cycles actually consumed by the offloaded tasks. Another example is that the cloud provider can charge users based on the number and the type of VMs their program use and how long they use them. In our model, we use the second model. Therefore, the monetary cost can be calculated by:

$$\zeta(s_k, a_k) = \begin{cases} \epsilon_c \times x_k \times \delta_{(prs)}(s_k, a_k), & \ell_k = c \text{ and } x_k > q_k^c \\ 0, & \text{otherwise} \end{cases} \tag{3.7}$$

where ϵ_c denotes the base price charged for an extra unit of cpu on the cloud (i.e., the monetary cost per computing resource per second).

3.1.5 The Weighted Cost Function

Then, similar to [84, 85], the total cost, $g(s_k, a_k)$, of running U_k following an offloading decision $a_k = (\ell_k, x_k)$ given the resource availability state s_k can be calculated as a weighted sum of the consumed energy, delay and monetary cost.

$$\begin{aligned} g(s_k, a_k) = \alpha_1 \times W(\delta_{(tr)}(s_k, a_k) + \delta_{(prs)}(s_k, a_k)) \\ + \alpha_2 \times (\xi_{(tr)}(s_k, a_k) + \xi_{(prs)}(s_k, a_k)) \\ + \alpha_3 \times \zeta(s_k, a_k) \end{aligned} \tag{3.8}$$

where $\alpha_1+\alpha_2+\alpha_3 = 1$, are constant coefficients that can be used by the application provider to achieve the desired trade off for the delay, energy and monetary cost in $g(s_k, a_k)$. Here, $W(\cdot)$ denotes a general delay cost function which is defined to be increasing and convex with respect to total delay. $W(\cdot)$ well depicts the intuition in practice that task execution is commonly less sensitive to a small delay, but the delay cost grows significantly when the delay continuously increases and becomes very large. The physical meaning of $W(\cdot)$ can be interpreted as the cost resulted by the service dissatisfaction [86].

It is worth noting here that since, in general, the data resulting from the task execution is much smaller than the task uploaded data itself and to simplify the illustration of our scheme we do not consider the energy consumption and delay when the edge/cloud transmit the results back to the MU. However, including this cost in the proposed model is a straightforward extension.

3.2 Proposed CMDP Model

In this section, we formulate our finite horizon multi-epoch offloading sequential decision problem as a finite horizon Constrained Markov Decision Process (CMDP) in which state

transitions occur in irregular times. The fundamental elements for our CMDP can be defined as: $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \mathcal{C} \rangle$. $\mathcal{S}$ is the set of the environment states and $\mathcal{A}$ is the set of permissible actions or decisions. $\mathcal{R}$ and $\mathcal{C}$ represents the reward function and cost function that measures the obtained reward and incurred cost when an action is selected given a certain state. The transition probability between states is represented by $\mathcal{P}$. In the following, we define each of these elements.

3.2.1 States and Actions

The set of states $\mathcal{S}$ in our model represents the set of all possible $s_k \in \mathcal{S}, \forall k \in \{0, 1, ..., K\}$. With a slight abuse of notation, we redefine $s_k = (s'_k, s''_k)$ to describe the set of occupied as well as free resources as follows. Here $s'_k = (U_k, b_k, q_k^m, q_k^{e_1}, \cdots, q_k^{e_I}, q_k^c)$ is used to reflect the battery status along with the number of available cpus on the MU, the edge e_i and on c. Also, we use $s''_k = (\bar{t}_k^1, \bar{\ell}_k^1, \bar{x}_k^1, \cdots, \bar{t}_k^N, \bar{\ell}_k^N, \bar{x}_k^N)$ to describe the cpu occupancy state by all running tasks at time t_k. More precisely, let $N < K$ be the maximum number of tasks that is expected to be executed concurrently at any given time, then each triplet $(\bar{t}_k^n, \bar{\ell}_k^n, \bar{x}_k^n), n \in \{1, \cdots, N\})$ denotes the remaining execution time, the execution location, and the occupied resources of n-th executing task. These running tasks are ordered as per their earliest remaining running times $\bar{t}_k^n$. Initially, at t_0, s''_k is empty and a new triplet is added to it whenever a decision a_k is made to execute a task U_k as will be shown next.

The set of allowable actions at each state s_k, $\mathcal{A}_k$, can also be defined as,

$$\begin{aligned} \mathcal{A}_k = \{a_k = (\ell_k, x_k) : & \ell_k \in \mathcal{L}, x_k \in \{1, \cdots, q_k^{\ell_k}\} \\ & \vee (\ell_k = c, x_k \in \mathbb{R} \text{ if } q_k^c < x_k)\} \end{aligned} \tag{3.9}$$

3.2.2 State Transition

The state s_k is updated right after an action a_k is made and whenever a running task terminates before t_{k+1}. These events move s_k into post-decision states until a new task arrives at a random time t_{k+1}. We note that when the execution of task U_k starts at t_k and x_k resources are allocated at ℓ_k and the availability part of state s_k moves into a

post-decision state $\tilde{s}'_k = (U_k, \tilde{b}_k, \tilde{q}_k^m, \tilde{q}_k^{e_1}, \cdots, \tilde{q}_k^{e_I}, \tilde{q}_k^c,)$, which is calculated as follows:

$$\tilde{q}_k^\ell = \begin{cases} q_k^\ell & \ell \neq \ell_k, \\ q_k^\ell - x_k & \ell = \ell_k, x_k \leq q_k^\ell \\ q_k^\ell & \ell = \ell_k = c, x_k > q_k^\ell, \end{cases} \tag{3.10}$$

The transition of the remaining SoC is denoted by:

$$\tilde{b}_k = b_k - b(s_k, a_k) \tag{3.11}$$

where $b(s_k, a_k)$ is the consumed SoC due to executing U_k and is calculated using the coulomb counting method [81]. Combined 3.4) and (3.6), we have,

$$b(s_k, a_k) = \begin{cases} \frac{\xi_{(tr)k}^{\ell_k}}{V_{tr} B_n}, & \ell_k \neq m \\ \frac{\xi_{(prs)k}^{\ell_k}}{V_{loc} B_n}, & \ell_k = m \end{cases} \tag{3.12}$$

where V_{tr} and V_{loc} denote the transmission voltage and local processing voltage respectively, and B_n is the nominal capacity of the MU battery.

When a new task arrives at t_{k+1}, we reach a new state s_{k+1} with the new U_{k+1}, the resources from the post-decision state, but only updates $\bar{t}_{k+1}^n$ to reflect the remaining times for the running tasks and q_{k+1}^ℓ to reflect any resources freed by tasks terminating in $[t_k, t_{k+1}]$:

$$\bar{t}_{k+1}^n = \begin{cases} \bar{t}_k^n - (t_{k+1} - t_k) & \bar{t}_k^n > t_{k+1} - t_k, \\ 0 & \bar{t}_k^n \leq t_{k+1} - t_k \end{cases} \tag{3.13}$$

For the resource availability part of next state s_{k+1}, we have:

$$q_{k+1}^\ell = \tilde{q}_k^\ell + \sum_j \bar{x}_k^j | \bar{\ell}_k^j = \ell, \bar{t}_k^j < t_{k+1} - t_k, \quad \ell \in \mathcal{L}, \tag{3.14}$$

3.2.3 Reward Function and Resource Constraints

Our objective is to minimize a weighted sum of delay, energy consumption and monetary cost, so the reward function should be negatively correlated with this weighted cost, which is determined as:

$$r(s_k, a_k) = \frac{\phi_1 - g(s_k, a_k)}{\phi_2} \tag{3.15}$$

Where ϕ_1 and ϕ_2 are constants. The cost function $\mathcal{C}$ in our CMDP is given by the consumed battery charge (3.12).

We aim to maximize the total reward while satisfying the constraints on the battery. Our main objective is that at each time t_k, we can select a parameterized action $a_k = (\ell_k, x_k)$, that selects the offloading location ℓ_k and assigned resources x_k, given the resource availability described by s_k. Therefore, we have the following CMDP formulation for the offloading problem,

$$\max_{a_0,\ldots,a_{K-1}} \mathbb{E}\left(\sum_{k=0}^{K-1} r(s_k, a_k)|s_0\right) \tag{3.16}$$

$$\text{s.t.} \quad a_k \in \mathcal{A}_k, \forall k = 0, 1, \cdots, K-1 \tag{3.17}$$

$$\mathbb{E}\left[\sum_{k=0}^{K-1} b(s_k, a_k)\right] \leq (1 - \underline{b}), \tag{3.18}$$

where $\mathbb{E}$ means taking expectation over all possible state trajectories starting from s_0. The first constraint (3.17) ensures that the action a_k taken at each state allocate only available resources to each task. On the other hand, the second constraint (3.18) ensure that executing all K tasks may not reduce the battery state of charge (SoC) to a value below the threshold $\underline{b}$, which is the amount of charge that is sufficient to bring the IoT device to a recharging station.

Clearly, an action a_k selected at time t_k will affect the remaining battery lifetime for the MU and the computational resource availability. In other words, any decision at time t_k must take into consideration all future decisions in order to be optimal.

3.3 Summary

In this chapter, a cloud-assisted MEC offloading infrastructure was proposed, where a battery constrained MU executes a number of data processing tasks whose arrival times and sizes are stochastic. The arrival tasks can be executed locally, or offloaded to one of edge servers and a cloud data center. In contrast to existing models, we consider that the transmission power can change as batteries deplete. Then, taking account of the battery constraint and the limited computation resources, we formalized an optimization problem of minimizing the weighted sum of delay, energy consumption and monetary cost of the entire system as a CMDP.

Chapter 4

Proposed Offloading Policy

As shown in the last chapter, because the remaining running time $\bar{t}_k^n$ in s_k'' is continuous, it is difficult to enumerate all possible state transitions. Therefore, to solve the above constrained optimization problem (3.16) without the need to explicitly obtain the state transition probabilities, we develop a constrained deep reinforcement learning (CDRL) algorithm to find the solution. We first relax the battery constraint using Lagrangian transformation and then design a model-free Advantage Actor-Critic (A2C) learning module that is executed on the edge to learn an optimal offloading policy by repeated interactions with the MEC environment.

4.1 Lagrangian Transformation

We first relax the constrained problem (3.16) and define a stationary policy π which corresponds to a mapping from states to a probability distribution over actions, i.e., $\pi(a|s) = P(a_k = a|s_k = s)$ is the probability of selecting action a in state s. Then, problem (3.16) can be transformed to find the policy π,

$$\max_{\pi} \mathbb{E}_{\pi}\left(\sum_{k=0}^{K-1} r(s_k, a_k)|s_0\right) \tag{4.1}$$

where $\mathbb{E}_{\pi}$ represents taking expectation according to the law induced by the policy π.

For a given initial state, we convert the primary problem into a Lagrangian dual problem by adding the battery constraint term to the original objective function, and the corresponding Lagrangian can be defined as:

$$\begin{aligned}\mathcal{L}(\pi,\eta) =& \mathbb{E}_\pi\left[\sum_{k=0}^{K-1} r(s_k,a_k)\right] - \eta\cdot\left(\mathbb{E}_\pi\left[\sum_{k=0}^{K-1} b(s_k,a_k)\right] - (1-\underline{b})\right) \\ =& \mathbb{E}_\pi\left[\sum_{k=0}^{K-1} r^\eta(s_k,a_k)\right] + \eta(1-\underline{b})\end{aligned} \tag{4.2}$$

where $\eta \geq 0$ is the Lagrangian multiplier. $\mathcal{L}(\pi,\eta)$ can be regarded as the total reward of the policy π with the constrained reward function $r^\eta(s_k,a_k)$ plus $\eta(1-\underline{b})$. For a given Lagrangian multiplier η, $\eta(1-\underline{b})$ is a constant and policy π has no effect on it.

Further notice that, every fixed Lagrange multiplier η induces an unconstrained MDP with the corresponding reward function for each decision epoch given by $r^\eta(s_k,a_k)$, thereafter referred to as r_k^η.

$$r^\eta(s_k,a_k) = r(s_k,a_k) - \eta b(s_k,a_k) \tag{4.3}$$

From the above constrained reward function, we can observe that r_k^η not only depends on the original reward function $r(s_k,a_k)$, but also on the battery consumption $b(s_k,a_k)$ and the Lagrange multiplier η. This shows that the battery consumption is also a factor to affect the action, which is something that the original reward function did not take into account. The value of Lagrange multiplier η reflects the influence of battery consumption on the decision of actions.

4.1.1 Primary Problem

We define a function with respect to π,

$$\theta_P(\pi) = \min_{\eta\geq 0} \mathcal{L}(\pi,\eta) \tag{4.4}$$

Next this function is analyzed below in terms of whether π satisfies the constraint or not.

- We first consider that if π cannot satisfy the battery constraint,

i.e., $\mathbb{E}_\pi\left[\sum_{k=0}^{K-1} b(s_k, a_k)\right] - (1-\underline{b}) \geq 0$, then

$$\begin{aligned}\theta_P(\pi) =& \min_{\eta\geq 0} \mathbb{E}_\pi\left[\sum_{k=0}^{K-1} r(s_k, a_k)\right] - \eta \cdot \left(\mathbb{E}_\pi\left[\sum_{k=0}^{K-1} b(s_k, a_k)\right] - (1-\underline{b})\right) \\ =& -\infty\end{aligned} \tag{4.5}$$

It can be easily observed that if $\mathbb{E}_\pi\left[\sum_{k=0}^{K-1} b(s_k, a_k)\right] - (1-\underline{b}) \geq 0$, making $\eta \to +\infty$ can minimize θ_P. Thus, when the constraint are not satisfied, θ_P tends to be infinitesimal.

- Then we consider if π satisfy the constraint, i.e., $\mathbb{E}_\pi\left[\sum_{k=0}^{K-1} b(s_k, a_k)\right] - (1-\underline{b}) \leq 0$, then

$$\begin{aligned}\theta_P(\pi) =& \min_{\eta\geq 0} \mathbb{E}_\pi\left[\sum_{k=0}^{K-1} r(s_k, a_k)\right] - \eta \cdot \left(\mathbb{E}_\pi\left[\sum_{k=0}^{K-1} b(s_k, a_k)\right] - (1-\underline{b})\right) \\ =& \min_{\eta\geq 0} \mathbb{E}_\pi\left[\sum_{k=0}^{K-1} r(s_k, a_k)\right] \\ =& \mathbb{E}_\pi\left[\sum_{k=0}^{K-1} r(s_k, a_k)\right]\end{aligned} \tag{4.6}$$

In this case, due to $\mathbb{E}_\pi\left[\sum_{k=0}^{K-1} b(s_k, a_k)\right] - (1-\underline{b}) \leq 0$, the value of η have to be 0 in the process of minimizing θ_P.

By analysing the above two cases, we can get:

$$\theta_P(x) = \begin{cases} \mathbb{E}_\pi\left[\sum_{k=0}^{K-1} r(s_k, a_k)\right], & \pi \text{ satisfy the constraint} \\ -\infty, & \text{otherwise} \end{cases} \tag{4.7}$$

Therefore, in the situation that the battery constraint is satisfied, we have

$$\max_\pi \theta_P(\pi) = \max_\pi \min_{\eta\geq 0} \mathcal{L}(\pi, \eta) = \max_\pi \mathbb{E}_\pi\left[\sum_{k=0}^{K-1} r(s_k, a_k)\right] \tag{4.8}$$

The above equation shows that $\max_{\pi} \theta_P(\pi)$ is equivalent with the original optimization problem (3.16). We use $\max_{\pi} \theta_P(\pi)$ to represent the primary problem.

4.1.2 Dual Problem

Now we define the dual problem of the primary problem, and we define a function with respect to η:

$$\theta_D(\eta) = \max_{\pi} \mathcal{L}(\pi, \eta) \tag{4.9}$$

Then we consider the minimization of $\theta_D(\eta)$. Therefore, the dual problem of primary problem is as follows:

$$\min_{\eta \geq 0} \theta_D(\eta) = \min_{\eta \geq 0} \max_{\pi} \mathcal{L}(\pi, \eta) \tag{4.10}$$

Because strong duality is known to hold for CMDP [49] and we invoke the minimax theorem,

$$\max_{\pi} \min_{\eta \geq 0} \mathcal{L}(\eta, \pi) = \min_{\eta \geq 0} \max_{\pi} \mathcal{L}(\eta, \pi) \tag{4.11}$$

Therefore, we can solve the primary problem through the dual problem. The constrained problem (3.16) can be converted to the following unconstrained problem:

$$\min_{\eta \geq 0} \max_{\pi} \mathcal{L}(\pi, \eta) \tag{4.12}$$

Here we refer π as the primary variable and η as the dual variable. To solve the above problem, a canonical approach is to use the iterative primal-dual method where in each iteration we update the primal policy π and the dual variable η in turn until reach a saddle point (η^*, π^*). η^* denotes the optimal Lagrangian multiplier and π^* denotes the optimal policy.

4.2 Primal-Dual Optimization

Our CMDP problem can be solved using two-time scales; gradient-ascent is applied on a faster time scale to find the optimal policy for a given Lagrangian multiplier, and dual variable update is performed on a slower time scale [49]. Therefore, We can solve the CMDP problem by solving the following two sub-problems.

- **the inner maximization problem:** With fixed η, the minimization problem can be viewed as a non-constrained MDP problem with instant reward $r^{\eta}(s_k, a_k)$. Let π_θ be a parameterized policy where the choice of actions is dependent on the parameter θ, and define $J(\pi_\theta) = \mathcal{L}(\pi_\theta, \eta = \eta^{(n)})$, where $\eta^{(n)}$ denotes the Lagrange multiplier in n-th iteration.

  The optimality equation of this maximization problem can be given below,

  $$V^{\eta}(s_0) = \max_{\pi_\theta} \mathbb{E}_{j \in \mathcal{S}} \Big[r^{\eta}(s_0, a_0) + V^{\eta}(j) \Big] \tag{4.13}$$

  where $\mathbb{E}_{j \in \mathcal{S}}$ denotes taking expectation over all possible next state j. $V^{\eta}(s_0)$ is the value function starting from the initial state s_0.

  More generally, it should solve the following optimistic problem:

  $$\pi^*(s) = \arg\max_{a} \mathbb{E}_{s' \in \mathcal{S}} \Big[r^{\eta}(s, a) + V^{\eta}(s') \Big] \tag{4.14}$$

  where $\pi^*(s)$ denotes the optimal action at state s.

  The advantage actor-critic (A2C) scheme can be adopted to find the optimal offloading policy with fixed Lagrange multiplier, where the actor searches for a local maximum in $J(\pi_\theta)$ within a set Π_θ of parameterized policies π_θ by performing policy gradient ascent with respect to the parameters θ. We will introduce details of the specific DRL algorithm in the next subsection.

- **the outer minimization problem:** The outer minimization over the multiplier η is a linear programming problem and we can update the Lagrange multiplier using

sub-gradient method. The updating process is given as follows,

$$\eta^{(n+1)} = \max\{\eta^{(n)} + \beta_n(B - (1 - \underline{b})), 0\} \tag{4.15}$$

where n is the iteration number and $B = \sum_{k=0}^{K-1} b(s_k, a_k)$. β_n is the updating rate which decrease with the increase of the iteration number. The Lagrange multiplier will converge to the optimal value through the sub-gradient algorithm.

4.3 Advantage Actor-Critic Algorithm

Because of the use of simple function approximator, it is difficult for traditional reinforcement learning to deal with large scale practical problems. To overcome this challenge, DRL is developed successfully, which uses DNN to be the function approximator.

In this thesis, the advantage actor-critic (A2C) algorithm, one of the novel DRL method, is applied to update the primal policy. The A2C method is composed of two components, i.e., the actor network and the critic network, which combines the benefit of value-based methods and policy-based methods. In the following, we describe the details of the implementation of A2C algorithm.

4.3.1 Actor Network

The actor network is responsible for outputting appropriate actions according to states. Generally speaking, there are two kind of approaches to implement an actor using deep neural network (DNN). The first one is to input all combinations of all actions and the same state in turns to the DNN and output values of these combinations, then to select action according to these values. Another approach is just to input the state to the DNN, and the DNN output the probabilities of executing each action, and select action according to the probabilities. The former approach requires several end-to-end network operations according to the size of the action space, and the latter one only needs to calculate once to get the execution probability of all actions in the action space. Therefore, in this thesis,

we will use the latter approach to implement the actor network. The dimensionality of input layer is same as the dimensionality of the state space. The dimensionality of output layer is same as the dimensionality of the action space.

When we initialize an actor function, we use a set of random parameters θ. At the beginning of the training, it is equivalent to a trial and error process that takes random action, and we obtain the data through the iterative interaction with the environment, and then updating the parameters according to the policy-gradient algorithm. For our offloading problem, according to the policy gradient theorem introduced before (2.33), the policy gradient can be written as:

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\pi_\theta}\Big[\nabla_\theta \log \pi_\theta(a|s) Q^\eta(s,a)\Big] \tag{4.16}$$

where $\pi_\theta(a|s)$ denotes the probability of selecting action a in state s according to parameter θ, and $Q^\eta(s,a)$ is the Q-value obtained by taking the action a under state s via policy π_θ. Equation (4.16) is the gradient used by the policy-gradient algorithm to update parameters. For advantage actor-critic algorithm, we need to substitute $Q^\eta(s,a)$ with the advantage function when updating the actor function, the advantage function will be introduced in the next subsection.

4.3.2 Critic Network

Based on Eq.(4.16), like we introduced in Algorithm 4, Monte Carlo (MC) learning uses accumulated reward starting from the current time step as an unbiased sample $Q^\eta(s,a)$ to update parameter θ. Although MC learning is unbiased, the noise is relatively large, which means that variance of MC is high. If the value of state can be estimated relatively accurately and be used to guide the update of policy, better learning results can be achieved, which is also the basic idea of actor-critic algorithm. By adding critic into this framework, variance can be effectively reduced.

In order to better deal with the problem of high variance, the actor-critic algorithm can be improved by using a baseline function. The basic idea is to subtract a baseline function

$B(s)$ from the policy gradient. $B(s)$ is only related to state but has nothing to do with action, so it does not change the gradient. The characteristic of $B(s)$ is reducing variance while not changing the expectation of action values, which is also the main idea of the use of advantage function.

In principle, any functions that are not related to action can be used as $B(s)$. But the best choice of baseline function $B(s)$ is the state value function, i.e., $B(s) = V^{\eta}(s)$. Therefore, the definition of the advantage function for our offloading problem is given as follows:

$$A^{\eta}(s,a) = Q^{\eta}(s,a) - V^{\eta}(s) \tag{4.17}$$

The advantage function can estimates how much better the chosen action a for a given state s is compared to the average action at that state. If the advantage function is positive, it suggests that the tendency to select action a should be strengthened for the future, whereas if the advantage function is negative, it suggests the tendency should be weakened. In the following, we will show how do we estimate the advantage function.

According to [87], we introduce a parameter γ to reduce variance further in obtaining the advantage function and we define $\psi_k = r_k^{\eta} + \gamma V^{\eta}(s_{k+1}) - V^{\eta}(s_k)$, i.e., the Temporal difference (TD) residual with discount γ, where $V^{\eta}(s_k)$ is the value function of state s_k. Let us consider taking the sum of y of these ψ terms, which we will denote by $A_k^{(y)}$,

$$\begin{aligned}
A_k^{(1)} &:= \psi_k & &= -V^{\eta}(s_k) + r_k^{\eta} + \gamma V^{\eta}(s_{k+1}) \\
A_k^{(2)} &:= \psi_k + \gamma\psi_{k+1} & &= -V^{\eta}(s_k) + r_k^{\eta} + \gamma r_{k+1}^{\eta} + \gamma^2 V^{\eta}(s_{k+2}) \\
A_k^{(3)} &:= \psi_k + \gamma\psi_{k+1} + \gamma^2\psi_{k+2} & &= -V^{\eta}(s_k) + r_k^{\eta} + \gamma r_{k+1}^{\eta} + \gamma^2 r_{k+2}^{\eta} + \gamma^3 V^{\eta}(s_{k+3})
\end{aligned} \tag{4.18}$$

$$A_k^{(y)} := \sum_{l=0}^{y-1} \gamma^l \psi_{k+l} = -V^{\eta}(s_k) + r_k^{\eta} + \cdots + \gamma^{y-1} r_{k+y-1}^{\eta} + \gamma^y V^{\eta}(s_{k+y}) \tag{4.19}$$

where the value of y ranges from 0 to K, and K is the maximum number of time steps before updating.

We give two examples to illustrate the implementation of the above function. When $k = K - 1$, i.e., the time point of the last action, the advantage function can be denoted

by:

$$A_k^{(y)} = r_{K-1}^{\eta} + \gamma V^{\eta}(s_K) - V^{\eta}(s_{K-1}) \tag{4.20}$$

Here the value of y is 1.

When $k = 0$, i.e., the time point of first action, the advantage function can be denoted by:

$$A_k^{(y)} = \sum_{i=0}^{K-1} \gamma^i r_i^{\eta} + \gamma^K V^{\eta}(s_K) - V^{\eta}(s_0) \tag{4.21}$$

In the situation, the value of y is K. Therefore, we can notice that in equation (4.19), the value of y is dependent on the value of k, and the two variables satisfy $k + y = K$.

We use another DNN (critic network) with parameter ω to estimate the state value function, i.e., $V^{\eta}(s_k) \approx V_{\omega}^{\eta}(s_k)$. Therefore, the estimated advantage function $\hat{A}_k^{(y)}$ can be given as follows:

$$\hat{A}_k^{(y)} := \sum_{l=0}^{y-1} \gamma^l \psi_{k+l} = -V_{\omega}^{\eta}(s_k) + r_k^{\eta} + \cdots + \gamma^{y-1} r_{k+y-1}^{\eta} + \gamma^y V_{\omega}^{\eta}(s_{k+y}) \tag{4.22}$$

We can consider $\hat{A}_k^{(y)}$ to be an estimator of the advantage function. As a result, the estimated policy gradient is given by:

$$\nabla_{\theta} J(\pi_{\theta}) = \mathbb{E}_{\pi_{\theta}} \left[\nabla_{\theta} \log \pi_{\theta}(a|s) \hat{A}_k^{(y)} \right] \tag{4.23}$$

Therefore, the update of the policy π_{θ} with respect to the parameters θ can be given:

$$\theta^{new} = \theta^{old} + \alpha \nabla_{\theta} J(\pi_{\theta}) \tag{4.24}$$

where $\alpha \geq 0$ is the learning rate for the policy update.

We use the advantage function as the prediction error and use the squared error function as the loss function to update the parameters of the critic network. Loss function is given as follows:

$$L(\omega) = [\hat{A}_k^{(y)}]^2 \tag{4.25}$$

The loss function is a function that maps the network evaluation to a real number, representing the squared difference between the actual value function and the expected value function. The optimization of the network is the process to minimize the loss value by tuning parameters ω. Backpropagation is used to modify the neural network through a method called chain rule, which adjusts each weight in the network in proportion to how much it contributes to overall error.

Now we propose the optimal stationary offloading and resource allocation algorithm in Algorithm 5 combining the DRL process and the sub-gradient process.

From the perspective of the policy updated, the Lagrange multiplier is treated as a constant. Therefore, (4.13) can be regarded as the classic deep reinforcement learning for the unconstrained MDP induced by a fixed Lagrange multiplier, with the corresponding reward function given by (4.3). From the perspective of the Lagrange multiplier, the optimal policy of the unconstrained MDP can converge quickly. By comparing the difference between the battery consumption of the current policy and the desired battery consumption, the Lagrange multiplier can be adjusted gradually.

In general, when the state transition probability information is perfect, the CMDP can be solved by model-based method, i.e., policy iteration and value iteration, which is an offline approach. Our proposed approach not only can be applied to the offline setting but also the online setting where the state transition probability is unknown. In addition, the proposed algorithm can follow the non-stationary environment dynamics (i.e., the environment has small change) if the learning rate has not close to zero.

Algorithm 5 Advantage actor-critic algorithm

Initialization:

Actor network with random weights θ

Critic network with random weights ω

Clear replay memory $\mathcal{D}$. set the maximum number of iterations $N_{\max}$ and $n = 0$.

while $n < N_{\max}$ **do**
  Obtain initial state s_0.
  for k in $\{0, 1, \ldots, K-1\}$ **do**
    Select a_k according to π_θ.
    Execute action a_k and calculate reward r_k^η.
    Observe the next state s_{k+1}.
    Store (s_k, a_k, r_k^η) in $\mathcal{D}$
  end for
  $R = V_\omega^\eta(s_K)$
  for k in $\{K-1, K-2, \ldots, 0\}$ **do**
    $R \leftarrow r_k^\eta + \gamma R$
    update θ by $\theta \leftarrow \theta + \alpha \frac{\partial \log \pi_\theta(a_k|s_k)}{\partial \theta} \left(R - V_w^\eta\left(s_k\right)\right)$
    update ω by minimizing the loss function
    $L(\omega) = (R - V_\omega^\eta(s_k))^2$
  end for
  Update the Lagrange multiplier by Eq.(4.15)
  Clear $\mathcal{D}$
  $n \leftarrow n + 1$
end while

4.4 Summary

In this chapter, we presented a constrained deep reinforcement learning (CDRL) algorithm based on primal-dual optimization. We first relaxed the CMDP problem into a regular MDP using Lagrangian transformation. Then, a novel deep reinforcement learning algorithm, i.e., advantage actor-critic method, was introduced to optimally select the offloading location and allocate the computation resources. Specifically, the actor defines parameterized policy and is responsible for giving the action based on the current state. The critic is used to evaluate and criticize the current action by processing the reward obtained from the environment. In the next chapter, we will evaluate our proposed algorithm and compare its performance against baseline methods.

Chapter 5

Experimental Analysis

In this section, we use PYTHON simulation to evaluate the performance of our proposed algorithm. The A2C algorithm is conducted by using Keras plus TensorFlow [88], where Keras is adopted to build DNN training model and TensorFlow supplies the background support. Keras [89] is one of the leading high-level neural networks APIs built on TensorFlow.

5.1 Parameter Setting

We evaluate the performance of our proposed algorithm where we have a single edge server and a core cloud and we set initial state and we set initial state $q_0^m = 6, q_0^e = 10, q_0^c = 4$. The number of tasks K is set as 200. The inter-arrival time is exponential distribution and the mean value is λ. The upload data size follow a normal distribution $u_k \sim \mathcal{N}(250, 10)$MB. The required CPU cycles v_k is proportional to upload size, as $v_k = \mu u_k$. Here μ is 1000. Satisfaction function is set as $W(T) = 0.6T^{1.8}$. We set the bandwidth for the BS $B_1 =$ 2MHz, the bandwidth for WiFi AP $B_2 =$ 4MHz. If the task is offloaded to cloud and use additional resources, we set $x_k = 2$. The low battery threshold $\underline{b}$ is set as 0.73. We set $\alpha_1 = \alpha_2 = 0.3, \alpha_3 = 0.4$. Table 5.1 shows the additional experiment configuration parameters.

Table 5.1: Simulation parameters setup

Parameter	Value
f^m	1.2×10^9 cycles/s
f^e	5×10^9 cycles/s
f^c	2×10^{10} cycles/s
B_n	2500 mAh
P_{max}	0.9 W
P_{loc}	1.2 W
V_{tr}	2.1 V
V_{loc}	2.8 V
R	5 MB/s
σ_1^2 and σ_2^2	1.5×10^{-8}
ϵ_c	60
γ	0.95

5.2 Network Structure

The actor and critic networks share network parameters except the output layer during the training process. For the hyper-parameter setting, both of them are three-layer neural networks and the hidden layer has 128 neurons. ReLU function is used as activation function for the hidden layer. The output layer of the actor network is activated by softmax, which is introduced in Eq.(2.5). For the kernel initializer, we set it follows a normal distribution $\mathcal{N}(0, 0.01)$, which can guarantee the probability of selecting each action is almost the same in the initial stages of training. This setting is to enhance the exploration. The learning rate for actor and critic network is 0.0001 and 0.0005 respectively. The simulations were performed on a PC with Intel Core i7-9750H processor @ 2.60 GHz CPU and 16 GB of RAM.

5.3 Convergence Performance

Figure 5.1 illustrates the training curves with different learning rate in the first 1000 iterations. The corresponding learning rate of critic network is five times as much as that of actor network. As is shown, the convergence curve varies in different orders of magnitude for learning rate. In fact, for all hyper-parameters of the training of DNN, learning rate is

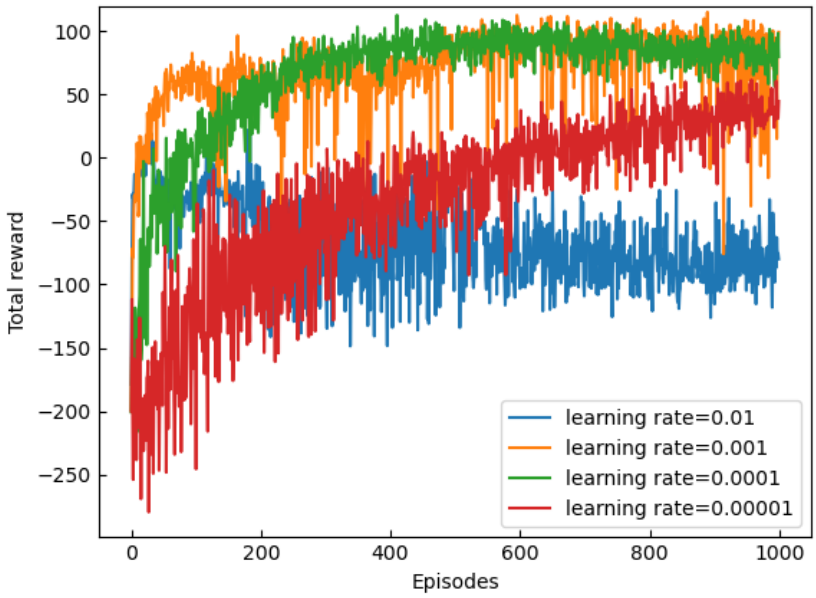

Figure 5.1: The total reward during the training process with different learning rates of actor network ($\lambda = 4.5$)

one of the important factors to have an affect on convergence performance. A small value of learning rate will result in a slower convergence speed. However, blindly increasing the learning rate can also make the training process divergent [90]. We can see that when the learning rate is 0.01, the convergence is very bad and it shows a terrible performance. In addition, the proposed scheme with learning rate 0.001 can reach the maximum reward at some spots, however, a learning rate 0.001 is still too large to guarantee convergence. Furthermore, a learning rate 0.00001 converges very slow and it does not reach the optimum before 1000 iterations. Therefore, the order of magnitude of 0.0001 is an appropriate choice for the learning rate.

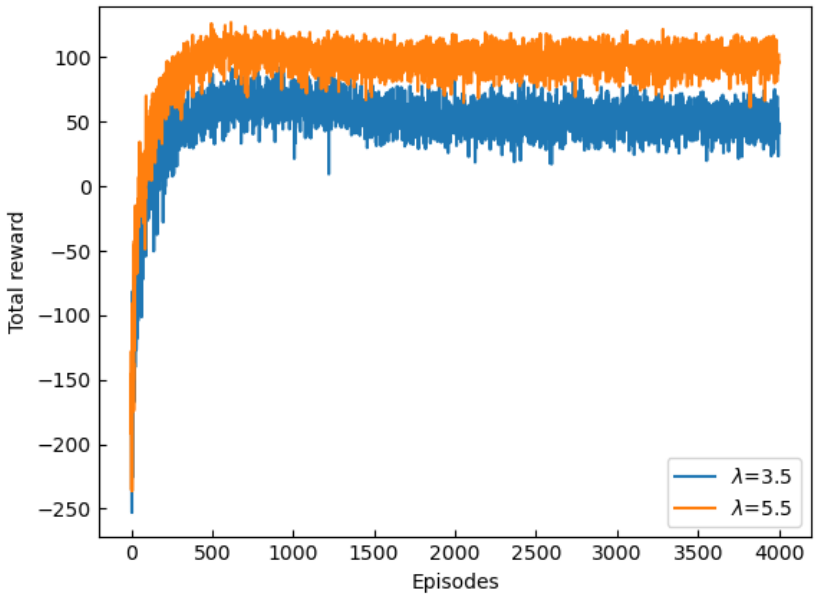

Figure 5.2: The total reward during the training process

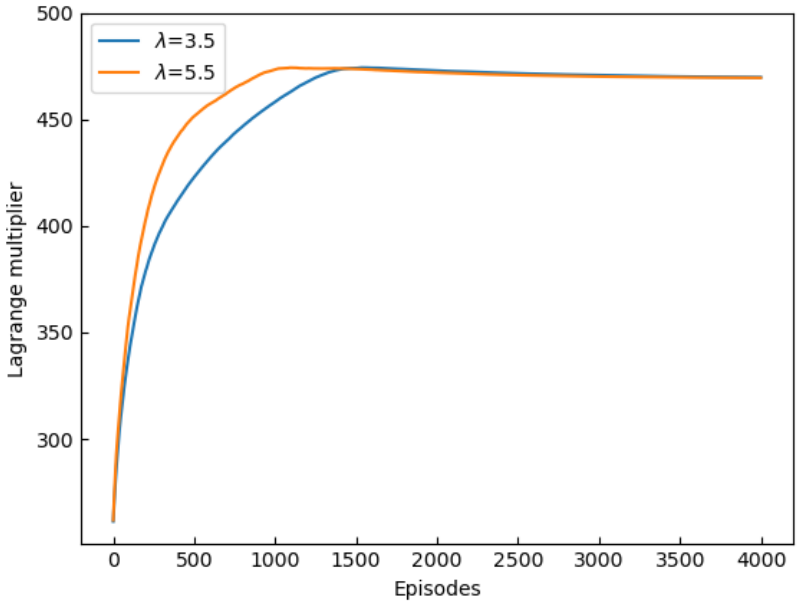

Figure 5.3: The Lagrange multiplier during the training process

We then increase the iteration to 4000 and Figure 5.2 depicts the total reward of the

proposed approach during the training process with respect to different inter-arrival time. It can be seen that the total reward received per episode converges to about 60 and 110 for $\lambda = 3.5$ and $\lambda = 5.5$, respectively. Both of them stabilize after around 1500 iterations. The high variance attributed to two uncertainties in our model: the task arrival time and upload data size.

Table 5.2: The number of episodes that exceed constraint tolerance during the training process

Episodes	$\lambda = 3.5$	percentage	$\lambda = 5.5$	percentage
0-500	500	100%	500	100%
500-1000	500	100%	495	99%
1000-1500	479	95.8%	204	40.8%
1500-2000	95	19%	81	16.2%
2000-2500	51	10.2%	64	12.8%
2500-3000	38	7.6%	81	16.2%
3000-3500	32	6.4%	67	13.4%
3500-4000	43	8.6%	71	14.2%
4000-4500	35	7.0%	57	11.4%
4500-5000	41	8.2%	43	8.6%
5000-5500	27	5.4%	29	5.8%
5500-6000	33	6.6%	27	5.4%
6000-6500	38	7.6%	29	5.8%
6500-7000	28	5.6%	25	5.0%
7000-7500	23	4.6%	24	4.8%

We keep increasing the iteration to 7500 and what can be clearly seen in Table 5.2 is the rapid decrease in the number of episodes that violate the battery constraint during the training process. At the beginning of the training process, because the agent is still in the process of exploration and the policy is almost equivalent to random policy, almost every episodes cannot satisfy the battery constraints. After 1500 iterations, this number decrease sharply. When the number of iterations is between 7000 and 7500, we can see the percentage of unsuccessful episodes have been decrease below 5%. Figure 5.3 reveals that there has been a steep increase for the Lagrange multiplier at the beginning of training process. This is because the difference between the battery constraint and the actual battery consumption is huge at the beginning of training. With the increase of iteration, the increase trend becomes slow, because the policy is getting close to the optimal one and the difference between actual battery consumption and the constraint becomes small.

After almost 1500 iterations, the Lagrange multiplier remains stable, which is consistent with Figure 5.2.

5.4 Performance Comparison

For comparison, we also trained an unconstrained actor-critic learning module within the same environment. A random method baseline and a greedy method baseline are also used for comparison. For the random method, the action is selected randomly from the set of the allowable actions. For the greedy method, the agent always select the action with the maximum reward from the set of the allowable actions.

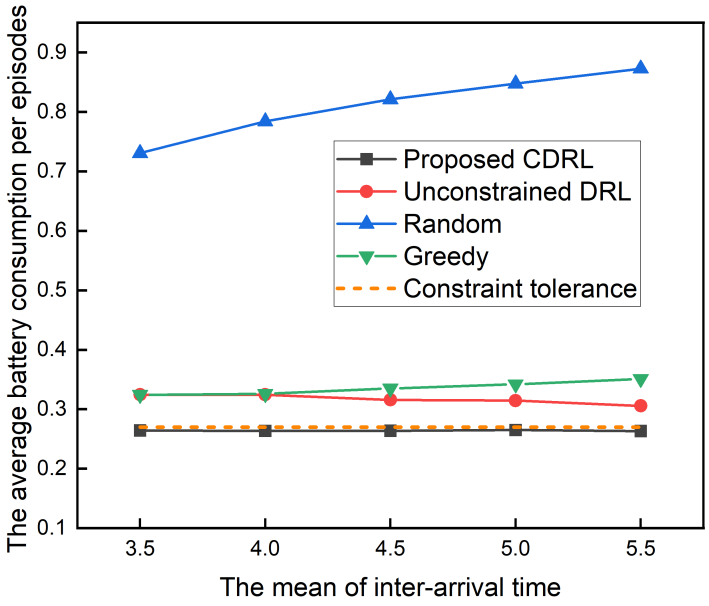

Figure 5.4: The battery consumption versus the mean of inter-arrival time

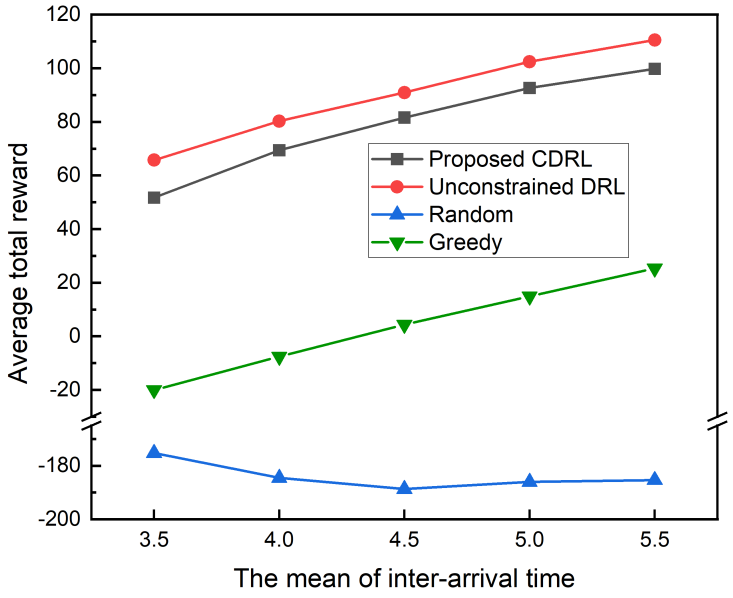

Figure 5.5: The total reward versus the mean of inter-arrival time

Figure 5.4 shows the battery consumption is maintained below the battery constraints $1 - \underline{b}$ for our proposed CDRL algorithm. The random method has the highest battery consumption. Figure 5.5 on the other hand, shows that the unconstrained DRL method achieves a higher reward than the proposed Constrained Deep Reinforcement Learning (CDRL) method. This is because in order to satisfy the battery constraint, sometimes the scheme would choose an action which has a higher cost but with a lower battery consumption. For example, when both the edge the cloud have no free resources but the cpus at the device are free, our proposed scheme may choose to offload the task to cloud and pay for additional resources at the cloud whenever the battery level is at a critical point. This action can be justified by comparing the lower amount of energy consumed to transmit the data compared to the energy needed for the local processing of the task.

From Figures 5.8 and 5.10, we can see that even though the greedy method has the minimum delay, it incurs the maximum monetary cost. This is because the greedy method always just considers the current task and ignores the requirement of future tasks. Normally, the greedy method always uses all available VM to execute the task and there is no

reservation for next few tasks. This action has very low cost performance because it will result in a large number of future tasks have to offload to the cloud and pay for additional resources. That is the reason why the greedy method has the highest monetary cost. The similar phenomenon can be found in Figures 5.11 and 5.13 where we fix the inter-arrival time and show the relationship between corresponding performance and the average upload data size for tasks.

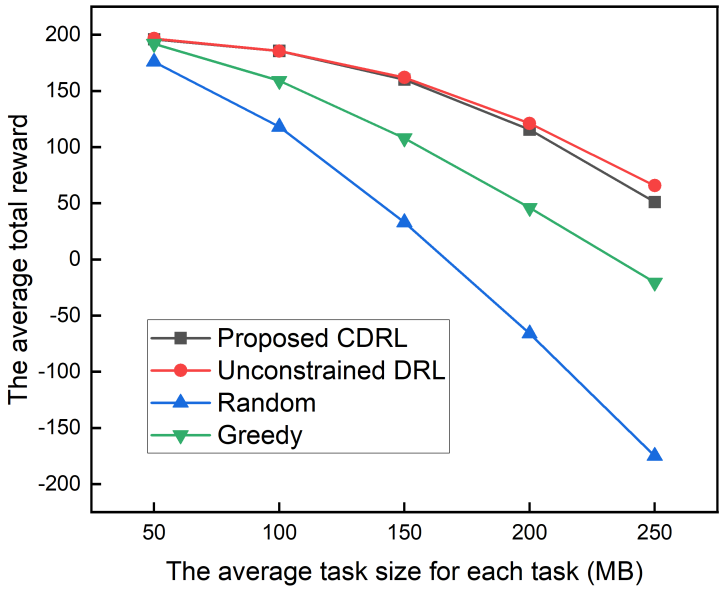

Figure 5.6: The average total reward versus the average task size

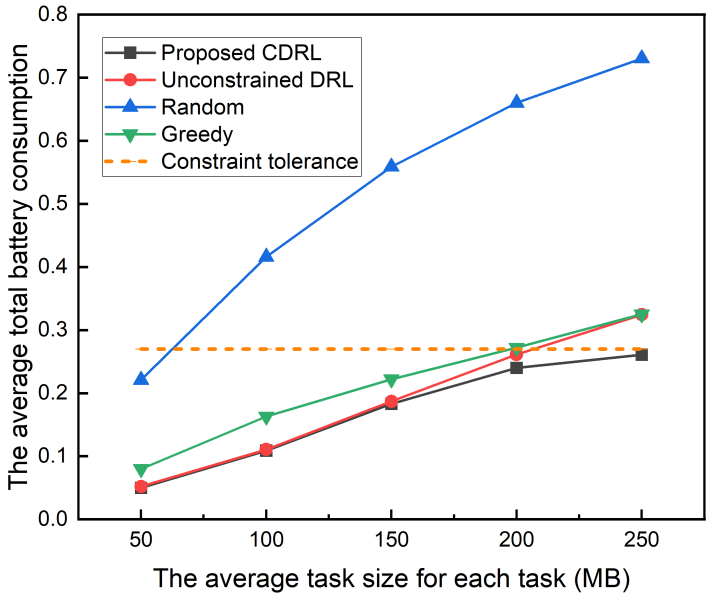

Figure 5.7: The battery consumption versus the average task size

Figure 5.6 shows the total reward received with respect to the average upload data size of offloading tasks, where the inter-arrival time of tasks is 3.5. As shown in Figure 5.6, the reward received of all methods decreases with the increasing average data size of offloading tasks, because bigger data size leads to more delay and energy consumption and each task will occupy VM for a longer time. Figures 5.11, 5.12 and 5.13 show there have been a gradual increase in the average delay, energy consumption and monetary cost of each task respectively. For the unconstrained DRL and proposed CDRL, when the average task size is small, the two methods show the same performance. This is because in the training process, the Lagrange multiplier will decrease to 0 since the battery constraints always can be satisfied for each episode. Therefore, the two methods are equivalent when the average task size is small. When the average task size becomes large, the reward received for unconstrained DRL is a little bit higher than that for the proposed CDRL. This is because the proposed CDRL method needs to consider the affect of battery constraint and to keep the battery consumption below this constraint. From Figure 5.7, it can be seen clearly that the proposed CDRL method satisfies the battery constraint well. The difference between

the CDRL algorithm and unconstrained DRL becomes obvious when the average task size is bigger than 200MB, which coincides with the corresponding reward curve in Figure 5.6. We also can observe that the reward of the random and greedy method decrease much more rapidly than the other two DRL methods with the increasing data size, which shows that the bigger data size of tasks, the more profit we get from our proposed method.

Compared to our proposed scheme, the greedy and random methods show apparently worse performance. These results demonstrate that the proposed primal-dual algorithm is successful at learning how to minimize the cost and to approximately enforce the constraints for the formulated CMDP.

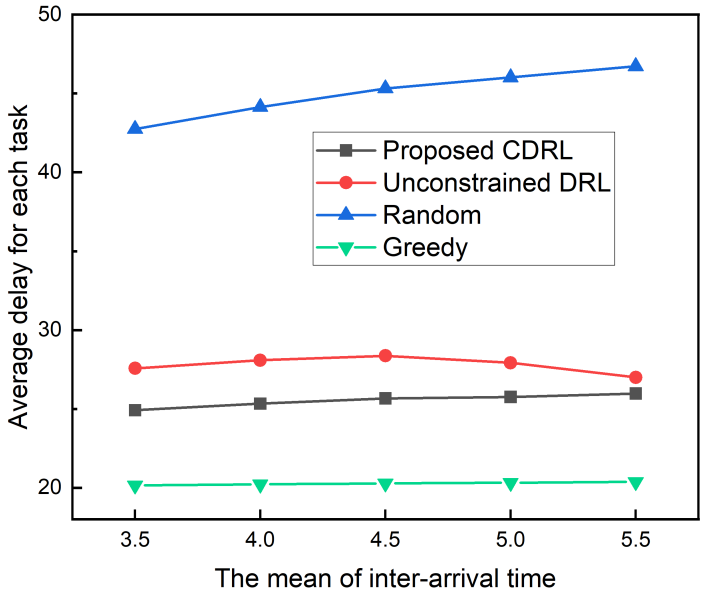

Figure 5.8: The average delay of each task versus the mean of inter-arrival time

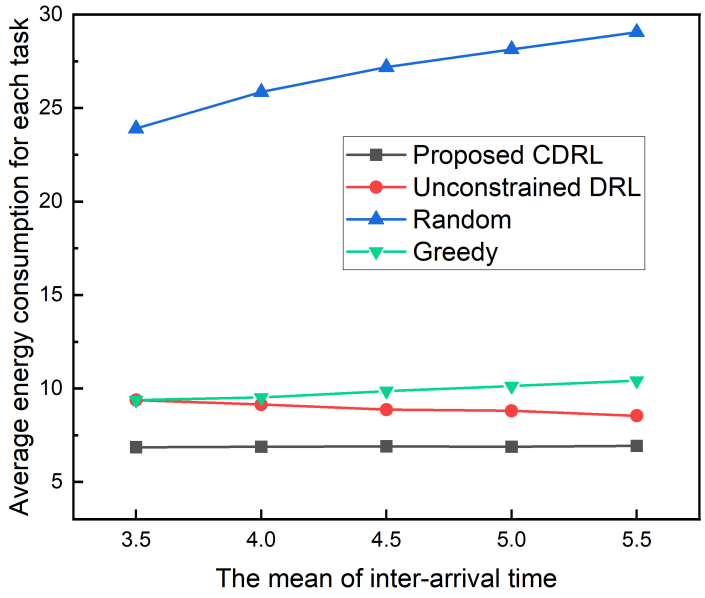

Figure 5.9: The average energy consumption of each task versus the mean of inter-arrival time

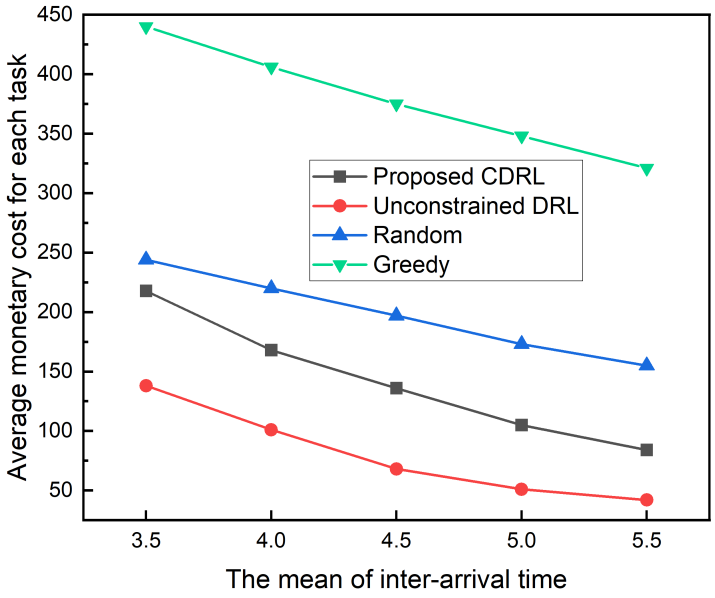

Figure 5.10: The average monetary cost of each task versus the mean of inter-arrival time

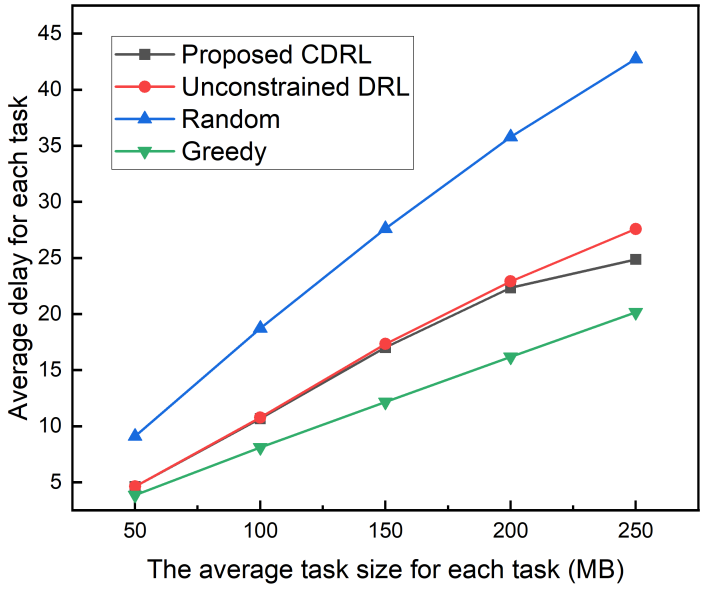

Figure 5.11: The average delay of each task versus the average task size

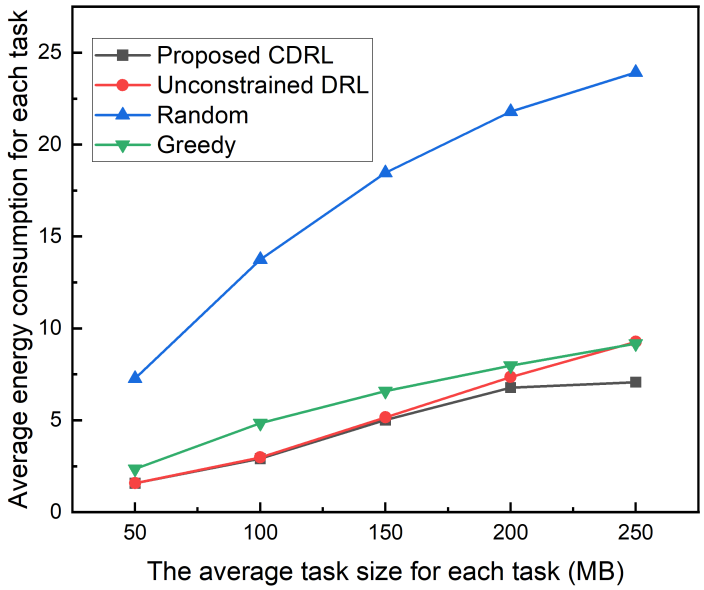

Figure 5.12: The average energy consumption of each task versus the average task size

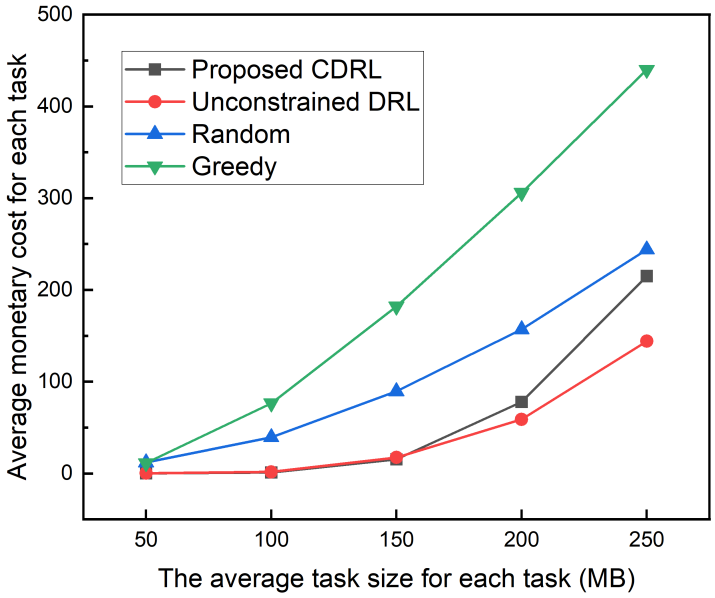

Figure 5.13: The average monetary cost of each task versus the average task size

5.5 Summary

In this chapter, we first evaluated the convergence performance of the proposed CDRL algorithm. The convergence performance shows that the proposed algorithm and the Lagrange multiplier can converge well under a proper learning rate. Then we compared its performance against the random method, the greedy method, and an unconstrained DRL algorithm. Specifically, the random method shows the worst performance. The greedy method can minimize the delay over other methods, but it generates the highest monetary cost. On the other hand, although the proposed CDRL algorithm's reward is not as high as the unconstrained DRL method, the proposed algorithm can successfully satisfy battery consumption requirements.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

The offloading and computation resource allocation strategy in mobile edge computing architecture can affect directly quality of service, so intelligent offloading algorithm renders better user experience. In this thesis, we have presented a novel scheme for the efficient location and resource selection for offloading tasks running on a mobile IoT unit using a multi-access edge computing infrastructure as well as a cloud data center. In contrast to existing schemes, we consider both the battery power limitation and the constraints of pre-allocated virtual machines (VMs) on the edge and cloud data-center servers by the application provider. We formulate the offloading and resource allocation problem as a constrained Markov decision process (CMDP) problem. We then propose a novel constrained deep reinforcement learning (CDRL) algorithm based on primal-dual optimization to find the near-optimal offloading policy. The proposed scheme can be used offline to train the mobile unit which then only needs to store an approximate set of offloading policies thereafter. Simulation results demonstrate that our proposed offloading and computation resources allocation approach achieves better performance than other baseline solutions in terms of the mobile unit cost savings while satisfying the battery consumption and VM resource availability constraints.

6.2 Future work

This thesis is based on specific scenarios and related assumptions, and there are many more that can be studied in-depth and optimized. In addition, with the continuous development of mobile edge computing technology, mobile edge computing offloading technology faces many new scenarios and new challenges, which are worthy of further study. The shortcomings of the thesis and the future research work are summarized as follows.

Mobility Prediction

The research content of this thesis assumes that the user is still or the range of motion is not broad, but as a matter of fact, the user is inevitably moving. Especially in an ultra-dense network, the offloading task failure or re-transmission phenomenon caused by the user crossing the area will be more obvious. Moreover, the VM migration impose high load on the backhaul and results in high delay, which makes it unsuitable for those real-time applications. To establish an effective mathematical model, based on accurately predicting the user's movement trajectory, designing an effective computation offloading strategy to reduce the task offload interruption is a challenge. Combined with some mobility prediction techniques [91], pre-migrate the computation in advance has been considered as a suggestion.

Multi-user Offloading Problem

In this thesis, we just consider a single user scenario. However, in real life, the multi-user scenario is more realistic. For most existing works, they need centralized control to achieve global optimal performance. In other words, existing work has a strong assumption that all users should share their information, e.g., quality of network connection and preference on energy efficiency [92]. However, users may be not willing to share the personal information owing to security and privacy concerns. Moreover, users are always selfish and hope to maximize their own performance. Therefore, to design a proper multi-user offloading and computation resource allocation algorithm is also a challenging problem.

Delayed Control Problem

When applying DRL into offloading problems, we generally believe that when the agent selects an action, it will be performed immediately, and a corresponding reward is obtained instantly. However, in real life, there is a control delay between obtaining the state of the environment and executing the action. This control delay always exists because it takes time to transfer observing information to the agent and compute the next action [74]. To better apply DRL, Katsikopoulos et al. [93] incorporate the past actions taken during the length of the delay into the current state in formulating an MDP model, so that the classical RL methods can be applied. This method focus on the constant delay problem. However, the actual delay in the real world may be stochastic. Therefore, considering the stochastic control delay when leveraging DRL in offloading problems is also an open issue.

Adaptation Problem

Although DRL can achieve very impressive results in simulations, there are two main challenges to applying DRL in the real-life offloading problem: generating training samples is exceedingly expensive, and unexpected change of the environment causes the well-trained model to fail in real-world application [94]. Dynamic changes include bandwidth variation, intermittent connectivity, resource provisioning, and workload management [95]. In these cases, the agent may not give the optimal policy anymore, and the agent has to be re-trained. Therefore, how to make the agent achieve online adaption in dynamic environments is a challenge for researchers.

References

[1] Khadija Akherfi, Micheal Gerndt, and Hamid Harroud. Mobile cloud computing for computation offloading: Issues and challenges. *Applied computing and informatics*, 14(1):1–16, 2018.

[2] Huaming Wu. Multi-objective decision-making for mobile cloud offloading: A survey. *IEEE Access*, 6:3962–3976, 2018.

[3] P Jonsson, S Carson, G Blennerud, J Kyohun Shim, B Arendse, A Husseini, et al. Ericsson mobility report. 2020. *Ericsson: Stockholm, Sweden*, 2020.

[4] Zheng Gong Zhiyuan Xu, Xi Chen et al. Virtual (augmented) reality white papers. *China Academy of Information and Communication Technology and HUAWEI TECHNOLOGIES CO.LTD.*, 2018.

[5] Ashiq Anjum, Tariq Abdullah, Muhammad Tariq, Yusuf Baltaci, and Nick Antonopoulos. Video stream analysis in clouds: An object detection and classification framework for high performance video analytics. *IEEE Transactions on Cloud Computing*, 2016.

[6] Lei Lei, Huijuan Xu, Xiong Xiong, Kan Zheng, and Wei Xiang. Joint computation offloading and multiuser scheduling using approximate dynamic programming in nb-iot edge computing system. *IEEE Internet of Things Journal*, 6(3):5345–5362, 2019.

[7] Fotios Zantalis, Grigorios Koulouras, Sotiris Karabetsos, and Dionisis Kandris. A review of machine learning and iot in smart transportation. *Future Internet*, 11(4):94, 2019.

[8] Fadi Al-Turjman and Arman Malekloo. Smart parking in iot-enabled cities: A survey. *Sustainable Cities and Society*, 49:101608, 2019.

[9] Pavel Mach and Zdenek Becvar. Mobile edge computing: A survey on architecture and computation offloading. *IEEE Communications Surveys & Tutorials*, 19(3):1628–1656, 2017.

[10] Ke Zhang, Yuming Mao, Supeng Leng, Quanxin Zhao, Longjiang Li, Xin Peng, Li Pan, Sabita Maharjan, and Yan Zhang. Energy-efficient offloading for mobile edge computing in 5g heterogeneous networks. *IEEE access*, 4:5896–5907, 2016.

[11] Lei Zhao, Jiadai Wang, Jiajia Liu, and Nei Kato. Optimal edge resource allocation in iot-based smart cities. *IEEE Network*, 33(2):30–35, 2019.

[12] Jianyu Wang, Jianli Pan, Flavio Esposito, Prasad Calyam, Zhicheng Yang, and Prasant Mohapatra. Edge cloud offloading algorithms: Issues, methods, and perspectives. *ACM Computing Surveys (CSUR)*, 52(1):1–23, 2019.

[13] Bin Cao, Long Zhang, Yun Li, Daquan Feng, and Wei Cao. Intelligent offloading in multi-access edge computing: A state-of-the-art review and framework. *IEEE Communications Magazine*, 57(3):56–62, 2019.

[14] Fagui Liu, Zhenxi Huang, and Liangming Wang. Energy-efficient collaborative task computation offloading in cloud-assisted edge computing for iot sensors. *Sensors*, 19(5):1105, 2019.

[15] Wei Cao, Gang Feng, Shuang Qin, and Mu Yan. Cellular offloading in heterogeneous mobile networks with d2d communication assistance. *IEEE Transactions on Vehicular Technology*, 66(5):4245–4255, 2016.

[16] Kuljeet Kaur, Sahil Garg, Georges Kaddoum, Syed Hassan Ahmed, and Mohammed Atiquzzaman. Keids: Kubernetes-based energy and interference driven scheduler for industrial iot in edge-cloud ecosystem. *IEEE Internet of Things Journal*, 7(5):4228–4237, 2019.

[17] Kaiyi Zhang and Nancy Samaan. Optimized look-ahead offloading decisions using deep reinforcement learning for battery constrained mobile iot devices. In *2020 IEEE International Conference on Smart Cloud (SmartCloud)*, pages 181–186, 2020.

[18] Qi Chen, Wei Wang, Fangyu Wu, Suparna De, Ruili Wang, Bailing Zhang, and Xin Huang. A survey on an emerging area: Deep learning for smart city data. *IEEE Transactions on Emerging Topics in Computational Intelligence*, 3(5):392–410, 2019.

[19] Li Ping Qian, Yuan Wu, Bo Ji, Liang Huang, and Danny HK Tsang. Hybridiot: Integration of hierarchical multiple access and computation offloading for iot-based smart cities. *IEEE Network*, 33(2):6–13, 2019.

[20] Liang Zhou, Dan Wu, Jianxin Chen, and Zhenjiang Dong. Greening the smart cities: Energy-efficient massive content delivery via d2d communications. *IEEE Transactions on Industrial Informatics*, 14(4):1626–1634, 2017.

[21] Quoc-Viet Pham, Fang Fang, Vu Nguyen Ha, Mai Le, Zhiguo Ding, Long Bao Le, and Won-Joo Hwang. A survey of multi-access edge computing in 5g and beyond: Fundamentals, technology integration, and state-of-the-art. *arXiv preprint arXiv:1906.08452*, 2019.

[22] Yangzhe Liao, Liqing Shou, Quan Yu, Qingsong Ai, and Quan Liu. Joint offloading decision and resource allocation for mobile edge computing enabled networks. *Computer Communications*, 2020.

[23] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Caceres, and Nigel Davies. The case for vm-based cloudlets in mobile computing. *IEEE pervasive Computing*, 8(4):14–23, 2009.

[24] Xu Chen, Lei Jiao, Wenzhong Li, and Xiaoming Fu. Efficient multi-user computation offloading for mobile-edge cloud computing. *IEEE/ACM Transactions on Networking*, 24(5):2795–2808, 2015.

[25] Rui Li, Zhi Zhou, Xu Chen, and Qing Ling. Resource price-aware offloading for edge-cloud collaboration: A two-timescale online control approach. *IEEE Transactions on Cloud Computing*, 2019.

[26] Yun Chao Hu, Milan Patel, Dario Sabella, Nurit Sprecher, and Valerie Young. Mobile edge computing—a key technology towards 5g. *ETSI white paper*, 11(11):1–16, 2015.

[27] Ju Ren, Deyu Zhang, Shiwen He, Yaoxue Zhang, and Tao Li. A survey on end-edge-cloud orchestrated network computing paradigms: Transparent computing, mobile edge computing, fog computing, and cloudlet. *ACM Computing Surveys (CSUR)*, 52(6):1–36, 2019.

[28] Sami Kekki, Walter Featherstone, Yonggang Fang, Pekka Kuure, Alice Li, Anurag Ranjan, Debashish Purkayastha, Feng Jiangping, Danny Frydman, Gianluca Verin, et al. Mec in 5g networks. *ETSI white paper*, 28:1–28, 2018.

[29] Milan Patel, Brian Naughton, Caroline Chan, Nurit Sprecher, Sadayuki Abeta, Adrian Neal, et al. Mobile-edge computing introductory technical white paper. *White paper, mobile-edge computing (MEC) industry initiative*, pages 1089–7801, 2014.

[30] Nasir Abbas, Yan Zhang, Amir Taherkordi, and Tor Skeie. Mobile edge computing: A survey. *IEEE Internet of Things Journal*, 5(1):450–465, 2017.

[31] Subashini Subashini and Veeraruna Kavitha. A survey on security issues in service delivery models of cloud computing. *Journal of network and computer applications*, 34(1):1–11, 2011.

[32] Delphine Christin. Privacy in mobile participatory sensing: Current trends and future challenges. *Journal of Systems and Software*, 116:57–68, 2016.

[33] Akhil Gupta and Rakesh Kumar Jha. A survey of 5g network: Architecture and emerging technologies. *IEEE access*, 3:1206–1232, 2015.

[34] Evgeny Khorov, Anton Kiryanov, Andrey Lyakhov, and Giuseppe Bianchi. A tutorial on ieee 802.11 ax high efficiency wlans. *IEEE Communications Surveys & Tutorials*, 21(1):197–216, 2018.

[35] Moo-Ryong Ra, Jeongyeup Paek, Abhishek B Sharma, Ramesh Govindan, Martin H Krieger, and Michael J Neely. Energy-delay tradeoffs in smartphone applications. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, pages 255–270, 2010.

[36] Lin Zhang, Ming Xiao, Gang Wu, Muhammad Alam, Ying-Chang Liang, and Shaoqian Li. A survey of advanced techniques for spectrum sharing in 5g networks. *IEEE Wireless Communications*, 24(5):44–51, 2017.

[37] Thinh Quang Dinh, Jianhua Tang, Quang Duy La, and Tony QS Quek. Offloading in mobile edge computing: Task allocation and computational frequency scaling. *IEEE Transactions on Communications*, 65(8):3571–3584, 2017.

[38] Yuyi Mao, Jun Zhang, and Khaled B Letaief. Dynamic computation offloading for mobile-edge computing with energy harvesting devices. *IEEE Journal on Selected Areas in Communications*, 34(12):3590–3605, 2016.

[39] Tuyen X Tran and Dario Pompili. Joint task offloading and resource allocation for multi-server mobile-edge computing networks. *IEEE Transactions on Vehicular Technology*, 68(1):856–868, 2018.

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

[41] Fengxian Guo, Heli Zhang, Hong Ji, Xi Li, and Victor CM Leung. An efficient computation offloading management scheme in the densely deployed small cell networks with mobile edge computing. *IEEE/ACM Transactions on Networking*, 26(6):2651–2664, 2018.

[42] Heungsik Eom, Pierre St Juste, Renato Figueiredo, Omesh Tickoo, Ramesh Illikkal, and Ravishankar Iyer. Machine learning-based runtime scheduler for mobile offloading framework. In *2013 IEEE/ACM 6th International Conference on Utility and Cloud Computing*, pages 17–25. IEEE, 2013.

[43] Heungsik Eom, Renato Figueiredo, Huaqian Cai, Ying Zhang, and Gang Huang. Malmos: Machine learning-based mobile offloading scheduler with online training. In *2015 3rd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering*, pages 51–60. IEEE, 2015.

[44] Siyun Wu, Weiwei Xia, Wenqing Cui, Qian Chao, Zhuorui Lan, Feng Yan, and Lianfeng Shen. An efficient offloading algorithm based on support vector machine for mobile edge computing in vehicular networks. In *2018 10th International Conference on Wireless Communications and Signal Processing (WCSP)*, pages 1–6. IEEE, 2018.

[45] Chaoyun Zhang, Paul Patras, and Hamed Haddadi. Deep learning in mobile and wireless networking: A survey. *IEEE Communications Surveys & Tutorials*, 21(3):2224–2287, 2019.

[46] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 315–323, 2011.

[47] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[48] Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[49] Eitan Altman. *Constrained Markov decision processes*, volume 7. CRC Press, 1999.

[50] Sami Khairy, Prasanna Balaprakash, Lin X Cai, and Yu Cheng. Constrained deep reinforcement learning for energy sustainable multi-uav based random access iot networks with noma. *arXiv preprint arXiv:2002.00073*, 2020.

[51] Warren B Powell. What you should know about approximate dynamic programming. *Naval Research Logistics (NRL)*, 56(3):239–249, 2009.

[52] Elena Pashenkova, Irina Rish, and Rina Dechter. Value iteration and policy iteration algorithms for markov decision problem. In *AAAI'96: Workshop on Structural Issues in Planning and Temporal Reasoning*. Citeseer, 1996.

[53] Christopher John Cornish Hellaby Watkins. Learning from delayed rewards. 1989.

[54] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[55] Hado V Hasselt. Double q-learning. In *Advances in neural information processing systems*, pages 2613–2621, 2010.

[56] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Thirtieth AAAI conference on artificial intelligence*, 2016.

[57] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

[58] Richard S Sutton, David A McAllester, Satinder P Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*, pages 1057–1063, 2000.

[59] Julien Vitay. Deep reinforcement learning.

[60] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[61] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. 2014.

[62] Shuai Yu, Xin Wang, and Rami Langar. Computation offloading for mobile edge computing: A deep learning approach. In *2017 IEEE 28th Annual International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC)*, pages 1–6. IEEE, 2017.

[63] Liang Huang, Xu Feng, Anqi Feng, Yupin Huang, and Li Ping Qian. Distributed deep learning-based offloading for mobile edge computing networks. *Mobile Networks and Applications*, pages 1–8, 2018.

[64] Liang Huang, Xu Feng, Luxin Zhang, Liping Qian, and Yuan Wu. Multi-server multi-user multi-task computation offloading for mobile edge computing networks. *Sensors*, 19(6):1446, 2019.

[65] Yiming Miao, Gaoxiang Wu, Miao Li, Ahmed Ghoneim, Mabrook Al-Rakhami, and M Shamim Hossain. Intelligent task prediction and computation offloading based on mobile-edge cloud computing. *Future Generation Computer Systems*, 102:925–931, 2020.

[66] Chaoxiong Cui, Ming Zhao, and Kelvin Wong. An lstm-method-based availability prediction for optimized offloading in mobile edges. *Sensors*, 19(20):4467, 2019.

[67] Xianlong Zhao, Kexin Yang, Qimei Chen, Duo Peng, Hao Jiang, Xianze Xu, and Xinzhuo Shuang. Deep learning based mobile data offloading in mobile edge computing systems. *Future Generation Computer Systems*, 99:346–355, 2019.

[68] Haleh Shahzad and Ted H Szymanski. A dynamic programming offloading algorithm for mobile cloud computing. In *2016 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*, pages 1–5. IEEE, 2016.

[69] Qiong Wu, Hongmei Ge, Hanxu Liu, Qiang Fan, Zhengquan Li, and Ziyang Wang. A task offloading scheme in vehicular fog and cloud computing system. *IEEE Access*, 8:1173–1184, 2019.

[70] Ji Li, Hui Gao, Tiejun Lv, and Yueming Lu. Deep reinforcement learning based computation offloading and resource allocation for mec. In *2018 IEEE Wireless Communications and Networking Conference (WCNC)*, pages 1–6. IEEE, 2018.

[71] Minghui Min, Liang Xiao, Ye Chen, Peng Cheng, Di Wu, and Weihua Zhuang. Learning-based computation offloading for iot devices with energy harvesting. *IEEE Transactions on Vehicular Technology*, 68(2):1930–1941, 2019.

[72] Hua Zuo, Guangquan Zhang, Witold Pedrycz, Vahid Behbood, and Jie Lu. Fuzzy regression transfer learning in takagi–sugeno fuzzy models. *IEEE Transactions on Fuzzy Systems*, 25(6):1795–1807, 2016.

[73] Xianfu Chen, Honggang Zhang, Celimuge Wu, Shiwen Mao, Yusheng Ji, and Medhi Bennis. Optimized computation offloading performance in virtual edge computing systems via deep reinforcement learning. *IEEE Internet of Things Journal*, 6(3):4005–4018, 2018.

[74] Lei Lei, Yue Tan, Kan Zheng, Shiwen Liu, Kuan Zhang, and Xuemin Shen. Deep reinforcement learning for autonomous internet of things: Model, applications and challenges. *IEEE Communications Surveys & Tutorials*, 2020.

[75] Renchao Xie, Qinqin Tang, Chenghao Liang, F Richard Yu, and Tao Huang. Dynamic computation offloading in iot fog systems with imperfect channel state information: A pomdp approach. *IEEE Internet of Things Journal*, 2020.

[76] Nan Cheng, Feng Lyu, Wei Quan, Conghao Zhou, Hongli He, Weisen Shi, and Xuemin Shen. Space/aerial-assisted computing offloading for iot applications: A learning-based approach. *IEEE Journal on Selected Areas in Communications*, 37(5):1117–1129, 2019.

[77] Jie Feng, F Richard Yu, Qingqi Pei, Xiaoli Chu, Jianbo Du, and Li Zhu. Cooperative computation offloading and resource allocation for blockchain-enabled mobile edge computing: A deep reinforcement learning approach. *IEEE Internet of Things Journal*, 2019.

[78] Zhao Chen and Xiaodong Wang. Decentralized computation offloading for multi-user mobile edge computing: A deep reinforcement learning approach. *arXiv preprint arXiv:1812.07394*, 2018.

[79] Hongchang Ke, Jian Wang, Lingyue Deng, Yuming Ge, and Hui Wang. Deep reinforcement learning-based adaptive computation offloading for mec in heterogeneous vehicular networks. *IEEE Transactions on Vehicular Technology*, 2020.

[80] Haodong Lu, Xiaoming He, Miao Du, Xiukai Ruan, Yanfei Sun, and Kun Wang. Edge qoe: Computation offloading with deep reinforcement learning for internet of things. *IEEE Internet of Things Journal*, 2020.

[81] Wen-Yeau Chang. The state of charge estimating methods for battery: A review. *International Scholarly Research Notices*, 2013, 2013.

[82] Xi Zhang and Jingqing Wang. Joint heterogeneous statistical-qos/qoe provisionings for edge-computing based wifi offloading over 5g mobile wireless networks. In *2018 52nd Annual Conference on Information Sciences and Systems (CISS)*, pages 1–6. IEEE, 2018.

[83] Shashidhar Lakkavalli and Suresh Singh. Using remaining battery lifetime information and relaying to decrease outage probability of a mobile terminal. In *2003 IEEE 58th Vehicular Technology Conference. VTC 2003-Fall (IEEE Cat. No. 03CH37484)*, volume 3, pages 1843–1847. IEEE, 2003.

[84] Yu Liu, Qimei Cui, Jian Zhang, Yu Chen, and Yanzhao Hou. An actor-critic deep reinforcement learning based computation offloading for three-tier mobile computing networks. In *2019 11th International Conference on Wireless Communications and Signal Processing (WCSP)*, pages 1–6. IEEE, 2019.

[85] Weiheng Jiang, Yi Gong, Yang Cao, Xiaogang Wu, and Qian Xiao. Energy-delay-cost tradeoff for task offloading in imbalanced edge cloud based computing. *arXiv preprint arXiv:1805.02006*, 2018.

[86] Changyan Yi, Jun Cai, and Zhou Su. A multi-user mobile computation offloading and transmission scheduling mechanism for delay-sensitive applications. *IEEE Transactions on Mobile Computing*, 19(1):29–43, 2019.

[87] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015.

[88] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283, 2016.

[89] François Chollet et al. Keras: The python deep learning library. *ascl*, pages ascl–1806, 2018.

[90] Jia Luo, F Richard Yu, Qianbin Chen, and Lun Tang. Adaptive video streaming with edge caching and video transcoding over software-defined mobile networks: A deep reinforcement learning approach. *IEEE Transactions on Wireless Communications*, 19(3):1577–1592, 2019.

[91] David HS Lima, Andre LL Aquino, and Marilia Curado. A review of mobility prediction models applied in cloud/fog environments. In *European Conference on Parallel Processing*, pages 263–274. Springer, 2018.

[92] Yufeng Zhan, Song Guo, Peng Li, and Jiang Zhang. A deep reinforcement learning based offloading game in edge computing. *IEEE Transactions on Computers*, 69(6):883–893, 2020.

[93] Konstantinos V Katsikopoulos and Sascha E Engelbrecht. Markov decision processes with delays and asynchronous cost collection. *IEEE transactions on automatic control*, 48(4):568–574, 2003.

[94] Anusha Nagabandi, Ignasi Clavera, Simin Liu, Ronald S Fearing, Pieter Abbeel, Sergey Levine, and Chelsea Finn. Learning to adapt in dynamic, real-world environments through meta-reinforcement learning. *arXiv preprint arXiv:1803.11347*, 2018.

[95] Arani Bhattacharya and Pradipta De. A survey of adaptation techniques in computation offloading. *Journal of Network and Computer Applications*, 78:97–115, 2017.