System Optimisation for Multi-access Edge Computing Based on Deep Reinforcement Learning

Jin Wang

College of Engineering, Mathematics and Physical Sciences
University of Exeter

Submitted by Jin Wang to the University of Exeter as a thesis for the degree of *Doctor of Philosophy in Computer Science*

I certify that all material in this thesis which is not my own work has been identified and that any material that has previously been submitted and approved for the award of a degree by this or any other University has been acknowledged.

Signature:.....

Department of Computer Science November 2021

I would like to dedicate this thesis to my loving parents.

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements. This dissertation contains fewer than 100,000 words including appendices, bibliography, footnotes, tables and equations and has fewer than 150 figures.

Jin Wang
November 2021

Acknowledgements

First and foremost I am extremely grateful to my supervisors, Prof. Geyong Min and Dr.. Jia Hu for their invaluable advice, continuous support, and patience during my PhD. study. Their immense knowledge and plentiful experience have pointed me in the right direction and provide inspiration to do the best work I could.

When I started the journey of my PhD. at the University of Exeter, I was a bit scared but anticipated the unknowns and uncertainties in future life. I would like to thank my colleagues in the lab over the years: Dr. Zhengxin Yu, Dr. Chengqiang Huang, Dr. Haozhe Wang, Dr. Miao Wang, Dr. Zhiwei Zhao, Dr. Yuan Zuo, Ms. Yang Mi, Mr. Zheyi Chen, Mr. Han Xu, Mr. Jed Mills, Mr. Zi Wang, Ms. Rui Jin, Mr. Zhe Wang, Mr. Mohammad Rami Koujan, Ms. Sakine Yalman, and all members in the lab who offer me lots of help and care for me. With their accompany, I have enjoyed a colourful PhD. life. In particular, I would like to extend my extra gratitude to Dr. Lejun Chen, Dr. Chengqiang Huang, Dr. Ke Li, and Dr. Chao Wang. They were senior PhD. students, lecturers, or postdocs when I was a rookie and knew nothing about the basics of research. They patiently guided me on how to do research and treated me equally. They are my role models for doing academic research. I would also like to thank Dr. Wenhan Zhan, who was my closest collaborator on the main work in the thesis, and with whom I shared many great conversations.

I wish to show my appreciation to the Alan Turing Institute for providing me with an opportunity to do an internship there from Oct. 2019 to July 2020, where I had the chance to know world-leading research. I would like to express my sincere gratitude to Mr. Lizhi Zhang, Ms. Chenxia Gu, Dr. Shaoxiong Hu, Mr. Tiejun Wei, Mr. Jun Sun, and Mr. Lorenzo Rimella, who come from different disciplines including statistics, medical science, and physics. They offered lots of help for my research and inspired me from different perspectives. Also, thanks for their accompany during my time in London.

Finally, I am deeply grateful to my beloved parents who always support me and love me. They are open-minded and respect every decision I have made for my life. This thesis is dedicated to them. I believe, with their support, I will have the courage to face every challenge in the future.

List of Publications

1. **J. Wang**, J. Hu, G. Min, A. Y. Zomaya, and N. Georgalas, "Fast Adaptive Computation Offloading in Edge Computing based on Meta Reinforcement Learning", *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 1, pp. 242–253, 2020.
2. **J. Wang**, J. Hu, G. Min, W. Zhan, Q. Ni, and N. Georgalas, "Computation Offloading in Multi-Access Edge Computing Using a Deep Sequential Model Based on Reinforcement Learning", *IEEE Communications Magazine*, vol. 57, no. 5, pp. 64–69, 2019.
3. **J. Wang**, J. Hu, G. Min, W. Zhan, "Energy-Aware Dependent Task Offloading for Multi-Access Edge Computing: A Deep Reinforcement Learning-driven Approach". (Minor revision, *IEEE Transactions on Computers*, 2021)
4. **J. Wang**, J. Hu, G. Min, "Online Service Migration in Edge Computing with Incomplete Information: A Deep Recurrent Actor-Critic Method", *arXiv preprint arXiv:2012.08679 2020.* (Under review, *IEEE Transactions on Mobile Computing*, 2021)
5. **J. Wang**, J. Hu, J. Mills, G. Min, "Federated Ensemble Model-based Reinforcement Learning", arXiv:2109.05549, 2021
6. J. Mills, J. Hu, G. Min, R. Jin, S. Zheng, **J. Wang**, "Accelerating Federated Learning with a Global Biased Optimiser", arXiv:2108.09134, 2021
7. Z. Wang, J. Hu, G. Min, Z. Zhao, **J. Wang**, "Data Augmentation based Cellular Traffic Prediction in Edge Computing-Enabled Smart City", *IEEE Transactions on Industrial Informatics*, vol. 17, no. 6, pp 4179–4187, 2021
8. W. Zhan, C. Luo, **J. Wang**, C. Wang, G. Min, H. Duan, and Q. Zhu,"Deep Reinforcement Learning-Based Offloading Scheduling for Vehicular Edge Computing", *IEEE Internet of Things Journal*, vol. 7, no. 6, pp. 5449–5465, 2020.
9. W. Zhan, C. Luo, **J. Wang**, G. Min, and H. Duan "Deep reinforcement learning-based computation offloading in vehicular edge computing", pro. in *IEEE GLOBECOM'19*, 2019.

Abstract

Multi-access edge computing (MEC) is an emerging and important distributed computing paradigm that aims to extend cloud service to the network edge to reduce network traffic and service latency. Proper system optimisation and maintenance are crucial to maintaining high Quality-of-service (QoS) for end-users. However, with the increasing complexity of the architecture of MEC and mobile applications, effectively optimising MEC systems is non-trivial. Traditional optimisation methods are generally based on simplified mathematical models and fixed heuristics, which rely heavily on expert knowledge. As a consequence, when facing dynamic MEC scenarios, considerable human efforts and expertise are required to redesign the model and tune the heuristics, which is time-consuming.

This thesis aims to develop deep reinforcement learning (DRL) methods to handle system optimisation problems in MEC. Instead of developing fixed heuristic algorithms for the problems, this thesis aims to design DRL-based methods that enable systems to learn optimal solutions on their own. This research demonstrates the effectiveness of DRL-based methods on two crucial system optimisation problems: task offloading and service migration. Specifically, this thesis first investigate the dependent task offloading problem that considers the inner dependencies of tasks. This research builds a DRL-based method combining sequence-to-sequence (seq2seq) neural network to address the problem. Experiment results demonstrate that our method outperforms the existing heuristic algorithms and achieves near-optimal performance. To further enhance the learning efficiency of the DRL-based task offloading method for unseen learning tasks, this thesis then integrates meta reinforcement learning to handle the task offloading problem. Our method can adapt fast to new environments with a small number of gradient updates and samples. Finally, this thesis exploits the DRL-based solution for the service migration problem in MEC considering user mobility. This research models the service migration problem as a Partially Observable Markov Decision Process (POMDP) and propose a tailored actor-critic algorithm combining Long-short Term Memory (LSTM) to solve the POMDP. Results from extensive experiments based on real-world mobility traces demonstrate that our method consistently outperforms both the heuristic and state-of-the-art learning-driven algorithms on various MEC scenarios.

Table of contents

List of figures

List of tables

Nomenclature

Acronyms / Abbreviations

CMDP Constrained Markov Decision Process

CSP Constraint-Satisfaction Problems

DAG Directed Acyclic Graphs

DNN Deep Neural Network

DP Dynamic programming

LP Linear Programming

DRL Deep Reinforcement Learning

FL Federated Learning

GNN Graph Neural Network

HEFT Heterogeneous Earliest Finish Time

ICT Information and Communications Technology

LSTM Long Short-Term Memory

MBA Multi-armed Bandit

MC Monte Carlo

MDP Markov Decision Process

MEC Multi-access Edge Computing

MILP Mixed Integer Linear Programming

MINLP Mixed-Integer Non-Linear Programming

MIQP Mixed Quadratic Programming

MRL Meta Reinforcement Learning

NFV Network Function Visualization

POMDP Partially Observable Markov Decision Process

QoS Quality-of-Service

RL Reinforcement Learning

RNN Recurrent Neural Network

SDN Software-Defined Network

seq2seq Sequence to Sequence

TD Temporal Difference

TSP Travelling Salesman Problem

UE User Equipment

VPG vanilla policy gradient

VR Virtual Reality

WAN Wide Area Network

Chapter 1

Introduction

In recent years, mobile devices including smartphones, tablets, wearable devices, smart sensors, etc. have infiltrated every aspect of individuals' life, work, and entertainment. The rapidly increasing number of modern mobile devices and the emerging diverse applications raise an inevitable requirement for more processing, operation, and optimisation of future computing and networking systems. When facing mobile applications, the centralized Cloud computing systems are encountering noticeable challenges including lagged data transmission, security vulnerability, low coverage, so on and so forth. To address the above challenges, Multi-access edge computing (MEC) [104] is proposed to provide high Quality-of-Service (QoS) to the end devices by providing computing and storage resources to the network edges.

MEC provides many fundamental functionalities (e.g., task offloading [105, 74] and content caching [154]) to improve the QoS of end-users. However, optimisation and maintenance for MEC systems are non-trivial due to the growing complexity of the MEC architecture. Traditional optimisation solutions for MEC systems rely heavily on expert knowledge. For example, due to the NP-hardness of most of the optimisation problems in MEC systems, many solutions are based on heuristic algorithms. Typical steps for those heuristic-based solutions include: 1) building a simplified model by analysing the problem and optimisation target; 2) designing heuristics/rules to achieve the target based on the simplified models; 3) extensively tuning those heuristics/rules to achieve good performance in real systems. Some researchers decompose the original optimisation problems into several sub-problems and solve all/parts of sub-problems by using convex optimisation [130, 66, 161]. However, those methods do not directly optimise for the global objective, and the optimisation results are often far from optima. In a nutshell, when facing ever-changing MEC scenarios, traditional optimisation solutions require considerable human efforts and expertise to redesign the model or decomposition methods. Besides, they need to tune the heuristics/rules to adapt to new

scenarios, which is time-consuming and sometimes unrealistic due to the various mobile applications and complicated architecture of MEC systems.

To address the above challenges, a natural way is to enable the systems to learn effective optimisation policies on their own. Deep Reinforcement Learning (DRL) [63], which combines Reinforcement Learning (RL) [118] with Deep Neural Networks (DNN) [59], delivers a promising solution to achieve the goal. DRL learns an effective policy (i.e., a mapping from environment states to actions) through interacting with the environment so as to maximize numerical rewards. With the help of the powerful representation ability of DNNs, DRL can effectively solve complex decision-making problems with large and high-dimensional state/action spaces. Recent breakthroughs in DRL have led to many successful applications in a wide range of areas including gaming [111, 112], robotics [65], networking [159, 24], etc., which inspired us to develop DRL-based methods for system optimisation problems that generally includes sequential decision-making processes. In this thesis, I focus on developing tailored DRL-based solutions for two vital and challenging problems in MEC: task offloading and service migration [102]. In the next section, I first give a brief review of the basic concept of MEC.

1.1 Multi-access Edge Computing

Over the past decades, the evolution of wireless communication networks has brought huge changes to every aspect of our lives, society, culture, politics, and economics. Back in the 1980s, the first generation (1G) of wireless cellular networks was developed with compliance of analogue modulation and mobility support and continued until being replaced by 2G digital telecommunications that were entirely digital. Next, 3G was proposed to provide better data transfer rate and multimedia application coherence by using radio access networks (RAN) [99]. By addressing the speed and network congestion issues of 3G networks, 4G can provide higher wireless speed and QoS [51]. In a nutshell, the main target of wireless systems from 1G to 4G is to pursue higher wireless speed to support the transition from voice-centric to multimedia-centric traffic. When it comes to 5G that has a similar transmission speed as its wireline counterparts, the mission becomes much more complex, aiming at supporting the explosive evolution of Information and Communications Technology (ICT) and the Internet. Specifically, 5G systems need to support multiple functionalities including communications, computing, control, and content delivery (4C) [77] and various emerging applications and services such as real-time online gaming, VR, AR, and ultra-high-definition video streaming. Therefore, 5G systems require unprecedented high access speed, computation and storage capability, and low latency.

Cloud computing is an important computing paradigm, which provides powerful centralized computing and storage capability for big data processing. Specifically, Cloud computing offers elastic resources and on-demand services for end-users over a Wide Area Network (WAN) without direct active management by users. Therefore, users can easily have seemingly unlimited resources without building and maintaining their computing infrastructures. Despite the remarkable benefits of cloud computing, it still has difficulties in satisfying the service demands of the new trend of delay-sensitive applications in 5G networks. First, end-users need to directly communicate with the centralised Cloud through a WAN that generally has high latency and is unlikely to improve in the foreseeable future [106]. Second, since the mobile users need to upload the data to the central Cloud for analysis, the volume of traffic data can bring a huge burden to the backhaul network. For example, there are thousands of video cameras deploying in an airport for security purposes, each of which produces data at 12 Mbps. If all video data is sent to the central Cloud, hundreds of Gbps bandwidth is required, which far exceeds the traffic capacity of current WANs. Last but not least, the long-distance data exchange between end-users and the central Cloud may carry a data security risk.

To address the above issues, MEC is proposed to offer Cloud capabilities and an IT service environment in the proximity of end-users [104]. MEC utilizes powerful servers deployed at the network edge for running applications and computation tasks. Typically, the MEC servers provide computing resources, storage capacity, connectivity and access to the Internet, thus it can significantly alleviate network congestion and reduce service latency. Fig. 1.1 shows the architecture of an emerging MEC system composed of the user level, edge level, and remote level. Here, the user level includes heterogeneous user devices (e.g., smartphone, camera, Virtual Reality (VR) glasses, and smart vehicle), the edge level contains MEC servers that provide edge computing services, and the remote level consists of Cloud servers. Specifically, user devices communicate with MEC servers through the RAN while MEC servers incorporate virtualisation infrastructures that provide computing, storage, and network resources. With the help of network function visualization (NFV), [38] and software-defined network (SDN) [54], MEC systems can offer various flexible and reliable services to end-users with high QoS guarantee.

MEC is a key enabling technology in the 5G network and beyond [153]. The trend of pushing cloud computing services to the network edge is expected to continue to accelerate in years to come. There are lots of challenges and open issues related to MEC systems including resource management, service orchestration, data management, and security. To overcome those challenges, numerous solutions have been proposed by both academia and industry to enhance the performance of MEC, such as modelling [110], multi-user resource

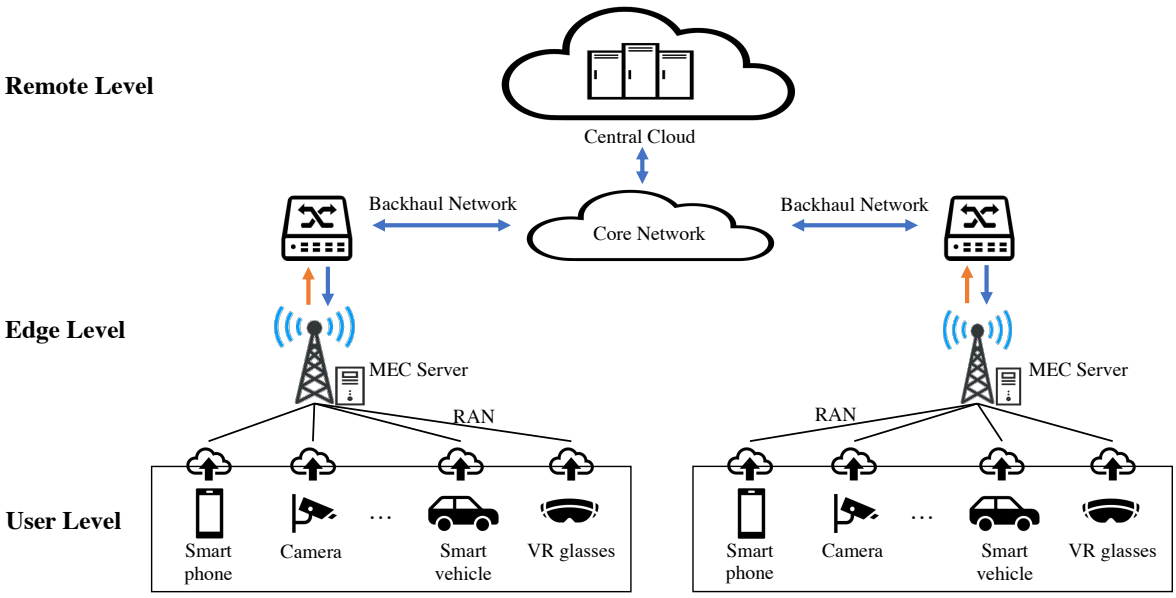


Fig. 1.1 Architecture of Multi-access Edge Computing.

allocation [130, 125], and system implementation [50], and so on and so forth. As the system architecture of MEC systems becomes increasingly sophisticated, it brings huge challenges for effectively optimising and maintaining the systems by human experts. To address the challenges, embedding machine learning to MEC systems becomes a trending solution [139, 160] because of the significant success achieved by modern artificial intelligence. In the next section, I give a brief overview of DRL, which is one of the flagships of machine learning.

1.2 Deep Reinforcement Learning

RL is a learning paradigm that learns through trial-and-error for sequential decision-making problems [118]. In the RL paradigm, a learning agent observes states from the environment and tries to make smart actions based on the observed states to maximise the total rewards as shown in Fig. 1.2. Although traditional RL methods have achieved some success in various areas [48], they are limited in domains where useful features can be extracted with expert knowledge, or to domains with fully observed, low-dimensional state spaces. To broaden the application of RL, DRL integrates DNNs, which have strong representation abilities towards high-dimensional datasets, for feature extraction. Learning with DNNs enables automatic feature engineering and end-to-end training via gradient descent, thus reliance on domain knowledge is significantly reduced or even removed.

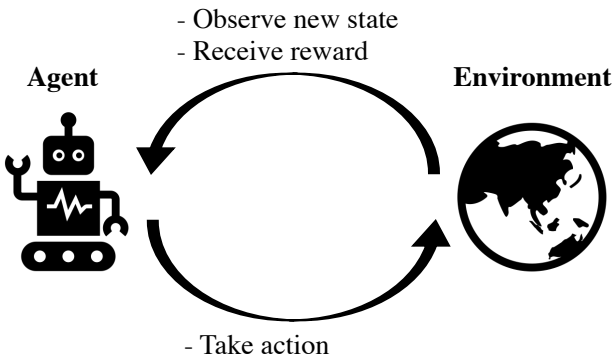


Fig. 1.2 An agent interacts with the environment, learning to make smart actions so that it can obtain maximal cumulative rewards.

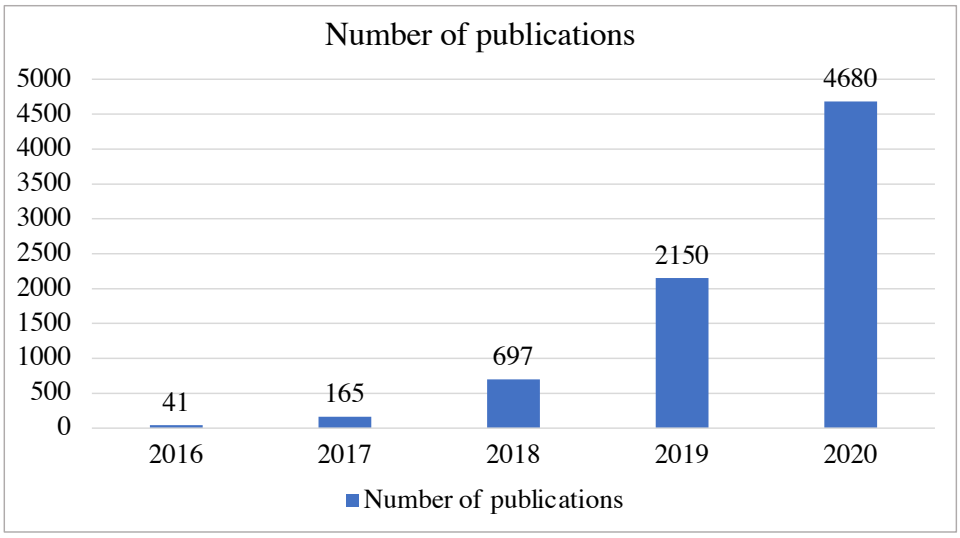


Fig. 1.3 The number of publications related to RL and edge computing systems from 2016 to 2020. (Source: google scholar)

The successes of DRL in multiple areas such as gaming [83, 112], robotics [164, 5], natural language processing [62, 64] have inspired researchers in the field of computer systems and networks to developed DRL-based methods for system optimisation [72, 160, 76]. Fig 1.3 illustrates the number of publications related to RL and edge computing from 2016 to 2020, showing a clear trend of the integration of RL and edge computing systems. Perusing recent studies in the field of MEC system optimisation, this thesis summarises the benefits of DRL-based solutions compared to traditional methods as follows:

- DRL can obtain effective solutions to sophisticated system optimisation problems by continuously learning from the real experiences of interacting with a system environment without complete system information and accurate analytical models.

- With the powerful representation abilities of DNNs, DRL is able to handle high-dimensional raw data collected from MEC systems (e.g., traffic data, mobile application structures, system logs, etc.) to improve decisions across heterogeneous system environments.
- Many system optimisation problems involve inherent properties of combinatorial optimisation. For example, the task offloading problem in MEC can be approximated by the knapsack problem [134]. However, traditional solutions highly rely on heuristics designed by human experts, which lack generic methods. On the contrary, DRL can automatically discover the hidden heuristics through learning thus reduce the reliance on expert knowledge.

Despite the above benefits of DRL-based solutions, efficiently applying DRL for system optimisation problems in MEC still face several challenges. In many cases, directly using off-the-peg DRL methods to address the system optimisation problems generally leads to unsatisfactory results. Efficient DRL problem formulations (i.e., proper definitions of the state, action spaces, and the reward function of DRL) are keys to achieve sufficiently good performance. Besides, the high sample complexity of DRL (i.e., many existing DRL methods require a large number of training samples to achieve good performance) hinders the application of existing DRL methods for system optimisation problems, since obtaining samples in real-world MEC systems can be costly. To address the above challenges, this thesis develops efficient RL problem formulation for different problems and tailors the DRL methods to better fit the properties of the MEC systems. In the following section, I will overview the research problems and the proposed novel DRL-based solutions for concrete system optimisation problems in MEC.

1.3 Research Problems, Challenges, and Objectives

System optimisation problems in MEC generally include sequential decision-making processes while DRL is one of the promising technologies to address these problems. However, effectively combining DRL methods with MEC systems still face some key challenges. This thesis focuses on address two crucial system optimisation problems in MEC: task offloading and service migration.

1.3.1 Problems and Challenges

Dependent Task Offloading based on Deep Reinforcement Learning

Task offloading (aka, computation offloading) is one of the key functionalities of MEC, which enables user devices to offload computation-intensive tasks of applications to MEC servers, thus reducing latency and energy consumption at user devices during the processing of the application. A crucial part of task offloading is to decide whether to offload or not. Typically, the offloading decisions for a mobile application can result in three schemes: 1) local execution: the whole application is done locally at the mobile device. 2) full offloading: the whole application is offloaded and processed by edge servers. 3) partial offloading: a part of the mobile application is processed locally while the rest is offloaded to edge servers. This thesis considers partial offloading scheme which is a very complex process affected by various factors including radio and backhaul connection quality, users' preference, workloads of mobile devices and cloud servers [92]. Specifically, many mobile applications are composed of dependent tasks where the outputs of some tasks are the input of others. In task offloading, we must decide whether a task should be offloaded to an MEC server, depending on the task profile (i.e., the required CPU cycles and input/output data size) and the MEC environment. Many existing studies [67, 28, 20, 29, 1] developed heuristic or approximation algorithms, since the above offloading problem is NP-hard. However, they rely heavily on expert knowledge or accurate analytical models. As a consequence, considerable human efforts and expertise are required to tune these heuristics or analytical models to adapt to new scenarios, which is time-consuming and even unrealistic due to the increasing complexity of applications and system architecture of MEC. To address the above issue, DRL has been adopted to handle task offloading problems in MEC [45, 131, 159, 168]. However, these methods assume the offloading tasks are independent without considering the inherent dependencies among tasks of real-world applications. In practice, many applications are composed of dependent tasks where the outputs of some tasks are the inputs of others. Ignoring the task dependencies when making task offloading decisions will severely affect the QoS of applications and waste the edge resources. How to designing DRL-based methods to learn effective offloading strategies? More specifically, how to extract the representative features of the tasks for better decision-making?

Fast Adaptive Task Offloading with Meta Reinforcement Learning

Although DRL-based methods can automatically learn effective offloading policy through interacting with the environment, the learned policy might have weak adaptability for unexpected perturbations or unseen situations (i.e., new environments). Full retraining to learn an updated policy is required for conventional DRL-based methods, which is time-consuming. Meta learning [127] is a promising method to address the aforementioned issues by leveraging previous experiences across a range of learning tasks to significantly accelerate learning

of new tasks. In the context of RL problems, meta reinforcement learning (MRL) aims to learn policies for new tasks within a small number of interactions with the environment by building on previous experiences. In general, MRL conducts two "loops" of learning, an "outer loop" which uses its experiences over many task contexts to gradually adjust parameters of the meta policy that governs the operation of an "inner loop". Based on the meta policy, the "inner loop" can adapt fast to new tasks through a small number of gradient updates [15].

There are significant benefits of adapting MRL to solving the computation offloading problem. Firstly, specific policies for new mobile users can be fast learned based on their local data and the meta policy. Secondly, MRL training in the MEC system can leverage resources from both the MEC servers and user devices. More specifically, training for the meta policy (outer loop) is run on the MEC server, and training for the specific offloading policy (inner loop) is processed on user devices. Normally, the "inner loop" training only needs several training steps and a small amount of sampling data, thus the user devices with limited computation resources and data are able to complete the training process. Finally, MRL can significantly improve the training efficiency in learning new tasks and make the offloading algorithm more adaptive to the dynamic MEC environment. However, how to effectively integrate the MRL method in MEC systems to achieve fast adaptive learning process for dynamic scenarios remains a big challenge.

Online Service Migration with Incomplete System Information

Another crucial problem in MEC is service migration. In general, a mobile application includes two parts: a front-end component running on mobile devices, and a back-end service that runs the tasks offloaded from the application on MEC servers [102]. When considering the user mobility along with the limited coverage of MEC servers, the communications between a mobile user and the user service running on an edge server may go through multiple hops, which would severely affect the QoS and even interrupt the ongoing services. To address this problem, the service could be dynamically migrated to a more suitable MEC server so that the QoS is maintained. Unfortunately, finding an optimal migration policy for such a problem is non-trivial, due to the complex system dynamics and user mobility. Many existing works [94, 135, 147, 137, 91] proposed service migration solutions based on MDP or Lyapunov optimisation under the assumption of knowing the complete system-level information (e.g., available computation resources of MEC servers, profiles of offloaded tasks, and backhaul network conditions). Thus, they designed centralized controllers (i.e., controllers are placed on edge servers or central cloud) that make migration decisions for mobile users in the MEC system.

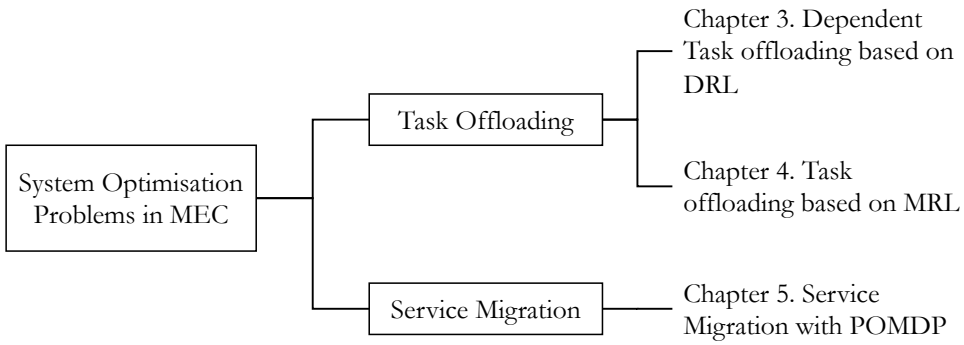


Fig. 1.4 Key contents of the thesis.

The aforementioned methods have two potential drawbacks: 1) in a real-world MEC system, gathering complete system-level information can be difficult and time-consuming; 2) the centralized control approach will have the scalability issue since its time complexity rapidly increases with the number of mobile users. To address the above issues, some works proposed decentralized service migration methods based on contextual Multi-Armed Bandit (MAB) [116, 93, 115], where the migration decisions are made by the user side with partially observed information. However, they did not consider the intrinsically large state space and complex dynamics in the MEC system, which may lead to unsatisfactory performance. Therefore, we need a new online service migration framework based on incomplete system information to address the above challenges.

1.3.2 Objectives

The goal of this thesis is to develop learning-based system optimisation methods for MEC systems, letting machine "learn to optimise on its own". In particular, this research aims to 1) build a policy gradient method for the dependent task offloading problem, 2) enhance the proposed task offloading method with meta reinforcement learning for fast adaptive learning, 3) build a DRL-based method combining recurrent neural network (RNN) to tackle the service migration problem with incomplete system information. Further details will be outlined in Section 1.4.

1.4 Thesis Organisation and Contributions

The rest of the thesis is organised as follows. Chapter 2 first gives a brief review of RL, DRL, MRL, and RL with partially observable environments and then discusses the state-of-the-art related work about task offloading, service migration, and learning-based combinatorial

optimisation. As shown in Fig 1.4, from Chapter 3 to Chapter 5, I present the detailed learning-based methods to address the task offloading and service migration problems in MEC. In Chapter 6, I summarise the thesis with a list of suggestions for future research directions. The outlines of contributions of each chapter are illustrated as follows:

- Chapter 2 gives a thorough review of deep reinforcement learning methods including conventional RL, value-based DRL, policy-based DRL, MRL, and RL with partially observable environments. Besides, the state-of-the-art solutions for task offloading and service migration are discussed. Furthermore, since both task offloading and service migration can be seen as combinatorial optimisation problems, I present the state-of-the-art learning-based combinatorial optimisation algorithms at the end of this chapter.

- In Chapter 3, I propose a new DRL-based Task Offloading (DRLTO) scheme leveraging off-policy RL empowered by a tailored Sequence-to-Sequence (seq2seq) neural network. The DRLTO is able to reduce the latency of running applications and the energy consumption at user devices [131]. In DRLTO, the task offloading problem is modelled as a Markov Decision Process (MDP). Applications are represented by Directed Acyclic Graphs (DAG), where vertices and edges denote tasks and their dependencies, respectively. To effectively extract the key features of task dependencies, a tailored seq2seq neural network is developed to represent the policy and value function of the MDP. Specifically, the input of the seq2seq neural network is the DAG represented by a sequence of embedding vectors, while the output is the offloading plan for the DAG. To improve the training efficiency, I develop an off-policy DRL algorithm with a clipped surrogate objective that can prevent the training from getting stuck in local optima and stabilize the training process. The DRLTO learns to make efficient offloading decisions through directly interacting with the environment and only requires minimal expert knowledge.

- In Chapter 4, I propose an MRL-based method that synergizes the first-order MRL algorithm with a seq2seq neural network [132]. The proposed method learns a meta offloading policy for all user devices and fast obtains the effective policy for each user device based on the meta policy and local data. There are three major benefits of my method: 1) personalized offloading policies for new mobile users can be fast learned based on their local data and the meta policy. 2) the training of my method in the MEC system can leverage resources from both the MEC servers and the mobile devices. 3) my method can significantly improve the training efficiency facing new environments and make the offloading algorithm more adaptive to the dynamic MEC scenarios.

- Chapter 5 investigates the service migration in MEC systems. I develop a new learning-driven method that is user-centric and can make effective online migration decisions given incomplete system-level information. In particular, the service migration problem is modelled as a Partially Observable Markov Decision Process (POMDP). To solve the POMDP, I design an encoder network that combines a Long Short-Term Memory (LSTM) and an embedding matrix for the effective extraction of hidden information. I then propose a tailored off-policy actor-critic algorithm with a clipped surrogate objective for efficient training. Results from extensive experiments based on real-world mobility traces demonstrate that my method consistently outperforms both the heuristic and state-of-the-art learning-driven algorithms while achieving near-optimal results on various MEC scenarios.

To summarise the key contents, this thesis researches the system optimisation problems in MEC, especially task offloading and service migration problems. The proposed DRL-based solutions are expected to contribute positively to the system optimisation and maintenance in MEC. In the next chapter, a comprehensive review of DRL backgrounds, and existing solutions for task offloading and service migration is provided. In addition to this, each primary chapter also makes some extra efforts to explain its contributions and most related works to make clear the contents in the chapter.

Chapter 2

Backgrounds

In this chapter, I first give a brief review of deep reinforcement learning (DRL) including conventional reinforcement learning (RL) methods, model-free DRL, meta reinforcement learning (MRL), and DRL with partial observable environments. Next, I present the state-of-the-art work related to task offloading, service migration, and learning-based combinatorial optimisation methods.

2.1 Primer of Deep Reinforcement Learning

RL can solve sequential decision-making problems by learning from interaction with the environment. In general, RL models a learning task $\mathcal{T}$ as a Markov Decision Process (MDP), which is defined by a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{P}_0, \mathcal{R}, \gamma)$, to represent the interaction between a learning agent and its environment. Specifically, $\mathcal{S}$ is the state space, $\mathcal{A}$ denotes the action space, $\mathcal{P}(\cdot|S_t, A_t)$ is the environment dynamics that gives the probability distribution of the next state given the current state and action, $\mathcal{P}_0$ is the distribution of the initial state, $\mathcal{R}(S_t, A_t)$ represents the reward function, and $\gamma \in [0, 1]$ is the discounted factor. The policy, $\pi(\cdot|s_t)$, represents the distribution over actions given a state s_t. I define a trajectory sampled from the environment according to the policy π as, the The return from state s_0 following policy π, which is defined as $G_t(\tau_\pi) = \sum_{i=t}^{T} \gamma^{i-t} r_t$, is the sum of discounted rewards along a trajectory $\tau_\pi := \{s_0, a_0, r_0, s_1, a_1, r_1, ..., s_T, a_T, r_T\}$. The goal of RL is to find an optimal policy π^*, so that the expected return, $\mathbb{E}_{\tau \sim p(\tau|\pi^*)}[G_0(\tau)]$, is maximal. The value functions (i.e., state-value function and action-value function) are used to describe how good a state or state-action pair is in RL. The sate-value function is defined as the expected total reward starting from state s,

which is formally given by

$$V_\pi(s) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^k R(S_t, A_t) \Big| S_0 = s \right]. \tag{2.1}$$

Likewise, the action-value function is defined by the expected return after taking an action a in state s and thereafter following policy π, which is formally denoted as

$$Q_\pi(s,a) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t R(S_t, A_t) \Big| S_0 = s, A_0 = a \right]. \tag{2.2}$$

An optimal state-value function $V^*(s)$ is the maximum state value achievable by any policy for state s, which is defined as $V^*(s) = \max_\pi V_\pi(s) = \max_a Q_{\pi^*}(s,a)$. The optimal state-value function can be decomposed into the Bellman equation:

$$V^*(s) = \max_a \sum_{s',r} p(s',r|s,a) \left[r + \gamma V^*(s') \right]. \tag{2.3}$$

Here, $p(s',r|s,a)$ is the joint distribution of dynamics $\mathcal{P}$ and the reward function $\mathcal{R}$ (we can treat the reward function as a conditional distribution given current state and action), s' denotes the next state of the current state s after taking action a.

Similarly, the optimal action-value function, $Q^*(s,a) = \max_\pi Q_\pi(s,a)$, is the maximum action value achievable by any policy for state s and action a. The Bellman optimality equation for $Q^*(s,a)$ is

$$Q^*(s,a) = \sum_{s',r} p(s',r|s,a) \left[r + \gamma \max_{a'} Q^*(s',a') \right], \tag{2.4}$$

where a' is the next-step action. Specifically, the Bellman optimality equations of V^* and Q^* express that the value of a state under an optimal policy must equal the expected return for the best action from that state:

$$\begin{aligned} V^*(s) &= \max_{a \in \mathcal{A}} Q^*(s,a) \\ &= \max_a \mathbb{E}_{\pi^*} \left[G_t \big| S_t = s, A_t = a \right] \\ &= \max_a \mathbb{E}_{\pi^*} \left[R_{t+1} + \gamma G_{t+1} \big| S_t = s, A_t = a \right] \\ &= \max_a \mathbb{E} \left[R_{t+1} + \gamma V^*(S_{t+1}) \big| S_t = s, A_t = a \right] \\ &= \max_a \sum_{s',r} p(s',r|s,a) \left[r + \gamma V^*(s') \right]. \end{aligned} \tag{2.5}$$

$$
\begin{aligned}
Q^*(s,a) &= \max_a \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} Q^*(S_{t+1}, a') \middle| S_t = s, A_t = a\right] \\
&= \sum_{s',r} p(s', r|s, a)\left[r + \gamma \max_{a'} Q^*(s', a')\right].
\end{aligned} \tag{2.6}
$$

Fig. 2.1 shows graphically the spans of future states and actions considered the Bellman optimality equations for $V^*(s)$ and $Q^*(s,a)$, where each open circle represents a state and each solid circle represents a state–action pair. Starting from the root node s, the agent could take an action $a \in \mathcal{A}$ following the policy π. From each of these actions, the environment returns one of several next states, s', along with reward r, based on the system dynamics. The optimal Bellman equations V^* and Q^* recursively choose the branch with maximal discounted accumulated reward over all the possibilities.

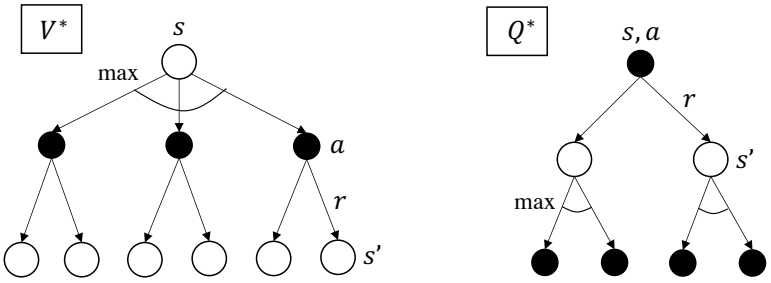


Fig. 2.1 Backup diagrams for V^* and Q^*.

Conventional RL algorithms aim to get the optimal value function of every state in the environment. When the model is known ahead, due to the Bellman optimality of the value function, obtaining the optimal value function becomes a planning problem which can be solved by **Dynamic Programming (DP)** [12]. When the model is unknown, sampling methods can be instead used. One common method is **Monte Carlo (MC)** method [39] which solves reinforcement learning problem based on averaging sample returns. Another common method is **Temporal Difference (TD)** [122] which learns by bootstrapping from the current estimate of the value function. However, all of these methods use the tabular way to store the state values. Hence, it is intractable for them to solve problems with extremely large or high dimensional state space.

In the past decade, deep learning (DL) has achieved remarkable successes in various areas, due to the advance of modern computing technologies and the strong representation ability of deep neural network (DNN). To address the curse of dimensionality in conventional RL algorithms, DRL integrates DNN to handle large and high-dimensional state space. From the perspective of the model dynamics, DRL methods can be roughly classified into model-based technologies [47, 84], which build a predictive model of an environment and derive a

controller from it, and model-free approaches [83, 37], which learns a direct mapping from states to actions. In the following subsection, I give an overview of existing model-free DRL algorithms.

2.1.1 Model-free Deep Reinforcement Learning

This thesis focuses on developing model-free DRL methods that learn effective policies without relying on a model, which generally falls into one of two categories: value-based methods and policy-based methods.

Value-based method: the valued-based DRL methods (e.g., deep Q-learning (DQL) [83] and double DQL [126]) use the deep neural network to approximate the optimal action-value function, $Q^*(s_t, a_t; \theta^Q)$ where θ^Q are parameters of the deep neural network. They obtain the optimal policy by greedily selecting the action with maximal action value, where $a_t = \arg\max_a Q^*(s_t, a; \theta^Q)$. In DQL, experiences $e_t = (s_t, a_t, r_t, s_{t+1})$ at each time step t are stored in the replay buffer D. The Q-network (i.e., a neural network that is used approximate the action-value function) is trained based on samples (or minibatches) of experience $(s_t, a_t, r_t, s_{t+1}) \sim U(D)$, drawn uniformly at random from the replay buffer. Formally, the training objective of the DQL is expressed as

$$L_Q(\theta^Q) = \mathbb{E}_{(s_t, a_t, r_t, s_{t+1}) \sim U(D)} \left[\left(r_t + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}; \theta^Q) - Q(s_t, a_t; \theta^Q) \right)^2 \right]. \tag{2.7}$$

However, since valued-based methods indirectly obtain a deterministic policy by training the Q-network, it generally has a low convergence rate [82]. The complex state space and large action space of the MEC environment exacerbate this issue. Besides, the training targets of value-based methods are generally obtained by one-step bootstrapping of the Q-network, which can be a highly biased estimation of the true action values. Introducing bias may harm the convergence of the algorithm, or cause converging to sub-optimal solutions. the above issues make value-based methods unfit to solve the system optimisation problems in MEC since the learned policies may lead to unsatisfied performance.

Policy-based methods: This thesis builds the system optimisation methods for MEC based on the policy-based methods. In contrast to the value-based methods, the policy-based methods (e.g., asynchronous actor-critic [82] and proximal policy optimization [108]) directly optimise the target (i.e., discounted total reward) and provide good convergence property for dealing with the complex state and action space of the environment. They parametrised the stochastic policy with a deep neural network rather than using a deterministic policy derived from the action-value function. The parameters of the policy network are updated by

performing gradient ascent on $\mathbb{E}[G_0(\tau)]$, which is formally defined by

$$\nabla_\theta \mathbb{E}_{\tau\sim p(\tau|\pi_\theta)}\left[\sum_{t=0}^{T}\gamma^t r_t\right] = \mathbb{E}_{\tau\sim p(\tau|\pi_\theta)}\left[\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_t|s_t) Q_{\pi_\theta}(s_t,a_t)\right]. \tag{2.8}$$

The core idea is of the policy-based methods is to estimate the gradient based on the trajectories sampled by the current policy. To estimate the state-action value function $Q_{\pi_\theta}(s_t,a_t)$, one solution is to apply MC method which samples multiple trajectories and use the return, G_t, as an unbiased estimate of $Q_{\pi_\theta}(s_t,a_t)$. The update rule of the policy is defined by

$$\theta \leftarrow \theta + \alpha \sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_t|s_t) G_t, \tag{2.9}$$

where α is the learning rate. Eq. (2.9) leads to a typical policy-based method, REINFORCE [144].

Although, REINFORCE has good theoretical convergence properties since it is a stochastic gradient method. The Monte Carlo estimation of the state-action value function introduces high variance and thus produces slow learning. To address this problem, a standard solution is to add an arbitrary *baseline* $b(s)$ [143]:

$$\theta \leftarrow \theta + \alpha \sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_t|s_t)(G_t - b(s_t)). \tag{2.10}$$

Here, the baseline $b(s_t)$ can be any function that it does not vary with action a. The baseline leaves the average value of the update unchanged, but it can significantly reduce the variance. There are some common choices of the baseline functions. For example, the time-based baseline aligns multiple trajectories on the same time step and calculates the average return at each time step as the baseline. Another natural choice for the baseline is an estimate of the state value function, $V(s_t;\theta^V)$, which is approximated by another neural network with parameters θ^V [82]. This results in a new update equation as:

$$\theta \leftarrow \theta + \alpha \sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_t|s_t) A(s_t,a_t), \tag{2.11}$$

where $A(s_t,a_t) := Q(s_t,a_t) - V(s_t;\theta^V)$ is the **advantage function**, which measures whether or not the action is better or worse than the policy's default behaviour.

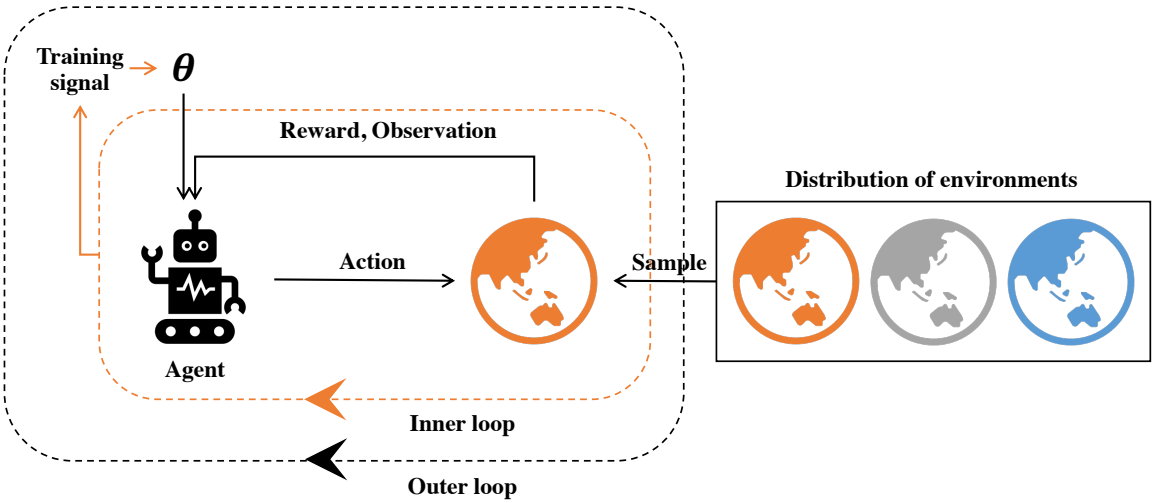


Fig. 2.2 Schematic of MRL, showing the two loops of training.

2.1.2 Meta Reinforcement Learning

The concept of meta learning, or learning to learn, has recently become a hot research topic [43]. In contrast to the conventional DL methods which solve complex tasks from scratch, meta learning provides a new learning paradigm that learns how to adapt new tasks faster by reusing previous experience. More specifically, a typical meta learning algorithm includes two loops of training, where the "inner loop" training solves a task defined by a dataset and objective, while the "outer loop" learns a meta model by using experiences across a distribution of learning tasks. The learned meta model can be used to speed up the training of the "inner loop". This new paradigm provides a variety of benefits. For example, meta learning can fast adapt to new learning tasks with few training samples and update steps, which has higher sample and computation efficiency than conventional DL methods.

MRL enhances the conventional RL methods with meta learning, which aims to learn a learning algorithm that can quickly find the policy for a learning task $\mathcal{T}_i$ drawn from a distribution of tasks $\rho(\mathcal{T})$. Each learning task $\mathcal{T}_i$ corresponds to a different environment, and these environments typically share the same state and action spaces but may differ in reward functions or their dynamics (i.e., $\mathcal{P}$ and $\mathcal{P}_0$). Similar to meta learning, the training of MRL generally contains two loops of optimisation, as illustrated in Fig. 2.2. The "outer loop" samples a new environment in every iteration and adjusts parameters θ that are used to determine the agent's behaviour. In the inner loop, the agent interacts with the environment and optimizes for the maximal reward based on the parameters θ. Here, θ represents the common experience that is extracted from multiple learning tasks.

Recent years have brought a wealth of methods focused on different aspects of the learning process of MRL. One typical example is context-based MRL [30, 133], which uses the recurrent neural network (RNN) to aggregate experiences of a range of learning tasks into a latent representation and quickly adapts new task learning conditioned on the aggregated experience. A second approach is gradient-based MRL [31, 103], which aims to learn initial parameters θ of policy neural network so that performing a single or few steps of policy gradient over θ with the given new task can lead to the optimal policy for that task. This research follows the formulations of model-agnostic meta-learning (MAML) [31], giving the target of gradient-based MRL as

$$J(\theta) = \mathbb{E}_{\mathcal{T}_i \sim \rho(\mathcal{T})} \left[J_{\mathcal{T}_i}(\theta') \right], \text{with } \theta' := U(\theta, \mathcal{T}_i), \tag{2.12}$$

where $J_{\mathcal{T}_i}$ denotes the objective function of task $\mathcal{T}_i$. For example, when using vanilla policy gradient (VPG), $J_{\mathcal{T}_i}(\theta) = \mathbb{E}_{\tau \sim P_{\mathcal{T}_i}(\tau|\theta)} \sum_{t=0} \left(\gamma^t r_t - b(s_t) \right)$, where $b(s_t)$ denotes an arbitrary baseline. U denotes the update function which depends on $J_{\mathcal{T}_i}$ and the optimization method. For instance, if we conduct k-step gradient ascent for $\mathcal{T}_i$, then $U(\theta, \mathcal{T}_i) = \theta + \alpha \sum_{t=1}^{k} g_t$, where g_t denotes the gradient of $J_{\mathcal{T}_i}$ at time step t and α is the learning rate. Therefore, the optimal parameters of policy network and update rules are

$$\begin{aligned} \theta^* &= \arg\max_{\theta} \mathbb{E}_{\mathcal{T}_i \sim \rho(\mathcal{T})} \left[J_{\mathcal{T}_i}(U(\theta, \mathcal{T}_i)) \right], \\ \theta &\leftarrow \theta + \beta \mathbb{E}_{\mathcal{T}_i \sim \rho(\mathcal{T})} \left[\nabla_\theta J_{\mathcal{T}_i}(U(\theta, \mathcal{T}_i)) \right], \end{aligned} \tag{2.13}$$

where β is the learning rate of "outer loop" training. The gradient-based MRL has good generalization ability. However, the second-order derivative in MAML may bring huge computation costs during training, which is inefficient. In addition, when combining with a complex neural network architecture (e.g., a seq2seq neural network), the implementation of second-order MAML becomes intractable. To address these challenges, some algorithms [31, 90] use the first-order approximation to MAML target.

2.1.3 Reinforcement Learning with Partially Observable Environments

MDP assumes that states include complete information for decision-making. However, in many real-world scenarios, observing such states is intractable. Therefore, the Partially observable Markov Decision Process (POMDP), an extension of MDP, is proposed as a general model for the sequential decision-making problem with a partially observable environment, which is defined by a tuple $(\mathcal{O}, \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \mathcal{U}, \gamma)$. Fig. 2.3 shows the graphical model of POMDP. Specifically, the state $s_t \in \mathcal{S}$ is latent and the observation $o_t \in \mathcal{O}$ contains

partial information of the latent state s_t. $\mathcal{U}(o_t|a_{t-1}, s_t)$ represents the observation distribution, which gives the probability of observing o_t if action a_{t-1} is performed and the resulting state is s_t.

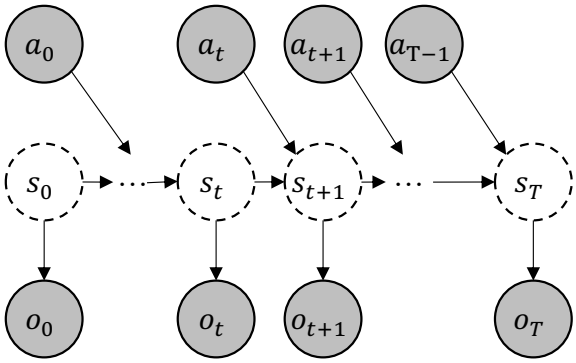


Fig. 2.3 Graphical model of POMDP.

Since the state is latent, the learning agent cannot choose its action directly based on the state. Alternatively, it has to consider a complete history of its past actions and observations to choose its current action. Specifically, the history up to time step t is defined by $H_t = \{o_0, a_0, ..., o_{t-1}, a_{t-1}, o_t\}$. Therefore, the key for RL-based methods to solve the POMDP is how to effectively infer the latent state based on the history, which is defined by $p(s_t|o_{\leq t}, a_{<t})$. In the literature, some RL methods [40, 166] assume the latent states as deterministic states, which encode the whole history by RNN and use the hidden state of RNN as input to the policy. Other works [142, 46, 163] explicitly infer the filtering distribution, $b_t := p(s_t|o_{\leq t}, a_{<t})$, named the **belief state**, which is defined by the distribution over latent states (i.e., stochastic latent state) given the history and sampling latent state from the distribution as input to the policy. Formally, the update rule of the belief state is given by

$$b_{t+1} = \frac{\int b_t \mathcal{U}(o_{t+1}|s_{t+1}, a_t)\mathcal{P}(s_{t+1}|s_t, a_t)\mathrm{d}s_t}{\int\int b_t \mathcal{U}(o_{t+1}|s_{t+1}, a_t)\mathcal{P}(s_{t+1}|s_t, a_t)\mathrm{d}s_t\mathrm{d}s_{t+1}}. \tag{2.14}$$

Due to the fact that deriving the belief state requires the model dynamics $\mathcal{U}$ and $\mathcal{P}$, I mainly apply RNN for the latent information extraction, which can achieve excellent performance and is much easier to be implemented in MEC scenarios compared to methods based on inferring the belief state.

2.2 Related work

One of the fundamental functionality of MEC systems is to offer computing services to the network edge. To improve the Quality of Service (QoS), appropriate task offloading and service migration strategies are of vital importance. In recent years, task offloading and service migration have been receiving intensive research interests from both the academic and industrial fields. From the theoretical perspective, both task offloading and service migration problems in MEC systems can be seen as the combinatorial optimisation problem. Machine learning (ML) methods have been widely applied to solve the combinatorial optimisation problem [78, 129, 19], which inspires us to apply DRL to address task offloading and service migration problems. In the following subsections, I provide a holistic view of the broad landscape of the contributions made so far in the literature related to the above topics.

2.2.1 Task Offloading in MEC

In MEC systems, due to the limited computation resources of mobile devices, it is crucial to offload the computation-intensive tasks to the powerful edge servers, especially for those delay-sensitive applications including virtual reality, augmented reality, and face recognition. From the angle of the offloading task model, the existing works can be divided into two groups: task model for binary offloading and that for partial offloading [77].

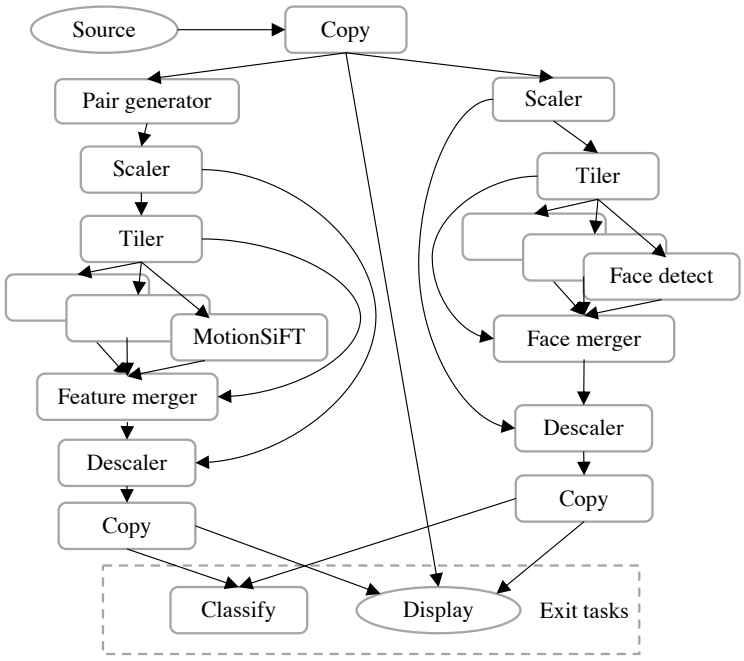


Fig. 2.4 An example of DAG for the gesture recognition application.

Binary Offloading: In the task model for binary offloading, the task cannot be partitioned and has to be executed as a whole either locally at the mobile device or offloaded to the MEC server. Note that, there is no inner dependency among tasks, thus a task is generally represented by a tuple with three elements: the task input size, the computation density (in CPU cycles per bit), and the task output size. These parameters are related to the nature of the applications and can be estimated through task profiles [80]. These three parameters can help capture the essential features (e.g., the computation and communication demands) of mobile applications. Besides, to help design effective algorithms, these parameters can facilitate a simple evaluation of the latency and energy consumption of the task.

Although this binary offloading model is seemingly simple, it is still NP-hard in most MEC scenarios. Hence, many existing works are highly based on heuristic algorithms. Dinh *et al.* [28] aimed to find an offloading plan for a set of tasks among different access points and MEC hosts, in order to achieve the minimal joint target of latency and energy. Chen *et al.* [20] focused on computation offloading for independent tasks in a software-defined ultra-dense network. They formulated the task offloading problem as Mixed-Integer Non-Linear Programming (MINLP) and solved it by using decomposition and heuristic methods. Hong *et al.* [42] proposed an approximate dynamic programming algorithm for computation offloading to achieve the optimal quality of experience. Zeng *et al.* [157] formulated the task offloading problem with joint consideration of task scheduling and image placement as an MINLP problem, and proposed a three-stage heuristic algorithm to solve it.

Despite the above heuristic-based solutions, convex-optimisation based methods are widely applied to address the binary offloading problem. In general, most binary offloading problems in MEC systems are non-convex. The core idea of many existing methods based on convex optimisation is to apply approximation or relaxation methods to convert the original problem into a convex optimisation problem and analyse the gap between the original problem and the simplified one. For example, Li *et al.* [66] proposed a drift-plus-penalty based Lyapunov optimisation approach to convert the original problem into an upper bound optimisation problem, which is later relaxed to a convex problem and solved by a convex optimisation method. Wang *et al.* [130] aimed at jointly tackling computation offloading, content caching, and resource allocation problems. They convert the original problem as a convex problem by using some relaxation and solve it based on decomposition and alternating direction method.

Different from the above methods that rely heavily on expert knowledge, DRL has recently been widely applied to solve the binary offloading problem. Li *et al.* [61] proposed a DQL-based offloading method to jointly optimize the offloading decision and computational resource allocation. Chen *et al.* [22] considered an ultra-dense network, where multiple base

stations can be selected for offloading. They also adopted deep Q-Learning to obtain the offloading strategy. Huang *et al.* [45] proposed a DRL-based offloading scheme in which both offloading decisions and resource allocations are considered. Dinh *et al.* [27] focused on multi-user multi-edge-node task offloading problem by using Q-learning in MEC. Zhan [159] *et al.* formulated the task offloading problem as a partially observable Markov decision process (POMDP) and applied a policy gradient DRL-based approach to solving the problem. Zou *et al.* [168] proposed a DRL-based offloading method by utilizing the asynchronous advantage actor-critic algorithm, to reduce the latency and energy consumption. Tan *et al.* [120] proposed a DQL-based offloading method considering constraints of limited resources, vehicle's mobility, and delay. The task model for binary offloading assumes the tasks are independent. However, most real-world applications consist of dependent tasks, ignoring the task dependencies when making offloading decisions can lead to severe performance degradation.

Partial Offloading: In the task model for partial offloading, applications were composed of tasks with inner dependencies, which is able to achieve a fine granularity of computation offloading, leading to better offloading performance. In general, those applications are modelled as Directed Acyclic Graphs (DAGs), where vertices and edges represent tasks and the inner dependencies, respectively. Fig 2.4 shows an example of DAG for the gesture recognition application. To address the partial offloading problem, heuristic algorithms have been widely applied. Wang *et al.* [138] modelled both the applications and the computing system as graphs and proposed an approximation algorithm for finding the task offloading plan to obtain the lowest cost. Neto *et al.* [88] implemented a user-level online offloading framework for Android applications, aiming at minimizing the remote execution overhead. Zanni *et al.* [156] proposed an innovative task selection algorithm for Android applications, achieving method-level granularity of offloading. Lin *et al.* [67] represented mobile applications as task graphs and proposed a heuristic-based algorithm to solve the task offloading problem in MEC. Yang *et al.* [150] proposed two types of offloading algorithms based on linear programming and group intelligence heuristic algorithm to handling task offloading problems in the Industrial Internet. These existing solutions rely on hand-tuned heuristic or approximation methods. However, turning heuristics for a given task offloading scenario is an expensive job that requires considerable human expertise. Consequently, facing the dynamic MEC scenarios, one might have to do this tuning repeatedly, which is time-consuming and sometimes impractical.

To address the above issue, some recent works introduced DRL methods to solve the partial offloading problems. As far as I know, our previous work [131] is the first to address the offloading problem with dependent tasks by combining DRL algorithms and sequence

to sequence (seq2seq) neural network. Very recently, Yan *et al.* proposed an actor-critic method that jointly optimised the offloading decision and the resource allocation under time-varying wireless fading channels and stochastic edge computing capability, considering task dependencies. In Chapters 3 and 4, this thesis presents the proposed DRL-based and MRL-based methods for partial offloading in detail.

2.2.2 Service Migration in MEC

Considering the users' mobility and the limited coverage of the MEC server, the communication between user devices and the MEC server might need to go through multiple hops, which may severely degrade the QoS. In order to mitigate the negative effects, services (e.g., virtual machines, containers, virtualized function, etc.) running on the MEC server must be dynamically migrated to a better placement with minimal running cost. Such requirements for migration in MEC systems can be observed in the advent of the "Follow Me" trend, where the new notions such as Follow Me Cloud [119, 94], Follow Me Edge-Cloud [4], and Move with Me [17] have recently emerged in the literature. However, finding an optimal migration policy for such a problem is highly challenging since it depends on several aspects including the distance between mobile devices and edge servers, the size of the service to be migrated, the workloads of the edge servers, so on and so forth.

Service migration in MEC has attracted intensive research interests in recent years. Rejiba *et al.* [102] published a comprehensive survey on mobility-induced service migration in fog, edge, and related computing paradigms. Most existing methods solve the service migration problem based on the time-slotted model which can be regarded as a sampled version of a continuous-time model. At each time slot, the algorithm needs to decide the placements of the services in the MEC system. Whenever a migration decision has to be taken, a trade-off has to be made between the potential benefits after the migration (e.g., the improvement of QoS) and the cost of migrating services. The key of service migration algorithms is to effectively address and model these trade-offs. Existing methods use different approaches to handle this issue including MDP [137], Markov chains [32], Mixed Integer Linear Programming (MILP) [113], Mixed-Integer Quadratic Programming (MIQP) [151], etc. Based on the placement of the decision-making logic, I roughly classify the related work into centralized control approach (the central cloud or MEC servers make service migration decisions for all mobile users) and decentralized control approach (each mobile user makes its own migration decisions).

Centralized control approach: Plenty of works focused on making centralized migration decisions (i.e., the migration decisions are made by ether central cloud or edge servers) based on the complete system-level information to minimize the total cost. Ouyang *et al.* [94]

converted the service migration problem as an online queue stability control problem and applied Lyapunov optimization to solve it. Ning *et al.* [91] formulate the service migration problem by jointly considering the constraints of server storage capability and service execution latency. They utilize Lyapunov optimization and distributed Markov approximation to enable dynamic service placement. Liu *et al.* [68] propose a multi-agent RL based method for the service migration where agents represent the controllers of MEC servers. Xu *et al.* [149] formulated the service migration problem as a multi-objective optimization framework and proposed a method to achieve a weak Pareto optimal solution. Wang *et al.* [137] formulated the service migration problem as a finite-state MDP and proposed an approximation of the underlying state space. They solve the finite-state MDP by using a modified policy-iteration algorithm. Other recent works tackled the service migration problem based on RL. Wang *et al.* [135] proposed a Q-learning based micro-service migration algorithm in mobile edge computing. Wu *et al.* [147] considered jointly optimizing the task offloading and service migration, and proposed a Q-learning based method combing the predicted user mobility. These works considered the case where the decision-making agent knows the complete system-level information. However, in a practical MEC system, collecting complete system-level information can be difficult and time-consuming. Moreover, the centralized control approach may suffer from the scalability issue when facing a rapidly increasing number of mobile users.

Decentralized control approach: some studies proposed to make migration decisions by the user side based on incomplete system-level information. Ouyang *et al.* [93] formulated the service migration problem as an MAB and proposed a Thompson-sampling based algorithm that explores the dynamic MEC environment to make adaptive service migration decisions. Sun *et al.* [115] proposed an MAB-based service placement framework for vehicle cloud computing, which can enable the vehicle to learn to select effective neighbouring vehicles for its service. Sun *et al.* [116] developed a user-centric service migration framework using MAB and Lyapunov optimization to minimize the latency with constraints of energy consumption. These methods simplify the system dynamics by modelling with MAB, which ignores the inherently large state space and complex transitions among states in a real-world MEC system. Distinguished from the above works, the proposed method models the service migration problem as a POMDP that has a continuous state space and models complex transitions between states. Moreover, the proposed method is model-free and adaptive to different scenarios, which can learn to make online service migration decisions with minimal expert knowledge. More recently, Yuan *et al.* [155] investigated the joint service migration and mobility optimization problem for vehicular edge computing. They modelled the MEC environment as a POMDP and proposed a multi-agent DRL method based on

independent Q-learning to learn the policy. However, using Q-learning based method to solve the environment with complex dynamics and continuous state space can be unstable and inefficient. In this thesis, the proposed method is implemented based on the policy-based method, which can achieve better performance than Q-learning based methods.

2.2.3 Learning-based Combinatorial Optimisation

Task offloading and service migration problems are generally modelled as combinatorial problems that can be solved by heuristic algorithms. Such heuristics are designed by domain experts, thus can be suboptimal due to the high complexity of the problems. To overcome this issue, one of the emerging trends over the past years is to solve combinatorial optimisation problems by using ML. An intuitive way is to train an ML model on the datasets that contain solutions generated by different solvers. The trained ML model can then be used to solve new problems with model inference. For example, Pointer networks [129] models the conditional probability of an output sequence with elements that are discrete tokens corresponding to positions in an input sequence, which can be trained to output satisfactory solutions to one type of combinatorial optimization problem where the output elements are selected from inputs, e.g., Travelling Salesman Problem (TSP).

Although Pointer networks can achieve good performance for some combinatorial optimisation problems, it still requires labeled data (i.e., solutions generated by existing solvers) for training. To overcome this issue, Bello *et al.* [13] proposed a framework to tackle combinatorial optimisation problems using Pointer networks and RL. Their method significantly outperformed the supervised learning approach (i.e., training Pointer networks with supervised learning) and obtained close to optimal results. Kool *et al.* [55] adopted attention layers rather than Pointer Networks as the policy neural network for solving TSP and obtained better performances compared to Pointer networks in multiple combinatorial optimisation problems. Very recently, there is a surge of interest in using graph neural networks (GNNs) [165] as a key building block for combinatorial tasks, either directly as solvers or by enhancing exact solvers [19, 11]. Dai *et al.* [25] provided a different way to solve TSP with RL by using graph embedding technology for learning an indirect policy. The learned greedy policy behaves like a meta-algorithm that incrementally constructs a solution. Prates *et al.* [98] proposed a new method that trains a GNN with supervised learning to predict the satisfiability of the decision version of TSP. All of the above works share the same fundamental philosophy, that is solving the abstracted combinatorial optimisation problem in an end-to-end pipeline that goes straight from raw inputs to general outputs. Among these methods, the structure design of neural networks is the key to achieve good performance. Specifically, the encoder-decoder architecture is widely used in these studies, which inspired

us to design encoder-decoder neural network structures to tackle task offloading and service migration problems in MEC.

2.3 Conclusion

In this chapter, I give an overview of the background knowledge related to this thesis. I first summarise the basic concept of RL and give a brief introduction of DRL algorithms including conventional model-free DRL (i.e., value-based and policy-based), MRL, and DRL with partially observable environments. I then discuss the state-of-the-art related work about task offloading and service migration in MEC systems. In addition, from the theoretical perspective of task offloading and service migration problems, they can be converted to combinatorial optimisation problems. I also present existing learning-based combinatorial optimisation algorithms which provide useful insights for us to design DRL-based system optimisation methods.

Chapter 3

Dependent Task Offloading Based on Deep Reinforcement Learning

In this chapter, this research starts to investigate how to adapt deep reinforcement learning (DRL) methods to address the dependent task offloading problem in MEC systems. In general, many mobile applications are composed of dependent tasks where the outputs of some tasks are the inputs of others. How to offload these tasks to the network edge is a vital and challenging problem in MEC, which aims to determine the placement (i.e., local or edge) of each running task of mobile applications to maximize the Quality-of-Service (QoS). Different from the existing works, I propose an intelligent dependent task offloading scheme leveraging off-policy reinforcement learning empowered by a tailored Sequence-to-Sequence (seq2seq) neural network. In the following sections, I present the details of the proposed method.

3.1 Introduction

One of the key functionalities of MEC is task offloading (aka, computation offloading), which enables to offload computation-intensive tasks of mobile applications from user equipment (UE) to MEC host at the network edge. A good task offloading strategy can save energy on devices and reduce the response time of applications. On the contrary, an inappropriate offloading policy can cause high energy consumption and poor response time. The existing works generally consider independent tasks offloading [28, 36] or dependent tasks offloading [67, 26]. For example, Lin et al. [67] propose an offloading method with task dependency on different processors based on Heterogeneous Earliest Finish Time (HEFT). De Maio et al. [26] propose a heuristic based approach to find a trade-off solution among application

runtime, battery lifetime and user cost. Most of the existing works on offloading strategies are based on heuristic algorithms, because of the NP-hardness of MEC offloading. However, with the increasing complexity of MEC applications and wireless network architecture, it is hard for any heuristic offloading strategy to fully adapt to the various scenarios in MEC.

DRL, which combines Reinforcement Learning (RL) with Deep Neural Networks (DNN), is a promising approach to achieve flexible and adaptive task offloading without expert knowledge. DRL learns an effective policy (i.e., a mapping from environment states to actions) through interacting with the environment so as to maximize numerical rewards. With the help of the powerful representation ability of DNNs, DRL can effectively solve complex decision-making problems with large and high-dimensional state/action spaces. Recent breakthroughs in DRL have led to many successful applications in a wide range of areas including gaming [111], robotics [65], networking [159], etc. DRL has been adopted to handle task offloading problems in MEC [45, 131, 159, 168]. However, these methods assume the offloading tasks are independent without considering the inherent dependencies among tasks of real-world applications. In practice, many applications are composed of dependent tasks where the outputs of some tasks are the inputs of others. Ignoring the task dependencies when making task offloading decisions will severely affect the QoS of applications and waste the edge resources.

To fill this gap, we propose a new DRL-based Task Offloading (DRLTO) scheme leveraging off-policy RL empowered by a Sequence-to-Sequence (S2S) neural network. The DRLTO is able to reduce the latency of running applications and the energy consumption at UE. In DRLTO, the task offloading problem is modelled as a Markov Decision Process (MDP). Applications are represented by Directed Acyclic Graphs (DAG), where vertices and edges denote tasks and their dependency, respectively. To effectively extract the key features of task dependency, an S2S neural network is applied to represent the policy and value function of the MDP. Specifically, the input of the S2S neural network is the DAG represented by a sequence of embedding vectors, while the output is the offloading plan for the DAG. To improve the training efficiency, we combine an off-policy DRL algorithm that includes a clipped surrogate objective and an entropy bonus to stabilize the training, provide better sample efficiency, and alleviate the issue of getting stuck in local optima. The DRLTO learns to make efficient offloading decisions through directly interacting with the environment and only requires minimal expert knowledge. The major contributions of this paper are summarized as follows:

- We develop an original DRL-based task offloading scheme, which leverages off-policy reinforcement learning with an S2S neural network to capture the intrinsic task dependency of applications. An MDP is designed to accurately model the dependent

task offloading problem with well-designed state space, action space, and reward function.

- We design a new embedding method that encodes vertices of the DAG representing an application to a sequence of embeddings including both the task profiles and dependency information. This method can convert the raw DAG as the input of the S2S neural network without information loss.
- We combine an S2S neural network with an attention mechanism to capture the long-term dependency of the input tasks. This S2S neural network can effectively approximate the policy and value function of the MDP model for the dependent task offloading problem. To effectively train the S2S neural network, we apply an off-policy DRL algorithm with a clipped surrogate objective function and an entropy bonus. This algorithm has a strong exploration ability and thus can prevent the training from getting stuck in the local optima.
- Extensive simulation experiments were conducted using the synthetic DAGs, covering a wide range of topologies, task numbers, and data rates that correspond to the characteristics of real-world applications. The performance results show that our method outperforms advanced heuristic baselines and can obtain near-optimal results under dynamic MEC scenarios.

3.2 Problem formulation: Task Offloading

The section presents the formulation of the task offloading problem. First, we give the details of the task offloading process in MEC. Next, the energy model and the optimization target used for task offloading are described.

3.2.1 Task Offloading Process

In practice, many applications consist of multiple tasks with general dependency, which should be considered when making decisions on task offloading. Fig. 3.1 gives an example of computation offloading in MEC. This example considers a real-world application—face recognition, which consists of dependent tasks such as tiler, detection, or feature mergence [100]. Formally, let $G = (\mathcal{T}, \mathcal{E})$ denote a DAG, where a vertex $t_i \in \mathcal{T}$ and a directed edge $e(t_i, t_j) \in \mathcal{E}$ represents a task t_i and the dependency between t_i and t_j, respectively. A task can start to run only when all of its predecessors are finished. The exit tasks are those without subsequent tasks.

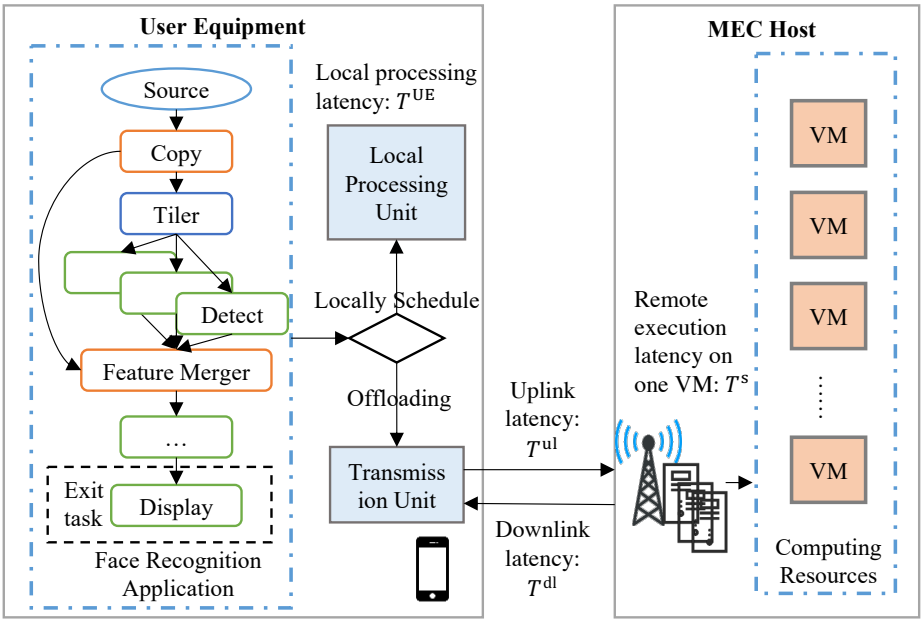


Fig. 3.1 An example of computation offloading in MEC.

We consider a block fading channel, where the fading coefficient remains unchanged (i.e., the transmission rate is fixed) during the task offloading process. In general, the MEC host runs multiple virtual machines (VMs) to process the offloaded tasks. In this work, we consider that each UE is associated with a dedicated VM providing provide computing, communications and storage resources to the UE as specified in many existing studies [114, 14]. Any task in the DAG has two scheduling choices: offloaded to the MEC host or run locally on the UE. If the task t_i is offloaded, it has three phases of execution: 1) *sending phase:* the UE sends t_i to the MEC host through a wireless channel. 2) *executing phase:* the MEC host executes the received task t_i. 3) *receiving phase:* the MEC host returns the results to the UE. On the other hand, if t_i is locally executed, there is no data transmission between the UE and MEC host. The local processor of the UE directly processes the task when it is ready. For all tasks in a DAG, let $A_{1:n} = [a_1, a_2, \ldots, a_i, \ldots, a_n]$ denote an offloading plan, where n represents the total task number and a_i represents the offloading decision of t_i. Specially, $a_i = 1$ denotes that t_i is offloaded to the MEC host, otherwise, $a_i = 0$ means that t_i is scheduled to the local processor. Fig. 3.2 shows a simple computation offloading plan for the face recognition application. Some tasks of the DAG run locally at the user equipment (e.g., Source, Copy, and Display), and others run remotely on the MEC server (e.g., Tiler, Detect, and Feature Merger). Note that the task dependency constraints the start time of task execution. For example, the Feature Merger can only start running on the MEC server when the Copy and all Detect tasks are finished. The goal of computation offloading is to find an

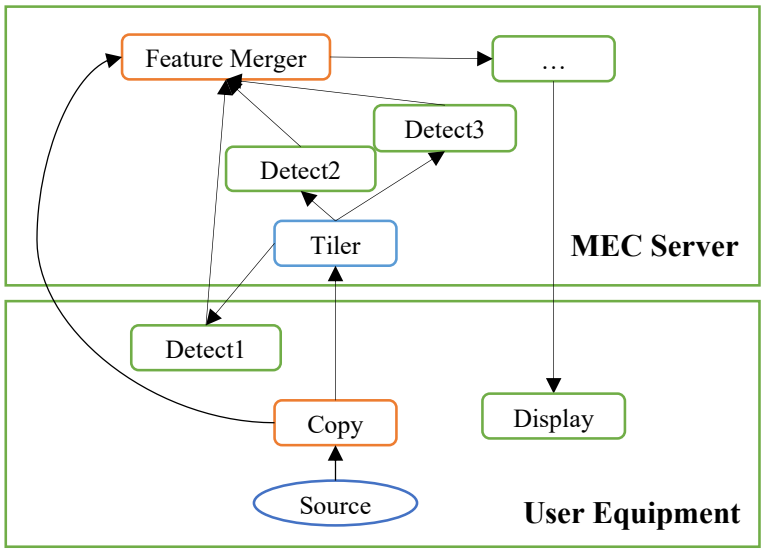


Fig. 3.2 A simple example of computation offloading for the face recognition application.

optimal partitioning plan so that the total running cost for the DAG is minimal. Whenever a computation offloading decision for a task has to be taken, a trade-off has to be made between the potential benefit that would result from offloading a task (e.g., QoS improvement) and the cost that may be incurred from doing so (e.g., introduce extra communication cost). In the following paragraphs, I define the running cost of tasks.

Let $\mathrm{T}_i^{\mathrm{ul}}$, $\mathrm{T}_i^{\mathrm{s}}$, and $\mathrm{T}_i^{\mathrm{dl}}$ denote the latency of sending, executing and receiving phase, respectively. Let $\mathrm{T}_i^{\mathrm{l}}$ denote the local execution latency for t_i. Besides, f_{l} and f_{s} represent the CPU clock speed of the UE and the VM in the MEC Host, respectively. R_{ul} denotes the uplink transmission rate while R_{dl} denotes the downlink transmission rate. Let $\mathrm{data}_i^{\mathrm{s}}$ and $\mathrm{data}_i^{\mathrm{r}}$ represent the data size of the task t_i that is offloaded to the MEC host and the result received, respectively. This research denotes the required CPU cycles for executing t_i as C_i. Therefore, all the above-defined latencies can be calculated by using the following equations:

$$\begin{aligned} \mathrm{T}_i^{\mathrm{ul}} &= \mathrm{data}_i^{\mathrm{s}}/R_{\mathrm{ul}}, \qquad \mathrm{T}_i^{\mathrm{s}} = C_i/f_{\mathrm{s}}, \\ \mathrm{T}_i^{\mathrm{dl}} &= \mathrm{data}_i^{\mathrm{r}}/R_{\mathrm{dl}}, \qquad \mathrm{T}_i^{\mathrm{l}} = C_i/f_{\mathrm{l}}. \end{aligned} \tag{3.1}$$

This research also assumes the available time of the uplink wireless channel, the MEC host, the downlink wireless channel, and the local processor, as $\mathcal{M}_i^{\mathrm{ul}}$, $\mathcal{M}_i^{\mathrm{s}}$, $\mathcal{M}_i^{\mathrm{dl}}$, $\mathcal{M}_i^{\mathrm{l}}$, respectively. Given a scheduling plan $A_{1:n}$, the available time of the resource depends on the finish time (FT) of the task scheduled immediately before t_i on that resource. If the task scheduled immediately before t_i does not utilize the resource, the FT on that resource is set as 0. I next analyse the local executing and remote offloading process in detail.

Local Executing: In this case, t_i is scheduled to the local processor. Note that t_i can only start execution until all its immediate predecessors are finished and the local processor is available to run. Besides, the immediate predecessors of t_i can be either run on the MEC host or the local processor. Therefore, the FT of task t_i on the local processor, $\mathrm{FT}_i^{\mathrm{l}}$, can be defined as

$$\begin{aligned} \mathrm{FT}_i^{\mathrm{l}} &= \max\left\{\mathcal{M}_i^{\mathrm{l}}, \max_{j\in \mathrm{pre}(t_i)}\left\{\mathrm{FT}_j^{\mathrm{l}}, \mathrm{FT}_j^{\mathrm{dl}}\right\}\right\} + \mathrm{T}_i^{\mathrm{l}}, \\ \mathcal{M}_i^{\mathrm{l}} &= \max\left\{\mathcal{M}_{i-1}^{\mathrm{l}}, \mathrm{FT}_{i-1}^{\mathrm{l}}\right\}. \end{aligned} \tag{3.2}$$

where $\mathrm{pre}(t_i)$ denotes the set of immediate predecessors of t_i. $\mathrm{FT}_j^{\mathrm{dl}}$ denotes the FT of transmitting a task t_j over the wireless downlink channel.

Remote Offloading: In this case, t_i is offloaded to the MEC host. As mentioned above, the offloading process for t_i includes three phases. In the sending phase, let $\mathrm{FT}_j^{\mathrm{ul}}$ denote the FT of transmitting the task t_j over the wireless uplink channel, where t_j is an immediate predecessor of t_i. If t_j is locally scheduled, t_i can only start its sending phase after t_j has finished local execution. Otherwise, if t_j is offloaded, t_i can only start sending after t_j has completed its sending phase. Therefore, the FT of task t_i on the wireless uplink channel can be calculated as:

$$\begin{aligned} \mathrm{FT}_i^{\mathrm{ul}} &= \max\left\{\mathcal{M}_i^{\mathrm{ul}}, \max_{j\in \mathrm{pred}(t_i)}\left\{\mathrm{FT}_j^{\mathrm{l}}, \mathrm{FT}_j^{\mathrm{dl}}\right\}\right\} + \mathrm{T}_i^{\mathrm{ul}}, \\ \mathcal{M}_i^{\mathrm{ul}} &= \max\left\{\mathcal{M}_{i-1}^{\mathrm{ul}}, \mathrm{FT}_{i-1}^{\mathrm{ul}}\right\}. \end{aligned} \tag{3.3}$$

In the executing phase, three conditions must be met before t_i can start running on the MEC host. First, t_i should finish its sending phase. Second, all predecessors of t_i should finish executing. Third, the MEC host is available to run the task. FT_i^{ul} is the FT of transmitting the task t_i over the wireless uplink channel and FT_j^{s} is the FT of running task t_j on the MEC host. Let $\mathrm{FT}_i^{\mathrm{s}}$ define the FT of t_i on the MEC host, we have:

$$\begin{aligned} \mathrm{FT}_i^{\mathrm{s}} &= \max\left\{\mathcal{M}_i^{\mathrm{s}}, \max\left\{\mathrm{FT}_i^{\mathrm{ul}}, \max_{j\in \mathrm{pred}(t_i)} \mathrm{FT}_j^{\mathrm{s}}\right\}\right\} + \mathrm{T}_i^{\mathrm{s}}, \\ \mathcal{M}_i^{\mathrm{s}} &= \max\left\{\mathcal{M}_{i-1}^{\mathrm{s}}, \mathrm{FT}_{i-1}^{\mathrm{s}}\right\}. \end{aligned} \tag{3.4}$$

Similarly, let $\mathrm{FT}_i^{\mathrm{dl}}$ define the FT of t_i on the downlink wireless channel. Note that $\mathcal{M}_i^{\mathrm{dl}}$ and T_i^{dl} are the available time and latency of task t_i on the downlink channel, respectively. Hence, we have:

$$\begin{aligned} \mathrm{FT}_i^{\mathrm{dl}} &= \max\left\{\mathcal{M}_i^{\mathrm{dl}}, \mathrm{FT}_i^{\mathrm{s}}\right\} + \mathrm{T}_i^{\mathrm{dl}}, \\ \mathcal{M}_i^{\mathrm{dl}} &= \max\left\{\mathcal{M}_{i-1}^{\mathrm{dl}}, \mathrm{FT}_{i-1}^{\mathrm{dl}}\right\}. \end{aligned} \tag{3.5}$$

3.2.2 Energy Model

Energy consumption of the UE is another important factor that this research should consider in MEC. In general, the energy consumption of an UE mainly consists of computation and transmission cost. Executing task locally or offloading task to the MEC host result in different energy costs.

When the task is locally executed, no transmission is needed, thus the energy consumption is mainly contributed by the computation process, which is defined as

$$\mathrm{E}_i^{\mathrm{l}} = \rho f_{\mathrm{l}}^{\zeta} \mathrm{T}_i^{\mathrm{l}}, \tag{3.6}$$

where $\rho f_{\mathrm{l}}^{\zeta}$ represents the computational power of the mobile device. ρ is a power coefficient while ζ is a constant (usually close to 3) [28].

When the task is offloaded, there is no computation process on the local device, thus the energy consumption is mainly contributed by wireless transmission, which is defined as

$$\mathrm{E}_i^{\mathrm{s}} = P^{\mathrm{Tx}} \mathrm{T}_i^{\mathrm{ul}} + P^{\mathrm{Rx}} \mathrm{T}_i^{\mathrm{dl}}, \tag{3.7}$$

where P^{Tx} and P^{Rx} are the sending and receiving power, respectively.

3.2.3 Optimisation Target

Based on the above definitions, giving the offloading plan, $A_{1:n}$, of a DAG, the latency and energy consumption can be calculated as

$$\mathrm{T}_{A_{1:n}}^{c} = \max_{t_i \in \mathcal{K}} \left\{ \max\{\mathrm{FT}_i^{\mathrm{l}}, \mathrm{FT}_i^{\mathrm{dl}}\} \right\}, \tag{3.8}$$

$$\mathrm{E}_{A_{1:n}}^{c} = \sum_{t_i \in \mathcal{T}, a_i = 1} \mathrm{E}_i^{\mathrm{s}} + \sum_{t_j \in \mathcal{T}, a_j = 0} \mathrm{E}_j^{\mathrm{l}}, \tag{3.9}$$

where $\mathcal{K}$ denotes the set of exit tasks.

QoS can be used to measure how good an offloading plan is, considering both latency and energy consumption. As in [158, 18], we use a similar definition of QoS as the optimization objective, which is a weighted sum of the normalized differences in latency and energy consumption between the offloading plan and local execution:

$$J_{A_{1:n}} = \lambda_t \frac{\mathrm{T}_l^c - \mathrm{T}_{A_{1:n}}^c}{\mathrm{T}_l^c} + \lambda_e \frac{\mathrm{E}_l^c - \mathrm{E}_{A_{1:n}}^c}{\mathrm{E}_l^c}, \tag{3.10}$$

where T_l^c and E_l^c are the latency and energy consumption of executing all tasks locally on the UE. λ_t and $\lambda_e \in [0, 1]$ are scalar weights. Eq. (3.10) represents a weighted sum approach of a multi-objective optimization problem. The weighted-sum approach is extensively used since it is generally effective and easy to implement. Besides, the weights reflect the relative importance of energy and latency, which can be set based on the user's preference. For general DAGs, the scheduling problem is NP-hard [58]. Therefore, it is extremely hard to find the optimal offloading plan with reasonable time complexity.

3.3 The DRLTO Scheme

This section presents the proposed DRLTO in detail. This research first presents the overall design of the DRLTO architecture is described. Next, the offloading model, the proposed seq2seq neural network, and the training algorithm are presented in detail.

3.3.1 The DRLTO Design

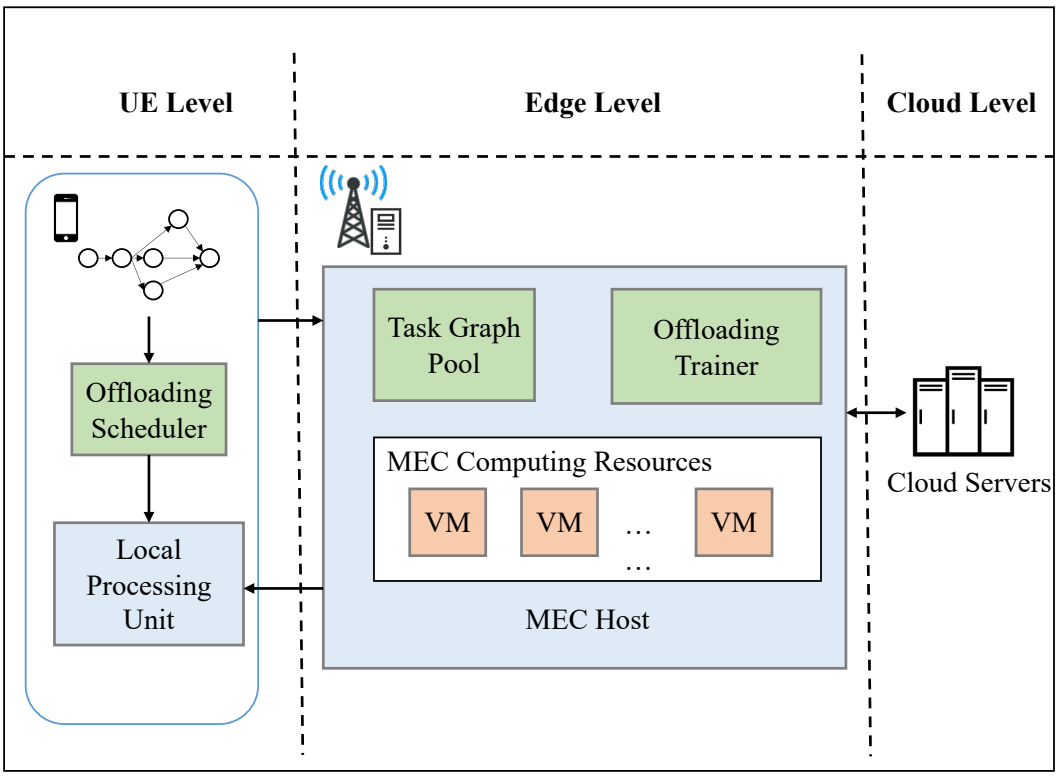


Fig. 3.3 The proposed DRL-based task offloading scheme.

The DRLTO can be integrated into an emerging MEC platform defined by ETSI [104]. As shown in Fig. 3.3, the MEC system consists of three levels: UE level, edge level, and cloud level. UE level includes various user devices (such as smartphones, tablets, vehicles) and software applications. In edge level, each MEC host contains the computing, storage and network resources for processing applications on its virtual machines (VMs). In cloud level, cloud servers are hired to process computation-intensive and resource hungry jobs. The DRLTO scheme contains three major components: offloading scheduler, task graph pool, and offloading trainer. Each UE has an offloading scheduler, which makes offloading decisions for user applications. Specifically, the trained seq2seq neural network is included in the offloading scheduler. The MEC host contains a task graph pool and an offloading trainer, which are used to gather DAGs from UE and conduct periodical training process, respectively.

The training for the seq2seq neural network is based on periodically gathered DAGs. At the off-peak time (e.g. midnight), the offloading trainer runs the training procedure. After training, the MEC host sends the parameters of the trained seq2seq neural network back to UE. The UE can then make the offloading decision via forward-propagation of the trained seq2seq neural network. The MEC host executes the offloaded tasks and returns the results to the UE. For those locally executed tasks, the local processing unit of the UE executes them when ready.

3.3.2 The Task Offloading Model

In order to adapt DRL to solve the task offloading problem, I model the problem as an MDP, where the state space, action space, and reward function of the MDP are defined as follows.

- **State Space:** When scheduling the task t_i, the state of the MEC system depends on the scheduling results of the previous tasks of t_i (i.e., the partial offloading plan). Hence, I define the state space as a combination of the DAG information (including DAG topology and task profiles) and the partial offloading plan. Formally, let G denote the encoded DAG and $A_{1:i}$ denote the offloading plan for the sequence of tasks from t_1 to t_i. The state space is thus denoted as

$$\mathcal{S} := \{s | s = (G, A_{1:i})\}. \tag{3.11}$$

  Specifically, G is comprised of a sequence of task embeddings. Each embedding consists of three elements: 1) a vector that includes an index of the task and the estimated task costs $\mathrm{T}_i^{\mathrm{l}}$, $\mathrm{T}_i^{\mathrm{ul}}$, $\mathrm{T}_i^{\mathrm{s}}$, and $\mathrm{T}_i^{\mathrm{dl}}$; 2) a vector of indices of immediate previous tasks; 3) a vector of indices of immediate subsequent tasks. The length of previous/subsequent

task indices vector is limited to p ($p = 12$ in the experiments). I pad the vector with -1, in case the number of previous/subsequent tasks is less than p. These three vectors are concatenated together to form the embedding vector. The embedding vector is then fed into the seq2seq neural network to obtain an offloading plan.

- **Action Space:** A task could be either offloaded to an MEC host or run locally on the UE. Let $a_i = 1$ represent offloading to an MEC host and $a_i = 0$ represent local execution for task t_i. Therefore, the action space can be defined as $\mathcal{A} := \{1, 0\}$.

- **Reward Function:** The objective of the proposed method is to maximize the QoS. In order to reach this objective, I define the reward function at each step as the estimated increment of the QoS:

$$\begin{aligned} R(s_i, a_i) &= \lambda_t \frac{\overline{\mathrm{T}_l^c} - \Delta \mathrm{T}_i}{\mathrm{T}_l^c} + \lambda_e \frac{\overline{\mathrm{E}_l^c} - \Delta \mathrm{E}_i}{\mathrm{E}_l^c}, \\ \Delta \mathrm{T}_i &= \mathrm{T}^c_{A_{1:i}} - \mathrm{T}^c_{A_{1:i-1}}, \\ \Delta \mathrm{E}_i &= \mathrm{E}^c_{A_{1:i}} - \mathrm{E}^c_{A_{1:i-1}}, \end{aligned} \tag{3.12}$$

where $\overline{\mathrm{T}_l^c}$ and $\overline{\mathrm{E}_l^c}$ are the average latency and energy consumption for a task in the DAG, which are given by $\overline{\mathrm{T}_l^c} = \mathrm{T}_l^c / n$ and $\overline{\mathrm{E}_l^c} = \mathrm{E}_l^c / n$, respectively.

3.3.3 The S2S Neural Network for DRLTO

According to the definition of our MDP model, the offloading problem can be converted to an S2S prediction problem where the input is the sequence of task embeddings and the output is a scheduling plan for those tasks. The policy $\pi(a_i | G, A_{1:i-1})$ is denoted as the probability of selecting action a_i for task t_i under the state $s = (G, A_{1:i-1})$. Moreover, $\pi(A_{1:n}|G)$ is the probability of having the offloading plan $A_{1:n}$ given the graph G with n tasks. Applying the chain rules of probability, we have

$$\pi(A_{1:n}|G) = \prod_{i=1}^{n} \pi(a_i | G, A_{1:i-1}). \tag{3.13}$$

To effectively approximate the policy defined in Eq. (3.13), an S2S neural network is a good option. As shown in Fig. 3.4, we combine an S2S neural network with the attention mechanism to approximate both the policy and the value function of the DRLTO. Formally, denote the sequence of task embeddings as $\mathbf{t} = [t_1, t_2, \ldots t_n]$ and the function of encoder

network as f_{enc}, then the hidden state of the encoder network can be obtained by:

$$e_i = f_{\text{enc}}(e_{i-1}, t_i; \theta_{\text{enc}}), \tag{3.14}$$

where e_i is the hidden state for encoding step i and θ_{enc} are the parameters of the encoder network. Let f_{dec} be the function of the decoder network. The hidden state of the decoder network d_j for decoding step j is calculated by:

$$d_j = f_{\text{dec}}(d_{i-1}, a_{j-1}, c_j; \theta_{\text{dec}}), \tag{3.15}$$

where θ_{dec} are the parameters of the decoder network. c_j is the context vector of the attention mechanism, which is defined by a weighted sum of the hidden states of the encoder network:

$$c_j = \sum_{i=1}^{n} \alpha_{ji} e_i. \tag{3.16}$$

The weight α_{ji} of each hidden state of the encoder network, e_i, is computed by

$$\alpha_{ji} = \frac{e^{f_{\text{score}}(d_{j-1}, e_i)}}{\sum_{k=1}^{n} e^{f_{\text{score}}(d_{j-1}, e_k)}} \tag{3.17}$$

where the score function, $f_{\text{score}}(d_{j-1}, e_i)$, is used to measure how well the input of the encoder network at position i matches the output of the decoder network at position j. In this paper, we define the score function as a trainable neural network as in the work [10].

The policy and value networks share most of the parameters (encoder and decoder) except for the top layer of the decoder. For the policy neural network, we add a fully connected layer on the output of the decoder, d_j , and use softmax function to convert the output into the distribution over actions, $\pi(a_j|s_j)$. For the value neural network, we add another fully connected layer on d_j and use the output to represent the state value, $v(s_j)$. The shared parameters in the S2S architecture are used to extract common features in DAGs, therefore training the policy neural network that can accelerate the training of the value neural network and vice versa.

The offloading decision for each task is made by the trained S2S neural network. Overall the steps for offloading process are follows:

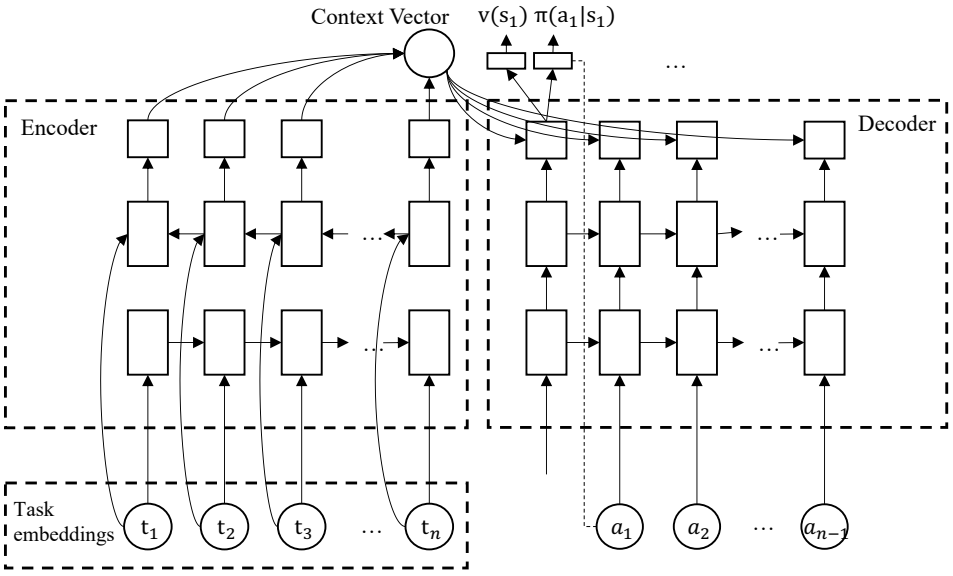


Fig. 3.4 Structure of the S2S neural network for DRLTO.

Step 1: A topological sorting is conducted to sort tasks of the DAG according to the rank values of tasks by decreasing order, which is defined as

$$rank(t_i) = \begin{cases} \mathrm{T_i^o} & \text{if } t_i \in \mathcal{K}, \\ \max_{t_j \in \mathrm{succ(t_i)}}(rank(t_j)) + \mathrm{T_i^o} & \text{if } t_i \notin \mathcal{K}, \end{cases} \tag{3.18}$$

where $\mathrm{succ(t_i)}$ represents the set of immediate successors of t_i and $\mathrm{T_i^o} = \mathrm{T}_i^{\mathrm{ul}} + \mathrm{T}_i^{\mathrm{s}} + \mathrm{T}_i^{\mathrm{dl}}$.

Step 2: Tasks are then embedded into a sequence of vectors as the input of the encoder (the details of the embedded vectors are presented in Section 3.3.2). Next, the output sequence of the encoder will be used to calculate the context vector. At the jth decoding step, the offloading decision can be generated through $a_j = \arg\max_{a_j} \pi(a_j|s_j)$.

Step 3: The UE and MEC host cooperatively finish executing all tasks according to the offloading decisions.

3.3.4 The Training Process of the DRLTO

The training goal is to find an optimal policy so that the accumulated reward is maximal, which is expressed as

$$\max_{\theta} L(\theta) = \mathbb{E}\left[\max_{\theta} \pi_\theta(A_{1:n}|G) \sum_{t}^{n} R(s_t, a_t)\right], \tag{3.19}$$

where θ are the parameters of the seq2seq neural network, n is the task number of the DAG, $R(s_t, a_t)$ is the reward function, s_t and a_t is the observed state and offloading decision for the tth task, respectively.

To improve the performance and training efficiency, the training target function is modified based on PPO [108], which generates trajectories using the old policy $\pi_{\theta_{\text{old}}}$ and updates the current policy π_θ whose initial value equals $\pi_{\theta_{\text{old}}}$ for several epochs. In order to avoid a large update of the policy, PPO penalizes changes to the policy via a clip function. The modified clipped target function for the proposed seq2seq neural network is given by

$$L^{\text{C}}(\theta) = \mathbb{E}\left[\sum_{t=1}^{n} \min\left(\text{pr}_t(\theta)\hat{A}_t, \text{clip}(\text{pr}_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\right)\right], \tag{3.20}$$

where $\hat{A}_t$ is the estimator of the advantage function at time step t and ϵ is a hyperparameter to control the clip range. $\text{pr}_t(\theta)$ is the policy probability ratio that is given by

$$\text{pr}_t(\theta) = \frac{\pi_\theta(a_t|G, A_{1:t})}{\pi_{\theta_{\text{old}}}(a_t|G, A_{1:t})}. \tag{3.21}$$

The clip function $\text{clip}(\text{pr}_t(\theta), 1-\epsilon, 1+\epsilon)$ aims to limit the value of $\text{pr}_t(\theta)$, which removes the incentive for moving $\text{pr}_t(\theta)$ outside of the interval $[1-\epsilon, 1+\epsilon]$. Finally, taking the minimum of the clipped and unclipped objective restricts the final objective as a lower bound on the unclipped objective.

As discussed in the previous subsection, a shared parameter S2S neural network is crafted for both policy and value function approximation. To train this neural network, the clipped objective function and the value function loss are integrated into the target function. Besides, adding an entropy bonus can ensure efficient exploration. The use of entropy bonus is designed to help keep the search alive by preventing convergence to a single choice of output, especially when several choices all lead to roughly the same results [145, 82]. Following the suggestion in work [108], we combine these terms and obtain the following function as the final training target:

$$L^{\text{PPO}}(\theta) = \mathbb{E}\left[L^{\text{C}}(\theta) - c_1 L^{\text{VF}}(\theta) + c_2 S[\pi_\theta](s_t)\right], \tag{3.22}$$

where c_1 and c_2 are coefficients, S denotes an entropy bonus, and L^{VF} is a squared-error loss between predicted state values $v_\pi(s)$ and target state values $\hat{v}_\pi(s)$:

$$L^{\text{VF}}(\theta) = \mathbb{E}\left[\sum_{t=1}^{n} (v_\pi(s_t) - \hat{v}_\pi(s_t))^2\right], \tag{3.23}$$

Algorithm 1: Off-policy Training for the seq2seq Neural Network

Create two seq2seq neural networks with θ_{old} and θ for the old policy $\pi_{\theta_{\text{old}}}$ and target policy π_θ with randomly generated initial values.
$\theta_{\text{old}} \leftarrow \theta$
for $i = 1, 2, ..., l$ **do**
    /** Exploration stage **/
    Collect set of trajectories D_i on policy $\pi_{\theta_{\text{old}}}$.
    for *each trajectory* $\tau \in D_i$ **do**
        Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_n$ according to Eq. (3.24).
        Compute the target state values $\hat{v}_\pi(s_1), \ldots, \hat{v}_\pi(s_n)$ following the equation: $\hat{v}_\pi(s_t) = \sum_{k=0}^{n-t+1} \gamma^k r_{t+k}$.
        Store advantage estimates and target state values in D_i.
    end
    /** Exploitation stage **/
    for $k = 1, 2, ..., m$ **do**
        Optimize the target function L^{PPO} w.r.t. θ_k by taking minibatch SGD (via *Adam*) using the sampled minibatches from D_i.
    end
    Synchronise old and new parameters: $\theta_{\text{old}} \leftarrow \theta$.
end

where $\hat{v}_\pi(s_t) = \sum_{k=0}^{n-t+1} \gamma^k r_{t+k}$.

Training the S2S neural network with RL is quite different from training DNN with RL, which can use transition segments (s_t, a_t, r_t, s_{t+1}) for back propagation. When training the S2S neural network, the entire trajectory should be divided into sequences which are then fed into the network. For example, two components of trajectories are firstly sampled from the environment, which are defined by a scheduling plan $A_{1:n}$ and a sequence of state values $[v_\pi(s_1), v_\pi(s_2), \ldots, v_\pi(s_n)]$ obtained from conducting a forward propagation of the S2S neural network with the tasks embedding sequence. Next, the reward sequences $[r_1, r_2, \ldots, r_n]$ can be obtained by applying the scheduling plan to the environment. Furthermore, the TD-error term δ at time step t can be calculated by $\delta_t = r_t + \gamma v_\pi(s_{t+1}) - v_\pi(s_t)$. Finally, the generalized advantage estimator (GAE) [107] is used to obtain the advantage function at time step t as

$$\hat{A}_t = \sum_{k=0}^{n-t+1} (\gamma\lambda)^k (\delta_{t+k}), \tag{3.24}$$

where λ $(0 < \lambda < 1)$ is used to control the trade-off between bias and variance.

The training algorithm is formally presented in Algorithm 1. First, the sampling and updating neural networks are initialized with the same parameters. In each loop, the sampling

neural network is used to sample a set of trajectories which are stored in D_i. Next, the sequence of advantage estimates $\hat{A}$ and the estimated state values $\hat{v}_\pi$ are calculated and stored in D_i. Since then, D_i consists of the tasks embedding sequences $[t_1, t_2, \ldots, t_n]$, sampled scheduling plans $A_{1:n}$, reward sequences $[r_1, r_2, \ldots, r_n]$, sampled state values sequences $[v_\pi(s_1), v_\pi(s_2), \ldots, v_\pi(s_n)]$, advantage estimates sequences $[\hat{A}_1, \hat{A}_2, \ldots, \hat{A}_n]$, and estimated state values sequences $[\hat{v}_\pi(s_1), \hat{v}_\pi(s_2), \ldots, \hat{v}_\pi(s_n)]$. At the exploitation stage, the updating neural network is improved by conducting minibatch Stochastic Gradient Descent (SGD) on D_i with the target function defined in Eq. (3.22) for m epochs. *Adam* is adopted as the optimization method for its efficiency and stability.

3.4 Results and Discussion

This section presents the experimental results and performance evaluation of the DRLTO. We first present how to set the simulation environment and hyperparameters. Next, the training results of the DRLTO are presented, including the average reward, value loss, and policy loss. Finally, we analyse the performance results of the DRLTO through comparing it with eight existing algorithms.

3.4.1 Simulation Environment and Hyperparameters

In our simulation experiments, we consider that the UE is in a small cell network with different transmission rates dependent on the distance between the UE and MEC host. The transmission rate of real-world 5G network (e.g., VZ and T-Mobile) is typically less than 60 Mbps, while for Sprint it is around 30 Mbps [87]. Hence, we set the set of transmission rate is {3 Mbps, 7 Mbps, 11 Mbps, 15 Mbps, 19 Mbps}, which covers the most area of a cell from a distal end to a proximal end. We set the constants in the energy model $\rho = 1.25 \times 10^{-26}$ and $\zeta = 3$ according to [28]. The CPU clock speed of the UE f_l is set to be 1 GHz. We set the total computation capacity of the edge server as 10 GHz. Moreover, P^{Tx} and P^{Rx} are 1.258 W and 1.181 W, respectively [28].

In MEC system, The task dependency can be obtained through parsing the programme of the mobile application. For example, we can use the compiler to parse the Android apk file or jar file into a syntax tree. The task dependency information is included in the syntax tree. We can then convert mobile applications into DAGs based on the parsed syntax tree. Different DAGs can have various topologies, for example, a data compression application is composed of multiple tasks with linear dependency, while face recognition and gesture capture applications involve more complex inner dependency among tasks (as shown in Fig.

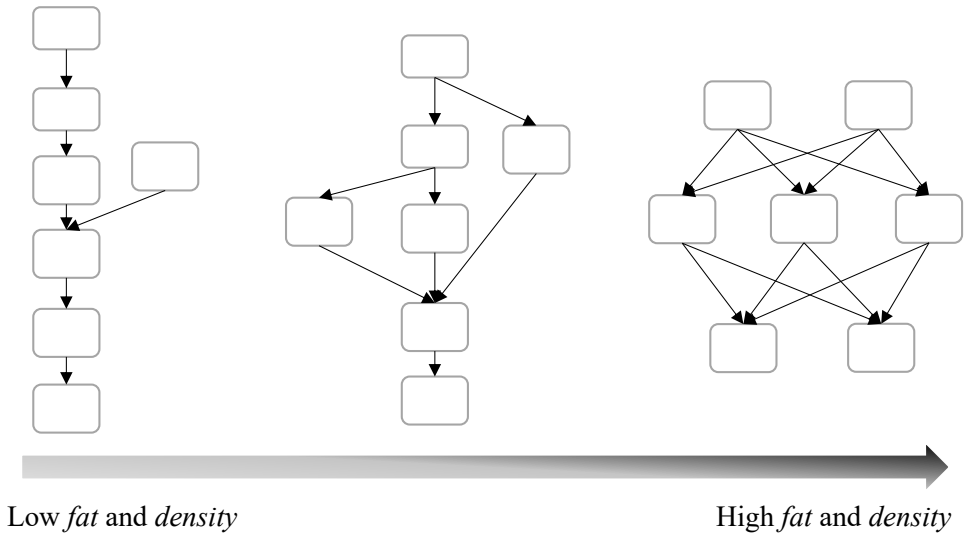


Fig. 3.5 The examples of synthetic DAGs from low *fat* and *density* to high *fat* and *density*.

3.1). Our scheme aims to obtain an effective offloading policy for general purpose, which can adapt to any DAG topologies. To effectively train our DRL-based algorithm, we need the information of task profiles and dependency for many different mobile applications. The current real datasets for mobile applications only contain information of a very limited number of applications [100]. Therefore, we use a synthetic DAG generator [8] to generate various DAGs representing heterogeneous applications. The properties of DAGs are controlled by several parameters including *fat*, *density*, and *ccr*. Here, *fat* is used to control the width and the height of a DAG. *density* determines the number of edges between two levels of a DAG. *ccr* denotes the communication-to-computation ratio, which is the ratio between the communication cost and the computation cost. Using a simulator to generate synthetic DAGs has many benefits. For example, it can fast generate a large number of DAGs representing various mobile applications. In addition, it is easy to control the features of generated DAGs by tuning simulator parameters.

For the DAG generation, we randomly pick *fat* from {0.3, 0.4, 0.5, 0.6, 0.7, 0.8}, *density* from {0.3, 0.4, 0.5, 0.6, 0.7, 0.8} and *ccr* from {0.3, 0.4, 0.5}. It is reasonable to set *ccr* less than 0.5, since many emerging applications are computation-intensive. Fig. 3.5 shows an example of DAG topologies from low *fat* and *density* to high *fat* and *density*. For the settings of task profile, the transmission data size of a task is set between 5 KB and 50 KB. The required CPU cycles for a task is set between 10^7 and 10^8 cycles/sec. The task number n of the generated DAG ranges from 10 to 50 with a step size of 5. According to the above setting, we randomly generate 500 graphs for each task number as the training dataset and

additional 100 graphs for each task number as testing dataset. Specifically, the parameters of our simulation environment are listed in Table 3.1.

The S2S neural network is implemented via Tensorflow. Specifically, the encoder is set as a two-layer bi-directed Long Short-Term Memory (LSTM) with 256 hidden units while the decoder is set as a two-layer dynamic LSTM also with 256 hidden units. Besides, the layer normalization [9] is added for both encoder and decoder. During the training process, the learning rate is set as 10^{-4}, the coefficients is set as $c_1 = 0.5$, $c_2 = 0.01$, and the batch size is set as 500. Hyperparameters can affect the training results and the convergence speed of DRLTO. We initially set the hyperparameters according to [108], and then run grid search on learning rate, batch size, discount factor, etc., to find the optimal hyperparameters (given in Table 3.2) for our algorithm. These hyperparameters have been applied to all scenarios in our experiments.

Table 3.1 The Parameters of Simulation Environment

Notation	Parameter	Value
$R_{\text{ul}}, R_{\text{dl}}$	UL/DL Transmission Rate	$\{3, 7, 11, 15, 19\}$ Mbps
f_{l}	CPU Clock Speed of UE	1 GHz
f_{s}	CPU Clock Speed of a VM in MEC Host	4×2.5 GHz
ζ, ρ	Constants in Energy Model	$\zeta = 3$, $\rho = 1.25 \times 10^{-26}$
P^{Tx}	Avg. Wireless Sending Power	1.258 W
P^{Rx}	Avg. Wireless Receiving Power	1.181 W
fat	Width of a DAG	$\{0.3, 0.4, 0.5, 0.6, 0.7, 0.8\}$
density	Density of Dependencies of a DAG	$\{0.3, 0.4, 0.5, 0.6, 0.7, 0.8\}$
ccr	Comm. to Comp. Ratio of a DAG	0.3 to 0.5
data_i^s, data_i^r	Sending/Receiving Data Size for a Task	5 KB to 50 KB
C	Required CPU Cycles for a Task	10^7 to 10^8 cycles/sec
p	Length of Task Indices Vector	12

3.4.2 Compared Algorithms

We compare the performance between the DRLTO and the following eight existing algorithms:

- *Optimal*: The exhaustive search is applied to list all possible solutions and find the optimal one with the highest QoS.
- *Local*: All tasks of the DAG are run on the local processor.

Table 3.2 The Neural Network and Training Hyperparameters

Hyperparameter	Value	Hyperparameter	Value
Encoder Layers	2	Encoder Layer Type	Bi-LSTM
Encoder Hidden Units	256	Encoder Layer Norm.	On
Decoder Layers	2	Decoder Layer Type	LSTM
Decoder Hidden Units	256	Decoder Layer Norm.	On
Learning Rate	10^{-4}	Activation function	Tanh
Optimization Method	Adam	Loss Coefficient c_1	0.5
Discount Factor γ	0.99	Entropy Coefficient c_2	0.01
Adv. Discount Factor λ	0.95	Clipping Constant ϵ	0.2

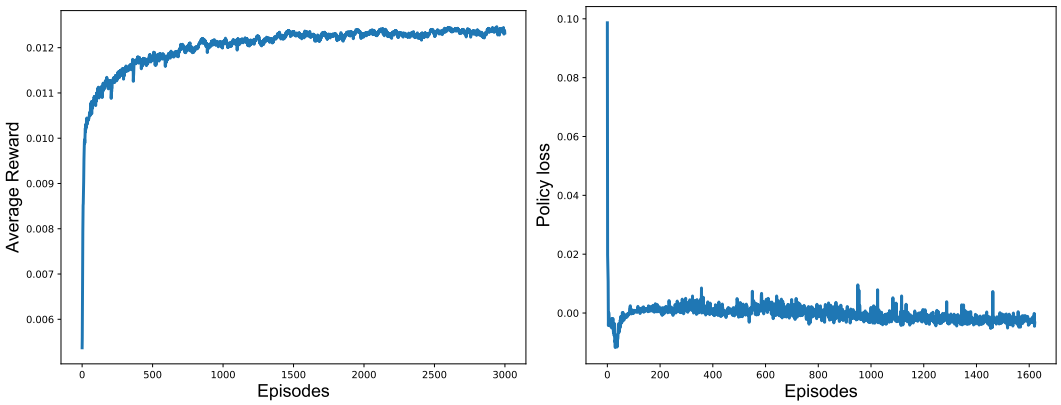
- *Remote*: All tasks of the DAG are offloaded to the MEC host.
- *Random*: Each task of the DAG is randomly assigned to the local processor or the MEC host.
- *Greedy*: Each task of the DAG is greedily assigned locally or remotely based on the estimated finish time on the local processor and the MEC host.
- *Round-Robin (RR)*: Tasks of the DAG are alternately scheduled to the local processor and MEC host.
- *HEFT-based*: Tasks are firstly prioritized according to Heterogeneous Earliest Finish Time (HEFT) as in [67, 124]. The prioritized tasks are then scheduled to the resource with earliest estimated finish time. This algorithm is an advanced heuristic algorithm and can be an important baseline to show the gap between learning-based algorithm and the heuristic algorithm.
- *Double Deep-Q Network based Task Offloading (DDQNTO)*: Tang *et al.* [121] combined LSTM, dueling deep Q-network (DQN), and double-DQN techniques to handle the task offloading problem without considering the inner dependency. Specifically, we use the same exploration-exploitation strategy as in the work [121] to train the Q-networks. This algorithm is a state-of-the-art learned-based algorithm without considering the dependency. I include this algorithm as a baseline to show the importance of the task dependency.

3.4.3 Training Performance and Convergence

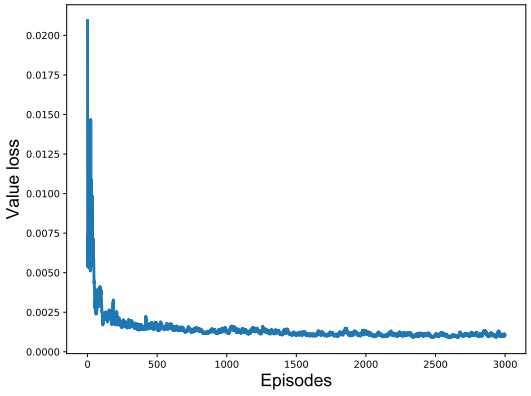
We first investigate the training overheads, convergence property, and inference latency in our algorithm. In our experiments, we set two different targets: *latency-optimal (LO)* and *energy-efficient (EE)*. The LO target aims at minimizing the latency, thus we set $\lambda_t = 1.0$ and $\lambda_e = 0.0$. The EE target aims to jointly optimize the latency and energy consumption, so we set $\lambda_t = \lambda_e = 0.5$. We conduct the training process for the DRLTO and record the average reward, the policy loss (i.e., L^{C} as defined in Eq. (3.20)), and the value loss (i.e., L^{VF} as defined in Eq. (3.23)). We train DRLTO on our workstation with GeForce RTX 2080. Each episode involves 40 mini-batch updates for the S2S neural network with the time overheads around 30 s/episode. Fig. 3.6 depicts the training results with the LO target. We notice that the average reward increases sharply at the beginning and steadily grows after 500 episodes. Our experimental results show that the training converges at around 1200 episodes. The total training overheads for a converged policy is around 10 hours. As the training dose not happen very frequently, the training overheads are acceptable. For the training loss, both the value loss and the policy loss approximate to zero after 500 episodes, which shows the good convergence property of the DRLTO. After training, we obtain two trained S2S neural networks, one of which is used for the LO target and the other is used for the EE target. Those trained networks will then be deployed back to the UE. To evaluate the inference speed, we use our laptop with CPU only (2.6 GHz 6-Core Intel Core i7) to do the inference. We feed 100 DAGs with $n = 15$ to the trained S2S neural networks for network inference. As a result, the total inference latency is around 108 ms with 1.08 ms per DAG. Compared to the time overheads of offloading tasks (as shown in Table 3.3), the inference overhead can be neglect. Note that, we do not involve any inference optimisation method to our trained model. However, many existing works show that the inference speed can be further improved on mobile devices. In the following sections, we evaluate DRLTO among different scenarios.

3.4.4 Evaluation with Different Task Numbers

First, we investigate the latency, energy consumption, and QoS of different offloading algorithms with varying numbers of tasks. In this case, we fix the transmission rate at 7 Mbps. When aiming at the LO target, we focus on the latency. Table 3.3 lists the average latency of executing a DAG with different numbers of tasks. The *Optimal (LO)* is implemented using exhaustive search to find the optimal solution. However, the *Optimal (LO)* has exponential time complexity, thus it is unable to find optimum in a reasonable amount of time when $n \geq 20$. The *Local* and the *Remote* both perform poorly in this scenario, which are even worse than the *Random*. *DDQNTO (LO)* cannot learn effective policy, which obtains higher average



(a) Average total rewards (b) Average policy loss



(c) Average value loss

Fig. 3.6 The values of average total reward, policy loss, and value loss in the training process of the DRLTO.

latency than the heretic search algorithm *HEFT-based*. Compared to the existing algorithms, *DRLTO* with the LO target significantly outperforms all heuristic baselines (from the *Local* to the *HEFT-based*) and the advanced DRL-based methods (i.e., *DDQNTO*). Moreover, *DRLTO (LO)* approximates the optimal solution when $n \leq 20$.

When aiming at the EE target, we jointly consider the latency and energy consumption. The comparison results of the energy consumption and QoS with different numbers of tasks are presented in Fig. 3.7 and Table 3.4, respectively. Offloading computation-intensive applications to the MEC host can help reduce the energy consumption on UE, therefore the *Remote* achieves the lowest energy consumption. However, the *Remote* has the highest latency as shown in Table 3.3. *DDQNTO (EE)* learns offloading policies that have similar energy consumption and latency as *Remote* in all cases. It seems that *DDQNTO (EE)* fails to learn a good trade-off between energy consumption and latency. On the contrary, *DRLTO* with the EE target can learn the optimal policy by taking both the latency and energy into account. Comparing the results in Table 3.3 and Fig. 3.7, we find that *DRLTO* with the EE

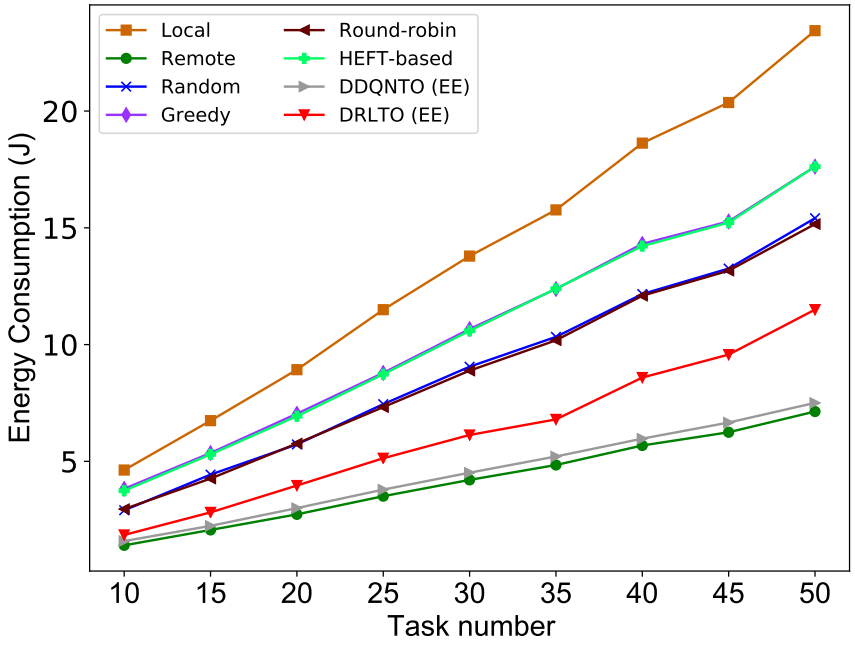


Fig. 3.7 The comparison of the DRLTO and existing algorithms in terms of average energy consumption (targeting at EE) with different numbers of tasks. EE denotes energy-efficient target.

target achieves the lowest energy consumption bar *Remote* and *DDQNTO (EE)*, while it still obtains acceptable latencies (close to the *Greedy*). Furthermore, Table 3.4 gives explicit results related to the QoS of all evaluated algorithms. Obviously, the QoS of the *Local* is always zero because we define QoS as a measurement of the gain/loss of an algorithm compared to the *Local*. Moreover, *DRLTO* with the EE target achieves the maximal QoS compared to all baseline algorithms (i.e., from *Local* to *DDQNTO*) and approximates the optimal solution.

Table 3.3 The comparison of the DRLTO and existing algorithms in terms of average latency (ms) of a DAG with different numbers of tasks (n). N/A denotes cases unable to find optimum. LO and EE denote latency-optimal and energy-efficient targets, respectively.

n	Optimal (LO)	Local	Remote	Random	Greedy	RR	HEFT-based	DDQNTO (LO)	DDQNTO (EE)	**DRLTO (LO)**	**DRLTO (EE)**
10	488.61	723.14	694.75	650.01	556.70	646.92	533.25	580.30	690.04	**491.98**	**583.03**
15	661.57	1053.44	991.49	915.54	780.73	886.51	743.20	836.01	987.36	**673.28**	**804.57**
20	851.71	1394.49	1322.87	1170.06	999.73	1134.41	958.82	1100.71	1317.71	**880.89**	**1010.94**
25	N/A	1796.05	1630.79	1430.03	1239.45	1395.91	1194.95	1349.15	1616.12	**1054.31**	**1221.08**
30	N/A	2154.71	1981.61	1727.75	1500.83	1692.38	1454.53	1629.57	1970.39	**1297.08**	**1507.35**
35	N/A	2463.69	2251.70	2043.32	1757.25	2025.92	1719.99	1906.46	2238.88	**1530.58**	**1767.64**
40	N/A	2910.47	2757.82	2360.31	1992.33	2246.18	1939.06	2286.65	2737.24	**1757.42**	**2067.79**
45	N/A	3182.15	2833.30	2442.66	2074.41	2323.27	2043.41	2365.28	2806.76	**1781.23**	**2051.24**
50	N/A	3662.96	3560.65	2897.15	2431.63	2754.37	2391.01	2948.88	3538.77	**2200.68**	**2533.18**

Table 3.4 The comparison of DRLTO and existing algorithms in terms of QoS (targeting at EE) with different numbers of tasks (n), N/A denotes cases unable to find optimum. EE denotes energy-efficient target.

n	Optimal (EE)	Local	Remote	Random	Greedy	RR	HEFT-based	DDQNTO (EE)	**DRLTO (EE)**
10	0.394	0	0.360	0.223	0.194	0.223	0.219	0.339	**0.389**
15	0.410	0	0.367	0.228	0.222	0.252	0.246	0.353	**0.399**
20	0.426	0	0.373	0.254	0.244	0.266	0.265	0.357	**0.415**
25	N/A	0	0.390	0.273	0.270	0.290	0.286	0.381	**0.434**
30	N/A	0	0.388	0.265	0.262	0.281	0.276	0.377	**0.426**
35	N/A	0	0.390	0.255	0.252	0.267	0.260	0.380	**0.426**
40	N/A	0	0.373	0.270	0.275	0.290	0.288	0.368	**0.417**
45	N/A	0	0.398	0.285	0.295	0.307	0.302	0.391	**0.441**
50	N/A	0	0.363	0.274	0.290	0.298	0.296	0.357	**0.411**

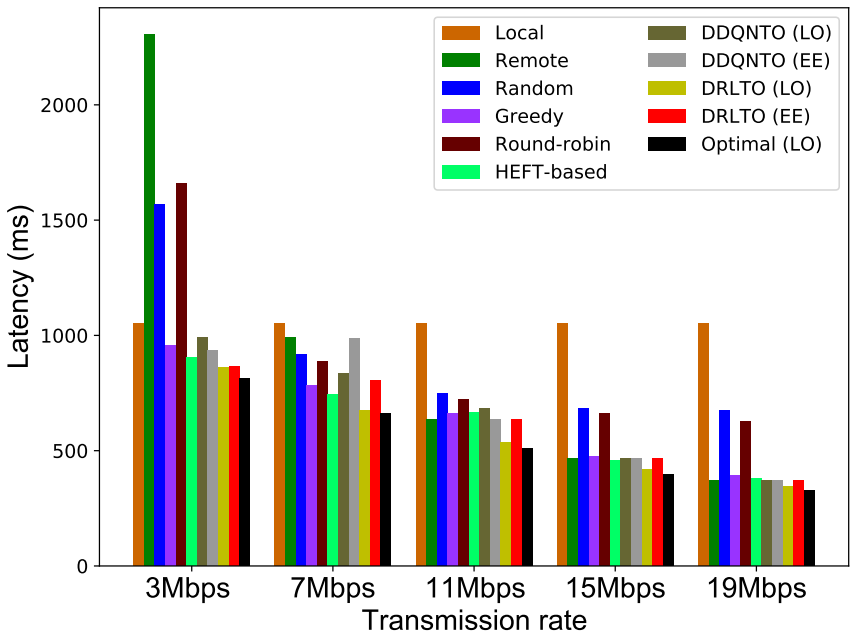


Fig. 3.8 The comparison of the DRLTO and existing algorithms in terms of average latency with different transmission rates. LO and EE denote latency-optimal and energy-efficient targets, respectively.

3.4.5 Evaluation with Different Transmission Rates

Next, we evaluate the performance of the *DRLTO* with different transmission rates. We first target at LO, and the results are shown in Fig. 3.8. When the transmission rate is low (meaning that the UE is far away from the MEC host), offloading tasks to the MEC host will result in high latency. On the contrary, if the transmission rate is high, offloading tasks can significantly reduce the latency. An efficient algorithm should automatically adapt its offloading policy to various transmission rates. As shown in Fig. 3.8, *DDQNTO (LO)* cannot learn effective policy, which achieves the worse performance than *HEFT-based* when the transmission rate increases from 3 Mbps to 15 Mbps. In contrast, *DRLTO (LO)* has high adaptability, which outperforms all baseline algorithms (i.e., from *Local* to *DDQNTO*) and approximates the optimal solution with various transmission rates.

When the target is EE, we need take energy consumption into consideration. Figs. 3.9 and 3.10 show the energy consumptions and QoS of various algorithms with different transmission rates, respectively. As expected, *Remote* consumes the least energy on UE since all tasks are run remotely. However, the latency of the *Remote* could be high and unacceptable under low or medium transmission rates. As shown in Fig. 3.10, the *Remote*, *Random*, and *RR* even achieve negative QoS when the transmission rate is 3 Mbps, meaning

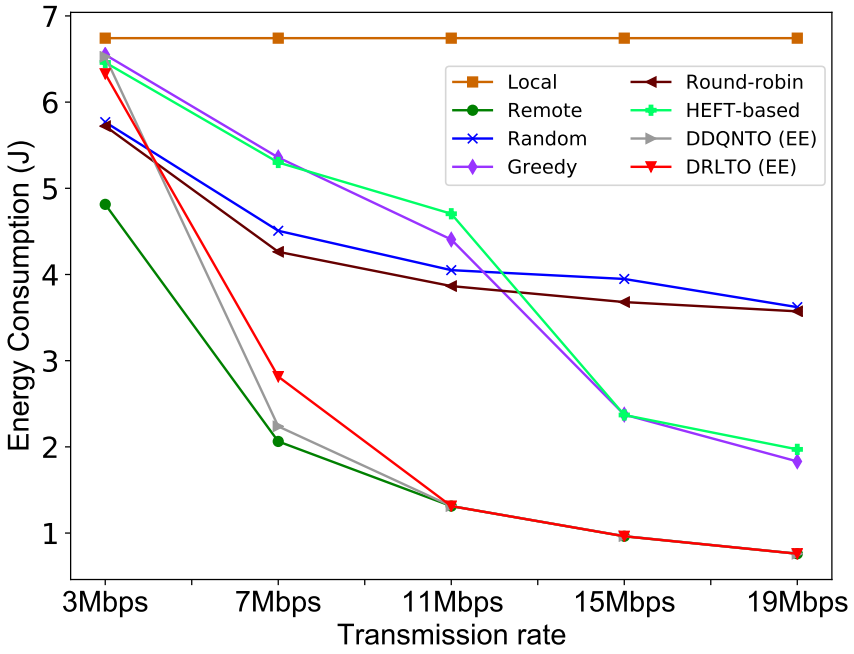


Fig. 3.9 The comparison of the DRLTO and existing algorithms in terms of average energy consumption (targeting at EE) with different transmission rates. EE denotes energy-efficient target.

that they are worse than the *local*. *DRLTO* with the EE target achieves the lowest energy consumption bar the *Remote* and *DDQNTO* when transmission rate is greater or equal than 7 Mbps, while it still leads to acceptable latencies. For example, when the transmission rate is 7 Mbps, *DRLTO* with the EE target obtains a similar latency as the *Greedy* but 50% less energy consumption than the latter, while both *DDQNTO* and *Remote* achieve the higher latency than *Random*. Fig. 3.10 demonstrates that the QoS of all algorithms (except the *Local*) increases with the transmission rate. This is because the communication cost declines as the transmission rate grows, offloading tasks to remote servers can be beneficial. In addition, *DRLTO* with the EE target obtains the maximal QoS compared with all baselines and approximates the optimal solution.

3.4.6 Evaluation with/without Dependency

The task dependency reflects that some tasks of the mobile application require input from others. This means that the offloaded tasks and tasks executed on the user equipment might not process in parallel. For example, in Fig. 3.2, Detect1 is assigned to local user equipment while Detect2 is offloaded to the MEC server. Since there is no dependency between Detect1 and Detect2, thus they can run in parallel. In contrast, Feature Merger can only start running

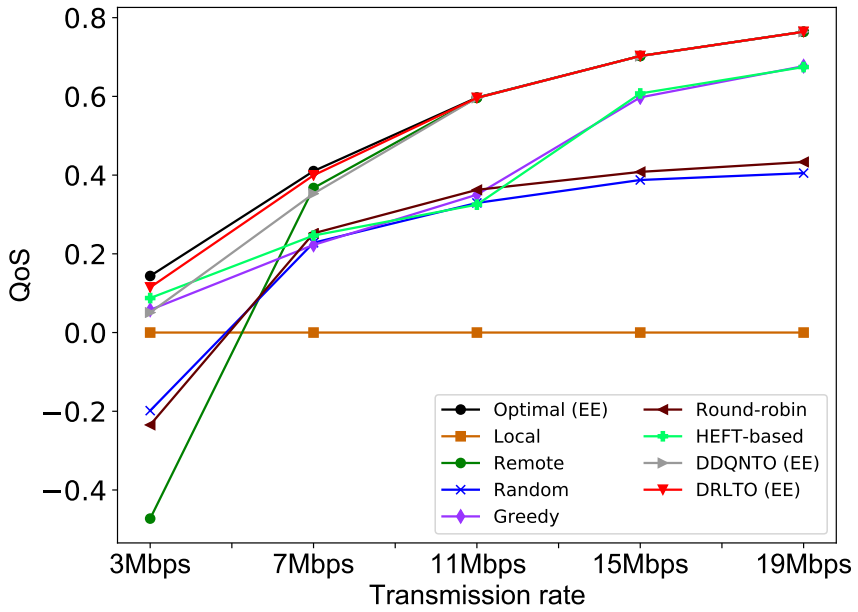


Fig. 3.10 The comparison of the DRLTO and existing algorithms in terms of QoS (targeting at EE) with different transmission rates. EE denotes energy-efficient target.

when Detect1 is finished due to the task dependency. Taking this dependency information into consideration or not can severely affect the task offloading performance. In the following content, I give the detailed evaluation results with/without dependency information.

We embed the dependent information into the task embeddings and use the encoder network with the attention mechanism to extract features from the task embedding. In order to show how the dependent information influences the results. As shown in Fig. 3.11, we remove the encoder network with attention mechanism from the S2S neural network and directly input the task embeddings to the decoder network. The output of the decoder network remains the same (i.e., the policy and the value function). More specifically, we remove the adjacent information from the task embeddings. We then train this policy network on the same training dataset as DRLTO targeting at LO and use the same hyperparameters (i.e., learning rate, batch size, and the number of gradient steps per episode). During the training, we evaluate the trained policy at each episode on the testing dataset with $n = 15$. Fig. 3.12 shows the evaluation results. We find that the training process cannot converge without considering the dependency information. Note that *DDQNTO* proposed in [121] does not involve the dependency information either. The above experiment results show that *DRLTO* can learn better policies than *DDQNTO* in all scenarios. Therefore, dependency information is one of the crucial factors in achieving good performance.

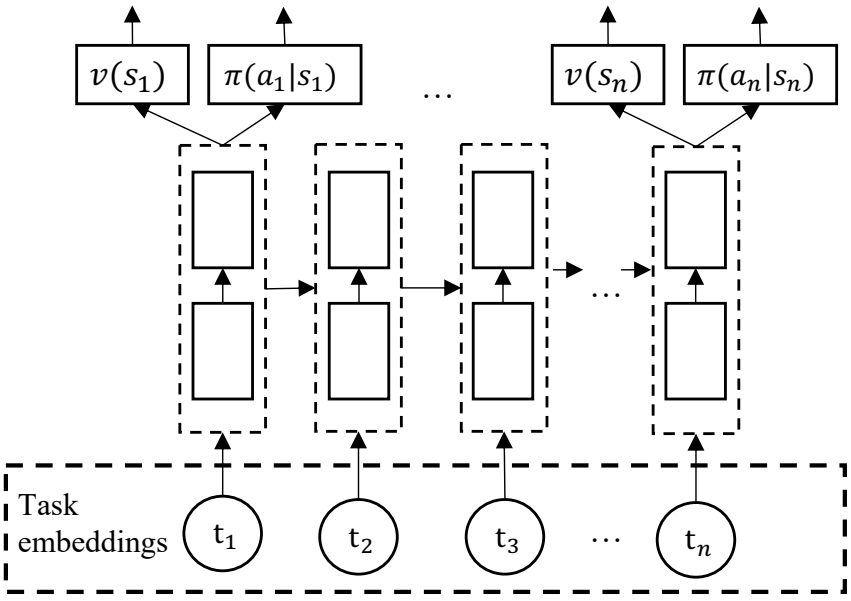


Fig. 3.11 The neural network architecture without considering the dependency among tasks.

3.5 Conclusion

In this chapter, I investigate the task offloading problem in MEC considering task dependency, with the aim of jointly optimizing the latency and energy consumption. To effectively adapt to dynamic scenarios, I propose a new offloading scheme that embeds DRL training and inference procedures into the MEC system. Specifically, I model the offloading problem as an MDP and combine an S2S neural network to approximate both the policy and value function of the MDP. An efficient policy gradient method is then applied for training the S2S neural network. The training results show that the DRLTO achieves excellent stability and convergence with reasonable training and inference overheads. Through comparing with the existing state-of-the-art heuristic and DRL-based algorithms, I demonstrate that the DRLTO has strong adaptability among different MEC scenarios and can obtain near-optimal solutions.

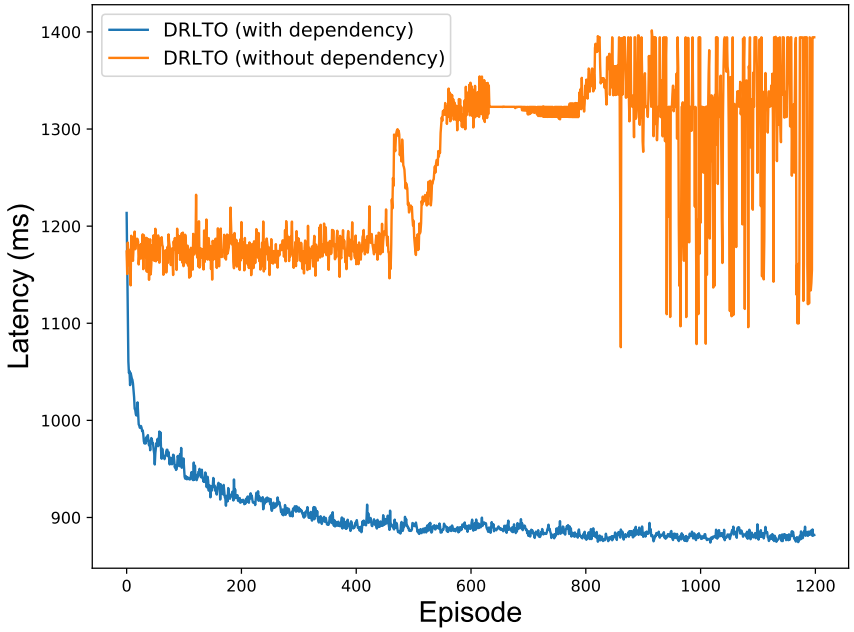


Fig. 3.12 Evaluation results for DRLTO with/without dependency information.

Chapter 4

Fast Adaptive Meta Reinforcement Learning based Task Offloading

In this chapter, I continue the investigation of learning-based task offloading in Multi-access Edge Computing (MEC) systems. In Chapter 3, I present a new method to tackle the dependent task offloading problem by combining Deep Reinforcement Learning (DRL) and Sequence to Sequence (seq2seq) neural network. However, it has low sample efficiency and weak adaptability to new learning tasks (e.g., obtaining an offloading policy for newly joined mobile users). When facing new learning tasks, full retraining is required to learn updated policies. To overcome this weakness, I propose a task offloading method based on meta reinforcement learning, which can adapt fast to new environments with a small number of gradient updates and samples. Similar to the previous work in Chapter 3, I model mobile applications as Directed Acyclic Graphs (DAGs) and the offloading policy by a custom seq2seq neural network.

4.1 Introduction

In real-world scenarios, many mobile applications (e.g., face recognition [100], gesture recognition [100], and augmented reality[6]) are composed of dependent tasks, which can be modelled as a Directed Acyclic Graph (DAG). Thus, offloading dependent tasks in a DAG with the minimum latency is a crucial problem in MEC. Since this problem is NP-hard, many existing solutions are based on heuristic or approximation algorithms [67, 28, 148]. However, these solutions rely heavily on expert knowledge or accurate mathematical models for the MEC system. Whenever the environment of the MEC system changes, the expert knowledge or mathematical models may need to be updated accordingly. Therefore, it is difficult for

one specific heuristic/approximation algorithm to fully adapt to the dynamic MEC scenarios arisen from the increasing complexity of applications and architectures of MEC.

Recently, researchers studied the application of DRL to various MEC task offloading problems [27, 23, 131, 44], including the previous work in Chapter 3. All these methods considered the MEC system including UE, wireless channels, and MEC host as one stationary RL environment and learn an offloading policy through interacting with the environment. However, these methods have weak adaptability for unexpected perturbations or unseen situations (i.e., new environments) like changes of applications, task numbers, or data rates. Because they have low sample efficiency and need full retraining to learn an updated policy for the new environment, they are time-consuming.

Meta learning [127] is a promising method to address the aforementioned issues by leveraging previous experiences across a range of learning tasks to significantly accelerate learning of new tasks. In the context of RL problems, meta reinforcement learning (MRL) aims to learn policies for new tasks within a small number of interactions with the environment by building on previous experiences. In general, MRL conducts two "loops" of learning, an "outer loop" which uses its experiences over many task contexts to gradually adjust parameters of the meta policy that governs the operation of an "inner loop". Based on the meta policy, the "inner loop" can adapt fast to new tasks through a small number of gradient updates [15].

There are significant benefits of adapting MRL to solving the computation offloading problem. Firstly, specific policies for new mobile users can be fast learned based on their local data and the meta policy. Secondly, MRL training in the MEC system can leverage resources from both the MEC host and UE. More specifically, training for the meta policy (outer loop) is run on the MEC host and training for the specific offloading policy (inner loop) is processed on UE. Normally, the "inner loop" training only needs several training steps and a small amount of sampling data, thus the UE with limited computation resources and data is able to complete the training process. Finally, MRL achieves strong adaptability by leveraging the previously learned knowledge and can significantly improve the training efficiency in learning new tasks. Specifically, for the computation offloading problem, the "outer loop" training is used to extract common features (e.g., the user preferences for mobile applications) from the existing mobile users. Note that, the common features can be treated as a good initialisation for the "inner loop" training so that it can help speed up the learning of the offloading policy for the new mobile user.

I propose an MRL-based method that synergizes the first-order MRL algorithm with a sequence-to-sequence (seq2seq) neural network. The proposed method learns a meta offloading policy for all UE and fast obtains the effective policy for each UE based on the

meta policy and local data. To evaluate the performance of the MRLCO under dynamic scenarios, this research considers the following scenarios: 1) Heterogeneous users with personal preferences of mobile applications which are represented as DAGs with different heights, widths, and task numbers. 2) Varying transmission rates according to the distance between the UE and the MEC host.

The major contributions of this chapter can be summarized as follows:

- As far as I known, I am the first of its kind to propose an MRL-based method (MRLCO) to address the computation offloading problem, achieving fast adaptation to dynamic offloading scenarios. MRLCO has high sample efficiency towards new learning tasks, thus it enables UE to run the training process by using its own data even with limited computation resources.
- I propose a new idea to model the dynamic computation offloading process as multiple MDPs, where the learning of offloading policies is decomposed into two parts: effectively learning a meta policy among different MDPs, and fast learning a specific policy for each MDP based on the meta policy.
- I convert the offloading decision process as a sequence prediction process and design a custom seq2seq neural network to represent the offloading policy. An embedding method is also proposed to embed the vertices of a DAG considering both its task profiles and dependencies. In addition, I propose a new training method which combines the first-order approximation and clipped surrogate objective to stabilize the training of the seq2seq neural network.
- I conduct simulation experiments using generated synthetic DAGs according to real-world applications, covering a wide range of topologies, task numbers, and transmission rates. The results show that MRLCO achieves the lowest latency within a small number of training steps compared to three baseline algorithms including a fine-tuning DRL method, a greedy algorithm, and a heterogeneous earliest finish time (HEFT) based heuristic algorithm.

4.2 Problem Formulation

This research shares a similar problem formulation as in Chapter 3 but consider minimal latency as the goal. In MEC systems, the UE makes offloading decisions for those tasks according to the system status and task profiles, thus some tasks are run locally on the UE while others are offloaded to the MEC host via wireless channels. In general, each MEC host

runs multiple virtual machines (VMs) processing the tasks. In this work, I consider that each UE is associated with a dedicated VM providing private computing, communications and storage resources to the UE, the same as in works [114, 14]. The computation capacity (i.e., the number of CPU cores times the clock speed of each core) of an MEC host is denoted as f_s. I consider an equal resource allocation for VMs, i.e., all VMs evenly share the computing resource of the MEC host. Therefore, assuming there are k users in the MEC systems, the computation capacity for each VM is $f_\mathrm{s} = f_s/k$. Formally, I model mobile applications as DAGs, $G = (T, E)$, where the vertex set T represents the tasks and the directed edge set E represents the dependencies among tasks, respectively. Each directed edge is denoted by $\overrightarrow{e} = (t_i, t_j)$, corresponding to the dependency between task t_i and t_j, where t_i is an immediate parent task of t_j, and t_j is an immediate child task of t_i. With the constraint of dependency, a child task cannot be executed until all of its parent tasks are completed. In $G = (T, E)$, this research calls a task without any child task as an *exit task*.

In computation offloading, a computation task can either be offloaded to the MEC host or executed locally on the UE. If task t_i is offloaded, there are three steps to execute t_i. First, the UE sends t_i to an MEC host through a wireless channel. Second, the MEC host runs the received task. Finally, the running result of t_i is returned to the UE. The latency at each step is related to the task profile and the MEC system state. Here, the task profile of t_i includes required CPU cycles for running the task, C_i, data sizes of the task sent, data_i^s, and the result received, data_i^r. Besides, the MEC system state contains the transmission rate of wireless uplink channel, R_ul, and rate of downlink channel, R_dl. Therefore, the latency for sending data, T_i^ul, executing on the MEC host, T_i^s, and receiving result, T_i^dl, of task t_i can be calculated as:

$$T_i^\mathrm{ul} = \mathrm{data}_i^s/R_\mathrm{ul}, \;\; T_i^\mathrm{s} = C_i/f_\mathrm{s}, \;\; T_i^\mathrm{dl} = \mathrm{data}_i^r/R_\mathrm{dl}. \tag{4.1}$$

If task t_i runs locally on the UE, there is only running latency on the UE, which can be obtained by $T_i^\mathrm{l} = C_i/f_\mathrm{l}$ where f_l denotes the computation capacity of the UE. The end-to-end latency of a task offloading process includes local processing, uplink, downlink, and remote processing latency, as shown in Fig. 3.1.

The scheduling plan for a DAG, $G = (T, E)$, is denoted as $A_{1:n} = \{a_1, a_2, \ldots, a_n\}$, where $|T| = n$ and a_i represents the offloading decision of t_i. Tasks are scheduled in a sequence based on the scheduling plan, where all parent tasks are scheduled before their child tasks. I denote FT_i^ul, FT_i^s, FT_i^dl, and FT_i^l as the finish time of task t_i on the uplink wireless channel, the MEC host, the downlink wireless channel, and the UE, respectively. I also denote the available time of these resources when scheduling task t_i as $\mathcal{M}_i^\mathrm{ul}$, $\mathcal{M}_i^\mathrm{s}$, $\mathcal{M}_i^\mathrm{dl}$, and $\mathcal{M}_i^\mathrm{l}$. The resource available time depends on the finish time of the task scheduled immediately before

t_i on that resource. If the task scheduled immediately before t_i does not utilize the resource, I set the finish time on the resource as 0.

If task t_i is offloaded to the MEC host, t_i can only start to send its data when its parent tasks are all completed and the uplink channel is available. Therefore, the finish time on the uplink channel, FT_i^{ul}, can be defined by

$$\begin{aligned} FT_i^{\mathrm{ul}} &= \max\left\{\mathcal{M}_i^{\mathrm{ul}}, \max_{j\in \mathrm{pre}(t_i)}\left\{FT_j^{\mathrm{l}}, FT_j^{\mathrm{dl}}\right\}\right\} + T_i^{\mathrm{ul}}, \\ \mathcal{M}_i^{\mathrm{ul}} &= \max\left\{\mathcal{M}_{i-1}^{\mathrm{ul}}, FT_{i-1}^{\mathrm{ul}}\right\}, \end{aligned} \tag{4.2}$$

where $\mathrm{pre}(t_i)$ denotes the set of immediate predecessors of t_i. Similarly, the finish time of t_i on the MEC host, FT_i^{s}, and that on the downlink channel, FT_i^{dl}, are given by

$$\begin{aligned} FT_i^{\mathrm{s}} &= \max\left\{\mathcal{M}_i^{\mathrm{s}}, \max\left\{FT_i^{\mathrm{ul}}, \max_{j\in \mathrm{pre}(t_i)} FT_j^{s}\right\}\right\} + T_i^{\mathrm{s}}, \\ FT_i^{\mathrm{dl}} &= \max\left\{\mathcal{M}_i^{\mathrm{dl}}, FT_i^{\mathrm{s}}\right\} + T_i^{\mathrm{dl}}, \\ \mathcal{M}_i^{\mathrm{s}} &= \max\left\{\mathcal{M}_{i-1}^{\mathrm{s}}, FT_{i-1}^{\mathrm{s}}\right\}, \\ \mathcal{M}_i^{\mathrm{dl}} &= \max\left\{\mathcal{M}_{i-1}^{\mathrm{dl}}, FT_{i-1}^{\mathrm{dl}}\right\}. \end{aligned} \tag{4.3}$$

If t_i is scheduled on the UE, the start time of t_i depends on the finish time of its parent tasks and the available time of the UE. Formally, the finish time of t_i on the UE, FT_i^{l}, is defined as

$$\begin{aligned} FT_i^{\mathrm{l}} &= \max\left\{\mathcal{M}_i^{\mathrm{l}}, \max_{j\in \mathrm{pre}(t_i)}\left\{FT_j^{\mathrm{l}}, FT_j^{\mathrm{dl}}\right\}\right\} + T_i^{\mathrm{l}}, \\ \mathcal{M}_i^{\mathrm{l}} &= \max\left\{\mathcal{M}_{i-1}^{\mathrm{l}}, FT_{i-1}^{\mathrm{l}}\right\}. \end{aligned} \tag{4.4}$$

Overall, the objective is to find an effective offloading plan for the DAG to obtain the minimal total latency. Formally, the total latency of a DAG given a scheduling plan $A_{1:n}$, $T_{A_{1:n}}^{c}$, is given by

$$T_{A_{1:n}}^{c} = \max\left[\max_{t_k\in\mathcal{K}}\left(FT_k^{\mathrm{l}}, FT_k^{\mathrm{dl}}\right)\right], \tag{4.5}$$

where $\mathcal{K}$ is the set of exit tasks. The problem in Eq. (4.5) is NP-hard, so finding the optimal offloading plan can be extremely challenging due to the highly dynamic DAG topologies and MEC system states. In the next section, I present the details of MRLCO for handling this problem.

Different from the problem defined in Chapter 3, I consider a more complex and dynamic environment where multiple mobile users involve in the MEC system. Specifically, different mobile users have different preferences for mobile applications and positions. Hence, mobile

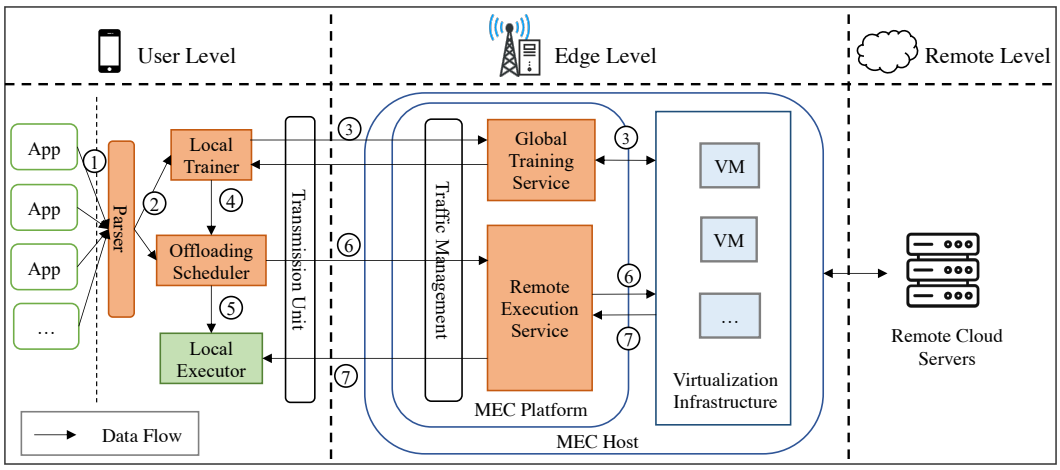


Fig. 4.1 The system architecture of the MRLCO empowered MEC system. The data flows in this architecture include: ① mobile applications, ② parsed DAGs, ③ parameters of the policy network, ④ the trained policy network, ⑤ tasks scheduled to local executor, ⑥ tasks offloaded to the MEC host, ⑦ results from the offloaded tasks.

users can have different distributions of DAG topologies and wireless transmission rates. In this Chapter, the goal is to fast learn an effective personalised offloading policy for each user without requiring uploading DAGs from local mobile devices to MEC servers.

4.3 MRLCO: An MRL-based Computation Offloading Solution

In this section, I first give an overview of the architecture of the MRLCO and explain how it works with the MEC system. Next, I present the detailed MDP modelling for the computation offloading problem. Finally, I describe the implementation of the MRLCO algorithm.

4.3.1 The MRLCO Empowered MEC System Architecture

The MRLCO aims to leverage the computation resources from both the UE and the MEC host for the training process. There are two loops of training — "inner loop" training for the task-specific policy and "outer loop" training for the meta policy. The "inner loop" training is conducted on the UE while the "outer loop" training on the MEC host.

Fig. 4.1 shows an architecture that integrates the MRLCO into an emerging MEC system [104] composed of the user level, edge level, and remote level. Here, the user level includes heterogeneous UE, the edge level contains MEC hosts that provide edge computing services, and the remote level consists of cloud servers. Specifically, mobile users communicate with

an MEC host through the local *Transmission unit*. The MEC host incorporates an MEC platform and a virtualization infrastructure that provides the computing, storage, and network resources. The MEC platform provides *Traffic management* (i.e., traffic rules control and domain name handling) and offers edge services. The five key modules of MRLCO (*parser*, *local trainer*, *offloading scheduler*, *global training service*, and *remote execution service*) can be deployed at the user and edge levels of the MEC system separately, as described below:

- At the user level, the *parser* aims to convert mobile applications into DAGs. The *local trainer* is responsible for the "inner loop" training, which receives the parsed DAGs from the *parser* as training data and uploads/downloads parameters of the policy network to/from the MEC host through local transmission unit. Once the training process is finished, the trained policy network will be deployed to the *offloading scheduler* that is used to make offloading decisions through policy network inference. After making decisions for all tasks of a DAG, the locally scheduled tasks will run on the local executor and the offloaded tasks will be sent to the MEC host for execution.

- At the edge level, the *global training service* and *remote execution service* modules are deployed to the MEC platform. The *global training service* is used to manage the "outer loop" training, which sends/receives parameters of the policy network to/from the UE and deploys the global training process on the virtualization infrastructure in the MEC host. The *remote execution service* is responsible for managing the tasks offloaded from the UE, assigning these tasks to associated VMs, and sending the results back to the UE.

Next, I describe the detailed training process of the MRLCO in the MEC system, as shown in Fig. 4.2. The training process for MRLCO includes four steps. First, the UE downloads the parameters of the meta policy from the MEC host. Next, an "inner loop" training is run on every UE based on the meta policy and the local data, in order to obtain the task-specific policy. The UE then uploads the parameters of the task-specific policy to the MEC host. Finally, the MEC host conducts an "outer loop" training based on the gathered parameters of task-specific policies, generates the new meta policy, and starts a new round of training. Once obtaining the stable meta policy, I can leverage it to fast learn a task-specific policy for new UE through "inner loop" training. Notice that the "inner loop" training only needs few training steps and a small amount of data, thus can be sufficiently supported by the UE. I will present the algorithmic details of the "outer loop" and "inner loop" training in Section 4.3.3.

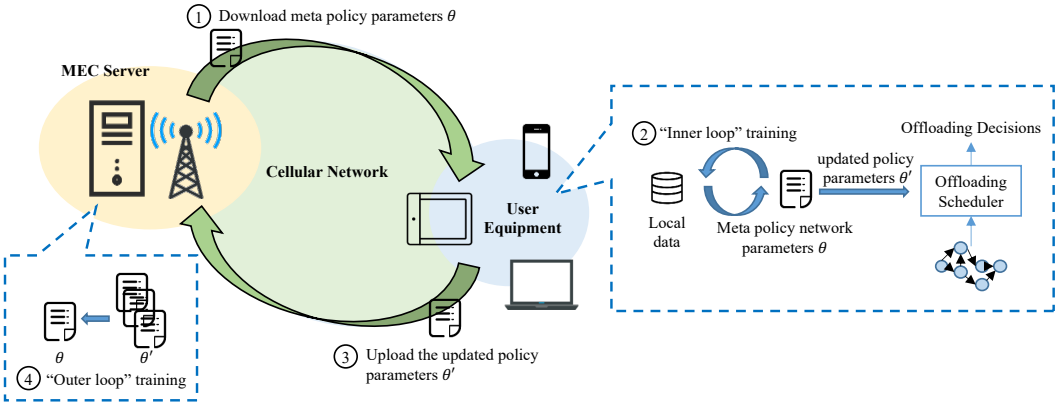


Fig. 4.2 The training process of the MRLCO empowered MEC system includes four steps: 1) the UE downloads the parameters of meta policy, θ, from the MEC host; 2) "inner loop" training is conducted on the UE based on θ and the local data, obtaining the parameters of task-specific policy, θ'; 3) the UE uploads θ' to the MEC host; 4) the MEC host conducts "outer loop" training based on the gathered updated parameters θ'.

4.3.2 Modelling the Computation Offloading Process as Multiple MDPs

To adapt MRL to solve the computation offloading problem, I firstly model the process of computation offloading under various MEC environments as multiple MDPs, where learning an effective offloading policy for one MDP is considered as a learning task. Formally, I consider a distribution over all learning tasks in MEC as $\rho(\mathcal{T})$, where each task $\mathcal{T}_i \sim \rho(\mathcal{T})$ is formulated as a different MDP, $\mathcal{T}_i = (\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{P}_0, \mathcal{R}, \gamma)$. In order to obtain the adaptive offloading policy for all learning tasks, I decompose the learning process into two parts: effectively learning a meta policy among all MDPs and fast learning a specific offloading policy for one MDP based on the meta policy. The definitions of the state, action, and reward for the MDP are listed as follows:

- **State:** When scheduling a task t_i, the latency of running the task depends on the task profile (i.e., required CPU cycles, data sizes), DAG topologies, the wireless transmission rate, and the state of MEC resources. According to Eqs. (4.2), (4.3), and (4.4), the state of MEC resources is related to the offloading decisions of task scheduled before t_i. Therefore, I define the state as a combination of the encoded DAG and the partial offloading plan:

$$\mathcal{S} := \{s_i | s_i = (G(T,E), A_{1:i})\} \quad \text{where} \quad i \in [1, |T|], \tag{4.6}$$

  where $G(T,E)$ is comprised of a sequence of task embeddings and $A_{1:i}$ is the partial offloading plan for the first i tasks. To convert a DAG into a sequence of task embed-

dings, I first sort and index tasks according to the ascending order of the rank value of each task, which is defined as

$$rank(t_i) = \begin{cases} T_i^o & \text{if } t_i \in \mathcal{K}, \\ T_i^o + \max\limits_{t_j \in child(t_i)} \left(rank(t_j)\right) & \text{if } t_i \notin \mathcal{K}, \end{cases} \tag{4.7}$$

where $T_i^o = T_i^{\text{ul}} + T_i^{\text{s}} + T_i^{\text{dl}}$ denotes the latency for task i from starting offloading to finishing execution, $child(t_i)$ represents the set of immediate child tasks of t_i. Each task is converted into an embedding that consists of three elements: 1) a vector that embeds the current task index and the normalized task profile, 2) a vector that contains the indices of the immediate parent tasks, 3) a vector that contains the indices of the immediate child tasks. The size of vectors that embed parent/child task indices is limited to p. I pad the vector with -1, in case the number of child/parent tasks is less than p.

- **Action:** The scheduling for each task is a binary choice, thus the action space is defined as $\mathcal{A} := \{0, 1\}$, where 0 stands for execution on the UE and 1 represents offloading.

- **Reward:** The objective is to minimize $T^c_{A_{1:n}}$ given by Eq. (4.5). In order to achieve this goal, I define the reward function as the estimated negative increment of the latency after making an offloading decision for a task. Formally, when taking action for the task t_i, the increment is defined as $\Delta T_i^c = T^c_{A_{1:i}} - T^c_{A_{1:i-1}}$.

Based on the above MDP definition, I denote the policy when scheduling t_i as $\pi(a_i|G(T,E), A_{1:i-1})$. For a DAG with n tasks, let $\pi\left(A_{1:n}|G(T,E)\right)$ denote the probability of having the offloading plan $A_{1:n}$ given the graph $G(T,E)$. Therefore, $\pi(A_{1:n}|G(T,E))$ can be obtained by applying chain rules of probability on each $\pi(a_i|G(T,E), A_{1:i-1})$ as

$$\pi(A_{1:n}|G(T,E)) = \prod_{i=1}^{n} \pi(a_i|G(T,E), A_{1:i-1}). \tag{4.8}$$

I use the similar seq2seq neural network architecture as in Chapter 3 to represent the policy defined in Eq. (4.8). Fig. 4.3 shows the design of a custom seq2seq neural network, which can be divided into two parts: encoder and decoder. In this work, both encoder and decoder are implemented by recurrent neural networks (RNN). The input of the encoder is the sequence of task embeddings, $[t_1, t_2, ..., t_n]$, while the output of the decoder is the offloading decisions of each tasks, $[a_1, a_2, ..., a_n]$. To improve the performance, I include the attention

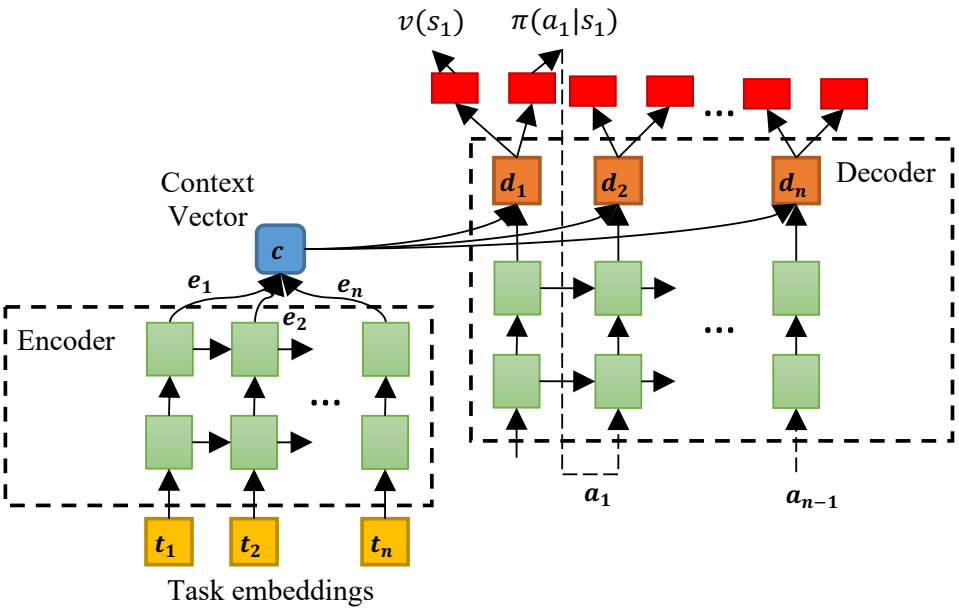


Fig. 4.3 Architecture of the seq2seq neural network in MRLCO. The architecture consists of an encoder and a decoder, where the input of the encoder is the sequence of task embeddings and the output of the decoder is used to generate both policy and value function.

mechanism [10] into the custom seq2seq neural network. Attention mechanism allows the decoder to attend to different parts of the source sequence (i.e., the input sequence of the encoder) at each step of the output generation, thus it can alleviate the issue of information loss caused by the original seq2seq neural network that encodes the input sequence into a vector with fixed dimensions.

Formally, I define the functions of the encoder and decoder as f_{enc} and f_{dec}, respectively. In this work, I use the Long Short-Term Memory (LSTM) as f_{enc} and f_{dec}. At each step of encoding, the output of the encoder, e_i, is obtained by

$$e_i = f_{enc}(t_i, e_{i-1}). \tag{4.9}$$

After encoding all the input task embeddings, I have the output vector as $\mathbf{e} = [e_1, e_2, ..., e_n]$. At each decoding step, I define the output of the decoder, d_j, as

$$d_j = f_{dec}(d_{j-1}, a_{j-1}, c_j), \tag{4.10}$$

where c_j is the context vector at decoding step j and is computed as a weighted sum of the outputs of the encoder:

$$c_j = \sum_{i=0}^{n} \alpha_{ji} e_i. \tag{4.11}$$

The weight α_{ji} of each output of encoder, e_i, is computed by

$$\alpha_{ji} = \frac{\exp\left(\text{score}(d_{j-1}, e_i)\right)}{\sum_{k=1}^{n} \exp\left(\text{score}(d_{j-1}, e_k)\right)}, \tag{4.12}$$

where the score function, $\text{score}(d_{j-1}, e_i)$, is used to measure how well the input at position i and the output at position j match. I define the score function as a trainable feedforward neural network according to the work [10]. I use the seq2seq neural network to approximate both policy $\pi(a_j|s_j)$ and value function $v_\pi(s_j)$ by passing the output of decoder $\mathbf{d} = [d_1, d_2, ..., d_n]$ to two separate fully connected layers. Notice that the policy and value function share most of the parameters (i.e., the encoder and decoder) which are used to extract common features of DAGs (e.g., the graph structure and task profiles). Therefore, training the policy can accelerate the training of value function and vice versa. During training for the seq2seq neural network, the action a_j is generated through sampling from the policy $\pi(a_j|s_j)$. Once the training is finished, the offloading decisions for a DAG can be made by inference through the seq2seq neural network, where the action a_j is generated by $a_j = \arg\max_{a_j} \pi(a_j|s_j)$. Therefore, the time complexity for my algorithm is the same as the inference of the seq2seq neural network with attention, which is $O(n^2)$ [129]. Normally, the task number, n, of a mobile application is less than 100 [75, 67, 28], thus the time complexity of the MRLCO is feasible.

4.3.3 Implementation of MRLCO

MRLCO shares a similar algorithm structure with gradient-based MRL algorithms, which consists of two loops for training. Instead of using VPG as the policy gradient method for the "inner loop" training [31], I define the objective function based on Proximal Policy Optimization (PPO) [108]. Compared to VPG, PPO achieves better exploring ability and training stability. For one learning task $\mathcal{T}_i$, PPO generates trajectories using the sample policy $\pi_{\theta_i^o}$ and updates the target policy π_{θ_i} for several epochs, where θ_i equals θ_i^o at the initial epoch. In order to avoid a large update of the target policy, PPO uses a clipped surrogate objective as

$$J^{\text{C}}_{\mathcal{T}_i}(\theta_i) = \mathbb{E}_{\tau \sim P_{\mathcal{T}_i}(\tau, \theta_i^o)} \left[\sum_{t=1}^{n} \min\left(\text{Pr}_t \hat{A}_t, \text{clip}_{1-\epsilon}^{1+\epsilon}\left(\text{Pr}_t\right) \hat{A}_t \right) \right]. \tag{4.13}$$

Here, θ_i^o is the vector of parameters of the sample policy network. Pr_t is the probability ratio between the sample policy and target policy, which is defined as

$$\text{Pr}_t = \frac{\pi_{\theta_i}(a_t|G(T,E), A_{1:t})}{\pi_{\theta_i^o}(a_t|G(T,E), A_{1:t})}. \tag{4.14}$$

Algorithm 2: Meta Reinforcement Learning based Computation Offloading

Require: Task distribution $\rho(\mathcal{T})$,
Randomly initialize the parameters of meta policy, θ
for *iterations* $k \in \{1, ..., K\}$ **do**
  Sample n tasks $\{\mathcal{T}_0, \mathcal{T}_1, ..., \mathcal{T}_n\}$ from $\rho(\mathcal{T})$
  for *each task* $\mathcal{T}_i \in \{\mathcal{T}_0, \mathcal{T}_1, ..., \mathcal{T}_n\}$ **do**
    Initialize $\theta_i^o \leftarrow \theta$ and $\theta_i \leftarrow \theta$
    Sample trajectories set $D = (\tau_1, \tau_2, \ldots)$ from $\mathcal{T}_i$ using sample policy $\pi_{\theta_i^o}$
    for *iterations* $j \in \{1, ..., m\}$ **do**
      Update parameters θ_i
        $\theta_i \leftarrow \theta_i + \alpha \nabla_{\theta_i} J_{\mathcal{T}_i}^{\mathrm{PPO}}(\theta_i)$
      by mini-batch gradient descent based on D with *Adam*
    end
  end
  Update $\theta \leftarrow \theta + \beta g^{\mathrm{MRLCO}}$ via *Adam*
end

The clip function $\mathrm{clip}_{1-\epsilon}^{1+\epsilon}(\mathrm{Pr}_t)$ aims to limit the value of Pr_t, in order to remove the incentive for moving Pr_t outside of the interval $[1-\epsilon, 1+\epsilon]$. $\hat{A}_t$ is the advantage function at time step t. Specially, I use general advantage estimator (GAE) [107] as the advantage function, which is defined by

$$\hat{A}_t = \sum_{k=0}^{n-t+1} (\gamma\lambda)^k (r_{t+k} + \gamma v_\pi(s_{t+k+1}) - v_\pi(s_{t+k})), \tag{4.15}$$

where $\lambda \in [0, 1]$ is used to control the trade-off between bias and variance. The value function loss is defined as

$$J_{\mathcal{T}_i}^{\mathrm{VF}}(\theta_i) = \mathbb{E}_{\tau \sim P_{\mathcal{T}_i}(\tau, \theta_i^o)} \left[\sum_{t=1}^{n} (v_\pi(s_t) - \hat{v}_\pi(s_t))^2 \right], \tag{4.16}$$

where $\hat{v}_\pi(s_t) = \sum_{k=0}^{n-t+1} \gamma^k r_{t+k}$.

Overall, I combine Eq. (4.13) and Eq. (4.16), defining the objective function for each "inner loop" task learning as:

$$J_{\mathcal{T}_i}^{\mathrm{PPO}}(\theta_i) = J_{\mathcal{T}_i}^{C}(\theta_i) - c_1 J_{\mathcal{T}_i}^{\mathrm{VF}}(\theta_i), \tag{4.17}$$

where c_1 is the coefficient of value function loss. Here, I do not add the entropy bonus as the regularisation term since the "outer loop" training can be seen as a kind of regularisation for the objective function.

Table 4.1 The Neural Network and Training Hyperparameters

Hyperparameter	Value	Hyperparameter	Value
Encoder Layers	2	Encoder Layer Type	LSTM
Encoder Hidden Units	256	Encoder Layer Norm.	On
Decoder Layers	2	Decoder Layer Type	LSTM
Decoder Hidden Units	256	Decoder Layer Norm.	On
Learning Rate β	5×10^{-4}	Learning Rate α	5×10^{-4}
Optimization Method	Adam	Activation function	Tanh
Discount Factor γ	0.99	Loss Coefficient c_1	0.5
Adv. Discount Factor λ	0.95	Clipping Constant ϵ	0.2
Gradient Step m	3		

According to the target of gradient-based MRL defined in Eq. (2.12) and the objective function given by Eq. (4.17), the "outer loop" training target of MRLCO is expressed as

$$\begin{aligned} &J^{\mathrm{MRLCO}}(\theta) = \mathbb{E}_{\mathcal{T}_i\sim\rho(\mathcal{T}),\tau\sim P_{\mathcal{T}_i}(\tau,\theta_i')}[J^{\mathrm{PPO}}_{\mathcal{T}_i}(\theta_i')], \\ &\text{where} \quad \theta_i' = U_{\tau\sim P_{\mathcal{T}_i}(\tau,\theta_i)}(\theta_i,\mathcal{T}_i), \quad \theta_i = \theta. \end{aligned} \tag{4.18}$$

Next, I can conduct gradient ascent to maximize the $J^{\mathrm{MRLCO}}(\theta)$. However, optimizing this objective function involves gradients of gradients, which introduces large computation cost and implementation difficulties when combining a complex neural network such as the seq2seq neural network. To address this challenge, this research uses the first-order approximation to replace the second-order derivatives as suggested in [90], which is defined as

$$g^{\mathrm{MRLCO}} := \frac{1}{n}\sum_{i=1}^{n}\left[(\theta_i'-\theta)/\alpha/m\right], \tag{4.19}$$

where n is the number of sampled learning tasks in the "outer loop", α is the learning rate of the "inner loop" training, and m is the conducted gradient steps for the "inner loop" training.

I present the overall design of the algorithm in Algorithm 2. The parameters of the meta policy neural network are denoted as θ. I firstly sample a batch of learning tasks $\mathcal{T}$ with batch size n and conduct "inner loop" training for each sampled learning task. After finishing the "inner loop" training, I update the meta-policy parameters θ by using gradient ascent $\theta \leftarrow \theta + \beta g^{\mathrm{MRLCO}}$ via *Adam* [53]. Here, β is the learning rate of "outer loop" training.

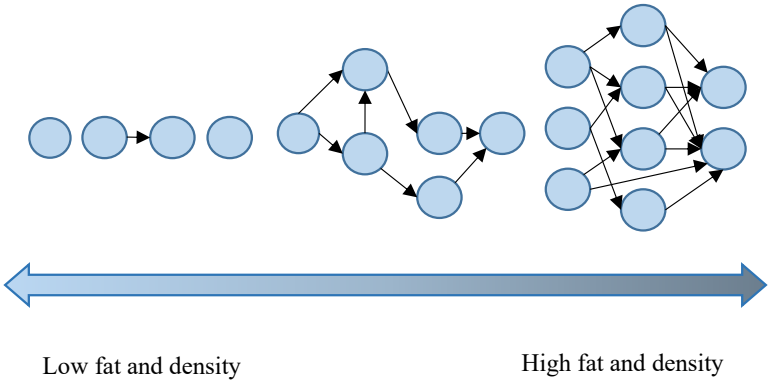


Fig. 4.4 Examples of generated DAGs.

4.4 Performance Evaluation

This section presents the experimental results of the proposed method. First, I introduce the algorithm hyperparameters of MRLCO and the simulation environment. Next, I evaluate the performance of MRLCO by comparing it with a fine-tuning DRL method and a heuristic algorithm.

4.4.1 Algorithm Hyperparameters

The MRLCO is implemented via Tensorflow. The encoder and decoder of the seq2seq neural network are both set as two-layer dynamic Long Short-Term Memory (LSTM) with 256 hidden units at each layer. Moreover, the layer normalization [9] is added in both the encoder and decoder. For the training hyperparameters setting in MRLCO, the learning rate of "inner loop" and "outer loop" are both set as 5×10^{-4}. The coefficient c_1 is set as 0.5 and the clipping constant ϵ is set as 0.2. The discount factors γ and λ are set as 0.99 and 0.95, respectively. The number of gradient steps for "inner loop" training, m, is set as 3. Overall, I summarize the hyperparameter setting in Table 4.1.

4.4.2 Simulation Environment

I consider a cellular network, where the data transmission rate varies with the UE's position. The CPU clock speed of UE, f_l, is set to be 1 GHz. There are 4 cores in each VM of the MEC host with the CPU clock speed of 2.5 GHz per core. The offloaded tasks can run in parallel on all cores, thus the CPU clock speed of a VM, f_s, is $4 \times 2.5 = 10$ GHz.

Many real-world applications can be modelled by DAGs, with various topologies and task profiles. To simulate the heterogeneous DAGs, I implement a synthetic DAG generator

according to [8]. There are four parameters controlling topologies and task profiles of the generated DAGs: n, *fat*, *density*, and *ccr*, where n represents the task number, *fat* controls the width and height of the DAG, *density* decides the number of edges between two levels of the DAG, and *ccr* denotes the ratio between the communication and computation cost of tasks. Fig. 4.4 shows the generated DAGs from low fat and density to high fat and density examples.

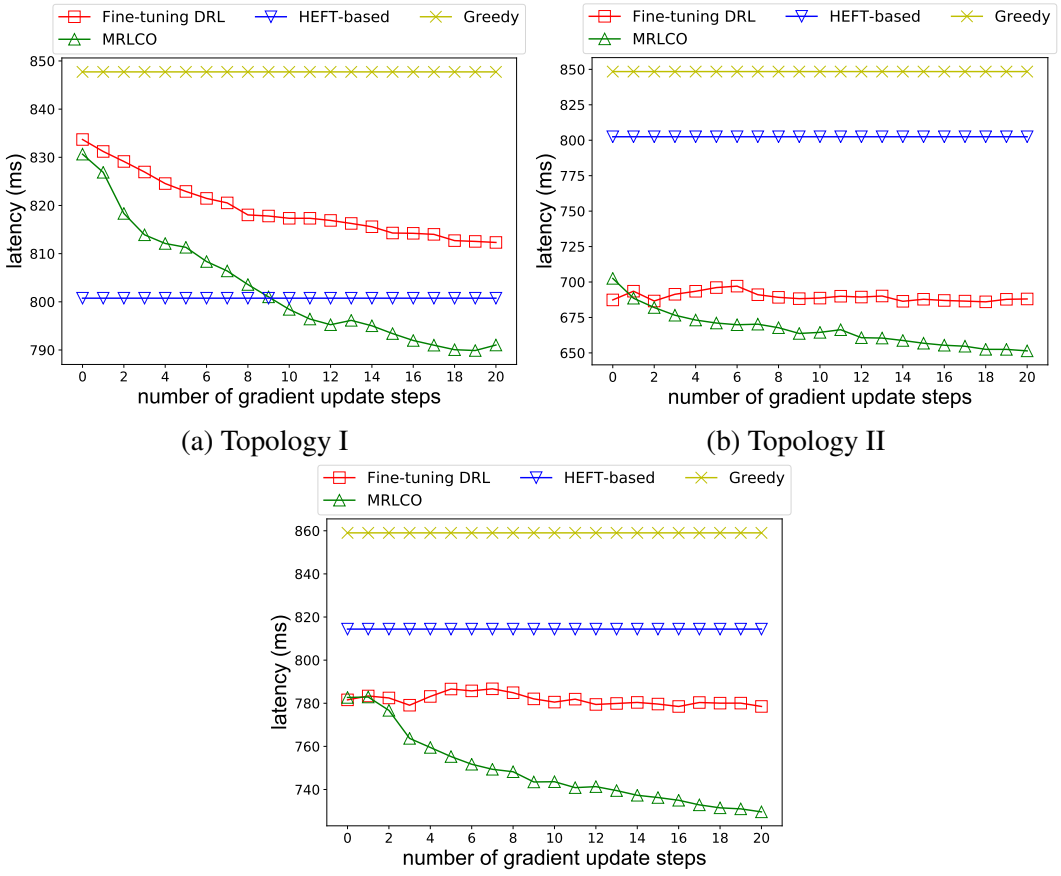
I design three experiments to evaluate the performance of MRLCO under dynamic scenarios. The first two experiments simulate the scenarios where UE has different application preferences represented by various topologies and task numbers. While the third experiment simulates the scenarios where UE has varying dynamic transmission rates. For all experiments, the data size of each task ranges from 5 KB to 50 KB; the CPU cycles required by each task ranges from 10^7 to 10^8 cycles [28]. The length of child/parent task indices vector p is set as 12. I randomly select *ccr* from 0.3 to 0.5 for each generated DAG, since most of mobile applications are computation-intensive. The generated datasets in each experiment are separated into "training datasets" and "testing datasets". I consider learning an effective offloading policy for each dataset as a learning task. The MRLCO firstly learns a meta policy based on "training datasets" by using Algorithm 2. The learned meta policy is then used as the initial policy to fast learn an effective offloading policy for the "testing datasets".

I compare MRLCO with three baseline algorithms:

- *Fine-tuning DRL*: It first pretrains one policy for all "training datasets" using the DRL-based offloading algorithm proposed in [131]. Next, it uses the parameters of the trained policy network as an initial value of the task-specific policy network, which is then updated on the "testing datasets".
- *HEFT-based*: This algorithm is adapted from [67], which firstly prioritizes tasks based on HEFT and then schedules each task with earliest estimated finish time.
- *Greedy*: Each task is greedily assigned to the UE or the MEC host based on its estimated finish time.

4.4.3 Results Analysis

In the first experiment, this research generates DAG sets with different topologies to simulate the scenario where users have different preferences of mobile applications. Each dataset contains 100 DAGs of similar topologies with the same *fat* and *density*, which are two key parameters influencing the DAG topology. I set the task number for each generated DAG as $n = 20$ and set $fat \in \{0.4, 0.5, 0.6, 0.7, 0.8\}$, $density \in \{0.4, 0.5, 0.6, 0.7, 0.8\}$. 25 DAG sets



(a) Topology I

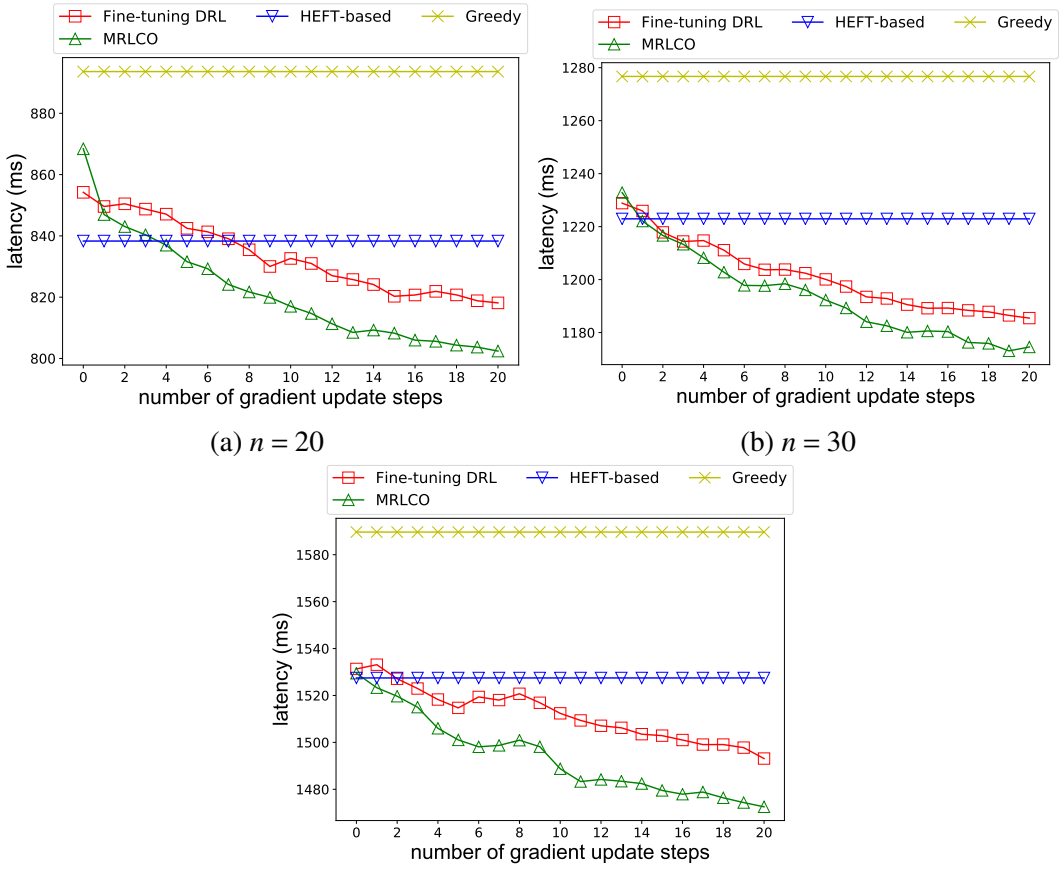
(b) Topology II

(c) Topology III

Fig. 4.5 Evaluation results with different DAG topologies.

are generated with different combinations of fat and density. Each DAG set represents the application preference of one mobile user and consider finding the effective offloading policy for a DAG set as a learning task. I randomly select 22 DAG sets as the training datasets and the other 3 as unseen testing datasets. I train the MRLCO and the fine-tuning DRL method on the training datasets and evaluate MRLCO and baseline algorithms on the testing datasets.

During training of MRLCO, I set the meta batch size as 10, thus 10 learning tasks are sampled from $\rho(\mathcal{T})$ in the "outer loop" training stage. At each "inner loop", I sample 20 trajectories for a DAG and conduct m policy gradient updates ($m = 3$) for the PPO target. After training, I evaluate the MRLCO and fine-tuning DRL method by running up to 20 policy gradient updates, each samples 20 trajectories for a DAG on the testing datasets. Fig. 4.5 shows the performance of the MRLCO and baseline algorithms with different DAG sets. Overall, the Greedy algorithm has the highest latency, while the MRLCO obtains the lowest latency. Fig. 4.5a demonstrates that the MRLCO is better than the HEFT-based algorithm after 9 steps of gradient update, while the fine-tuning DRL method consistently performs



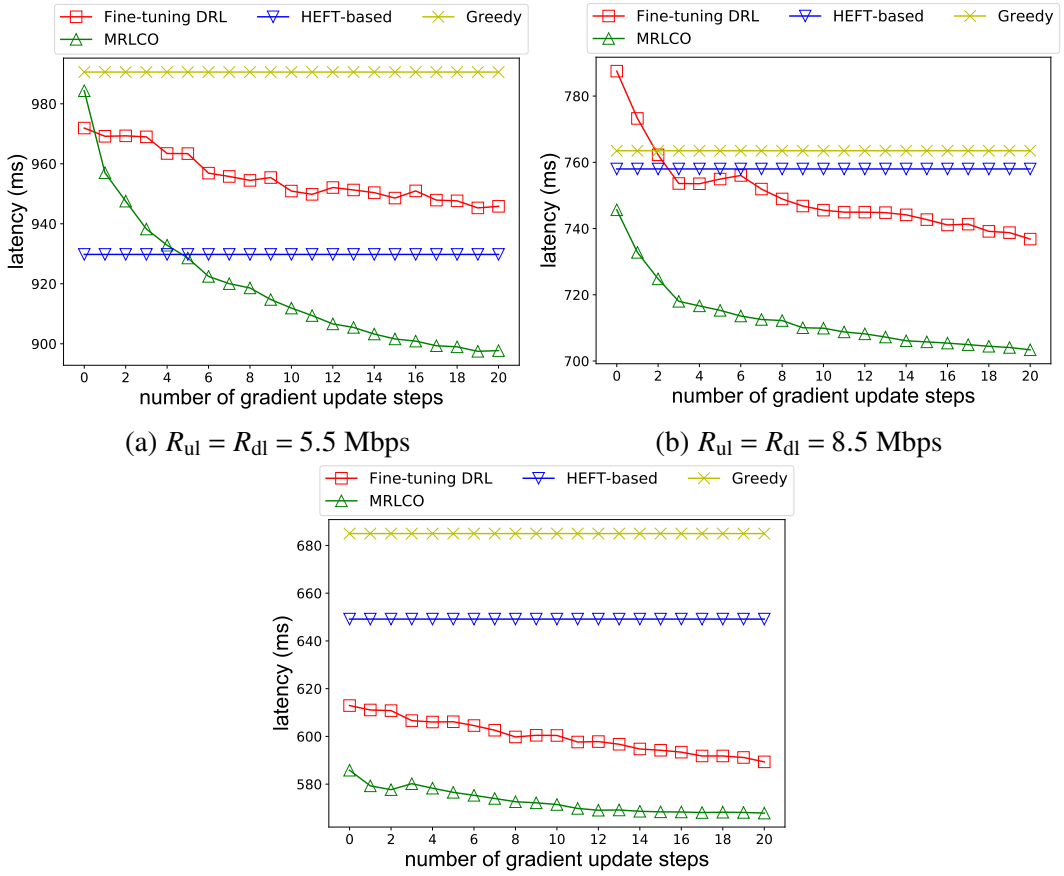
(a) $n = 20$ (b) $n = 30$

(c) $n = 40$

Fig. 4.6 Evaluation results with different task numbers.

worse than the HEFT-based algorithm. This indicates that the MRLCO can adapt to new tasks much more quickly than the fine-tuning DRL method. In Fig. 4.5b and Fig. 4.5c, the MRLCO and the fine-tuning DRL method with 0 step of gradient updates already beat the two heuristic-based algorithms: HEFT-based and the Greedy algorithms, because both the MRLCO and fine-turning DRL learn the updated policy based on the pre-trained models instead of learning from scratch. These heuristic-based algorithms use fixed policies to obtain the offloading plan, which cannot adapt well to different DAG topologies.

The second experiment aims to show the influence of the task number n on the performance of different algorithms. I randomly generate 6 training datasets with $n \in \{10, 15, 25, 35, 45, 50\}$ and 3 testing datasets with $n \in \{20, 30, 40\}$. In each dataset, I generate DAGs by randomly selecting *fat* from {0.4, 0.5, 0.6, 0.7, 0.8}, *density* from {0.4, 0.5, 0.6, 0.7, 0.8}, and *ccr* from 0.3 to 0.5, thus the distributions of DAG topologies of all datasets are similar. In this experiment, I set the meta batch size as 5 and the rest of the settings the same as the first experiment. Fig. 4.6 shows that both the MRLCO and the fine-tuning DRL method



(a) $R_{\text{ul}} = R_{\text{dl}} = 5.5$ Mbps

(b) $R_{\text{ul}} = R_{\text{dl}} = 8.5$ Mbps

(c) $R_{\text{ul}} = R_{\text{dl}} = 11.5$ Mbps

Fig. 4.7 Evaluation results with different transmission rates.

outperform the HEFT-based algorithms after a few gradient updates, and are consistently better than the Greedy from step 0 of gradient updates. Moreover, MRLCO adapts to new learning tasks faster than the fine-tuning DRL method. For example, Fig. 4.6b shows that, after one step gradient update, the latency of MRLCO decreases sharply and is less than both fine-tuning and HEFT-based algorithms. After 20 gradient updates, MRLCO obtains the lowest latency compared to the baseline algorithms.

I conduct the third experiment to evaluate the performance of MRLCO with different transmission rates. Learning the offloading policy for each transmission rate is considered as an individual learning task. I randomly generate the DAG dataset by setting $n = 20$ and other parameters the same as the second experiment. In addition, I implement *Optimal* algorithm via exhaustively searching the solution space to find the optimal offloading plan. I conduct meta training process based on randomly selected transmission rates from 4 Mbps to 22 Mbps with a step size of 3 Mbps. I then evaluate the trained meta policy among transmission rates from {5.5 Mbps, 8.5 Mbps, 11.5 Mbps}, which are unseen in the training procedure.

Fig. 4.7 shows that the MRLCO again adapts to new learning tasks much faster than the fine-tuning DRL method in all test sets and achieves the lowest latency after 20 gradient updates. In some cases (Fig. 4.7b and Fig. 4.7c), MRLCO even achieves the lowest latency at the initial point.

Table 4.2 summarizes the average latency of all algorithms on different testing datasets. Overall, the MRLCO outperforms all heuristic baseline algorithms after 20 gradient update steps. The MRLTO and fine-tuning DRL method will get better results with more update steps. Table 4.2 also shows the performance of the fine-tuning and the MRLCO algorithms after 100 update steps. Compared to the fine-tuning algorithm, the MRLCO achieves better result after both 20 and 100 update steps. However, there are still gaps between the results of MRLCO and the Optimal values. One possible solution could be to integrate the seq2seq neural network with another sample efficient off-policy MRL method [101], which is a direction for future work.

Table 4.2 The comparison of MRLCO and baseline algorithms in average latency (ms) on different testing datasets, N/A denotes cases unable to find optimum.

Test dataset	**Heuristic Algorithms**			**Fine-tuning DRL**		**MRLCO**	
	Optimal	HEFT-based	Greedy	update steps (20)	update steps (100)	update steps (20)	update steps (100)
Toploogy I	679.31	800.75	847.73	812.32	789.92	791.03	722.63
Toploogy II	555.46	802.46	848.43	688.05	636.49	651.42	601.93
Toploogy III	605.05	814.39	859.03	778.52	712.79	729.63	641.92
$n = 20$	689.21	838.31	893.62	818.14	802.50	802.41	743.42
$n = 30$	N/A	1222.93	1276.70	1185.47	1152.07	1174.55	1098.43
$n = 40$	N/A	1527.47	1589.66	1493.11	1432.41	1472.53	1397.63
$R_{\text{ul}} = R_{\text{dl}} = 5.5$ Mbps	770.10	929.79	990.58	945.82	901.36	897.73	831.58
$R_{\text{ul}} = R_{\text{dl}} = 8.5$ Mbps	628.21	757.99	763.49	736.81	701.75	703.38	674.93
$R_{\text{ul}} = R_{\text{dl}} = 11.5$ Mbps	524.14	649.15	684.97	589.33	570.19	567.88	548.26

MRLCO has many advantages over the existing RL-based task offloading methods, such as learning to fast adapt in a dynamic environment and high sample efficiency. Beyond the scope of task offloading in MEC systems, the proposed MRLCO framework has the potential to be applied to solve more decision-making problems in MEC systems. For instance, content caching in MEC aims to cache popular contents at MEC hosts to achieve high Quality-of-Service (QoS) for mobile users and reduce the network traffic. While MEC hosts can have different caching policies to suit the dynamic content preferences and network conditions of users in different areas. The proposed MRL framework can be adapted to solve this problem through executing the "outer loop" training at cloud servers to learn a meta caching policy and the "inner loop" training at MEC hosts to learn a specific caching policy for each MEC host.

Even though MRLCO has many benefits to MEC systems, there are several challenges for further exploration. In this chapter, I consider stable wireless channels, reliable mobile devices, and sufficient computation resources. Thus, the MRLCO will not break down when increasing the number of users. However, when operating at large-scale, some UE as stragglers may drop out due to broken network connections or insufficient power. Considering the synchronous process of "outer loop" training that updates the meta policy after gathering parameters from all UE, the stragglers might affect the training performance of MRLCO. One way to solve this issue is to apply an adaptive client selection algorithm which can automatically filter out stragglers and select reliable clients to join the training process based on their running states.

4.5 Conclusion

In this chapter, I propose an MRL-based approach, namely MRLCO, to solve the computation offloading problem in MEC. Distinguished from the existing works, the MRLCO can quickly adapt to new MEC environments within a small number of gradient updates and samples. In the proposed method, the target mobile applications are modelled as DAGs, the computation offloading process is converted to a sequence prediction process, and a seq2seq neural network is proposed to effectively represent the policy. Moreover, I adopt the first-order approximation for the MRL objective to reduce the training cost and add a surrogate clipping to the objective so as to stabilize the training. I conduct simulation experiments with different DAG topologies, task numbers, and transmission rates. The results demonstrate that, within a small number of training steps, MRLCO achieves the lowest latency compared to three baseline algorithms including a fine-tuning DRL method, a greedy algorithm, and an HEFT-based algorithm.

Chapter 5

Online Service Migration with Incomplete System Information

As a crucial problem in Multi-access Edge Computing (MEC), service migration needs to decide where to migrate user services for maintaining sustainable Quality-of-Service (QoS) when users roam between MEC servers with limited coverage and capacity. However, finding an optimal migration policy is intractable due to the highly dynamic MEC environment and user mobility. Many existing works make centralized migration decisions based on complete system-level information, which can be time-consuming and also suffer from the scalability issue with the rapidly increasing number of mobile users. To address these challenges, I propose a new learning-driven method, namely Deep Recurrent Actor-Critic based service Migration (DRACM), which is user-centric and can make effective online migration decisions by utilizing incomplete system-level information. In the next section, I give an overall introduction to the research problem, challenges, and contributions.

5.1 Introduction

MEC provides many computing and storage resources at the network edge (close to users), which can effectively cut down the application latency and improve the the Quality-of-Service (QoS). Specifically, a mobile application empowered by the MEC consists of a front-end component running on mobile devices, and a back-end service that runs the tasks offloaded from the application on MEC servers [102]. In this way, the MEC enables mobile devices with limited processing power to run complex applications with satisfied QoS.

When considering the user mobility along with the limited coverage of MEC servers, the communications between a mobile user and the user service running on an edge server may

go through multiple hops, which would severely affect the QoS. To address this problem, the service could be dynamically migrated to a more suitable MEC server so that the QoS is maintained. Unfortunately, finding an optimal migration policy for such a problem is non-trivial, due to the complex system dynamics and user mobility. Many existing works [94, 135, 147, 137, 91] proposed service migration solutions based on Markov Decision Process (MDP) or Lyapunov optimization under the assumption of knowing the complete system-level information (e.g., available computation resources of MEC servers, profiles of offloaded tasks, and backhaul network conditions). Thus, they designed centralized controllers (i.e., controllers are placed on edge servers or central cloud) that make migration decisions for mobile users in the MEC system.

The aforementioned methods have two potential drawbacks: 1) in a real-world MEC system, gathering complete system-level information can be difficult and time-consuming; 2) the centralized control approach will have the scalability issue since its time complexity rapidly increases with the number of mobile users. To address the above issues, some works proposed decentralized service migration methods based on contextual Multi-Armed Bandit (MAB) [116, 93, 115], where the migration decisions are made by the user side with partially observed information. However, they did not consider the intrinsically large state space and complex dynamics in the MEC system, which may lead to unsatisfactory performance. A recent work [155] modeled the joint optimization problem of service migration and path selection as a partially observable Markov decision process (POMDP) solved by independent Q-learning, which can be unstable and inefficient when handling the MEC environment with continuous state space (e.g., data size, CPU cycle, workload) and complex system dynamics.

To address the aforementioned challenges, I propose a new method, DRACM, which is user-centric and can learn to make online migration decisions with incomplete system-level information based on Deep Reinforcement Learning (DRL). Specifically, the incomplete system-level information includes the workloads of edge servers, the processing capacity of edge servers, and the network condition of the links among edge servers. DRL is able to solve complex decision-making problems in various areas, including robotics [35], games [152], networks [24], etc., making it an attractive approach. Distinguished from the existing works, I model the service migration problem as a POMDP with continuous state space and develop a tailored off-policy actor-critic algorithm to efficiently solve the POMDP.

This framework is highly related to the scenarios when system-level service placement management is difficult. In practice, a MEC system can include multiple networks and edge/cloud services that are managed by different operators and it is generally difficult to achieve efficient coordinated system-level service management across multiple operators. Therefore, the centralised service migration method may be difficult to deploy in real-world

MEC systems. In contrast, decentralised methods can avoid using system-level information and make efficient migration decisions. Moreover, user-managed service migration can achieve better-personalized service support tailored to user-specific preference when users' demands are highly diverse [93]. I then summarise the main contributions of this work as:

- This research models the service migration problem as a POMDP to capture the intrinsically complex system dynamics in the MEC. I solve the POMDP by proposing a novel off-policy actor-critic method, namely DRACM. Specifically, the distinguishing advantage of this new method is that it is model-free and can quickly learn effective migration policies through end-to-end reinforcement learning (RL), where the agent makes online migration decisions based on the sampled raw data from the MEC environment with minimal human expertise.
- A new encoder network that combines a Long Short-Term Memory (LSTM) and an embedding matrix is designed to effectively extract the hidden information from the sampled histories. Moreover, a tailored off-policy actor-critic algorithm with a clipped surrogate objective function is developed to substantially stabilize the training process and improve the performance.
- This research demonstrates how to implement the DRACM efficiently in an emerging MEC framework, where the migration decisions can be made online through the inference of the policy network, while the training of the policy network can be offline, thus saving the cost of directly interacting with the MEC environment.
- Extensive experiments are conducted to evaluate the performance of the DRACM using real-world mobility traces. The results demonstrate that the DRACM has a stable training process with high adaptivity to different scenarios. Furthermore, it outperforms the online baseline algorithms, and can achieve near-optimal results.

5.2 Problem Formulation of Service Migration

As shown in Fig. 5.1, I consider a typical scenario where mobile users move in a geographical area covered by a set of MEC servers, $\mathcal{M}$, each of which is co-located with a base station. In the MEC system, mobile users can offload their computation tasks to the services provided by MEC servers. I define the MEC server that runs the service of a mobile user as the user's *serving node*, and the MEC server that directly connects with the mobile user as the user's *local server*. In general, the MEC servers are interconnected via stable backhaul links, thus the mobile user can still access its service via multi-hop communication among MEC servers

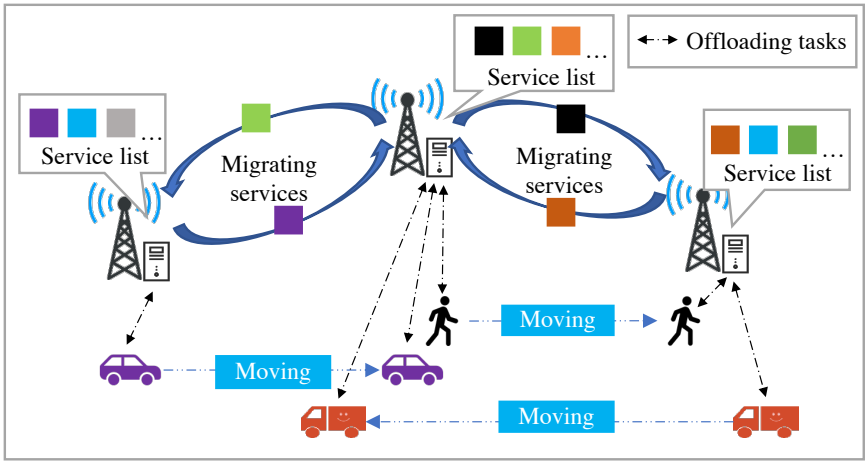


Fig. 5.1 An example of service migration in MEC.

when it is no longer directly connected to the serving node. To maintain satisfactory QoS, the service should be dynamically migrated among the MEC servers as the user moves. In this chapter, I use latency as the measurement for the QoS that consists of migration, computation, and communication delays.

I consider a time-slotted model, where a user's location may only change at the beginning of each time slot. The time-slotted model is widely used to address the service migration problem [137, 93, 135], which can be regarded as a sampled version of a continuous-time model. When a mobile user changes location, the user makes the migration decision for the current service and then offloads computation tasks to the serving node for processing. Denote the migration decision at time slot t as a_t ($a_t \in \mathcal{M}$), where a_t can be any of the MEC servers in this area. In general, the migration, computation, and communication delays are expressed as follows.

Migration delay: The migration delay is incurred when a service is moved out from the previous serving node. In general, the migration delay $B(d_t) = m_t^c d_t$ is a non-decreasing function of d_t [137, 93, 136], where d_t is the hop distance between the current serving node a_t and the previous one a_{t-1}, and m_t^c is the coefficient of migration delay. The migration delay can capture the service interruption time during migration, which increases with the hop distance due to the involved propagation and switching delay of service data transmission.

Computation delay: At each time slot, the mobile user may offload computation tasks to the serving node for processing. The computing resources of MEC servers are shared by multiple mobile users to process their applications. At time slot t, I denote the sum of the required CPU cycles for processing the offloaded tasks as c_t, the workload of the serving node as $w_t^{a_t}$, and the total computing capacity of the serving node as f^{a_t}. I consider a weighted resource allocation strategy on each MEC server, where tasks are allocated with

computation resources proportional to their required CPU cycles. Therefore, the computation delay of running the offloaded tasks at time slot t, can be calculated as

$$D(a_t) = \frac{c_t}{(\frac{c_t}{w_t^{a_t}+c_t} f^{a_t})} = \frac{w_t^{a_t}+c_t}{f^{a_t}}. \tag{5.1}$$

Communication delay: After migrating the service, the communication delay is incurred when the mobile user offloads computation tasks to the serving node. Generally, the communication delay consists of two parts: access delay between the mobile user and the local server, and backhaul delay between the local server and the serving node. The access delay is determined by the wireless environment and the data size of the offloaded tasks. At time slot t, I denote the data size of the offloaded tasks as $data_t$, the average upload rate of the wireless channel as ρ_t. Hence, the access delay can be expressed as

$$R(data_t) = \frac{data_t}{\rho_t}. \tag{5.2}$$

While the backhaul delay is incurred by data transmission, propagation, processing, and queuing between the serving node and the local server through backhaul networks, which mainly depends on the hop distance along the shortest communication path and the data size of the offloaded tasks [155, 137, 93]. I denote the local server at time slot t as u_t ($u_t \in M$) and the hop distance between the serving node a_t and the local server u_t as y_t. The bandwidth of the outgoing link of the local server is denoted as η_t. Generally, the transmission delay of the computation results can be ignored because of the small data size. Consequently, the backhaul delay can be given by

$$P(y_t, data_t) = \begin{cases} 0, & \text{if} \quad y_t = 0, \\ \dfrac{data_t}{\eta_t} + 2\lambda_{\text{bh}} y_t, & \text{if} \quad y_t \neq 0, \end{cases} \tag{5.3}$$

where λ_{bh} is a coefficient of the backhaul delay [155]. Especially, when the serving node and mobile user are directly connected ($y_t = 0$), there is no backhaul cost. Overall, the total communication delay at time slot t can be obtained by

$$E(y_t, data_t) = R(data_t) + P(y_t, data_t). \tag{5.4}$$

Given a finite time horizon T, the objective for the service migration problem is to obtain optimal migration decisions, $\{a_1, a_2, ..., a_T\}$, so that the sum of all the above costs (i.e., total

latency) is minimal. Formally, the objective is expressed as:

$$\begin{aligned} &\min_{a_0,a_1,\ldots,a_T} \sum_{t=0}^{T} B(d_t) + D(a_t) + E(y_t, data_t), \\ &\text{s.t.} \quad a_t \in \mathcal{M}. \end{aligned} \tag{5.5}$$

Obtaining the optimal solution for the above objective is challenging, which requires user mobility and complete system-level information over the entire time horizon. However, in real-world scenarios, it is impractical to gather all the relative information in advance. To address this challenge, I propose a learning-based online service migration method that can make efficient migration decisions based on partially observed information. In the next section, I present the solution in detail.

5.3 Online Service Migration with Incomplete Information

Service migration in MEC is intrinsically a sequential decision-making problem with a partially observable environment (i.e., with incomplete system information), which can be naturally modeled as a POMDP. I solve the POMDP with the proposed DRACM method to provide effective online migration decisions.

5.3.1 POMDP modeling for service migration problem

Key factors that affect the migration decision of a mobile user at a time slot are the mobility of the user, the offloading tasks' profile, the workloads of edge servers, and the resource allocations of edge servers, etc. Ideally, the user can make optimal migration decisions if knowing complete information related to the decision-making process. However, some information are hard to obtain for the user side. For example, at each time slot, the workloads of edge servers are determined by the task requests from their associated mobile users and the available computation resources of edge servers. However, it is unlikely for a mobile user to get such information. To make effective decisions based on partially observable information, POMDP is a natural choice to model the problem, which gives the agent the ability to effectively estimate the outcome of its actions even when it cannot exactly observe the state of its environment. In the POMDP modeling, the mobile user treats the unobserved information (e.g., workloads and resource allocations of MEC servers) as a part of the latent state. Differing from the simplified model such as MAB, POMDP does not ignore the intrinsic large state space and complex dynamics of the service migration problem, thus solving the POMDP can result in more effective decisions.

The detailed POMDP model of service migration is defined as follows:

- *Observation*: The observation contains information that is accessible from the user side, which is defined by a tuple of the local server u_t, the transmission rate of wireless network ρ_t, the required CPU cycles of computation tasks c_t, and the sizes of transmission data, $data_t$:

$$o_t := (u_t, \rho_t, c_t, data_t). \tag{5.6}$$

  Note that the geographical location of the mobile user is an indirect factor that affects the migration decisions, which determines the local server associated with the mobile user and affects the transmission rate (included in the definition of the observation, Eq. (5.6)). Therefore, I define the local server u_t as a component of the observation rather than the geographical location of the mobile user.

- *Action*: At each time slot, the service can be migrated to any of the MEC servers in the area. Therefore, an action is defined as $a_t \in \mathcal{M}$.

- *Reward*: The reward at each time slot is defined as the negative sum of migration, computation, and communication delays, which is formally expressed as

$$r_t := -\left(B(d_t) + D(a_t) + E(y_t, data_t)\right). \tag{5.7}$$

Solving the above POMDP is non-trivial due to the complex dynamics and continuous state space of the MEC environment. In the next subsection, I present the method, DRACM, to solve the above POMDP.

5.3.2 Deep Recurrent Actor-Critic based service Migration (DRACM)

Fig. 5.2 shows the overall architecture of the DRACM, which follows an end-to-end principle with raw history sampled from the environment as input and the migration decisions as output. The DRACM consists of two parts: the encoder network and the learning agent, where the encoder network learns to effectively represent the latent state of the POMDP based on the history and the learning agent learns to make effective migration decisions. The goal of the encoder network is to infer the latent state of the POMDP based on the observed history:

$$p(s_{1:T}|o_{1:T}, a_{1:T-1}) = \prod_{t=1}^{T} p(s_t|s_{t-1}, a_{t-1}, o_t) \tag{5.8}$$

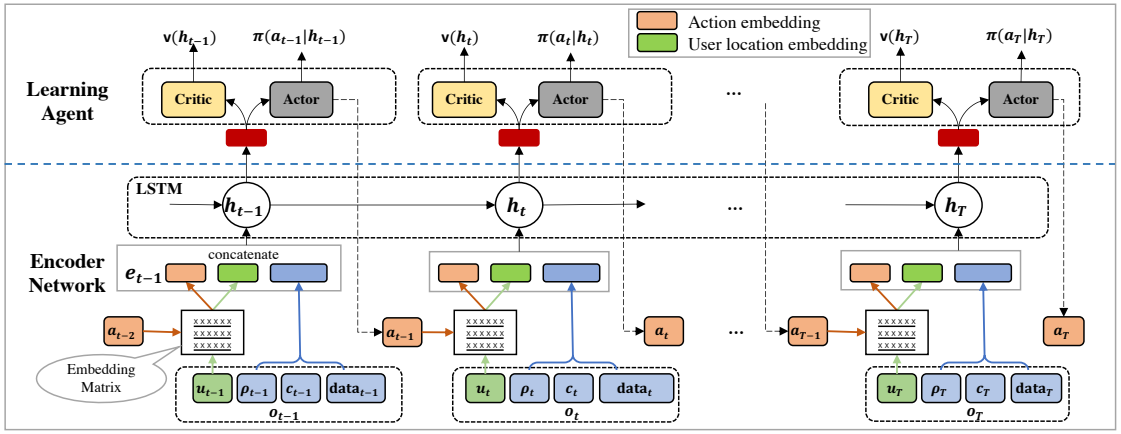


Fig. 5.2 The architecture of the DRACM.

Here, I include a LSTM to approximate the above function where the hidden state of the LSTM, h_t, is used to represent the latent state s_t of the POMDP, thus I have

$$h_t = f_{\text{enc}}([o_{\leq t}, a_{<t}]; \theta) = f_{\text{enc}}([o_t, a_{t-1}], h_{t-1}; \theta), \tag{5.9}$$

where $t \in [1, T]$, f_{enc} and θ represent the inner process and parameters of the encoder network, respectively.

To improve the representation ability of the features u_t and a_{t-1}, I convert them into embeddings by looking up a trainable $|\mathcal{M}| \times d_e$ matrix, where d_e is the dimension of embedding vectors. Subsequently, the action embedding, user location embedding, and the rest components of the observation are concatenated as a vector, e_t, feeding into the LSTM to produce the hidden state h_t.

The learning agent is based on a standard actor-critic structure. Both actor and critic are parametrized by neural networks with the hidden state h_t as input. I denote ϕ and ψ as the parameters of actor and critic networks, respectively. The actor network aims at approximating the policy, $\pi(a_t|h_t;\phi)$, which outputs a distribution over the action space at time step t given h_t. Meanwhile, the critic network, $v(h_t, \psi)$, approximates the value function that is an estimation of the expected return when starting in h_t and following the policy π thereafter.

I denote the trajectory sampled from the environment following the policy π as $\tau = \{o_0, a_0, r_0, ..., o_T, a_T, r_T\}$. The critic network can be updated by minimizing the mean square error of one-step temporal differences δ_t based on the sampled trajectories, which is formally defined as

$$L^{\text{critic}}(\psi, \theta) = \mathbb{E}_{\tau \sim p(\tau|\pi)} \left[\sum_{t=0}^{T} \delta_t^2 \right], \tag{5.10}$$

$$\delta_t = r_t + \gamma v(h_{t+1};\psi) - v(h_t;\psi), \tag{5.11}$$

where the h_t can be obtained by Eq. (5.9) and r_t is the immediate reward obtained at time step t. The objective of the actor is to find an optimal policy that maximizes the accumulated reward, which can be formally expressed as

$$L^{\text{act}}(\phi,\theta) = \mathbb{E}_{\tau\sim p(\tau|\pi)}\left[\sum_{t=0}^{T}\gamma^t r_t\right]. \tag{5.12}$$

The optimal policy can then be obtained by gradient assent through policy gradient with one-step actor-critic [118], where the gradient of the above objective function can be calculated by

$$\nabla_{\theta,\phi} L^{\text{act}} = \mathbb{E}_{\tau\sim p(\tau|\pi)}\left[\sum_{t=0}^{T}\delta_t \nabla_{\theta,\phi}\log\pi(a_t|h_t;\phi)\right]. \tag{5.13}$$

However, directly applying the above on-policy (i.e., using the same policy for training and sampling) objective has some drawbacks when solving the service migration problem. First, I cannot train the policy network offline with mini-batches by using on-policy objective. This can lead to severe sample efficiency problem, since the learning agent needs to resample trajectories from the environment after each gradient update. Especially, in the MEC system, frequently interacting with the environment to get the training samples is costly. Second, the on-policy objective has limited exploring ability, thus the policy can easily get stuck in a local optima. Third, to reduce the variance of the objective function, Eq. (5.13) includes a biased estimator δ_t. However, introducing bias may harm the convergence of the algorithm. To address the above problems, I design an off-policy (i.e., training a policy different from that was used to sample the data) algorithm that can train the policy with mini-batches and reduce the interaction frequency with the environment. Inspired by the previous works on RL [108, 37, 107], I introduce an off-policy training method with a surrogate objective as follows:

$$L_{\text{c}}^{\text{act}}(\phi,\theta) = \mathbb{E}_{\tau\sim p(\tau|\pi')}\left[\sum_{t=0}^{T} g_{\text{clip}}(\pi'_t,\pi_t,\hat{A}_t) + c_h\mathcal{H}(\pi_t)\right], \tag{5.14}$$

$$g_{\text{clip}}(\pi'_t,\pi_t,\hat{A}_t) = \min\left(\frac{\pi_t}{\pi'_t}\hat{A}_t, \text{clip}_{1-\epsilon}^{1+\epsilon}\left(\frac{\pi_t}{\pi'_t}\right)\hat{A}_t\right), \tag{5.15}$$

$$\hat{A}_t(h_t;\psi) = \sum_{l=0}^{T}(\gamma\lambda)^l\delta_{t+l}, \tag{5.16}$$

Algorithm 3: Deep Recurrent Actor-Critic based service Migration (DRACM)

Initialize the parameters of behavior policy ϕ', behavior encoder network θ', target policy ϕ, target encoder network θ, and critic network ψ,
for $k = 0, 1, 2, ..., n$ **do**
  /**Start sampling process**/
  Synchronize the parameters: $\theta' \leftarrow \theta$, $\phi' \leftarrow \phi$.
  Sample a set of trajectories $D_\tau = \{\tau_0, \tau_1, ...\tau_n\}$ by running the behavior policy $\pi'(a_t|h'_t;\phi')$ in the environment, where $h'_t = f_{\text{enc}}([o_{\leq t}, a_{<t}];\theta')$.
  Compute the advantage estimator, $\hat{A}_t$, according to Eq. (5.16).

  /**Start target policy updating process**/
  for $j = 0, 1, 2, ..., m$ **do**
    Update the parameters of encoder network θ, target policy network ϕ, and critic network ψ,
    $\theta \leftarrow \theta + \nabla_\theta L_{\text{c}}^{\text{act}}(\phi, \theta) - \nabla_\theta L^{\text{critic}}(\psi, \theta)$,
    $\phi \leftarrow \phi + \nabla_\phi L_{\text{c}}^{\text{act}}(\phi, \theta)$,
    $\psi \leftarrow \psi - \nabla_\psi L^{\text{critic}}(\psi, \theta)$,
    by mini-batch gradient updates based on collected trajectories D_τ with *Adam*.
  end
end

where $\pi'(a_t|h'_t;\phi')$ is the behavior policy for sampling trajectories, which does not participate in gradient updates. $\pi(a_t|h_t;\phi)$ is the target policy for optimization. $\frac{\pi_t}{\pi'_t}$ is the importance sampling ratio which is used to correct the distribution errors caused by the difference between the behavior and target policies. Besides, I introduce $c_h\mathcal{H}(\pi_t)$ as a regularization term to further encourage exploration during training, where $\mathcal{H}(\pi_t)$ denotes the entropy of the policy and c_h is a coefficient. However, the off-policy method is known for being unstable and hard to coverage. To address this issue, the clip function, $\text{clip}_{1-\epsilon}^{1+\epsilon}$, is used to limit the value of the importance sampling ratio by removing the incentive for moving the ratio outside of the interval $[1-\epsilon, 1+\epsilon]$, thus it can prevent very large policy updates and stabilize the training. To balance the trade-off between variance and bias of the training objective, I utilize the generalized advantage estimator (GAE) [107], $\hat{A}_t$, as given by Eq. (5.16), where $\lambda \in [0, 1]$ is used to control the trade-off between bias and variance. GAE can dramatically reduce the variance of the objective while keeping a tolerable bias level. Different from the work [108], I include an encoder network as a shared part of both actor and critic networks. The encoder network can effectively extract the latent state of POMDP based on the history (i.e., the sequence of previous observations and actions).

Algorithm 3 summarizes the training process of the DRACM. Each training loop consists of the sampling process and the target policy updating process. In the sampling process,

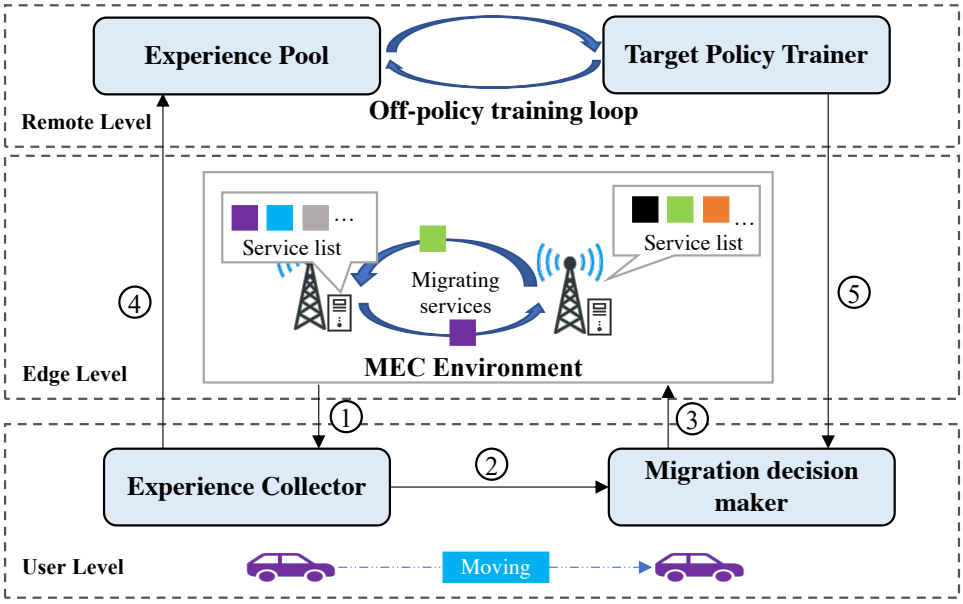


Fig. 5.3 The framework of DRACM empowered MEC system. The data flows in this framework are: ① the observation o_t and reward r_t from the MEC environment, ② the history $H_t = \{o_0, a_0, ..., a_{t-1}, o_t\}$ for migration decision-making, ③ the migration action, a_t, made by the behavior policy, ④ the collected trajectories uploaded to the experience pool, ⑤ the parameters of the trained target policy and encoder networks for service migration.

I firstly synchronize the parameters of the behaviour and target networks (include policy network and encoder network), and then sample a set of trajectories from the environment using the behaviour encoder and policy networks. The advantage estimator, $\hat{A}_t$, can then be obtained based on the sampled trajectories. Next, in the target policy updating process, I conduct training of m loops to update the parameters of the encoder network, policy network, and critic network via mini-batch stochastic gradient descent with *Adam* [53]. After training, the target policy and encoder networks can be deployed to the end device for making online migration decisions by neural network inference, which has a linear time complexity of $O(n)$, where n is the length of the history. In the next subsection, I discuss how to implement the DRACM in the emerging MEC system.

5.3.3 The DRACM empowered MEC framework

The emerging MEC system defined by ETSI consists of three levels: user level, edge level, and remote level [104]. The user level includes various mobile devices such as smartphones and vehicles. The edge level consists of multiple edge servers where each server provides services for processing tasks that are offloaded by mobile users. The edge servers are connected through backhaul links, thus the service can be migrated among them. The remote

level includes data centers with large storage and computing capacity. Fig. 5.3 shows the overall framework of integrating the DRACM into the three-level MEC system. Four key components (*experience collector*, *migration decision maker*, *experience pool*, and *target policy trainer*) of the DRACM are deployed at the user and remote level:

- At the user level, the *experience collector* is responsible of collecting the information of observations and rewards from the MEC environment (Step ①). It sends the history $H_t = \{o_0, a_0, ..., a_{t-1}, o_t\}$ to the *migration decision maker* for online decision-making (Step ②), and the collected trajectories to the *experience pool* for the target policy training (Step ④). The *migration decision maker* includes behavior policy and encoder networks. It downloads parameters from the *target policy trainer* as the initial values of the behavior policy and encoder networks (Step ⑤), and decides the migration actions based on the observed history (Step ③).
- At the remote level, the *experience pool* stores the sampled trajectories from mobile users. The *target policy trainer* is in charge of training the target policy based on the sampled trajectories.

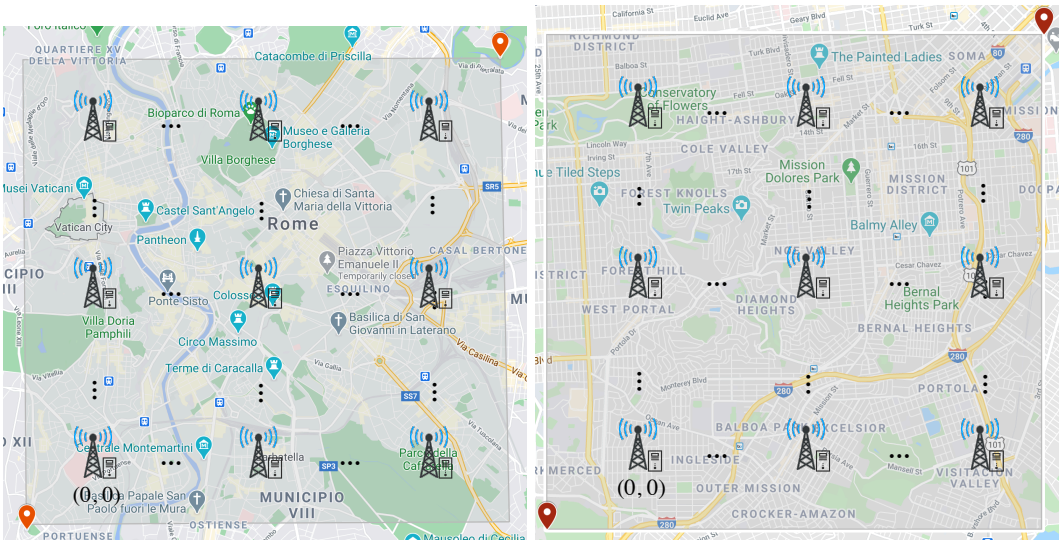
According to Algorithm 3, the *target policy trainer* conducts multiple training loops with mini-batch gradient updates based on the collected trajectories in the *experience pool*. Note that the training can be offline without directly interacting with the MEC environment. After training, the *target policy trainer* sends the updated parameters of policy and encoder networks to mobile users for the next-round of sampling process.

5.4 Experiments

In this section, I present the comprehensive evaluation results of the DRACM in detail. The experiments demonstrate that: 1) the DRACM has a stable and efficient training process; 2) the DRACM can autonomously adapt to different MEC scenarios including various user's task arriving rates, applications' processing densities, and coefficients of migration delay. I firstly introduce the experiment settings based on a real-world MEC environment. Next, I present the baseline algorithms for comparison. Finally, I evaluate the performance of the DRACM and baseline algorithms in different MEC scenarios.

5.4.1 Experiment settings

I evaluate the DRACM with two real-world mobility traces of cabs in Rome, Italy [16] and San Francisco, USA [97]. Specifically, I focus the analysis to the central parts of Rome



(a) The central area of Rome, Italy (b) The central area of San Francisco, US

Fig. 5.4 The central areas of Rome, Italy (8 km × 8 km area bounded by the coordinate pairs [41.856, 12.442] and [41.928, 12.5387]) and San Francisco (8 km × 8 km area bounded by the coordinates pairs [37.709, -122.483] and [37.781, -122.391]).

and San Franscisco, as shown in Fig. 5.4. I consider that 64 MEC servers are deployed in each area, where each MEC server covers a 1 km × 1 km grid with a computation capacity f = 128 GHz (i.e., four 16-core servers with 2 GHz for each core). According to [87], the upload rate of real-world commercial 5G networks is generally less than 60 Mbps. Therefore, in the environment, the upload rate ρ_t in each grid is set as 60, 48, 36, 24, and 12 Mbps from a proximal end to a distal end. The hop distances between two MEC servers are calculated by Manhattan distance. The location of an MEC server is represented by a 2-D vector (i, j) with respect to a reference location at $(0, 0)$. To calculate the propagation latency, I set the bandwidth of backhaul network, η_t, as 500 Mbps [73] and the coefficient of backhaul delay, λ_{bh}, as 0.02 s/hop [155]. The migration delay varies with various service types and network conditions, e.g., the migration delay of Busybox (a type of service) ranges from 2.4 to 3.3 seconds [73] with different bachkhaul bandwidths. Following some related work on MEC [93, 135, 73], I assume the coefficient of migration delay is uniformly distributed in $[1.0, 3.0]$ s/hop during the training.

At each time slot, the tasks arriving at a mobile user and those arriving at an MEC server are sampled from Poisson distributions with rates λ_p^u and λ_p^s, respectively. In the experiments, I show the performance of the DRACM under different task arriving rates of mobile users. According to the current works [89, 21, 158], the data size of an offloaded task in real-world mobile applications often varies from 50 KB (sensor data) [89] to 5 MB

Table 5.1 Parameters of the Simulated Environment.

Parameter	Value
Computation capacity of an MEC server, f	128 GHz
Upload rate of wireless network, ρ_t	{60, 48, 36, 24, 12} Mbps
Bandwidth of backhaul network, η_t	500 Mbps
Coefficient of backhaul delay, λ_{bh}	0.02 s/hop
Coefficient of migration delay, m_t^c	$U[1.0, 3.0]$ s/hop
Data size of each offloaded task	$U[0.05, 5]$ MB
Processing density of an offloaded task, κ	$U[200, 10000]$ cycles/bit
User's task arriving rate λ_p^u	2 tasks/slot
MEC server's task arriving rate λ_p^s	$U[5, 20]$ tasks/slot

Table 5.2 Hyperparameters of the DRACM.

Hyperparameter	Value	Hyperparameter	Value
LSTM Hidd. Units	256	Embedding Dim. d_e	2
Actor Layer Type	Dense	Actor Hidd. Units	128
Critic Layer Type	Dense	Critic Hidd. Units	128
Learning Rate	0.0005	Optimizer	Adam
Discount λ	0.95	Discount γ	0.99
Coefficient c_h	0.01	Clipping Value ϵ	0.2

(image data) [21]. Hence, I set the data size of each offloaded task uniformly distributed in $[0.05, 5]$ MB. The required CPU cycles of each task can be calculated by the product of the data size and processing density, κ, which is uniformly distributed in $[200, 10000]$ cycles/bit, covering a wide range of tasks from low to high computation complexity [57]. I summarize the parameter settings of the simulation environment in Table 5.1.

5.4.2 Baseline algorithms

I compare the performance of the DRACM to that of five baseline algorithms:

- **Always migrate (AM):** A mobile user always selects the nearest MEC server to migrate at each time slot.
- **Never migrate (NM):** The service is placed on an MEC server and never migrate during the time horizon.
- **Multi-armed Bandit with Thompson Sampling (MABTS)**: Some exiting works [116, 93] solve the service migration problem based on MAB. According to the work

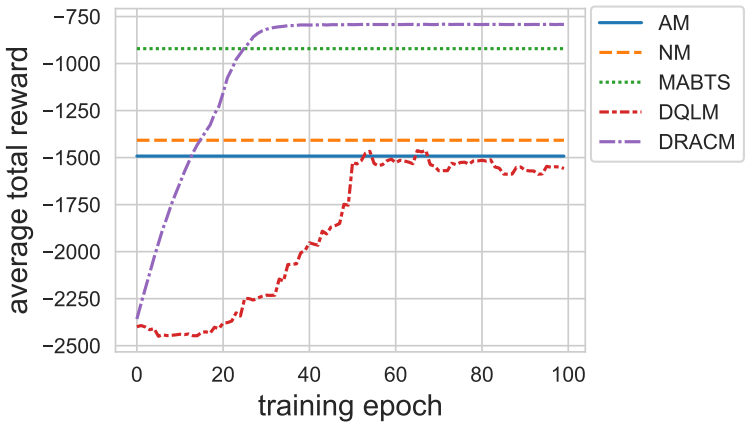


Fig. 5.5 Average total reward of the DRACM and baseline algorithms with the mobility traces of Rome.

[93], MABTS uses a diagonal Gaussian distribution to approximate the posterior of the cost for each arm and applies Thompson sampling to handle the trade-off between exploring and exploiting.

- **DQL-based migrate (DQLM):** Some recent works [135, 147, 155] adapt DQL to tackle the service migration problem. For a fair comparison, I use similar neural network structure as DRACM to approximate the action-value function for DQLM, but use the objective function of the DQL method as the training target. Moreover, I use ϵ-greedy to control the exploring-exploiting trade-off as the above works do.

- **Optimal migrate (OPTIM):** Assuming the user mobility trace and the complete system-level information over the time horizon are known ahead, the service migration problem can be transformed to the shortest-path problem [94, 135], which can be solved by the Dijkstra algorithm.

The NM, AM, MABTS, and DQLM algorithms can run online, while the OPTIM is an offline algorithm which defines the performance upper-bound of service migration algorithms.

5.4.3 Evaluation of the DRACM and baseline algorithms

This research first evaluates the training performance of the DRACM and DQLM on two different mobility trace datasets [16, 97]. Each training dataset includes 100 randomly picked mobility traces, where each trace has 100 time slots of three-minute length each. Table 5.2 lists the hyperparameters in training. The neural network structure of the DQLM is similar to the DRACM with the same encoder network. The difference is that, rather than using the

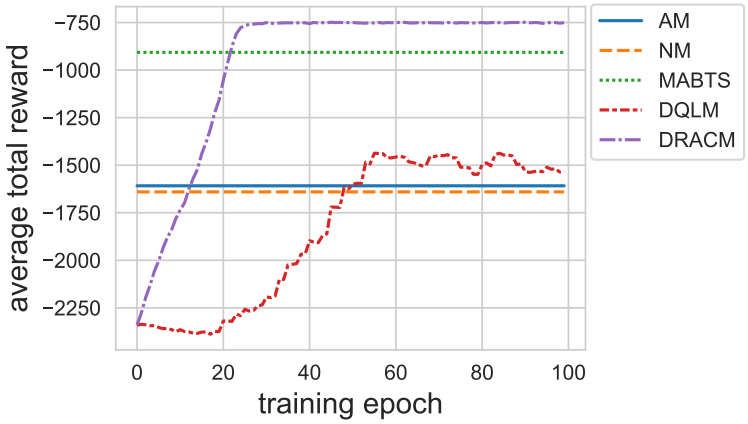


Fig. 5.6 Average total reward of the DRACM and baseline algorithms with the mobility traces of San Francisco.

actor-critic structure, the DQLM is based on the Q-network that includes a fully connected layer with 128 hidden units to approximate the action-value function and chooses the action with the largest action-value at each time step. I train the DQLM and DRACM with the same learning rate, mini-batch size, and number of gradient update steps.

Figs. 5.5 and 5.6 show the training results of DRACM and DQLM on mobility traces of Rome and San Francisco, respectively. The other baseline algorithms do not involve the training process for neural networks, thus I show their final performance. The network parameters of both DRACM and DQLM are initialized by random values, thus they randomly select actions to explore the environment and achieve the worst results compared to other baseline algorithms before training. However, the DRACM quickly surpasses NM and AM after 12 epochs and keeps growing on both mobility traces. After 25 training epochs, the average total reward of the DRACM remains stable, which shows the excellent convergence property of the DRACM. Besides, the final stable results of the DRACM on both mobility traces beat all baseline algorithms.

To evaluate the generalization ability of the DRACM, I test the trained target policy on testing datasets of both mobility traces, where each test dataset includes 30 randomly picked mobility traces that were not included in the training dataset. Figs. 5.7 and 5.8 present the results of the average total latency of DRACM and baseline algorithms on Rome and San Francisco mobility traces, respectively. I found the DRACM achieves the best performance compared to online baseline algorithms on both mobility traces. Specifically, Fig. 5.7 shows that the DRACM outperforms the DQLM and MABTS by 18% and 13%, respectively. Fig. 5.8 indicates that the DRACM surpasses the DQLM and MABTS by 44% and 23%, respectively. Furthermore, the DRACM achieves near-optimal results within 12%

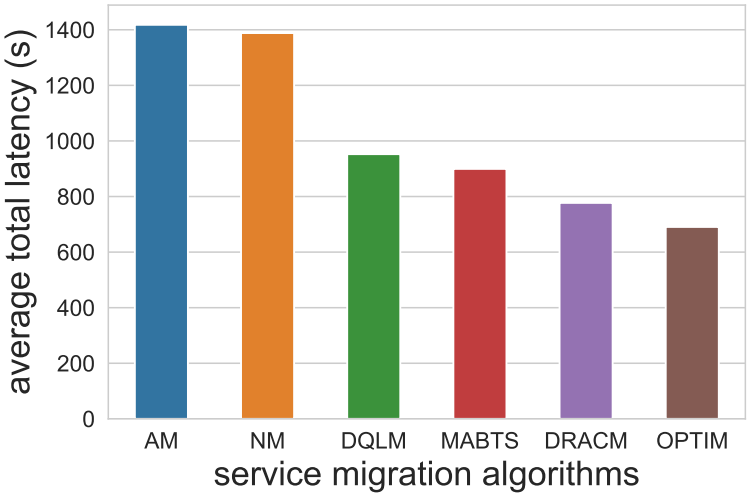


Fig. 5.7 Average total latency (s) of service migration over the time horizon (250 minutes) on the testing dataset from mobility traces of Rome.

of the optimum on both mobility traces. MABTS models the service migration problem as a bandit model which can be seen as a one-step MDP. The bandit model only considers one-step reward which is "short-sighted" compared to the POMDP that includes multi-step rewards. As a consequence, MABTS can easily get stuck in local optima thus achieving unsatisfactory performance. Although DQLM models the service migration problem as POMDP, it uses Q-learning to train the policy, which involves a slow learning process and can be unstable. Hence, the performance of DQLM is even worse than MABTS. In contrast, our method models the service migration problem as a POMDP and solve it with an efficient off-policy policy gradient method.

I then test the DRACM and baseline algorithms with different task arriving rates of users on both mobility traces. As shown in Figs. 5.9 and 5.10, the average total latencies of all evaluated algorithms increase with the rise of user's task arriving rate, since the average number of offloaded tasks increases at each time slot. The evaluation results show that the DRACM adapts well among different task arriving rates of users, where it outperforms the DQLM and MABTS by up to 24% and 45%, respectively. Moreover, in all cases, the results of DRACM are close to the optimal values.

Next, I investigate the performance of the DRACM with different processing densities. For a real-world mobile application, the higher is the processing density, the more computation power is required for processing the application. Figs. 5.11 and 5.12 depict the average total latency of DRACM on Rome mobility traces and San Francisco mobility traces, respectively. I find that the DRACM adapts well to the change of processing density on both mobility traces, where it outperforms all online baselines.

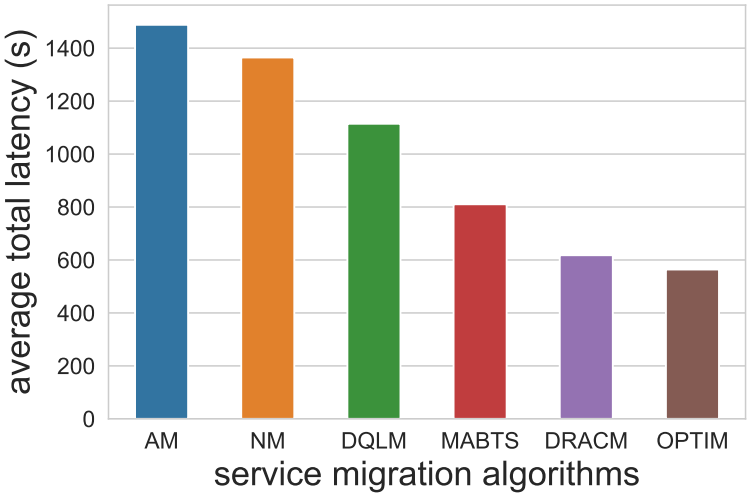


Fig. 5.8 Average total latency (s) of service migration over the time horizon (250 minutes) on the testing dataset from mobility traces of San Francisco.

Migration delay is another important factor that influences the overall latency. To investigate the impact of the migration delay, I evaluate the DRACM and baseline algorithms on the testing datasets with different coefficients of migration delay. Intuitively, when the migration delay is high, a mobile user may not choose to frequently migrate services. As shown in Figs. 5.13 and 5.14, the NM algorithm keeps the similar performance in all cases while the performance of other algorithms drops with the increase of m_c^t. This is because that the NM does not involve the migration process and thus has no migration delay. In Fig. 5.13, I find the MABTS suffers serious performance degradation as m_t^c increases. When the m_c^t is low (e.g., $m_c^t = 1.0$), the MABTS achieves similar results as the DRACM. However, when $m_c^t > 4$, the performance of MABTS becomes even worse than the DQLM. Compared to RL-based methods like the DQLM and DRACM, MABTS is "short-sighted" since it only considers the one-step reward rather than explicitly optimizes the total reward over the entire time horizon. Overall, the DRACM autonomously learns to adapt among the scenarios with different migration delays, which achieves the best performance compared to the online baselines (with up to 25% improvement over the MABTS and up to 42% improvement over the DQLM), and obtains near-optimal results in the experiments.

The DRACM method has many advantages: 1) the learning-based nature of the DRACM makes it flexible among different scenarios with few human expertise; 2) the user-centric design is scalable for the increasing number of mobile users, where each mobile user makes effective online migration decisions based on the incomplete system information; 3) the tailored off-policy training objective improves both performance and stability of the training process; 4) the design of online decision-making and offline policy training makes the

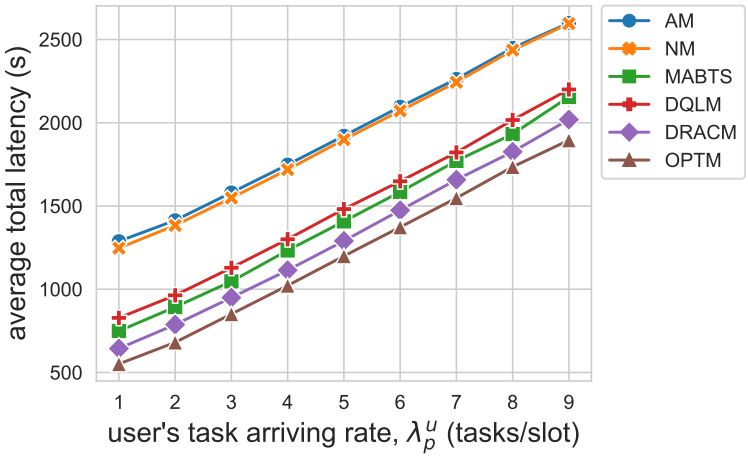


Fig. 5.9 Average total latency (s) of service migration over the time horizon (250 minutes) with different task arriving rates of users (mobility traces of Rome).

DRACM more practical in real-world MEC systems. Beyond the scope of service migration, the framework of the DRACM has the potential to be applied to solve more decision-making problems in MEC systems such as task offloading and resource allocation [77].

Despite that the DRACM has many benefits to solve the service migration problem, there are several challenges for further investigation. In this paper, I consider minimizing the individual cost of each mobile user for service migration. Through POMDP modeling, each mobile user independently learns to solve the service migration problem through trial-and-error with the MEC environment, without considering actions from other mobile users (the actions from other mobile users are treated as part of the environment dynamics). However, when minimizing the global cost of all mobile users, I need to explicitly consider the cooperation among users to find the best policy during training. One way to solve this problem is to include cooperative multi-agent reinforcement learning [162] in the training process, where agents exchange/share local information with their neighbours. In addition, since sampling experiences from real-world MEC systems can be time-consuming and costly, the sample efficiency of the DRACM needs to be further improved. During each training loop, the DRACM only uses the experiences that are sampled from the current behaviour policy and discards the previous gathered experiences, which might not make full use of the gather experiences. A potential solution to address this issue is to develop a more efficient offline reinforcement learning [60] method that can utilize previously collected experiences from different migration policies. Offline reinforcement learning algorithms hold tremendous application promise in the field of edge computing since it can learn effective policy through previous gathered experiences (those experiences can be sampled by carefully designed

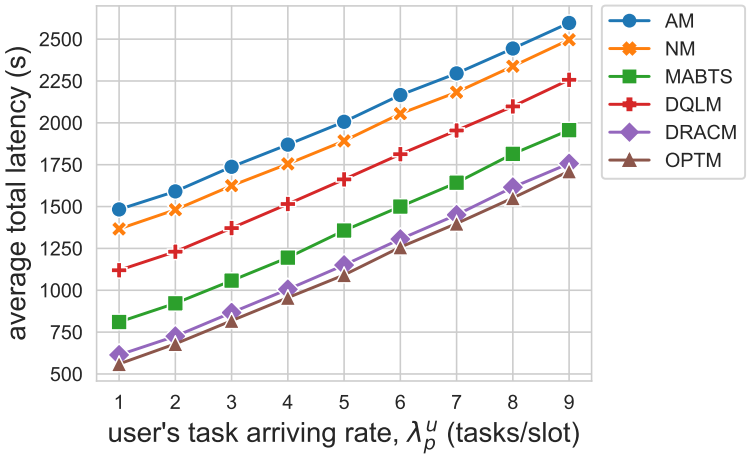


Fig. 5.10 Average total latency (s) of service migration over the time horizon (250 minutes) with different task arriving rates of users (mobility traces of San Francisco).

expert algorithms or human operators) even without further interaction with the environment [33].

5.5 Conclusion

In this chapter, I propose the DRACM, a new method for solving the service migration problem in MEC given incomplete system-level information. The proposed method is completely model-free and can learn to make online migration decisions through end-to-end RL training with minimal human expertise. Specifically, the service migration problem in MEC is modelled as a POMDP. To solve the POMDP, I design an encoder network that combines an LSTM and an embedding matrix to effectively extract hidden information from sampled histories. Besides, I propose a new tailored off-policy actor-critic algorithm with a clipped surrogate objective to improve the training performance. I demonstrate the implementation of the DRACM in the emerging MEC framework, where migration decisions can be made online from the user side and the training for the policy can be offline without directly interacting with the environment. This research evaluates the DRACM and four online baseline algorithms with real-world datasets and demonstrates that DRACM consistently outperforms the online baselines and achieves near-optimal results on a diverse set of scenarios.

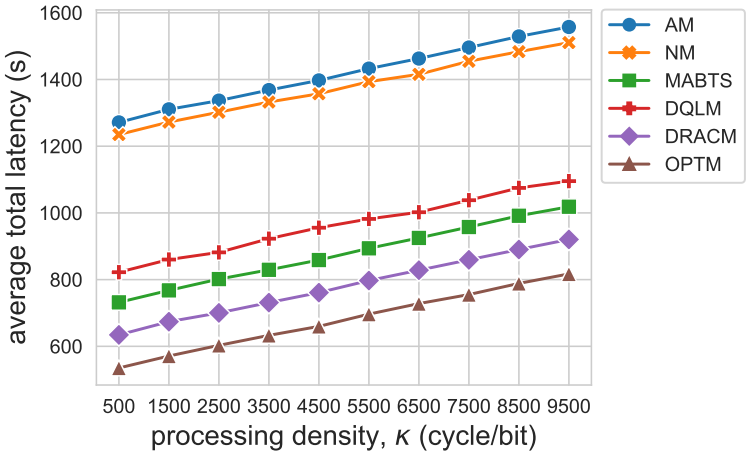


Fig. 5.11 Average total latency (s) of service migration over the time horizon (250 minutes) with different processing densities (mobility traces of Rome).

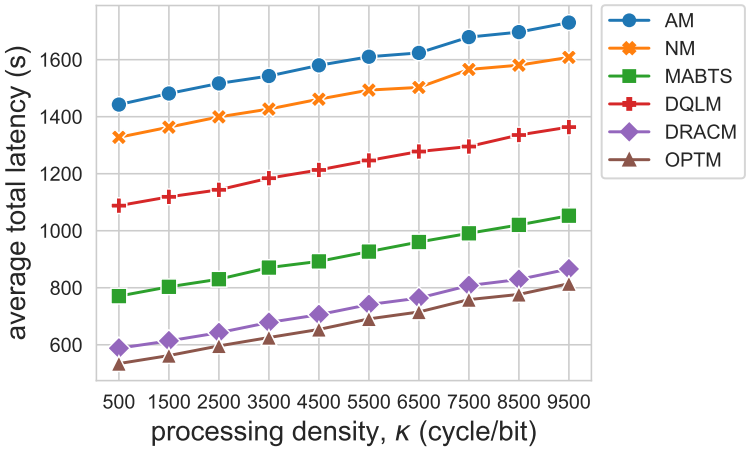


Fig. 5.12 Average total latency (s) of service migration over the time horizon (250 minutes) with different processing densities (mobility traces of San Francisco).

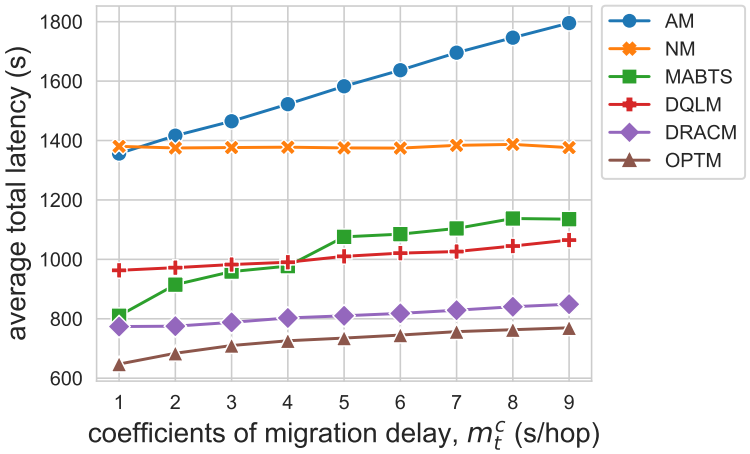


Fig. 5.13 Average total latency (s) of service migration over the time horizon (250 minutes) with different coefficients of migration delay (mobility traces of Rome).

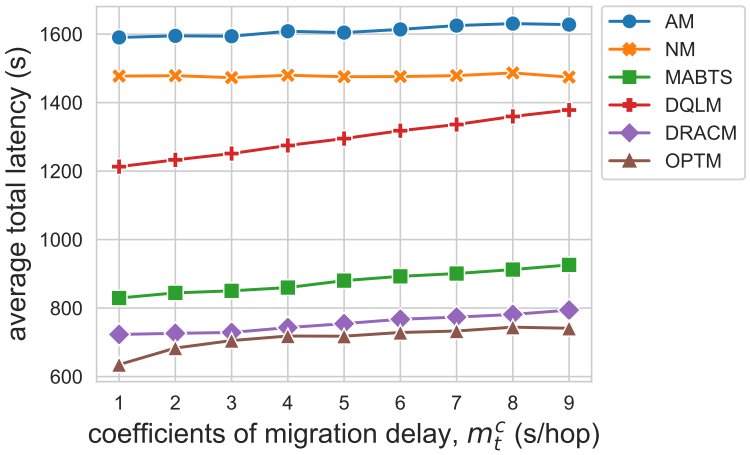


Fig. 5.14 Average total latency (s) of service migration over the time horizon (250 minutes) with different coefficients of migration delay (mobility traces of San Francisco).

Chapter 6

Conclusion

6.1 Summary

Optimisation and maintenance for Multi-access Edge Computing systems have been of vital importance for maintaining high Quality-of-Service (QoS), reducing costs, and saving energy. Over the past decades, system optimisation and maintenance are traditionally the responsibility of human experts. However, with the increasing complexity of the system architecture, effectively optimising such sophisticated systems becomes intractable. This thesis takes a step forward to solve system optimisation problems in MEC with Deep Reinforcement Learning (DRL), letting machines learn optimal policies to optimise themselves. To demonstrate the effectiveness of DRL-based methods, this thesis investigates two crucial optimisation problems in MEC systems: task offloading and service migration.

Chapter 3 considers the task offloading problem for mobile applications, where the applications are modelled as Directed Acyclic Graphs (DAG). The optimisation target is to minimise a weighted sum of energy consumption and processing latency. To address the problem, this research proposes a DRL-based offloading framework, DRTO, which combines an off-policy policy gradient method with a Sequence-to-Sequence (seq2seq) neural network. Extensive experiments demonstrate that the proposed method achieves superior performance than human-designed heuristic algorithms and close to the optimal solution with quadratic time complexity.

Chapter 4 keep investigating the dependent task offloading problem. Although the proposed DRL-based method in Chapter 3 can achieve good performance for a learning task, retraining for the policy is required when facing a new learning task, which is time-consuming. To address the problem, this research enhances the DRL-based task offloading framework with Meta Reinforcement Learning (MRL), which learns a meta-policy from a range of learning tasks and uses the meta-policy to speed up the learning of new tasks. Simulation

experiments demonstrate that the proposed method can fast learn effective offloading policies for new learning tasks with few update steps and training data.

Chapter 5 studies the service migration problem in MEC considering user mobility. Different from many existing studies, this thesis considers a decentralised decision-making process where migration decisions are made by mobile users. This research then models the service migration problem as a Partially Observable Markov Decision Process (POMDP) and proposed an actor-critic algorithm that combines Long Short-Term Memory (LSTM) for the extraction of hidden state to solve the POMDP. This research evaluates the proposed method with real-world mobility datasets. The results show that the proposed method consistently outperforms the state-of-the-art online migration algorithms and achieves near-optimal results on a diverse set of scenarios.

6.2 Future Works

Integrating machine learning technologies into MEC systems has become a promising trend. Recently, lots of pristine and interesting research problems and directions have been proposed. In the following section, this thesis outlines some important research directions for future work.

6.2.1 Offline Model-based Reinforcement Learning Algorithms for System Optimisation

To solve the system optimisation problems by DRL, there are still some challenges for the implementation of DRL-based methods in real-world MEC systems. First, training DRL generally requires a large amount of data, however collecting experiences from real-world MEC systems can be costly. Besides, in some cases, the learning agent can not directly interact with the edge system during training due to the delayed system rewards. Therefore, the training algorithm should be sample efficient and offline (i.e. the training is based on the pre-gathered log data from the system). There are two promising advanced reinforcement learning (RL) methods can address the above issue: Model-based RL [84] and offline RL algorithms [60]. Specifically, model-based RL algorithms have been demonstrated to achieve much higher sample efficiency than model-free one while offline RL algorithms utilize previously collected data without additional online data collection.

Recently, both model-based RL and offline RL has attracted lots of research interest. Dyna [117] can be seen as one of the early model-based RL algorithms, which alternates between model learning, data generation under a model, and policy learning using the model

data. More recently, Luo *et al.* [71] proposed a novel algorithmic framework for designing and analysing model-based RL with theoretical guarantees. Janner *et al.* [47] developed a new algorithm that uses short model-generated rollouts branched from real experience. Their method can match the asymptotic performance of the best model-free algorithms while achieve much lower sample complexity. Offline RL algorithms aim to learn effective policy from static datasets without further interaction with the environment. However, there are still challenges to develop effective offline RL methods. For example, how to properly tackle the shift between the dataset and the learned policy? Kumar *et al.* [56] proposed conservative Q-learning (CQL) to avoid the overestimation of Q-values caused by the distribution shift. A recent work [52] integrated model-based RL and offline RL which learned a pessimistic MDP (P-MDP) from the offline datasets and train a near-optimal policy for the P-MDP.

Although some model-based RL and offline RL methods have demonstrated their effectiveness in gaming or robotics environment, seldom of them have been integrated into edge computing systems. However, the high sample complexity and online training paradigm of conventional RL algorithms are the main factors that hind them from deploying in real-world systems. To tackle these issue, model-based RL and offline RL are promising methods for system optimisation problems.

6.2.2 Reinforcement Learning for System Optimisation Problems with Constraints

In general, the optimisation targets of traditional RL methods do not explicitly consider the constraints. However, many real-world system optimisation problems are Constraint-Satisfaction Problems (CSPs) where the state of the objectives must satisfy several constraints or limitations. For example, in some scenarios, the task offloading problem needs to satisfy hard deadline constraints (i.e., deadline constraints for some tasks should never be violated) [41, 123] or computing resource constraints [69]. Most of the existing RL methods are built on the MDP without considering the constraints on states, thus it is hard for them to tackle CSPs.

To solve the above challenges, it is necessary to redesign the existing RL algorithms to satisfy the system constraints in MEC. A theoretical framework of Constrained Markov Decision Processes (CMDPs) is introduced by Altman [7] in 1999. In CMDPs, the objective is to maximize accumulated reward while satisfying some linear constraints over auxiliary costs. Altman [7] presented a Linear Programming (LP) based methods to handle CMDPs that the model dynamics are known. In high-dimensional environment settings, Achiam *et al.* [2] approximated the constrained objective with an unconstrained one and sought to

ensure approximate constraint satisfaction during the training process. Nachum *et al.* [85] developed a special duality-based solution to RL algorithms, which transforms the traditional constraint-satisfaction mathematical form to an unconstrained one. Miryoosefi *et al.* [81] proposed an algorithmic scheme that can handle various types of constraints with rigorous theoretical guarantees. Despite the above RL solutions for CSPs, most of them have some assumptions that can hardly be satisfied in real-world systems (e.g., the constraints should be convex). Exploring new RL frameworks to address system optimisation problems under constraints can be a promising and important research direction.

6.2.3 Safe and Robust Reinforcement Learning in MEC systems

RL fundamentally involves exploring and exploiting processes where the agent needs to visit different states and actions in order to find an optimal policy. The context-free random exploring methods such as ϵ-greedy have no constraints for the action, thus may bring the system into an unsafe region and cause catastrophic consequences. In addition, the trained policies of conventional RL algorithms are fragile to the environment perturbation. For safety-critical applications, the failure of a learned policy may lead to dangerous situations. Unfortunately, MEC systems generally contain unstable and noisy dynamics due to the complex and dynamic wireless communication and highly heterogeneous edge devices. Some applications in MEC systems including self-driven vehicles [70] and cloud robotics [3] are highly safety-sensitive, thus we need to assure a safe exploring process during RL training and generate robust policies for deployment.

Safe and robust RL has been an active area in recent years [95, 34, 96]. A safe and robust RL system should ensure reasonable system performance and/or respect safety constraints during the learning and/or deployment processes [34]. In the software engineering area, formal verification is concerned with the rigorous mathematical specification, design, and verification of systems [146]. Therefore, a natural starting point is to consider formal methods to verify the DRL-based solutions. However, since DRL combines DNNs that are black-box models for decision-making, verifying DRL still faces lots of challenges [109]. To prevent the failure of the trained policy in turbulent environments, the policy should be robust against the adversarial input. To address this problem, some studies [128, 96] explicitly involve adversarial training into their algorithm, thus create policies that are robust to differences in training/test conditions. However, these methods still lack theoretical guarantees. Building safe and robust RL for real-world systems is an open research area where lots of issues and challenges remain unsolved.

6.2.4 Federated Reinforcement Learning in Edge Computing Systems

In MEC systems, the training of RL algorithms is generally conducted on the central Cloud, thus requiring mobile users to upload their local data. However, in many real-world applications, data from clients is often decentralised and privacy sensitive while the client devices (e.g., smartphones, tablets, and IoT devices) generally have limited samples and computing resources. Consequently, it is challenge for traditional RL methods to learn effective models for those applications while keep the data stored on the individual client without harming the privacy.

To address the above challenges, Federated Learning (FL) is an emerging solution that aims to collaboratively train Machine Learning (ML) models in a distributed fashion without sensitive user data leaving the devices where it was generated. Many popular FL algorithms for supervised learning, such as Federated Averaging (FedAvg) [79], work in an iterative fashion, with rounds of local training on clients, followed by model uploading and aggregation on a server. Therefore only model parameters, and not sensitive user data, is uploaded by clients. However, the majority of FL works consider training supervised models, how to extend FL to RL paradigm remains an interesting and open challenge [49].

Several previous works have investigated training RL policies in the FL setting. Nadiger et al. [86] proposed a system for training virtual Pong players (controlled via a Deep Q-network) in the FL setting to match the skill levels of (simulated) players. In [141] and [140], the authors design systems for training RL policies to maximise cache-hits for user content on base stations near the network edge. Zhou et al. [167] designed the FedRL system for training a policy, where individual FL clients do not each have access to the full state-space of the RL task. Despite this previous research interest, none of these previous works provide rigorous theoretical analysis about combining RL and FL. Therefore, investigating effective methods for the integration of RL and FL in MEC systems and providing rigorous theoretical analysis create interesting opportunities and challenges for the applications of RL methods in real-world MEC systems.

References

[1] Abbas, N., Zhang, Y., Taherkordi, A., and Skeie, T. (2021). Reward-oriented task offloading under limited edge server power for multi-access edge computing. *IEEE Internet of Things Journal*.

[2] Achiam, J., Held, D., Tamar, A., and Abbeel, P. (2017). Constrained policy optimization. In *International Conference on Machine Learning*, pages 22–31. PMLR.

[3] Afrin, M., Jin, J., Rahman, A., Rahman, A., Wan, J., and Hossain, E. (2021). Resource allocation and service provisioning in multi-agent cloud robotics: A comprehensive survey. *IEEE Communications Surveys & Tutorials*.

[4] Aissioui, A., Ksentini, A., Gueroui, A. M., and Taleb, T. (2018). On enabling 5g automotive systems using follow me edge-cloud concept. *IEEE Transactions on Vehicular Technology*, 67(6):5302–5316.

[5] Akkaya, I., Andrychowicz, M., Chociej, M., Litwin, M., McGrew, B., Petron, A., Paino, A., Plappert, M., Powell, G., Ribas, R., et al. (2019). Solving rubik's cube with a robot hand. *arXiv preprint arXiv:1910.07113*.

[6] Al-Shuwaili, A. and Simeone, O. (2017). Energy-efficient resource allocation for mobile edge computing-based augmented reality applications. *IEEE Wireless Communications Letters*, 6(3):398–401.

[7] Altman, E. (1999). *Constrained Markov decision processes*, volume 7. CRC Press.

[8] Arabnejad, H. and Barbosa, J. G. (2013). List scheduling algorithm for heterogeneous systems by an optimistic cost table. *IEEE Transactions on Parallel and Distributed Systems*, 25(3):682–694.

[9] Ba, J. L., Kiros, J. R., and Hinton, G. E. (2016). Layer normalization. In *Advances in Neural Information Processing Systems, NIPS*.

[10] Bahdanau, D., Cho, K., and Bengio, Y. (2015). Neural machine translation by jointly learning to align and translate. In *International Conference of Learning Representations, ICLR*.

[11] Battaglia, P. W., Hamrick, J. B., Bapst, V., Sanchez-Gonzalez, A., Zambaldi, V., Malinowski, M., Tacchetti, A., Raposo, D., Santoro, A., Faulkner, R., et al. (2018). Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*.

[12] Bellman, R. (1966). Dynamic programming. *Science*, 153(3731):34–37.

[13] Bello, I., Pham, H., Le, Q. V., Norouzi, M., and Bengio, S. (2016). Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940*.

[14] Borcea, C., Ding, X., Gehani, N., Curtmola, R., Khan, M. A., and Debnath, H. (2015). Avatar: Mobile distributed computing in the cloud. In *2015 3rd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering*, pages 151–156. IEEE.

[15] Botvinick, M., Ritter, S., Wang, J. X., Kurth-Nelson, Z., Blundell, C., and Hassabis, D. (2019). Reinforcement learning, fast and slow. *Trends in cognitive sciences*.

[16] Bracciale, L., Bonola, M., Loreti, P., Bianchi, G., Amici, R., and Rabuffi, A. (2014). Crawdad dataset roma/taxi (v. 2014-07-17). Downloaded from https://crawdad.org/roma/taxi/20140717.

[17] Bruschi, R., Davoli, F., Lago, P., and Pajo, J. F. (2018). Move with me: Scalably keeping virtual objects close to users on the move. In *2018 IEEE International Conference on Communications (ICC)*, pages 1–6. IEEE.

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

[19] Cappart, Q., Chételat, D., Khalil, E., Lodi, A., Morris, C., and Veličković, P. (2021). Combinatorial optimization and reasoning with graph neural networks. *arXiv preprint arXiv:2102.09544*.

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

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

[22] Chen, X., Zhang, H., Wu, C., Mao, S., Ji, Y., and Bennis, M. (2018). Optimized computation offloading performance in virtual edge computing systems via deep reinforcement learning. *IEEE Internet of Things Journal*, 6(3):4005–4018.

[23] Chen, X., Zhang, H., Wu, C., Mao, S., Ji, Y., and Bennis, M. (2019). Optimized computation offloading performance in virtual edge computing systems via deep reinforcement learning. *IEEE Internet of Things Journal*, 6(3):4005–4018.

[24] Chinchali, S., Hu, P., Chu, T., Sharma, M., Bansal, M., Misra, R., Pavone, M., and Katti, S. (2018). Cellular network traffic scheduling with deep reinforcement learning. In *Association for the Advancement of Artificial Intelligence (AAAI)*.

[25] Dai, H., Khalil, E. B., Zhang, Y., Dilkina, B., and Song, L. (2017). Learning combinatorial optimization algorithms over graphs. In *Advances in Neural Information Processing Systems, NIPS*, pages 6348–6358.

[26] De Maio, V. and Brandic, I. (2018). First hop mobile offloading of dag computations. In *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 83–92. IEEE.

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

[28] Dinh, T. Q., Tang, J., La, Q. D., and Quek, T. Q. (2017). Offloading in mobile edge computing: Task allocation and computational frequency scaling. *IEEE Transactions on Communications*, 65(8):3571–3584.

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

[30] Duan, Y., Schulman, J., Chen, X., Bartlett, P. L., Sutskever, I., and Abbeel, P. (2016). Rl^2: Fast reinforcement learning via slow reinforcement learning. *arXiv preprint arXiv:1611.02779*.

[31] Finn, C., Abbeel, P., and Levine, S. (2017). Model-agnostic meta-learning for fast adaptation of deep networks. In *Proceedings of ICML*, pages 1126–1135.

[32] Frangoudis, P. A. and Ksentini, A. (2018). Service migration versus service replication in multi-access edge computing. In *2018 14th International Wireless Communications & Mobile Computing Conference (IWCMC)*, pages 124–129. IEEE.

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

[34] Garcıa, J. and Fernández, F. (2015). A comprehensive survey on safe reinforcement learning. *Journal of Machine Learning Research*, 16(1):1437–1480.

[35] Gu, S., Holly, E., Lillicrap, T., and Levine, S. (2017). Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 3389–3396. IEEE.

[36] Guo, H., Liu, J., and Zhang, J. (2018). Computation offloading for multi-access mobile edge computing in ultra-dense networks. *IEEE Communications Magazine*, 56(8):14–19.

[37] Haarnoja, T., Zhou, A., Abbeel, P., and Levine, S. (2018). Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International Conference on Machine Learning*, pages 1861–1870. PMLR.

[38] Han, B., Gopalakrishnan, V., Ji, L., and Lee, S. (2015). Network function virtualization: Challenges and opportunities for innovations. *IEEE Communications Magazine*, 53(2):90–97.

[39] Hastings, W. K. (1970). Monte carlo sampling methods using markov chains and their applications.

[40] Hausknecht, M. and Stone, P. (2015). Deep recurrent q-learning for partially observable mdps. In *Association for the Advancement of Artificial Intelligence (AAAI)*.

[41] Hekmati, A., Teymoori, P., Todd, T. D., Zhao, D., and Karakostas, G. (2019). Optimal mobile computation offloading with hard deadline constraints. *IEEE Transactions on Mobile Computing*, 19(9):2160–2173.

[42] Hong, S.-T. and Kim, H. (2019). Qoe-aware computation offloading to capture energy-latency-pricing tradeoff in mobile clouds. *IEEE Transactions on Mobile Computing*, 18(9):2174–2189.

[43] Hospedales, T., Antoniou, A., Micaelli, P., and Storkey, A. (2020). Meta-learning in neural networks: A survey. *arXiv preprint arXiv:2004.05439*.

[44] Huang, L., Bi, S., and Zhang, Y. J. (2019a). Deep reinforcement learning for online computation offloading in wireless powered mobile-edge computing networks. *IEEE Transactions on Mobile Computing*.

[45] Huang, L., Feng, X., Zhang, C., Qian, L., and Wu, Y. (2019b). Deep reinforcement learning-based joint task offloading and bandwidth allocation for multi-user mobile edge computing. *Digital Communications and Networks*, 5(1):10–17.

[46] Igl, M., Zintgraf, L., Le, T. A., Wood, F., and Whiteson, S. (2018). Deep variational reinforcement learning for pomdps. In *International Conference on Machine Learning (ICML)*, pages 2117–2126.

[47] Janner, M., Fu, J., Zhang, M., and Levine, S. (2019). When to trust your model: Model-based policy optimization. In *Advances in Neural Information Processing Systems, NIPS*, volume 32.

[48] Kaelbling, L. P., Littman, M. L., and Moore, A. W. (1996). Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285.

[49] Kairouz, P., McMahan, H. B., Avent, B., Bellet, A., Bennis, M., Bhagoji, A. N., Bonawitz, K., Charles, Z., Cormode, G., Cummings, R., et al. (2019). Advances and open problems in federated learning. *arXiv preprint arXiv:1912.04977*.

[50] Kekki, S., Featherstone, W., Fang, Y., Kuure, P., Li, A., Ranjan, A., Purkayastha, D., Jiangping, F., Frydman, D., Verin, G., et al. (2018). Mec in 5g networks. *ETSI white paper*, 28:1–28.

[51] Khan, A. H., Qadeer, M. A., Ansari, J. A., and Waheed, S. (2009). 4g as a next generation wireless network. In *2009 International conference on future computer and communication*, pages 334–338. IEEE.

[52] Kidambi, R., Rajeswaran, A., Netrapalli, P., and Joachims, T. (2020). Morel: Model-based offline reinforcement learning. *arXiv preprint arXiv:2005.05951*.

[53] Kingma, D. P. and Ba, J. (2015). Adam: A method for stochastic optimization. In *International Conference on Learning Representations (ICLR)*.

[54] Kirkpatrick, K. (2013). Software-defined networking. *Communications of the ACM*, 56(9):16–19.

[55] Kool, W. and Welling, M. (2019). Attention, learn to solve routing problems! In *International Conference of Learning Representations, ICLR*.

[56] Kumar, A., Zhou, A., Tucker, G., and Levine, S. (2020). Conservative q-learning for offline reinforcement learning. *arXiv preprint arXiv:2006.04779*.

[57] Kwak, J., Kim, Y., Lee, J., and Chong, S. (2015). Dream: Dynamic resource and task allocation for energy minimization in mobile cloud systems. *IEEE Journal on Selected Areas in Communications*, 33(12):2510–2523.

[58] Kwok, Y.-K. and Ahmad, I. (1999). Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys (CSUR)*, 31(4):406–471.

[59] LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *nature*, 521(7553):436–444.

[60] Levine, S., Kumar, A., Tucker, G., and Fu, J. (2020). Offline reinforcement learning: Tutorial, review, and perspectives on open problems. *arXiv preprint arXiv:2005.01643*.

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

[62] Li, J., Monroe, W., Ritter, A., Galley, M., Gao, J., and Jurafsky, D. (2016). Deep reinforcement learning for dialogue generation. *arXiv preprint arXiv:1606.01541*.

[63] Li, Y. (2017). Deep reinforcement learning: An overview. *arXiv preprint arXiv:1701.07274*.

[64] Li, Z., Jiang, X., Shang, L., and Li, H. (2017). Paraphrase generation with deep reinforcement learning. *arXiv preprint arXiv:1711.00279*.

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

[66] Lin, R., Zhou, Z., Luo, S., Xiao, Y., Wang, X., Wang, S., and Zukerman, M. (2020). Distributed optimization for computation offloading in edge computing. *IEEE Transactions on Wireless Communications*, 19(12):8179–8194.

[67] Lin, X., Wang, Y., Xie, Q., and Pedram, M. (2015). Task scheduling with dynamic voltage and frequency scaling for energy minimization in the mobile cloud computing environment. *IEEE Transactions on Services Computing*, 8(2):175–186.

[68] Liu, C., Tang, F., Li, K., Tang, Z., and Li, K. (2021). Distributed task migration optimization in mec by extending multi-agent deep reinforcement learning approach. *IEEE Transactions on Parallel and Distributed Systems*, 32(7):1603–1614.

[69] Liu, M. and Liu, Y. (2017). Price-based distributed offloading for mobile-edge computing with computation capacity constraints. *IEEE Wireless Communications Letters*, 7(3):420–423.

[70] Liu, S., Liu, L., Tang, J., Yu, B., Wang, Y., and Shi, W. (2019). Edge computing for autonomous driving: Opportunities and challenges. *Proceedings of the IEEE*, 107(8):1697–1716.

[71] Luo, Y., Xu, H., Li, Y., Tian, Y., Darrell, T., and Ma, T. (2018). Algorithmic framework for model-based deep reinforcement learning with theoretical guarantees. *arXiv preprint arXiv:1807.03858*.

[72] Luong, N. C., Hoang, D. T., Gong, S., Niyato, D., Wang, P., Liang, Y.-C., and Kim, D. I. (2019). Applications of deep reinforcement learning in communications and networking: A survey. *IEEE Communications Surveys & Tutorials*, 21(4):3133–3174.

[73] Ma, L., Yi, S., Carter, N., and Li, Q. (2019). Efficient live migration of edge services leveraging container layered storage. *IEEE Transactions on Mobile Computing*, 18(9):2020–2033.

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

[75] Mahmoodi, S. E., Uma, R., and Subbalakshmi, K. (2019). Optimal joint scheduling and cloud offloading for mobile applications. *IEEE Transactions on Cloud Computing*, 7(2):301–313.

[76] Mao, H. (2020). *Network system optimization with reinforcement learning: methods and applications*. PhD thesis, Massachusetts Institute of Technology.

[77] Mao, Y., You, C., Zhang, J., Huang, K., and Letaief, K. B. (2017). A survey on mobile edge computing: The communication perspective. *IEEE Communications Surveys & Tutorials*, 19(4):2322–2358.

[78] Mazyavkina, N., Sviridov, S., Ivanov, S., and Burnaev, E. (2021). Reinforcement learning for combinatorial optimization: A survey. *Computers & Operations Research*, page 105400.

[79] McMahan, B., Moore, E., Ramage, D., and y Arcas, B. A. (2017). Communication-efficient learning of deep networks from decentralized data. *20th International Conference on Artifical Intelligence and Statistics (AISTATS)*.

[80] Melendez, S. and McGarry, M. P. (2017). Computation offloading decisions for reducing completion time. In *2017 14th IEEE Annual Consumer Communications & Networking Conference (CCNC)*, pages 160–164. IEEE.

[81] Miryoosefi, S., Brantley, K., Daumé III, H., Dudík, M., and Schapire, R. (2019). Reinforcement learning with convex constraints. In *Advances in Neural Information Processing Systems, NIPS*.

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

[83] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. (2015). Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533.

[84] Moerland, T. M., Broekens, J., and Jonker, C. M. (2020). Model-based reinforcement learning: A survey. *arXiv preprint arXiv:2006.16712*.

[85] Nachum, O. and Dai, B. (2020). Reinforcement learning via fenchel-rockafellar duality.

[86] Nadiger, C., Kumar, A., and Abdelhak, S. (2019). Federated reinforcement learning for fast personalization. In *2019 IEEE Second International Conference on Artificial Intelligence and Knowledge Engineering (AIKE)*, pages 123–127. IEEE.

[87] Narayanan, A., Ramadan, E., Carpenter, J., Liu, Q., Liu, Y., Qian, F., and Zhang, Z.-L. (2020). A first look at commercial 5g performance on smartphones. In *Proceedings of The Web Conference 2020*, pages 894–905.

[88] Neto, J. L. D., Yu, S.-Y., Macedo, D. F., Nogueira, J. M. S., Langar, R., and Secci, S. (2018). Uloof: a user level online offloading framework for mobile edge computing. *IEEE Transactions on Mobile Computing*, 17(11):2660–2674.

[89] Nguyen, D. C., Pathirana, P. N., Ding, M., and Seneviratne, A. (2020). Privacy-preserved task offloading in mobile blockchain with deep reinforcement learning. *IEEE Transactions on Network and Service Management*, 17(4):2536–2549.

[90] Nichol, A., Achiam, J., and Schulman, J. (2018). On first-order meta-learning algorithms. *arXiv preprint arXiv:1803.02999*.

[91] Ning, Z., Dong, P., Wang, X., Wang, S., Hu, X., Guo, S., Qiu, T., Hu, B., and Kwok, R. (2021). Distributed and dynamic service placement in pervasive edge computing networks. *IEEE Transactions on Parallel and Distributed Systems*, 32(6):1277–1292.

[92] Othman, M., Madani, S. A., Khan, S. U., et al. (2013). A survey of mobile cloud computing application models. *IEEE communications surveys & tutorials*, 16(1):393–413.

[93] Ouyang, T., Li, R., Chen, X., Zhou, Z., and Tang, X. (2019). Adaptive user-managed service placement for mobile edge computing: An online learning approach. In *IEEE International Conference on Computer Communications (INFOCOM)*, pages 1468–1476. IEEE.

[94] Ouyang, T., Zhou, Z., and Chen, X. (2018). Follow me at the edge: Mobility-aware dynamic service placement for mobile edge computing. *IEEE Journal on Selected Areas in Communications*, 36(10):2333–2345.

[95] Pecka, M. and Svoboda, T. (2014). Safe exploration techniques for reinforcement learning–an overview. In *International Workshop on Modelling and Simulation for Autonomous Systems*, pages 357–375. Springer.

[96] Pinto, L., Davidson, J., Sukthankar, R., and Gupta, A. (2017). Robust adversarial reinforcement learning. In *International Conference on Machine Learning*, pages 2817–2826. PMLR.

[97] Piorkowski, M., Sarafijanovic-Djukic, N., and Grossglauser, M. (2009). Crawdad data set epfl/mobility (v. 2009-02-24).

[98] Prates, M., Avelar, P. H., Lemos, H., Lamb, L. C., and Vardi, M. Y. (2019). Learning to solve np-complete problems: A graph neural network for decision tsp. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 4731–4738.

[99] Qingyang Hu, Y. Q. (2013). *Heterogeneous Cellular Networks*. Wiley.

[100] Ra, M.-R., Sheth, A., Mummert, L., Pillai, P., Wetherall, D., and Govindan, R. (2011). Odessa: enabling interactive perception applications on mobile devices. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, pages 43–56. ACM.

[101] Rakelly, K., Zhou, A., Quillen, D., Finn, C., and Levine, S. (2019). Efficient off-policy meta-reinforcement learning via probabilistic context variables. *arXiv preprint arXiv:1903.08254*.

[102] Rejiba, Z., Masip-Bruin, X., and Marín-Tordera, E. (2019). A survey on mobility-induced service migration in the fog, edge, and related computing paradigms. *ACM Computing Surveys (CSUR)*, 52(5):1–33.

[103] Rothfuss, J., Lee, D., Clavera, I., Asfour, T., and Abbeel, P. (2019). Promp: Proximal meta-policy search. In *Proceedings of ICLR*.

[104] Sabell, D., Sukhomlinov, V., Trang, L., Kekki, S., Paglierani, P., Rossbach, R., Li, X., Fang, Y., Druta, D., Giust, F., et al. (2019). Developing software for multi-access edge computing. *ETSI White Paper*, 20.

[105] Saeik, F., Avgeris, M., Spatharakis, D., Santi, N., Dechouniotis, D., Violos, J., Leivadeas, A., Athanasopoulos, N., Mitton, N., and Papavassiliou, S. (2021). Task offloading in edge and cloud computing: A survey on mathematical, artificial intelligence and control theory solutions. *Computer Networks*, page 108177.

[106] Satyanarayanan, M., Bahl, P., Caceres, R., and Davies, N. (2009). The case for vm-based cloudlets in mobile computing. *IEEE pervasive Computing*, 8(4):14–23.

[107] Schulman, J., Moritz, P., Levine, S., Jordan, M., and Abbeel, P. (2016). High-dimensional continuous control using generalized advantage estimation. In *International Conference of Learning Representations, ICLR*.

[108] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

[109] Seshia, S. A., Sadigh, D., and Sastry, S. S. (2016). Towards verified artificial intelligence. *arXiv preprint arXiv:1606.08514*.

[110] Shojaee, R. and Yazdani, N. (2020). Modeling and performance analysis of smart map application in the multi-access edge computing paradigm. *Pervasive and Mobile Computing*, 69:101280.

[111] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., et al. (2018). A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144.

[112] Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., et al. (2017). Mastering the game of go without human knowledge. *nature*, 550(7676):354–359.

[113] Sun, X. and Ansari, N. (2016). Primal: Profit maximization avatar placement for mobile edge computing. In *2016 IEEE International Conference on Communications (ICC)*, pages 1–6. IEEE.

[114] Sun, X. and Ansari, N. (2019). Adaptive avatar handoff in the cloudlet network. *IEEE Transactions on Cloud Computing*, 7(3):664–676.

[115] Sun, Y., Guo, X., Zhou, S., Jiang, Z., Liu, X., and Niu, Z. (2018). Learning-based task offloading for vehicular cloud computing systems. In *IEEE International Conference on Communications (ICC)*, pages 1–7. IEEE.

[116] Sun, Y., Zhou, S., and Xu, J. (2017). Emm: Energy-aware mobility management for mobile edge computing in ultra dense networks. *IEEE Journal on Selected Areas in Communications*, 35(11):2637–2646.

[117] Sutton, R. S. (1990). Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Machine learning proceedings 1990*, pages 216–224. Elsevier.

[118] Sutton, R. S. and Barto, A. G. (2018). *Reinforcement learning: An introduction*. MIT press.

[119] Taleb, T., Ksentini, A., and Frangoudis, P. A. (2016). Follow-me cloud: When cloud services follow mobile users. *IEEE Transactions on Cloud Computing*, 7(2):369–382.

[120] Tan, L. T. and Hu, R. Q. (2018). Mobility-aware edge caching and computing in vehicle networks: A deep reinforcement learning. *IEEE Transactions on Vehicular Technology*, 67(11):10190–10203.

[121] Tang, M. and Wong, V. W. (2020). Deep reinforcement learning for task offloading in mobile edge computing systems. *IEEE Transactions on Mobile Computing*.

[122] Tesauro, G. (1995). Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68.

[123] Teymoori, P., Todd, T. D., Zhao, D., and Karakostas, G. (2020). Efficient mobile computation offloading with hard task deadlines and concurrent local execution. In *IEEE Global Communications Conference, GLOBECOM*, pages 1–6. IEEE.

[124] Topcuoglu, H., Hariri, S., and Wu, M. (2002). Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274.

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

[126] Van Hasselt, H., Guez, A., and Silver, D. (2016). Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30.

[127] Vanschoren, J. (2018). Meta-learning: A survey. *arXiv preprint arXiv:1810.03548*.

[128] Vinitsky, E., Du, Y., Parvate, K., Jang, K., Abbeel, P., and Bayen, A. (2020). Robust reinforcement learning using adversarial populations. *arXiv preprint arXiv:2008.01825*.

[129] Vinyals, O., Fortunato, M., and Jaitly, N. (2015). Pointer networks. In *Advances in Neural Information Processing Systems, NIPS*, pages 2692–2700.

[130] Wang, C., Liang, C., Yu, F. R., Chen, Q., and Tang, L. (2017a). Computation offloading and resource allocation in wireless cellular networks with mobile edge computing. *IEEE Transactions on Wireless Communications*, 16(8):4924–4938.

[131] Wang, J., Hu, J., Min, G., Zhan, W., Ni, Q., and Georgalas, N. (2019a). Computation offloading in multi-access edge computing using a deep sequential model based on reinforcement learning. *IEEE Communications Magazine*, 57(5):64–69.

[132] Wang, J., Hu, J., Min, G., Zomaya, A. Y., and Georgalas, N. (2020a). Fast adaptive task offloading in edge computing based on meta reinforcement learning. *IEEE Transactions on Parallel and Distributed Systems*, 32(1):242–253.

[133] Wang, J., Kurth-Nelson, Z., Tirumala, D., Soyer, H., Leibo, J., Munos, R., Blundell, C., Kumaran, D., and Botvinick, M. (2017b). Learning to reinforcement learn. *arXiv preprint arXiv:1611.05763*.

[134] Wang, J., Liu, T., Liu, K., Kim, B., Xie, J., and Han, Z. (2018). Computation offloading over fog and cloud using multi-dimensional multiple knapsack problem. In *2018 IEEE Global Communications Conference (GLOBECOM)*, pages 1–7. IEEE.

[135] Wang, S., Guo, Y., Zhang, N., Yang, P., Zhou, A., and Shen, X. S. (2021a). Delay-aware microservice coordination in mobile edge computing: A reinforcement learning approach. *IEEE Transactions on Mobile Computing*, 20(3):939–951.

[136] Wang, S., Urgaonkar, R., He, T., Chan, K., Zafer, M., and Leung, K. K. (2016). Dynamic service placement for mobile micro-clouds with predicted future costs. *IEEE Transactions on Parallel and Distributed Systems*, 28(4):1002–1016.

[137] Wang, S., Urgaonkar, R., Zafer, M., He, T., Chan, K., and Leung, K. K. (2019b). Dynamic service migration in mobile edge computing based on markov decision process. *IEEE/ACM Transactions on Networking*, 27(3):1272–1288.

[138] Wang, S., Zafer, M., and Leung, K. K. (2017c). Online placement of multi-component applications in edge computing environments. *IEEE Access*, 5:2514–2533.

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

[140] Wang, X., Li, R., Wang, C., Li, X., Taleb, T., and Leung, V. C. M. (2021b). Attention-Weighted Federated Deep Reinforcement Learning for Device-to-Device Assisted Heterogeneous Collaborative Edge Caching. *IEEE Journal on Selected Areas in Communications*, 39(1):154–169.

[141] Wang, X., Wang, C., Li, X., Leung, V. C. M., and Taleb, T. (2020c). Federated Deep Reinforcement Learning for Internet of Things With Decentralized Cooperative Edge Caching. *IEEE Internet of Things Journal*, 7(10):9441–9455.

[142] Watter, M., Springenberg, J., Boedecker, J., and Riedmiller, M. (2015). Embed to control: A locally linear latent dynamics model for control from raw images. In *NIPS*, pages 2746–2754.

[143] Weaver, L. and Tao, N. (2001). The optimal reward baseline for gradient-based reinforcement learning. In *Proceedings of the Seventeenth Conference on Uncertainty in Artificial Intelligence*, page 538–545. Morgan Kaufmann Publishers Inc.

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

[145] Williams, R. J. and Peng, J. (1991). Function optimization using connectionist reinforcement learning algorithms. *Connection Science*, 3(3):241–268.

[146] Wing, J. M. (1990). A specifier's introduction to formal methods. *Computer*, 23(9):8–22.

[147] Wu, C.-L., Chiu, T.-C., Wang, C.-Y., and Pang, A.-C. (2020). Mobility-aware deep reinforcement learning with glimpse mobility prediction in edge computing. In *IEEE International Conference on Communications (ICC)*, pages 1–7. IEEE.

[148] Wu, H., Knottenbelt, W., and Wolter, K. (2019). An efficient application partitioning algorithm in mobile environments. *IEEE Transactions on Parallel and Distributed Systems*, 30(7):1464–1480.

[149] Xu, J., Ma, X., Zhou, A., Duan, Q., and Wang, S. (2020). Path selection for seamless service migration in vehicular edge computing. *IEEE Internet of Things Journal*, 7(9):9040–9049.

[150] Yang, L., Zhong, C., Yang, Q., Zou, W., and Fathalla, A. (2020). Task offloading for directed acyclic graph applications based on edge computing in industrial internet. *Information Sciences*, 540:51–68.

[151] Yao, H., Bai, C., Zeng, D., Liang, Q., and Fan, Y. (2015). Migrate or not? exploring virtual machine migration in roadside cloudlet-based vehicular cloud. *Concurrency and Computation: Practice and Experience*, 27(18):5780–5792.

[152] Ye, D., Liu, Z., Sun, M., Shi, B., Zhao, P., Wu, H., Yu, H., Yang, S., Wu, X., Guo, Q., et al. (2020). Mastering complex control in moba games with deep reinforcement learning. In *Association for the Advancement of Artificial Intelligence (AAAI)*, volume 34, pages 6672–6679.

[153] You, X., Wang, C.-X., Huang, J., Gao, X., Zhang, Z., Wang, M., Huang, Y., Zhang, C., Jiang, Y., Wang, J., et al. (2021). Towards 6g wireless communication networks: Vision, enabling technologies, and new paradigm shifts. *Science China Information Sciences*, 64(1):1–74.

[154] Yu, Z., Hu, J., Min, G., Zhao, Z., Miao, W., and Hossain, M. S. (2020). Mobility-aware proactive edge caching for connected vehicles using federated learning. *IEEE Transactions on Intelligent Transportation Systems*.

[155] Yuan, Q., Li, J., Zhou, H., Lin, T., Luo, G., and Shen, X. (2020). A joint service migration and mobility optimization approach for vehicular edge computing. *IEEE Transactions on Vehicular Technology*, 69(8):9041–9052.

[156] Zanni, A., Yu, S.-Y., Bellavista, P., Langar, R., and Secci, S. (2017). Automated selection of offloadable tasks for mobile computation offloading in edge computing. In *2017 13th international conference on network and service management (CNSM)*, pages 1–5. IEEE.

[157] Zeng, D., Gu, L., Guo, S., Cheng, Z., and Yu, S. (2016). Joint optimization of task scheduling and image placement in fog computing supported software-defined embedded system. *IEEE Transactions on Computers*, 65(12):3702–3712.

[158] Zhan, W., Luo, C., Min, G., Wang, C., Zhu, Q., and Duan, H. (2020a). Mobility-aware multi-user offloading optimization for mobile edge computing. *IEEE Transactions on Vehicular Technology*, 69(3):3341–3356.

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

[160] Zhang, C., Patras, P., and Haddadi, H. (2019a). Deep learning in mobile and wireless networking: A survey. *IEEE Communications surveys & tutorials*, 21(3):2224–2287.

[161] Zhang, G., Zhang, S., Zhang, W., Shen, Z., and Wang, L. (2021). Joint service caching, computation offloading and resource allocation in mobile edge computing systems. *IEEE Transactions on Wireless Communications*.

[162] Zhang, K., Yang, Z., and Başar, T. (2019b). Multi-agent reinforcement learning: A selective overview of theories and algorithms. *arXiv preprint arXiv:1911.10635*.

[163] Zhang, M., Vikram, S., Smith, L., Abbeel, P., Johnson, M., and Levine, S. (2019c). Solar: Deep structured representations for model-based reinforcement learning. In *International Conference on Machine Learning (ICML)*, pages 7444–7453.

[164] Zhang, Y., Clavera, I., Tsai, B., and Abbeel, P. (2019d). Asynchronous methods for model-based reinforcement learning. In *Conference on Robot Learning (CoRL)*.

[165] Zhou, J., Cui, G., Hu, S., Zhang, Z., Yang, C., Liu, Z., Wang, L., Li, C., and Sun, M. (2020). Graph neural networks: A review of methods and applications. *AI Open*, 1:57–81.

[166] Zhu, P., Li, X., Poupart, P., and Miao, G. (2018). On improving deep reinforcement learning for pomdps. *arXiv preprint arXiv:1704.07978*.

[167] Zhuo, H. H., Feng, W., Xu, Q., Yang, Q., and Lin, Y. (2019). Federated reinforcement learning. *arXiv preprint arXiv:1901.08277*.

[168] Zou, J., Hao, T., Yu, C., and Jin, H. (2021). A3c-do: A regional resource scheduling framework based on deep reinforcement learning in edge scenario. *IEEE Transactions on Computers*, 70(2):228–239.