UNIVERSITY OF SOUTHAMPTON

Faculty of Engineering and Physical Science
School of Electronics and Computer Science

Deep Reinforcement Learning for Online Combinatorial Resource Allocation with Arbitrary State and Action Spaces

by

Tesfay Zemuy Gebrekidan

ORCiD: 0000-0002-0182-0997

A thesis for the degree of
Doctor of Philosophy

June 2024

University of Southampton

Abstract

Faculty of Engineering and Physical Science
School of Electronics and Computer Science

Doctor of Philosophy

Deep Reinforcement Learning for Online Combinatorial Resource Allocation with Arbitrary State and Action Spaces

by Tesfay Zemuy Gebrekidan

Online combinatorial resource allocation is the process of dynamically assigning limited resources to tasks that arrive arbitrarily. The allocation of resources is done without complete knowledge of future resource demands. Conventional resource allocation algorithms, such as mathematical optimization, are inefficient for online resource allocation problems because relevant information about the problem is not available in advance; they cannot predict and adapt to the dynamic changes of the problem; they have high online computational costs, making them impractical for real-time decisions; and they are inefficient for non-convex optimization problems.

Real-time optimization using artificial intelligence (AI) and machine learning (ML) algorithms is state-of-the-art in online resource allocation. The use of AI/ML is one of the key components in the evolution of 5G to 6G. Deep reinforcement learning (DRL) is a subfield of ML that integrates reinforcement learning (RL) and deep learning (DL), both of which are components of AI. Due to its ability to make sequential online decisions in dynamic and uncertain contexts, as well as its ability to learn from experience and lower online computational costs, DRL is a commonly used solution for online resource allocation problems.

Compared to other application areas, DRL encounters unique challenges in online combinatorial resource allocation problems. The resource allocation problem often involves elements that are of varying sizes and have no specific order, henceforth referred to as arbitrarily sized and orderless (ASO) elements. In mobile and cloud computing, for example, the problem can include tasks with varying numbers and sizes, as well as varying numbers of user devices (UDs). However, existing DRL algorithms use standard deep neural network (DNN) algorithms as function approximators. The neurons in DNNs are set to accept specific information at the specific index of a given input, whereas the order of the tasks in the input of combinatorial resource allocation problems does not matter. The DNN cannot generalize using the knowledge it learned with a different permutation of the same input. Furthermore, the number of UDs can vary,

but the number of neurons in the input of standard DNNs is fixed in size. Additionally, existing DRL algorithms make decisions by selecting a single action sequentially or a fixed number of actions at a time for the ASO input. However, online combinatorial resource allocation needs to select an arbitrary number of actions based on the resource constraint. This sequential action selection leads to increased dimensionality, suboptimal convergence in training, and greater computational complexity. Arbitrary action space, in particular, is understudied in DRL. Furthermore, existing DRL algorithms in resource allocation problems consider homogeneous constraints on either the UDs or the server side, while resource allocation problems usually include various resource constraints. There are continuous-valued resource constraints on UDs, discrete-valued number of channels on the communication network, and combinatorial competition of UDs on the server due to storage constraints. These challenges can generally be summarized as handling an arbitrary state space for an ASO input, an arbitrary action space for an ASO output, and various constraints on the UDs and the server.

The objective of this research is to advance DRL algorithms for online combinatorial resource allocation problems so that they can effectively handle arbitrary state and action spaces for the ASO inputs and outputs, as well as to consider heterogeneous resource constraints on the UDs and the server. Consequently, we propose three solutions as follows. 1) A novel DRL algorithm with coalition action selection for online combinatorial resource allocation. The coalition action selection enables DRL to simultaneously select an arbitrary number of actions without updating the state multiple times. By reducing state space and depth of decision, coalition action selection provides better performance, faster convergence, and lower execution complexity compared to conventional sequential action selection approaches, where the state is updated for every action taken. 2) We proposed a novel DRL algorithm with computationally efficient stationary ASO input transformation for online combinatorial resource allocation problems. By using a set of equations to transform the ASO input to a fixed-size vector, the stationary ASO input transformation provides better convergence and lower computational cost than a transformer neural network-based transformation, which is used as the state-of-the-art technique to handle ASO inputs in existing combinatorial optimization problems. The reason for the efficiency is that the transformer is designed to learn contextual relationships between the sequence of words in natural language processing (NLP), but the ASO inputs in the resource allocation are numerical and do not have as significant a relationship as the sequence of words. 3) By applying coalition action selection to the multiagent deep deterministic policy gradient (MADDPG), we propose a combinatorial client-master multiagent DRL (CCM_MADRL) algorithm for task offloading in mobile edge computing (CCM_MADRL_MEC) to handle various resource constraints.

The efficiencies of the proposed solutions, compared to state-of-the-art approaches, are assessed using online resource allocation problems with arbitrary arrival of tasks and

various resource constraints. The DRL algorithm with coalition action selection is evaluated using an online resource allocation problem with an arbitrary number of tasks. It has outperformed existing sequential action selection approaches in terms of proximity to offline optimal solutions, speed of convergence, and computational costs. The coalition action selection retains close to offline optimal performance in settings with different task arrival rates, whereas the sequential action selection approach drops in performance when the task arrival rate is high. The DRL with coalition action selection is implemented using the encoder of the transformer neural network. We evaluated the stationary ASO input transformation on the same problem. It has outperformed the transformer-based transformation in various sizes of task arrival rates. Furthermore, the stationary ASO input transformation yields a lower computational complexity than the transformer for various task arrival rates. The difference in computational cost between the transformer and the stationary ASO input transformation is greater in the sequential action selection than in the coalition action selection. Lastly, the CCM_MADRL algorithm is evaluated using a task-offloading problem in mobile edge computing with different constraints, such as battery level, task deadline, transmission power, computational resource, server storage capacity, and number of communication channels. By exploiting the different advantages of the policy iteration and value function and the coalition action selection approach, it has demonstrated better convergence than the existing MADDPG and heuristic algorithms.

Contents

List of Figures

List of Tables

Declaration of Authorship

I declare that this thesis and the work presented in it is my own and has been generated by me as the result of my own original research.

I confirm that:

1. This work was done wholly or mainly while in candidature for a research degree at this University;
2. Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
3. Where I have consulted the published work of others, this is always clearly attributed;
4. Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
5. I have acknowledged all main sources of help;
6. Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
7. Parts of this work have been published as presented in Section 1.4

Signed:..... Date:.....

Acknowledgements

First, I would like to sincerely thank my supervisors, Professor Sebastian Stein and Professor Timothy Norman for their unwavering support and guidance during my study. I express my gratitude to Professor Sebastian Stein for his patience and dedication in assisting me during a period when external obstacles coincided with my Ph.D. study.

Next, I want to acknowledge the International Technology Alliance in Distributed Analytics and Information Sciences (DAIS ITA) [1] for funding my Ph.D. study.

I am very grateful for the financial, academic, and moral support of my friends throughout my student life.

[1] https://dais-legacy.org/

Acronyms and Abbreviations

5G	Fifth Generation Network
6G	Sixth Generation Network
AI	Artificial Intelligence
ML	Machine Learning
DDPG	Deep Deterministic Policy Gradient
MADDPG	Multiagent Deep Deterministic Policy Gradient
DNN	Deep Neural Network
DL	Deep Learning
DRL	Deep Reinforcement learning
DQN	Deep Q Network
MADRL	Multiagent Deep Reinforcement Learning
MARL	Multiagent Reinforcement Learning
MEC	Mobile Edge Computing
MCC	Mobile Cloud Computing
CCM_MADRL	Combinatorial Client-Master MADRL Algorithm
CCM_MADRL_MEC	CCM_MADRL-based Task Offloading for MEC
MDP	Markov Decision Process
ASO	Arbitrarily Sized and Orderless
RL	Reinforcement Learning
SDN	Software Defined Network
DDPG	Deep Deterministic Policy Gradient
MADDPG	Multiagent Deep Deterministic Policy Gradient
QoE	Quality of Experience
QoS	Quality of Service
GNN	Graph Neural Network
TD	Temporal Difference
ReLU	Rectified Linear Unit
Tanh	Hyperbolic Tangent
ADAM	Adaptive Moment Estimation
UD	User Device
MB	Megabyte

TB	Terabyte
GHz	Gigahertz
dB	Decibels
dBm	Decibels Relative to Milliwatt
HPC	High Performance Computing
NLP	Natual Language Processing
ODUK	Optical Data Units
Ph.D.	Doctor of Philosophy
IP	Integer Programming

Chapter 1

Introduction

1.1 Overview

Online combinatorial resource allocation refers to the process of dynamically allocating limited resources to tasks arriving sequentially and arbitrarily (Tan et al., 2020), where allocation decisions must be made without complete knowledge of future resource requests. Unlike offline resource allocation, where all relevant information is known in advance and decisions are made based on a fixed dataset, online resource allocation makes decisions for incoming requests. Online combinatorial resource allocation is a challenging problem that arises in various application domains, including cloud computing (Xu et al., 2022), online job scheduling (Etesami, 2021), and auctions (Tan et al., 2020). Solving online combinatorial resource allocation problems is often computationally infeasible due to their combinatorial nature and the uncertainty of future requests.

Resource allocation algorithms have evolved with increasing resource allocation challenges. In the beginning, resource allocation was controlled by rule-based or heuristic methods that depended on human judgment and predetermined algorithms (Cichoń, 1997). However, these methods struggle to adapt to changing conditions and optimize complex resource distributions. Next, resource allocation has been transformed by mathematical optimization approaches such as linear programming and integer programming, which use mathematical models to discover the best solutions considering constraints and objectives (Sundermann et al., 2021). Heuristic algorithms, such as genetic algorithms, particle swarm optimization, and simulated annealing, offered more flexible and adaptable solutions than mathematical models to complex resource allocation problems (Cichoń, 1997). In contrast to mathematical models, which are based on a set of equations that explain the problem, heuristic algorithms are problem-solving strategies that use trial and error to find a solution (Varde et al., 2008).

Traditional resource allocation algorithms, such as mathematical optimization, have four drawbacks that make them ineffective for online resource allocation problems.

First, relevant information about the problem is not available in advance. Second, they cannot predict and adapt to the dynamic changes of the problem. Third, they incur high online computational costs (Nian et al., 2020b), making them impractical for real-time decisions. Fourth, they are inefficient for non-convex problems (Wang et al., 2018a; Tefera et al., 2023). Despite the applicability to non-convex problems, heuristic algorithms also suffer the same limitations. Moreover, heuristic algorithms are approximate solutions.

Real-time solutions using AI and ML algorithms are state-of-the-art in online resource allocation (Wang et al., 2018a). The use of AI/ML is one of the key components in the evolution of 5G to 6G (Kaur et al., 2021). DRL is a subfield of ML that integrates the advantages of RL and DL as presented in Section 2.2.5.1. It has attracted a lot of attention in many fields due to its ability to make consecutive decisions in dynamic and uncertain contexts (Wu et al., 2021a; Boute et al., 2022). It is a commonly used solution in resource allocation problems. A survey on the applicability of DRL to resource allocation by Luong et al. (2019) has concluded that the use of DRL has many advantages in solving complex problems where there is incomplete network information. It allows all entities in the network to learn and make autonomous decisions, improves learning speed, and provides the ease of modeling some problems, such as security, as games. Another survey of DRL in control systems (Nian et al., 2020b) has stated that, compared to mathematical programming-based control systems, DRL has the advantage of coping with uncertainties, non-stationarity, and reducing the online computational cost by having offline training, which makes it advantageous for systems where reducing the online computational cost is of importance.

However, because online combinatorial resource allocation problems involve ASO tasks and the presence of various discrete and continuous constraints, the application of DRL to resource allocation problems faces many challenges as presented in Section 1.2. Furthermore, the current DRL algorithms are centralized and not scalable. A survey by Luong et al. (2019) shows that most of the current applications of DRL algorithms in resource allocation problems are single-agent algorithms deployed to solve a single objective function. Despite the introduction of multi-agent DRL (MADRL) algorithms to resource allocation problems, most of them are made up of homogeneous agents aimed at solving the same objective with the same resource constraints.

In this research, we focus on addressing the challenges that the DRL algorithms face in resource allocation problems. The challenges of applying DRL to online resource allocation problems and research requirements are presented in the following section.

1.2 Research Challenges and Requirements

While DRL can overcome the limitations of traditional mathematical optimization algorithms in the context of online resource allocation, the application of DRL to resource allocation faces unique challenges compared to other application areas. We categorize the challenges into three categories: arbitrary state spaces, arbitrary action spaces, and various constraints.

The first challenge is handling the ASO input. Inputs in resource allocation problems are usually ASO, which means that the elements of the input are variable in number, and their order does not matter. In other words, the input can be a set. Standard DNNs, which are used as function approximators in DRL, have two challenges with variable-size and permutation-invariant elements in their inputs and outputs. The first challenge is that a permutation-invariant input is considered as different information when fed to the neural network in different orders because it is fed to different input neurons of DNN. As a result, different permutations of a problem instance end up being learned as distinct problem instances, which decreases generalizability and costs training time and resources. The second challenge of DNN is its inability to accept variable-size information because standard DNNs are usually configured with a fixed number of inputs and outputs. Existing DL approaches that handle permutation invariance and arbitrary size and their challenges are discussed in Chapter 2.

The second challenge is in the action space. Existing DRL algorithms make sequential decisions by updating their state after each decision, whereas the combinatorial problem is orderless. Sequential action selection increases the dimensionality of the problem (Janner et al., 2021) because it increases the number of states to be explored. This leads to increased uncertainty, lower convergence, and higher computational costs that affect execution speed. The DRL algorithms are also configured to make choices from a fixed number of action spaces, whereas resource allocation problems have dynamic action spaces. Computational complexity and performance can be improved by making coordinated simultaneous decisions for ASO inputs and outputs.

Third, current MADRL algorithms in online combinatorial resource allocation problems are homogeneous agents with the same resource constraints and objectives, while the online resource allocation problem can have a mix of continuous and discrete constraints, and combinatorial decisions. For example, UDs, communication channels, and servers in task-offloading problems (Islam et al., 2021) have different constraints. There are continuous-valued resource constraints on UDs, discrete-valued number channels on the communication network, and combinatorial selection of UDs on the server due to storage constraints. However, in existing DRL algorithms, only homogeneous constraints of the UDs are considered as a penalty in the reward function. This must be

addressed using heterogeneous agents that consider the different constraints. Furthermore, most MADRL-based resource allocation algorithms deploy multiagent deep deterministic policy gradient (MADDPG) algorithms that comprise actor and critic networks. Because only actors make decisions and the critic is only used to give feedback to the actors in training, they are not efficient in terms of convergence for combinatorial decisions due to the lack of cooperation as presented in Section 2.2.8.5.

The main requirements of the DRL and MADRL algorithms for the resource allocation problem can be grouped as follows.

R_I : The curse of dimensionality in DRL-based resource allocation algorithms must be minimized to increase convergence and decrease computational complexity, by making coordinated combinatorial decisions rather than sequential decisions that update the state for each decision. Action selection must work for an arbitrary action space.

R_{II} : DRL-based resource allocation algorithms must be able to accept an arbitrary number of inputs. They must be permutation-invariant to a change in the order of the input.

R_{III} : MADRL-based resource allocation algorithms must minimize the computational cost of online execution to give a real-time solution to the sensitive time system.

R_{IV} : MADRL-based resource allocation algorithms must allow the coexistence of different decision entities, with heterogeneous MADRL agents, in the resource allocation problem to consider the different types of constraints and objective functions.

R_V : Resource allocation algorithms must be resilient and generalizable to network dynamics, including changes in the number of users, communication channels, and permutations in the input.

R_{VI} : MADRL-based resource allocation algorithms must be scalable and resilient to allow new agents to enter and leave the agent team.

1.3 Research Contributions

This section describes the main contributions of this work. The detailed presentation of the contributions is presented as chapters as outlined in Section 1.5.

C_I : A novel DRL algorithm with coalition action selection for online combinatorial resource allocation with arbitrary action space. The algorithm enables the simultaneous selection of a coalition of an arbitrary number of actions, leading to

state-space reduction, faster learning and execution, and close to optimal convergence by avoiding sequential action selection and state update. This is proposed based on the Requirement R_I, the Requirement R_{III}, and the Requirement R_V.

C_{II} : A novel computationally efficient stationary ASO input transformation function that transforms an unorderly and arbitrarily sized input space into a fixed size vector before being fed to the DRL. The transformation is also permutaiton invariant. This is proposed to meet Requirement R_{II}, Requirement R_{III}, and Requirement R_V.

C_{III} : A novel combinatorial client-master MADRL algorithm for task offloading in mobile edge computing, with battery threshold, computational resource, transmission power constraints on the UDs; a storage constraint on the edge server; number of channels as a constraint on the wireless network. Heterogeneous MADRL agents are deployed on the UDs and on the server to consider the various constraints and make different decisions. The constraints of UDs are considered in the reward functions as a penalty, and the constraints of the server and channels are considered in the combinatorial action selection. This contribution applies the coalition action selection of Contribution C_I and the per-action DQN to form a master agent to extend existing DRL algorithms with homogeneous agents to heterogeneous agents of the value function and policy gradient methods in a client-master relationship. The CCM_MADRL_MEC is proposed to address many requirements, including Requirement R_{IV}, Requirement R_I, and Requirement R_V. By including Contribution C_{II} in its model, it also serves as a foundation for Requirement R_{VI} and Requirement R_{II}.

We have evaluated the proposed methods and algorithms. Contribution C_I is evaluated using an online resource allocation problem, as used by Zhang et al. (2009). It has significantly outperformed existing sequential approaches in terms of proximity to the offline optimal, speed of convergence, and computational costs. Contribution C_{II} is applied to the problem in Contribution C_I, which used the transformer neural network to handle the ASO input. A detailed experimental evaluation is conducted showing better convergence and relative to the transformer-based counterparts on both the coalition and sequential action selection approaches. Contribution C_{III} is evaluated in task-offloading problems in mobile edge computing with various constraints on the UDs, the wireless channel, and the server. It has shown better convergence than the heuristic and benchmark algorithms.

1.4 Published Works

Some of the contributions are published at a conference as follows at the time of submission.

- Contribution C_I is published as Deep Reinforcement Learning with Coalition Action Selection for Online Combinatorial Resource Allocation with Arbitrary Action Space, in Proceedings of the 23rd International Conference on Autonomous Agents and Multiagent Systems (Gebrekidan et al., 2024a).
- Contribution C_{III} is published as Combinatorial Client-Master Multiagent Deep Reinforcement Learning for Task Offloading in Mobile Edge Computing, in Proceedings of the 23rd International Conference on Autonomous Agents and Multiagent Systems (Gebrekidan et al., 2024b).

1.5 Structure of the Thesis

The thesis is structured as follows. Chapter 2 explores the literature review of DRL-based resource allocation algorithms and their limitations, followed by existing types of DRL algorithms in general, and summarizes how they are used and modified for the proposed algorithms. Chapter 3, Chapter 4, and Chapter 5 present Contribution C_I, Contribution C_{II}, and Contribution C_{III} respectively. Finally, Chapter 6 concludes the thesis.

Chapter 2

Literature Review

Since this research aims to address the challenges that DRL algorithms face regarding online resource allocation problems with arbitrary state and action spaces, the literature review is divided into two sections. The challenges and requirements mentioned in Section 1.2 are revisited in the first section, which examines current DRL-based resource allocation strategies. The second section reviews various DRL algorithms and explains how to customize and apply them for the contributions stated in Section 1.3. The limitations of existing DRL algorithms, in general, and their customization for our proposed algorithms are summarized in Section 2.2.9.

2.1 Reinforcement Learning-based Resource Allocation Algorithms and Their Challenges

As discussed in Chapter 1, because DRL is state-of-the-art and more suitable than traditional resource allocation algorithms for online resource allocation, this section focuses only on the review of DRL-based online resource allocation algorithms.

The fact that MARL algorithms can contain autonomous agents that can be readily added and removed allows them to meet Requirement R_{VI}'s scalability and the heterogenous decision entities to meet different constraints and objective functions in Requirement R_{IV}. For example, Sun et al. (2019) proposed a dynamic resource reservation and DRL-based autonomous virtual network resource slicing framework, where the resource provider dynamically collects its free and unused resources from virtual networks and reserves them in proportion to their minimum requirements, and then the virtual networks use autonomous DRL agents to adjust their resource allocation to satisfy their objective functions independently. Virtual networks and the DRL algorithms that decide their autonomous resource allocation can be freely added and removed.

The related work on the other requirements is presented in the following sections.

2.1.1 Deep Reinforcement Learning for Resource Allocation with Arbitrary Action Space

In regard to Requirement R_I, we review DRL algorithms with combinatorial action selection approaches concerning the DRL algorithm with coalition action selection for the online combinatorial resource allocation problem in Chapter 3. Note that the coalition action selection is also applied by the master agent of CCM_MADRL_MEC in Chapter 5.

A combinatorial action selection approach, inspired by the per-action-DQN (He et al., 2015), called deep reinforcement relevance network (DRRN), is proposed by He et al. (2016b) for recommendation systems. However, the action space is intended to represent a fixed number of combinations of the items to be recommended. The combinations of actions are then used as single actions in the per-action-DQN. This grouping of actions into combinations makes it not applicable to resource allocation, where arbitrary actions can be selected with the consideration of the resource constraint. Moreover, computational complexity is still high, as the action space is exponential with the number of tasks. They have also pointed out that computational complexity is a challenge they have not focused on in their work.

Similarly, although there are DRL algorithms with combinatorial action selection approaches in resource allocation (Huang et al., 2018, 2019b) and capacitated vehicle routing (Delarue et al., 2020) problems, they have a fixed number of outputs. A detailed review of their other limitations is discussed in Chapter 3.

In summary, combinatorial action space is not studied in DRL. Most existing approaches overlook it by simplifying their problem with the assumption that only one task arrives at a time (Almasan et al., 2022; Stein et al., 2020; Chen et al., 2018) or by applying MARL algorithms that are short of requirements R_{IV} as discussed in the next section.

2.1.2 Deep Reinforcement Learning for Resource Allocation with Various Constraints

Despite the capability to be customized so that they address the research requirements, existing DRL-based resource allocation algorithms do not address the Requirement R_{IV}'s coexistence of different decision entities, with heterogeneous MADRL agents, in the resource allocation problem to consider the different types of constraints and objective functions. A survey by Luong et al. (2019) shows that most of the existing DRL-based resource allocation algorithms optimize a single objective function either on the user or on the edge server side. The survey shows that single-agent DRL is widely applied to resource allocation problems, such as channel access, resource scheduling, and task offloading. Another recent survey by Islam et al. (2021), which has presented a

comprehensive survey on task-offloading algorithms by categorizing them from multiple perspectives, including types of algorithms, decision entities, and the computational model, has also found that there is little work in decentralized decision on task-offloading.

Although there exist many MADRL-based resource allocation algorithms, they are homogeneous agents with the same functionalities running on the user devices (UD). For example, Tang and Wong (2020) proposed MADRL-based task offloading in the mobile edge computing system without knowing the decisions of other users. Users use a Long-Short-Term Memory (LSTM) network to learn the load-level dynamics of edge devices. User devices that offload to the same edge device equally share the processor. However, edge devices must make combinatorial decisions on the UDs for better performance. They need another DRL agent to decide which of the users to serve to maximize their utility. Furthermore, this work and many other task offloading algorithms (Nguyen et al., 2023; Zhang et al., 2020; Jiang et al., 2023), which make optimization from the user perspective, overlook the storage constraint of the server.

There are also DRL-based resource allocation algorithms that approach the problem from the edge server side. Huang et al. (2019a) proposed a DRL-based joint task offloading and bandwidth allocation algorithm for multi-user mobile edge computing to minimize the overall offloading cost in terms of energy, computation, and latency. This designates the problem into a single agent centralized DRL algorithm running on the edge device with a single objective function. The action space is a single index vector of a concatenation of task-offloading decisions, an increase, and a decrease of the uplink and downlink bandwidth allocation. Because it slows training and execution due to the increased dimension of the action space: deciding on task offloading, the increase in uplink bandwidth, the decrease in uplink bandwidth, the increase in downlink bandwidth, or the decrease of downlink bandwidth at a time step, it not only does not address Requirement R_{IV} but also does not address Requirement R_I and Requirement R_{III}. MADRL algorithms with multiple objectives can solve task overload and bandwidth allocation differently. Cui et al. (2019) proposed MADRL-based joint user, power level, and channel selection for unmanned aerial vehicle (UAV) networks considering each UAV as an independent and non-coordinating agent without knowing the decisions of the others. The objective of each UAV is to maximize performance. Although it minimizes the state and action dimension of centralized decision-making by allowing each agent to make decisions based on their local observations, this setting is only from the resource provider's perspective. It does not work if users are also self-interested in their objective function and therefore do not satisfy the Requirement R_{IV}. The question of how UAVs know whether a user is selected by other UAVs or not is also not explained.

A detailed review of related work on task offloading is provided in the related work section of Chapter 5.

2.1.3 Handling ASO Inputs in Deep Reinforcement Learning

Regarding Requirement R_V, the fact that DRL algorithms learn to maximize long-term returns, and the use of DNN, demonstrates their ability to be generalizable and able to adapt their decisions to the dynamics of the problem. However, they have two limitations in learning with ASO inputs and outputs. The first limitation is that existing DRL-based resource allocation algorithms are centralized. Second, the fact that standard neural networks accept a fixed-sized input vector in a specific order and output a specific number of outputs in a specific order makes it difficult for them to work with ASO inputs and outputs. This affects the generalizability of the permutation of the same problem. For example, the task offloading algorithm of Tang and Wong (2020) assumes a fixed number of edge devices in its state representation. Additionally, it does not meet the requirement in Requirement R_{II}, because it does not work for a variable number of edge devices because neural networks, which are used as function approximators in DRL, usually work with a fixed number of inputs and outputs.

We classify existing techniques for handling ASO input as neural network-based input transformations and stationary input transformations. One-hot encoding, bag-of-words (Zhang et al., 2010), zero-padding (Lopez-del Rio et al., 2020), and set-pooling are examples of stationary input transformations. Mukhutdinov et al. (2019) proposed a MADRL-based routing system using one-hot encoding. For example, if we have a network with seven nodes, number 3 is encoded as 0010000 and the set {4,6} is encoded as 0001010. Although one-hot encoding is used to avoid depending on the order in which nodes are enumerated, and when nodes are variable in number, it still has two limitations to fully address Requirement R_{II}. First, it is a static representation. It cannot represent nodes when they enter and leave the network dynamically. Second, it has a maximum limit on the number of nodes that can be represented. Furthermore, the algorithm uses one-hot encoding to represent only the nodes, assuming that only one task is processed at a time. Using one-hot encoding, it is challenging to represent multiple tasks in the nodes. The numerical values that characterize the tasks are also not represented in the one-ot encoding. Bag-of-words is a technique commonly used in NLP to represent a text by the number of occurrences of words. However, it is not suitable for numeric state representation because it only represents the occurrences of words. Zero-padding techniques assume a fixed length of the input. If an input comes with a smaller size than the assumed value, it is zero-padded to make it the set value. Huegle et al. (2019) has summarized existing approaches to dealing with variable input in autonomous driving. Edwards and Storkey (2017) and Zaheer et al. (2017) proposed element-wise transformation followed by set-pooling to solve the ASO set in ML. First, each element of the set is fed to a DNN of fixed size. Then they are concatenated using a pooling operation $mean, sum, product, and maximum$. Note that although DNN is deployed, the primary technique of handling the ASO input is aggregation of the output

of DNN with set-pooling. Lee et al. (2019) extended set pooling, to consider the relationship between the elements of the set. These methods have limitations when used for DRL. First, they are used to output the same output irrespective of the permutation of the input but cannot help in the action representation to select an element of them. In other words, they cannot be used for coalition action selection because their final output is aggregated to a single output vector with set-pooling. Second, although the methods are permutation invariant and work for any size, they can often transform different sets to the same output. This makes it ineffective for DRL algorithms because they need to interpret different states differently, otherwise, it will cause ambiguity.

Various neural network-based transformations are available, such as set neural networks (Lee et al., 2019; Zaheer et al., 2017; Qi et al., 2017), permutation-invariant neural networks (PINN) (Tang and Ha, 2021), recurrent neural networks (RNN) (Hochreiter and Schmidhuber, 1997; Sutskever et al., 2014), graph neural networks (GNN) (Scarselli et al., 2009), pointer networks (Vinyals et al., 2015), and the attention-based transformer neural network known by its title "Attention is all you need" (Vaswani et al., 2017). Set neural networks use a neural network to transform input sets into output sets and then apply maximum pooling or sum pooling to form the context vector. PINN is designed to recognize elements of an ordered input vector, even when noise is present or when some elements are missing. RNN processes inputs sequentially, making it prone to vanishing gradients and is not suitable for orderless input. GNN is designed to work with graph-structured inputs. GNN is not important for the ASO input because the inputs have no graph-structured relationship, but it can be applied by considering the ASO input as a fully connected graph. Attention-based neural networks process input simultaneously. The transformer neural network (Vaswani et al., 2017) is the state-of-the-art technique for ASO input transformation based on attention neural networks. It uses self-attention mechanisms to weigh the importance of different input tokens, allowing it to capture the dependencies between the input elements effectively. Moreover, using the multihead attention layer, computes attention across multiple heads, enabling the model to focus on different parts of the input simultaneously and enhancing its ability to learn complex relationships. A detailed analysis of the suitability of GNN and the transformer is presented in Table 2.1. We use the encoder part of the transformer neural network to handle the ASO input in Chapter 3. However, although the transformer is efficient in processing an input of arbitrary length in parallel, it is not efficient for combinatorial problems with ASO input, because it is designed to learn complex relationships between elements of an input sequence, but the tasks in combinatorial problems are independent. This unnecessary computation can affect performance and execution speed. Therefore, we design another method in Chapter 4.

Type of DNN	How it works	Advantage	Suitability for ASO
GNN	Uses an iterative message passing and embedding to represent the input	If the input is graph-structured. For example, routing	ASO can be handled using GNN by considering the elements of the ASO input as a fully connected graph but is not important because they have no graph-structured relationship. It only incurs computational cost. GNN on the fully connected graph is similar to the transformer (Wu et al., 2021b; Veličković, 2023).
Transformer	Use an attention mechanism in every layer of the encoder and decoder network to learn the dependency between elements of a sequence of input. The encoder simultaneously transforms the input and the decoder produces the output sequentially	When there is a relationship between elements of the input. For example, sequence mapping, NLP, sorting, and prioritizing.	Transformer can represent ASO better than GNN because of the attention mechanism, but there is no as complex contextual relationship as in NLP in the numerical ASO input. The elements of the ASO are independent. This unnecessary computational complexity can affect its efficiency in ASO inputs.

TABLE 2.1: Suitability analysis of GNN and transformer for ASO inputs

2.1.4 Summary of Deep Reinforcement Learning-based Resource Allocation Algorithms

Although the literature shows that DRL can meet research requirements, existing DRL-based resource allocation algorithms do not meet these requirements. While DRL has the potential to achieve Requirement R_{III} since it can be trained offline or in a simulated environment before it is applied in the real world (Nian et al., 2020b), it needs further modification when applied to a combinatorial problem with ASO inputs and outputs, because existing algorithms make a costly sequential decision. Existing DRL-based resource allocation algorithms have considered a mechanism to handle an arbitrary action space to fulfill the Requirement R_I. Furthermore, most existing DRL algorithms do not handle ASO inputs regarding Requirement R_{II}, because they assume a fixed input and output size or have a limit on the maximum number that can be represented. Requirement R_V is not satisfied for the same reason. Concerning Requirement R_{III}, although the transformer is state-of-the-art in representing and performing parallel processing of the ASO input, it is computationally inefficient for combinatorial

problems where the tasks are ASO. Furthermore, most existing DRL-based resource allocation algorithms do not address Requirement R_{IV} because they are single-agent approaches with one objective function or homogeneous MADRL agents with the same constraints. Similarly, existing resource allocation algorithms are not scalable to solve multiple problems using MADRL to meet Requirement R_{VI}.

Therefore, the aim is to solve dynamicity and scalability in terms of ASO inputs and outputs as in requirements R_I and R_{II}, which also underpin R_I. We proposed an input transformation for the ASO input to address R_{II}. To further solve the permutation invariance and flexible size in the action space, we propose a novel coalition action selection for the ASO action space. We also proposed a combinatorial MADRL algorithm for the MARL scenario to satisfy all the research requirements.

2.2 Reinforcement Learning

Reinforcement learning (RL) and DRL are sometimes used interchangeably, but DRL is used when the RL uses DNN as a function approximator. Unless otherwise stated, the terms RL and DRL, as well as MARL and MADRL, are also used interchangeably in this thesis. Because this research aims to advance DRL and MADRL by addressing the challenges they face in online resource allocation problems, we perform a comprehensive review of existing types of RL algorithms and discuss their suitability for the specified research challenges. First, we present the introduction and types of RL algorithms and discuss their suitability for the proposed contributions in Section 1.3. Then, to deal with the Requirement R_I and the Requirement R_{II}, we review the representation of state and action spaces. Both are important to minimize the online computational cost of for Requirement R_{III}. Lastly, we conducted a detailed review of MADRL and the types of coordination in relation to the proposed CCM_MADRL_MEC.

2.2.1 Introduction

DRL is a branch of ML, which is a branch of AI, characterized by its learning ability through trial-and-error interaction of an agent with an environment that gives either a punishment or a reward for an action applied to an observed state. Training of the algorithm is carried out in such a way that it chooses actions to maximize its long-term reward. It is attracting attention due to its ability and success in solving many complex problems (Nguyen et al., 2020). David et al. (2021) have posited that the optimization of rewards is sufficient to simulate all artificial and natural intelligence.

The interaction between a DRL algorithm and the problem to be solved is described as a Markov Decision Process (MDP) (van Otterlo and Wiering, 2012). An MDP is a Markov

reward process plus an action. A Markov reward process is a Markov chain with a reward added. A Markov chain is a process with state and state transition probabilities. The description of an MDP consists of five main elements:

- A set of states S. At each time step, the state of the environment is an element $s \in S$ that is self-contained information to make decisions about the environment.
- A set of actions A. At each time step, the agent chooses an action $a \in A$ to perform in the environment.
- A state transition model $P(s_{t+1}|s_t, a_t)$ that describes the probability that the state of the environment changes to the state s_{t+1} when the agent acts a_t in the state s_t.
- A reward function $R(r|s, a)$ that describes the real-valued reward r that the agent receives from the environment after acting a in state s.
- A discount factor $\gamma \in [0, 1]$ that controls the importance of future rewards.

Almost all DRL algorithms are described using MDPs because they have at least the action, state, and reward. However, not all DRL algorithms require full knowledge of the MDP description. For example, model-free DRL algorithms do not use the state transition probability. The next state is also not used in some DRL algorithms if only the immediate reward is of importance. The discount factor is also not needed if only the immediate reward is considered. The contextual bandit problem is an example of where the next state is not needed.

In addition to the MDP, the DRL algorithm is also described by additional terminology, including agent, environment, and policy.

- The agent is the system that observes the state of the environment and executes a policy to select an action and apply it to the environment.
- The environment is the problem that the agent is solving.
- The policy maps the observations to actions.

The evolution of RL takes its root from K-armed bandit problems, where the agent only obtains values for its actions, taking the state as fixed (Nian et al., 2020b). The value is affected only by the action chosen because the state is fixed. A K-armed bandit problem is a simplified setting of RL (Intayoad et al., 2020). An extension of the K-armed bandit is the contextual bandit, where there can be many states that affect the choice of action selection. The contextual bandit is intermediate between the K-armed bandit and the full DRL (Intayoad et al., 2020). It focuses on immediate rewards, as does the K-armed bandit. At the same time, it involves complete DRL problems, since it uses a policy to

select actions based on contextual information. It is also called an associative DRL in the literature (Li et al., 2010). Lastly, full RL has the characteristics that the actions at a given state can have both a direct effect on the reward and an indirect effect on the state transition of the environment, which is then termed the next state. In addition to immediate reward, actions may have long-term consequences. As a result, immediate rewards can sometimes be sacrificed to obtain more in the long term (Intayoad et al., 2020).

RL originates from two main fields of research (Nian et al., 2020b): optimal control using value functions and dynamic programming, and trial-and-error search inspired by animal psychology. The optimal control optimizes the input trajectory using the functional equation known as the Bellman equation as shown in (2.1).

$$V^*(s) = \max_a(R(s,a) + \gamma \sum_{s'} P(s' \mid s,a).V(s')), \tag{2.1}$$

where $V^*(s)$ is the optimal goodness or value of being in state s, $V(s')$ is the goodness or value of being in the next state s', $R(s,a)$ is the reward of taking action a in a state s, γ is the discount factor, which is the current value of a future reward, and P is the transition probability that state s goes to state s' when action a is taken. If the value of taking an action in a given state matters more than the value of being in that state, the Bellman equation is updated as in (2.2).

$$Q^*(s,a) = R(s,a) + \gamma \underset{a'}{max} Q(s',a'), \tag{2.2}$$

where $Q^*(s,a)$ is the optimal goodness or value of taking action a in state s and $R(s,a)$ is the reward of taking action a in a state s.

When the problem to be solved by RL has a fixed number of states and actions, the policy is updated to tabular data using Equation (2.1). For large or continuous state spaces, DNN is used as a function approximator, instead of a table, to generalize for unseen inputs. This is discussed in detail in the discussion of the types of RL in the following. In DRL, the state, action, and reward experiences are stored in the replay memory at each time step. The DRL algorithm is trained using the data in replay memory.

During learning by trial and error, the DRL agent uses exploration or exploitation to select an action for a given state. Exploitation is the selection of the best action based on the learned knowledge, whereas exploration is the testing of any random action out of the existing choices. The agent is likely to use more exploration in its early stages, but gradually prefers more exploitation to take advantage of its knowledge. After being fully trained and deployed for a real application, the agent uses exploitation. Although exploration can help an agent test how good or bad unknown actions are, it often comes with a cost in the form of penalties or missed opportunities (Chalkiadakis and Boutilier, 2003) if it is executing and learning at the same time in the real world. As an extreme

example, a self-driving car cannot explore the real world because it will crash. Thus, the self-driving agent is trained with both exploration and exploitation in a simulation environment and deployed with the exploitation model in the real world. The exploration-exploitation trade-off is one of the challenges of MADRL. This is because, unlike single-agent DRL where only one agent is interacting with the environment, multiple agents need to coordinate their policies on equilibria to make an optimal joint action. The problem is that agents must distinguish whether other agents are exploring or exploiting to better track behavioral patterns and coordinate joint action. Chalkiadakis and Boutilier (2003) have proposed an optimal exploration mechanism for coordinated MADRL using a Bayesian approach to predict the exploration cost. The type of exploration varies with the type of DRL algorithm. For example, epsilon-greedy exploration is commonly used in Q-learning and its extensions; deterministic policy gradient algorithms usually add exploration noise to their actions to explore new actions.

In the next sections, we discuss different categories of DRL concerning their importance to our contributions.

2.2.2 Types of Reinforcement Learning Algorithms

RL algorithms differ in many contexts. In this section, we present the categorization of RL algorithms from various perspectives.

2.2.2.1 Single-Agent vs Multi-agent Reinforcement Learning

RL algorithms can be classified as single-agent and multi-agent by the number of agents interacting with the environment. Single-agent RL is an agent that interacts with the environment, while in MARL, multiple agents interact with the environment in a cooperative, competitive, or mixed setting. The review of the literature on MARL is presented separately in Section 2.2.8.

2.2.2.2 Model-based vs Model-free

This classification is mainly based on the state transition model of the environment. If a perfect environment model has already been given, dynamic programming or DRL can be used to solve the problem (Sutton and Barto, 2018). If the environment model is not given, but can be constructed by learning from experience, the model-based DRL is used to learn both the state transition model of the environment and the policy (Dayan and Niv, 2008). It uses a planning algorithm along with the learned model to solve the problem. The next state and the reward in the model-based DRL can be estimated before applying the action. Model-free DRL does not need the state transition model.

Q-learning (Watkins and Dayan, 1992) and DQN (Mnih et al., 2013) are examples of model-free DRL because they directly estimate the optimal Q values of each action in each state. Whether to choose model-based or model-free DRL depends on whether it is possible to formulate the state in a way that allows viewing the next state and reward before applying the action.

Most resource allocation algorithms use model-free DRL algorithms because the state transition model of network dynamics is difficult to model. This is because the state dynamics of the network is affected not only by the actions taken but also by other factors, including unstable signal strength, dynamic joining and dropping of users, and online arrival rate of the tasks. Therefore, we chose a model-free DRL for all of our proposed methods.

2.2.2.3 Policy Optimization vs Value-function Optimization

RL algorithms are also classified into two based on how they optimize their policy. Q-learning and its extensions learn by optimizing their state values or state-action values, called value functions. The state value is the goodness of being in a given state, whereas the action value is the goodness of selecting a given action in a given state. Q-learning (Watkins and Dayan, 1992), state-action-reward-state-action (SARSA) (Rummery and Niranjan, 1994), DQN (Mnih et al., 2013), and their extensions are examples of value function optimization in RL. In these RL algorithms, the policy is executed by selecting actions with the highest state value or action value (Q value). The extensions of the Q-learning algorithm are presented in Section 2.2.3.

RL algorithms based on policy optimization, on the other hand, directly optimize their policy. The RL agent directly optimizes the policy to generate actions instead of optimizing the value of the actions or states. They are also called actors. The deterministic policy gradient (Silver et al., 2014) and the stochastic policy gradient (REINFORCE) (Williams, 1992) with some other variants are RL algorithms based on policy optimization. The extensions of the policy gradient algorithm are presented in section 2.2.4.

In comparison, algorithms based on optimization of the value function are more likely to converge to the optimum action values because Q learning (Watkins and Dayan, 1992) is theoretically proven to converge to the optimum value with a probability of 1, while policy optimization algorithms are more likely to converge to a locally optimal policy (Sutton et al., 1999). On the other hand, policy gradients work with both discrete and continuous action spaces, whereas value function optimization-based algorithms, such as Q-learning and its extensions, work with discrete action spaces.

In our research, both policy gradient- and value function-based DRL algorithms are used. We use the value function-based DRL algorithm for Chapter 3 and Chapter 4.

Actor-critic methods use value function optimization in their critics and policy gradient optimization in their actors. To take advantage of the convergence of the value function and the suitability to both discrete and continuous action spaces of the policy optimization, we combined both as a client-master MADRL algorithm in Chapter 5.

2.2.2.4 Monte Carlo vs Temporal Difference

If the state transition model of the problem is not known, we have two approaches to update the value of taking actions in the states: an episodic update, known as Monte Carlo, and a step-by-step update, which is also known as temporal difference (TD). Note that if the model is already given, we can solve the problem using dynamic programming. The term Monte Carlo is often used more broadly for any estimation method whose operation involves a significant random component, but in DRL, Monte Carlo methods are ways to solve a problem based on the average returns of the samples (Sutton and Barto, 2018). The Monte Carlo method is simple. It undergoes a trajectory of states and actions and updates its values after the entire episode. The intuition behind Monte Carlo is that, after many episodes, the values for all state and action pairs will be approximated to the real values because they are being averaged over many samples. Thus, Monte Carlo methods have high variance because the values for the actions depend on the averages of the samples. In contrast, TD approaches update the values of actions after every step. Therefore, Monte Carlo is more appropriate when the rewards of taking actions at individual states are not very certain but depend on the overall performance of the episode, whereas time differences are more appropriate, with rewards certainly known at every step. Q-learning and its variants are examples of TD RL algorithms, whereas algorithms based on policy gradients are Monte Carlo RL algorithms. Actor-critic methods, a combination of policy gradient and Q learning algorithms, are also TD methods. All algorithms we propose in this work are TD methods.

2.2.2.5 On-policy vs Off-policy

In on-policy RL, the algorithm uses the same policy when selecting an action and training. SARSA (Rummery and Niranjan, 1994) is an example of on-policy because it always uses the same policy for training and selecting the action. On the other hand, off-policy can use different policies when selecting the action and when making a policy update. Q-learning is an example of an off-policy RL algorithm because it uses the best action to determine the next state in the policy update, but it can sometimes use a random action to interact with the environment. Policy gradient algorithms have both on-policy and off-policy variants. The on-policy-based policy gradient is when we use the samples of an episode generated from the same policy to train the policy,

too. The off-policy-based policy gradient (Imani et al., 2018) can mix different samples of different policies with its training sample.

In this research, we use off-policy RL algorithms because all of them learn from a sample of replay memory, which is collected by different policies.

2.2.3 DQN and Its Extensions

In this subsection, a brief review of some of the extensions of DQN is presented based on the survey in Luong et al. (2019).

The first deep Q-learning network (DQN) was proposed by Mnih et al. (2013) to play Atari. DQN is a breakthrough in RL because, unlike Q-learning, it works with continuous state spaces using the advantage of generalization in deep learning. Because DQN shows poor performance due to the overestimation of actions caused by the positive bias introduced because they use the same policy to select the best action and assess the expected value of the action in the Bellman equation, it is further extended to double deep Q-learning (DDQN) (Van Hasselt et al., 2016). DDQN is proposed to solve the positive bias of DQN using two Q values, Q1 and Q2. The main idea is that the first DQN, the weights of which are slowly copied from the second, is used to select the action as usual, but the evaluation of the action value comes from the second Q-value. DRL collects its data set by storing its interactions with environment experiences in its replay memory. The replay memory stores the experiences in the form of $< state, action, next\ state, reward >$ The aforementioned DRL algorithms learn from the replay memory sampled almost at the same frequency as what the agent has experienced. Prioritized replay memory is introduced by Schaul et al. (2016) to help take the sample in such a way that experiences are selected if there is much to learn from them. A downside of the DQN extensions explained so far is that sometimes, especially if the action space is too large, it could be unnecessary to learn all the action values at every state to reach convergence. To address this, dueling DQN (Wang et al., 2016) is proposed to decouple the state value and the action value into two separate streams that are then aggregated together. This helps the algorithm to learn faster without the need to learn every action value in every state because the state value has already been known by a separate stream. Asynchronous multistep DQN (Wang et al., 2018b) is proposed to reduce the computational cost of training from replay memory. In this, multiple agents cooperate asynchronously in their gradient descent updates after they are trained in parallel in their version of the environment. In some problems, using the estimated value may not be an accurate solution if the environment is stochastic and its reward follows a distributional value. A distributed DQN approach (Bellemare et al., 2017) is proposed to update the Q-values for a distributional rather than a scalar value. Lastly, the rainbow DQN (Hessel et al., 2018) integrates all the concepts of the DQN extensions.

In this work, as with all DDPG algorithms, we use DDQN to train the master agent of the proposed CCM_MADRL_MEC algorithm.

2.2.4 Extensions of Policy Gradient Algorithms

Policy-gradient DRL is called actor-only because it does not use any form of a stored value function (Grondman et al., 2012). For example, REINFORCE (Williams, 1992) is a stochastic policy gradient algorithm that directly optimizes the probability of selecting its actions based on a sample generated from a trajectory of states and actions over many episodes. Critic-only methods such as Q-learning, SARSA, and DQN use a value function and no explicit function for the policy (Grondman et al., 2012). A major advantage of policy gradient (actor-only) over value-function (critic-only) methods is that they allow the policy to generate actions in a continuous action space (Grondman et al., 2012). However, policy-gradient DRL algorithms suffer from the following problems:

- High variance: The reward of the actions is inconsistent because it is averaged or propagated to the subsequent actions of the episode.
- Delayed reward problem: The reward is usually known at the end of the episode.
- Sample inefficiency: The policy is usually updated using the sample generated in the last trajectory.

Below are the extensions of the policy gradient DRL algorithm to solve these problems.

2.2.4.1 Actor-critic Method

The actor-critic method (Witten, 1977; Barto et al., 1983; Mnih et al., 2016), can be classified as a combination of policy optimization and value-function optimization. The actor is policy optimization. It is used to select actions. The critic is value-function optimization. It is used to give feedback to the actor. Actor-critic is proposed to address the variance in the policy-gradient algorithms because, unlike the averaged or propagated reward over the actions of an episode in Monte Carlo, it directly gets critics for every action. Since REINFORCE updates its policy using the Monte Carlo method, it may consider all intermediate actions as good if the final reward of the episode is good because the final reward is propagated to all intermediate actions. Therefore, more samples are needed to eliminate the variance from the propagated reward. This causes slow learning. The actor-critic solves this by changing the policy update from Monte Carlo to TD. The total reward in the gradient is replaced by a Q-value of the current action for the current state, which is trained by a separate critic DRL algorithm. Inspired by the dueling DQN, an advantage can be introduced to the actor-critic to

stabilize learning by removing variability. The advantage is calculated by subtracting the average Q-value of actions taken in that state from the Q-value of the action. This indicates the extra reward gained by taking the action relative to the average taken in that state. The advantage, $A(s,a)$, is calculated by subtracting the action value, $Q(s,a)$, from the state value, $V(s)$, as $A(s,a) = Q(s,a) - V(s)$ where s is the state and a is the action selected in that state. If $A(s,a)$ is greater than zero, the gradient moves in that direction; otherwise, it is in the opposite direction.

There are many other extensions of the actor-critic methods. The DDPG (Lillicrap et al., 2016) handles both continuous states and actions using a combination of value-based and policy-based approaches with deep learning. Lowe et al. (2017) proposed a multi-agent version of DDPG called MADDPG.

2.2.4.2 Off-policy Policy Gradient

REINFORCE (Williams, 1992) is on-policy DRL because training samples are collected according to the target policy. Once the policy is updated, the old training samples are removed. Off-policy policy gradient algorithms (Imani et al., 2018) provide several additional advantages compared to on-policy DRL algorithms.

First, the off-policy approach does not require full trajectories and can reuse any past episodes for much better sample efficiency. Second, the sample collection follows a behavior policy different from the target policy, bringing better exploration.

In our research, the client agents of the CCM_MADRL_MEC are off-policy-based policy optimization algorithms like the DDPG. The master agent and the other algorithms in Chapter 3 and Chapter 4 are off-policy value functions.

2.2.5 State Space and Representation

This section discusses existing types of state spaces and their representation.

2.2.5.1 State Space

Q-learning is a tabular DRL approach for definite discrete state spaces. A Q-function performs the value update and action selection policies. Q-learning has two limitations. First, it is difficult to handle very large state spaces in tabular form. Second, it cannot support continuous state spaces. DQN, which substitutes the Q-function of Q-learning with DL, addresses both limitations of Q-learning. For the first limitation, it does not use a table, but a DNN, and regarding the second limitation, it works for continuous state spaces using the advantage of generalization from deep learning.

Due to the nature of resource requirements, all of our algorithms in this research assume a continuous state space and use DL. The action space is continuous in the client agents of CCM_MADRL_MEC but discrete and arbitrary in size in the others.

2.2.5.2 State Representation

In typical DRL algorithms, states are typically represented by a fixed-size feature vector. The elements that compose the feature vector are in a specific order in all states. However, there are sometimes states with a variable number of permutation-invariant elements, which can be represented as sets. For example, the knapsack problem must find the best combination of the items, regardless of the order of the items in the input. A resource allocation algorithm has to give the same solution no matter what the order of the users in the input vector is. Standard DRL does not recognize that an input vector is permutation-invariant, because standard neural networks accept inputs in the order given to them. This slows the learning of the algorithm if the elements that make up the state are permutation-invariant. For example, in DRL, if the state is {12, 34, 56, 67} and if its order does not matter, the algorithm will consider {56, 34, 12, 67} as a different state because the elements will be fed to the neural network of the DRL algorithm differently and it does not take advantage of its knowledge of one or more of its previous permutations. States with permutation-invariant elements in their feature vector need to take advantage of prior knowledge of the instance of their permutation. Furthermore, since neural networks are usually configured with a fixed number of inputs and outputs, they cannot accept a feature vector if its number of elements increases or decreases.

In natural languages, states with a permutation invariant number of elements and variable number are represented by bag-of-words (Zhang et al., 2010). Bag-of-words is information about the number of occurrences of words. This is not convenient for use for numeric state representation for two reasons. First, if the environment is a continuous state space, the dictionary is indefinite. Therefore, it is difficult to have a fixed dictionary. Second, it only represents the occurrence of numbers as words, not their values. A detailed discussion of ASO is given in Section 2.1.3

2.2.6 Action Space and Representation

In this section, we present a review of the existing types of action space and the representation of actions in DRL algorithms. We also clarify how our action representation is different.

2.2.6.1 Action space

DRL can have discrete and continuous action spaces. The action space is finite in discrete action DRL. In the continuous action space, the action is a real value between a given range. When a combination of discrete and continuous action spaces is necessary, a parameterized action (Masson et al., 2016) uses a combination of discrete and continuous actions. It includes continuous parameters in discrete actions.

2.2.6.2 Action Representation

Continuous action spaces are represented by a vector of continuous values. If the required action is a single scalar value, it is outputted as a single value from the DRL algorithm. If the required actions are multiple actions with continuous values, they are outputted as a vector of continuous values.

In the discrete action space, the action representation depends on the number of actions as follows.

2.2.6.3 Action Representation with Fixed vs Variable Number of Actions

If the number of actions is fixed, the actions are represented by labels equal to the number of actions. For example, if the actions are two: *left* and *right*, they are labeled 0 and 1, respectively, in the output of the DRL algorithm. The actions are selected by their Q-value or by their probability. There are two methods to handle problems with a varying number of actions in natural language processing using DRL: the per-action DQN and the deep reinforcement relevance network (DRRN) (He et al., 2015, 2016b). Their main intuition is that the algorithm learns the relevance of each action to the state by embedding each action within the state and taking it as input to the deep neural network with an aggregate output for every state-action combination. Because the actions are learned separately from the state, the action selection is made by inputting the state with all possible actions one by one into the trained policy, recording their Q-values, and choosing the best one. The state and action are inputted into the same neural network in the per-action DQN, whereas two separate neural networks that are finally integrated into one output are used in the DRRN.

In the resource allocation problem, there are likely to be varying numbers of users and resource providers. Therefore, as complementary to the per-action DQN, we also design a method to handle variable state sizes by transforming the input into a fixed state representation in Chapter 4.

2.2.6.4 Action Representation in Large Discrete Action Spaces

Discrete action spaces are intractable for large action spaces. Dulac-Arnold et al. (2015) (Wolpertinger Architecture) proposed a policy gradient algorithm that outputs a continuous proto-action that is used to select the nearest-neighbor discrete action. The efficiency of this continuous action representation can be impacted by how the discrete actions are sorted. Instead of selecting items with the closest index to the continuous action outputted by the actor, several approaches used a scoring function parameter outputted as a continuous action from the actor and then multiplied it with a predefined ranking of the actions computed using word embedding (Zhao et al., 2017; Hu et al., 2018; Liu et al., 2018). Chandak et al. (2019) proposed an action representation method by learning the structure of the action set from the observed transitions. The limitations with all of these action representations are that they assume that there is a predefined ranking parameter, the action set is fixed, or the actions have a stationary behavior.

In this work, we proposed a coalition action selection that minimizes the dimensionality by selecting a coalition of multiple actions at a time. We used the transformer to output the Q-value for all of the actions at the same time.

2.2.6.5 Single Action vs Multiple Action Selection

Most existing DRL algorithms select a single best action at a time. However, some problems require combinatorial selection of multiple actions simultaneously. For example, a service placement algorithm may need to decide on a combination of services to fetch from the cloud at one time. Two mechanisms enable such combinatorial action selection at the same time: action branching (Tavakoli et al., 2018) and DRRN. DRRN-based combinatorial action selection is inspired by the per-action DQN (He et al., 2015). The number of actions is a combination of items. Every possible combination, which is considered an action, is fed to the neural network one by one along with the state, and finally, the combination with the best Q-value is selected.

Although the DRRN-based method for combinatorial action selection handles the flexible number of combinations of a combinatorial problem, there is still a large computational cost for taking all possible combinations of items as the number of actions. Our research proposes a new combinatorial action selection DRL algorithm that has a linear action space. The number of actions is the same as the number of items, unlike the DRRN-based algorithm, which has several actions equal to the possible number of combinations of items.

The action branching assumes that the system is made up of components that run subactions with some independence. This works well if the components have different

tasks with independent decisions, but they affect the state of the system. However, if the components do the same thing or if there is no need for independent decisions in the components, the action space can grow to twice the number of items. For example, Zhang et al. (2019) used ten actions to decide whether to broadcast or not to broadcast five items. Furthermore, this algorithm does not work for flexible-size states and variable numbers of actions. We proposed a deep combinatorial DRL for a flexible number of inputs, which has minimized the action space and works for a varying number of inputs.

In the literature, there are multiplayer K-armed bandit approaches to perform multiple actions at the same time (Besson and Kaufmann, 2018). However, players are multiple agents who play with multiple arms. This is more of a multi-agent DRL algorithm than the multiple action selector agent, requiring a coordination mechanism for multiple players to avoid collisions on the same arms, whereas the multiple action selection algorithm we are seeking is one agent deciding the best combination of the actions.

2.2.7 Types of Environments

DRL environments can be classified into many categories. An environment can be classified by the number of agents as single-agent and multi-agent. It can also be classified as discrete or continuous in the action space. The environment can be classified as deterministic and stochastic, depending on whether the next state can be determined based on the current state of the environment and the action of the agent. The environment can be classified as episodic and sequential by the effect of the previous action. If the current action is affected by the previous action, the environment is sequential; otherwise, it is episodic.

In the resource allocation algorithms of this work, the next states are not always deterministic due to the random arrival of online tasks. We used an episodic environment because the tasks are instances and are independent of each other. In task offloading, the environment is sequential because the previous offloading decision affects the current decisions.

In terms of observability, the environment can be either fully observable if an agent has a full view of it, or partially observable if an agent has a partial view of the environment. Full observability uses Markov decision process modeling. An agent with partial observability only observes the environment rather than the state because the state is not fully observable. The partially observable environment in DRL is modeled by a partially observable Markov decision process (POMDP) (Jaakkola et al., 1995). The user is unable to directly observe the state of the environment, but models the probability of transitioning between the states and observations.

In resource allocation, the environment is usually fully observable if the DRL algorithm is deployed on a centralized server or resource provider. This is because the server usually has a full view of its network by collecting information from all users. The environment can also be partially observable if a decentralized MADRL is used, which is also the setting we are proposing, where every node in the network participates in the decision of resource allocation. Therefore, our work considers both full and partial observability.

Environments can be classified by the statistics of their model as stationarity and non-stationary. A stationary environment is an environment whose state or reward transition dynamics are stationary. In contrast, a non-stationary environment is an environment whose state or reward transition dynamics change with time (Padakandla et al., 2020). In a multi-agent DRL setting, non-stationarity occurs due to concurrent learning of multiple agents, which causes an action taken by an agent to affect the received reward and evolution of the state of opponent agents (Zhang et al., 2021). This is discussed in the challenges of multi-agent DRL in section 2.2.8.

2.2.8 Multi-agent Reinforcement Learning

Multi-agent DRL is a DRL algorithm with multiple agents involved. The transition of state and the reward received from an agent are influenced by the actions of the other agents (Zhang et al., 2021). This section presents the types of MARL algorithms by their type of setting and training, their advantages and challenges, and their types of coordination.

2.2.8.1 Types of Settings in Multi-agent Reinforcement learning

Agents in DRL can be classified by their setting as cooperative, competitive, and mixed agents. Cooperative agents work to maximize a shared objective or reward. In a competitive setting, some agents increase their reward, while other agents decrease.

The MARL algorithm in our research is a mixed setting because the resource provider and the users cooperate to maximize their utility. However, resource-requesting users are competitive because they compete to share the limited resources.

2.2.8.2 Types of Training in Multi-agent Reinforcement learning

MARL algorithms can be classified as independent, centralized, and decentralized based on how they are trained and executed. Independent Q-learning (IQL) (Tan, 1993) is the simplest MARL algorithm, where agents are trained and executed independently considering the other agents as part of the environment. They can optionally share their

experience data to maximize sample efficiency. Agents do not communicate or coordinate their actions or observations. As a result, the shortcoming of this IQL is the inability to efficiently handle partial observability and non-stationarity. This makes IQL not optimal. A similar but slightly different setting is isolated MARL where agents dynamically divide the environment at every time step and act independently on their partitions as in Sun et al. (2019). In centralized training, agents are trained centrally but executed in a decentralized environment. For example, counterfactual multiagent policy gradient (COMA) (Foerster et al., 2018b), MADDPG (Lowe et al., 2017), value decomposition networks (VDN) (Sunehag et al., 2018), and QMIX (Rashid et al., 2018). In decentralized training, agents are trained and executed differently, but they use coordination and communication among them. For example, the work by Zhang et al. (2018) is a decentralized training but shares its parameters with its neighbors. This distinguishes them from independent training.

In our work, CCM_MADRL_MEC is centralized training, but it can be deployed as decentralized or centralized in the execution. Client agents can run either on the UDs or on the server, whereas the master agent is always on the server.

2.2.8.3 Advantages and Challenges of Multi-agent Reinforcement learning

MARL has many advantages compared to a single agent; it converges faster due to experience sharing, skilled agents teach new agents or new agents imitate skilled agents; learning is faster due to parallel processing; if an agent fails, the other agents take the task (Buşoniu et al., 2010). Moreover, the dimensionality is reduced because agents make decisions based on local observation.

MARL also has many challenges. Zhang et al. (2021) has reviewed the challenges of MARL. Non-stationarity is one of the challenges. The state of the environment may become non-stationary for one agent because other agents, which change their choice of action as they learn, are applying actions to it. As a result, agents need to account for and adapt to the behavior of the other agents. There are many types of research for agents modeling other agents (He et al., 2016a; Foerster et al., 2018a; He et al., 2016a). In the CCM_MADRL_MEC algorithm, we ignored the nonstationarity, since ignoring may achieve satisfiable performance, as discussed in the review of the literature by Zhang et al. (2021). Third,

Coordination is also a challenge in MARL. Users must coordinate their actions. For the resource allocation problem, Tang and Wong (2020) used the LSTM network to predict the load on the edge device instead of directly coordinating the actions of the users. In our work, we designed our algorithm to be an actor-critic method to take advantage of their coordination method.

Another challenge in MARL is communication constraints. However, in networking, nodes can communicate in their usual update of routing information at the beginning of every time step.

2.2.8.4 Parallel Processing with Single Agent Reinforcement learning

In addition to the advantages of the MARL, many cases require MARL-like parallel processing for the simultaneous decision of many tasks in a single agent.

DRL can make simultaneous decisions using a single agent. In most of the existing work on problems that are made up of multiple items, such as recommendation systems, decisions are made sequentially. This is because the decision on one item gives information about the interests of the user. However, there are times when a decision on one item has no information to tell about the other item. For example, in the task offloading problem by Zhang et al. (2009), the tasks are decided sequentially one after the other, by updating the agent's state after each decision on the tasks. This incurs a computational cost. Since all tasks need to undergo a decision by the DRL agent at the same time step in time, there should be a method to make simultaneous decisions on the tasks with their state computed only once.

The difference in the MARL algorithm for simultaneous decisions in a single agent from the usual MARL algorithms used to improve performance is that it converts sequential decisions to simultaneous decisions. In the proposed MARL-based task offloading with simultaneous decisions, a node decides the task offloading decisions of its tasks simultaneously.

Moreover, dimensionality reduction, hybrid discrete-continuous actions, multiple objective functions, and multiple types of constraints are other factors that require MARL.

2.2.8.5 Types of Coordination in Multi-agent Reinforcement learning

There are many types of coordination in MARL. In this section, we have categorized them as coordination by the actor-critic method, coordination by factorization of the value function, and coordination by the graph network.

By taking advantage of the feedback from the critic, actor-critic methods are extended to coordinate multi-agent DRL algorithms for policy gradient-based algorithms. The types of coordination in actor-critic methods can be grouped into three as follows (Lyu et al., 2021). The first type of coordination using the actor-critic method is the joint actor-critic algorithm (JAC) (Wang et al., 2019; Bono et al., 2018). JAC learns in the joint state and action space of the agents, with a centralized actor and a centralized critic as a single agent. The second type of coordination by actor-critic is independent actor-critic

(IAC) (Foerster et al., 2018b; Lyu et al., 2021; Tan, 1993). In IAC, each agent learns its own independent actor and independent critic. The third is the independent actor and central critic (IACC) (Lowe et al., 2017; Foerster et al., 2018b). This learns independent actors but a centralized critic that uses the state and action of all of the actors as its input coordinates the actors by giving feedback about their joint actions. The IACC can also be classified into two: a fixed number of agents and a variable number of agents. In the fixed number of agents, the critic accepts information from a fixed number of actors. On the other hand, an IACC with a variable number of agents coordinates any number of agents in any permutation. GAMA (Chen et al., 2020) is an example of IACC with a variable number of agents.

The intuition behind using a centralized critic to coordinate MARL agents is that the critic can provide feedback to the agents on the best-combined decision. However, Lyu et al. (2021) have researched the contrast and comparison between centralized and decentralized critics and have found that a centralized critic is not better than a decentralized critic and that they have their advantages and disadvantages. While it is true, as they experimentally demonstrated, that it is not what actions other agents make but what is the best possible reward that it can get that matters as feedback from the critic for an agent, because the agents will make decisions based on local observation, the centralized critic has other advantages. If the critic has to not only give feedback but also participate in action selection, as in the work by Zhang et al. (2020) and Jiang et al. (2023) it needs information about states and actions of all actors, which the decentralized critic is not suitable for making a coordinated decision.

MARL can also be coordinated by factorizing the joint value function of individual agents. Unlike the actor-critic approach of coordination, which has a policy-based actor and a value-function-based critic, the value function factorization methods only have value functions. They use the joint value function instead of the critic. VDN learns the individual agent value functions by factorizing a joint value function, which is a sum of individual value functions. QMIX is an extension of VDN, which, instead of factorizing the joint value function into the agents, ensures that the *argmax* performed on the joint action-value function is the same as the *argmax* performed on the value function of the individual agents. It learns a joint action value function which is a nonlinear combination of the per-agent action value functions which only condition on local observation. Qtran (Son et al., 2019) is another method of coordination with a general factorization of the value function, avoiding the additive and monotonic constraints of VDN and Qmix.

The limitation of coordination by value function is that, like all value-based algorithms, it is complex to apply for large action spaces and variable numbers of actions.

Graph networks are also used to coordinate any number of agents, regardless of their permutation. Graph network-based coordination mechanisms can be used in training

and execution. A value function-based deep coordination to coordinate multi-agent DRL algorithms in a permutation-invariant and flexible-size action space. GAMA used a graph network and an attention mechanism to coordinate the training of a critic for any number and permutation of agents.

In the CCM_MADRL_MEC, we customized the centralized critic to a master agent to make a combinatorial decision based on the combined state and action of all actors and the per-action DQN. The master agent also serves to provide feedback to the client agents. By doing this, we used the critic not only for coordinated training, as in classical actor-critic, but also for coordinated action selection. Because the actors in actor-critic are policy gradient and the critic is a value function, using the master agent for providing feedback and for making combinatorial decisions leads to better convergence and cooperation than using a critic for providing feedback only.

2.2.9 Summary of Deep Reinforcement Learning

The existing resource allocation algorithms are summarized in Section 2.1.4. In this section, we summarize the existing DRL algorithms relevant to our problems.

Based on the literature review:

- Arbitrary action space and arbitrary action selection are understudied in DRL. There are DRl algorithms combinatorial action selection approaches, but they have fixed numbers of outputs.
- Existing MARL algorithms are homogeneous agents that are either a policy gradient or a value function and are not convenient for the various constraints of the resource allocation.

The existing DRL and MADRL algorithms are customized and used for the proposed algorithms as follows.

- The encoder part of the transformer is customized to make parallel processing of ASO input for the DRL with coalition action selection
- Per-action DQN is applied for making arbitrary action selection with stationary ASO transformation in Chapter 4 that replaces the transformer. Because a common state is computed for the ASO input before applying to the DQN, the coalition action selection with stationary ASO transformation also allows independent and parallel execution of the input with DRL as a single agent to get the Q-values as described in Section 2.2.8.4.

- The per-action DQN is applied to the CCM_MADRL_MEC in Chapter 5 as well to customize the critic to a master agent for combinatorial decision
- The use of a centralized critic for coordination is customized by modifying the critic to make combinatorial decisions for better coordination
- By adapting the coalition action selection and the per-action DQN, MADDPG is customized to CCM_MADRL_MEC
- DDQN and prioritized experience replay are applied to all proposed algorithms

Chapter 3

Deep Reinforcement Learning with Coalition Action Selection for Online Combinatorial Resource Allocation with Arbitrary Action Space

3.1 Introduction

Combinatorial optimization (CO) problems involve finding the best possible combination of discrete elements from a given set of feasible options. CO problems can be found in many fields, including resource allocation and routing. A resource allocation problem is considered a CO problem when it involves deciding on a combination of tasks to maximize an objective function. The *tasks* can include computational tasks in task offloading, which need to be transferred from low-capacity devices to other devices for faster computation (Zhang et al., 2009), or traffic demands or flows that require bandwidth resources over a communication network (Liu et al., 2021). deciding

As introduced in Section 1.1, mathematical optimization algorithms have been used to solve CO problems (Sundermann et al., 2021). However, if tasks in a CO problem arrive online and in arbitrary numbers, standard optimization algorithms are not efficient because tasks are not known in advance, that is, online CO problems require online decisions without knowing future arrivals (Tan et al., 2020), but mathematical optimization algorithms require complete information a priori. DRL is the state of the art for sequential online decisions in dynamic and uncertain contexts (Boute et al., 2022; Wu

et al., 2021a) with incomplete information because it plans future decisions by learning from experience and minimizes online computational cost (Nian et al., 2020a).

Current DRL algorithms make sequential decisions not only for each time step but also for each available task in a single time step. This comes with many drawbacks, such as the curse of dimensionality (Gu, 2003), large depth of decision, and increased uncertainty, which led to suboptimal results. One of the biggest advantages of applying DRL for resource allocation problems is minimizing online computational costs because it can be trained offline and executed online (Nian et al., 2020a). However, the sequential decision of DRL algorithms on CO problems is still inefficient in terms of computational cost. Furthermore, the curse of dimensionality (Gu, 2003) makes training DRL algorithms with sequential action selection slower and more challenging(Ota et al., 2020) for large problems. Executing the DRL algorithm sequentially for the tasks also incurs a delay in output. Moreover, DRL for arbitrary action space problems, such as CO problems, is understudied. Because the CO problem includes an arbitrary number of tasks in the online setting, the DRL algorithm must work with an arbitrary action space. While these are not negligible challenges, many existing DRL-based online resource allocation algorithms (Almasan et al., 2022; Stein et al., 2020; Chen et al., 2018) overlook this challenge by assuming that only one task arrives at each time step. A DRL-based online resource allocation algorithm by Liu et al. (2021) assumes that no new request arrives until the current flow is completed. However, real online resource allocation problems encounter arbitrary numbers of tasks. For example, Zhang et al. (2009) considered a scenario in which a cluster of computational units has to make task-offloading decisions on multiple tasks. They proposed a sequential action selection algorithm in which, at each step, the available tasks are selected sequentially until the resource constraint is exhausted. In other words, there are multiple sequential action selections in a single time step to select more tasks. Huang et al. (2018) and Huang et al. (2019b) considered multiple tasks and proposed a deep learning algorithm with a fixed number of outputs to produce binary outputs of fixed size. Other DRL-based combinatorial optimization and resource allocation algorithms (Bello et al., 2016; Sheng et al., 2020) apply the concept of sequence-to-sequence modeling using pointer neural networks (Vinyals et al., 2015) and transformer networks (Vaswani et al., 2017) of NLP to select combinatorial actions. Nevertheless, all existing multiple-action selection approaches suffer from either the curse of dimensionality or have fixed outputs. Although sequence-to-sequence modeling can be the right option when the order of the output matters, as in the traveling salesman problem (TSP)(Bello et al., 2016), they are not important when the output order does not matter. Therefore, learning to produce an ordered sequential output for non-orderly sets leads to computational costs in training and execution.

The dimensionality, execution complexity, and training complexity of DRL algorithms in online combinatorial problems can be minimized by changing the way current DRL algorithms select actions. Instead of selecting a sequence of actions one after the other,

using sequential execution, until the constraint is met, a coalition of actions can be selected simultaneously with a parallel execution. Coalition formation is a negotiation procedure that aims to resolve conflicts between entities by forming groups that can achieve mutually beneficial outcomes. Recent work by Sarkar et al. (2022) has studied coalition formation and its application to various multiagent systems. In this work, we applied it to combinatorial action selection, where CO tasks have to find the best coalition by learning different possible coalitions. We model the selection of coalitions as a single task and single coalition formation problem (Guo et al., 2020), where a group of tasks must form a single coalition to use the limited resources to maximize the combined long-term utility.

In addition to the sequential action selection approach in the output, the representation of arbitrary-sized and orderless (ASO) data in the input is another challenge in applying DRL to online resource allocation problems. That is, the information in the input to the DRL algorithm can be arbitrary in size, and its order does not matter. However, DRL algorithms, which use a standard neural network as a function approximator, have a predetermined number of inputs that accept a fixed number of inputs in a specific order (Tang and Ha, 2021). Because each input neuron of the neural network is set to accept and sense specific information from the input at a corresponding index, the DRL algorithm will consider different permutations of the same input as different information. This increases the size of the state space and slows down the training of the DRL algorithm. To benefit from DRL approaches in circumstances with such arbitrary inputs whose order does not matter, special architectural components are required (Huegle et al., 2019).

We can classify the existing techniques for handling the ASO input into two categories: neural network-based input transformations and stationary input transformations as presented in Section 2.1.3. Stationary transformations map an input of arbitrary length to a fixed-size vector before it is fed into the policy of the DRL approach. Because they can cause a collision of transformations by mapping two or more sets to the same vector, current stationary ASO transformation strategies are less expressive and can cause ambiguity in the DRL algorithm. On the other hand, neural network-based input transformations are more expressive, as the original input is fed directly to the policy. The neural network of the policy is used to learn both the input transformation and the policy. Therefore, there is no ambiguity in the DRL. For this reason, we adopt the neural network-based input transformation to deal with ASO input.

The main contributions of this work are three-fold:

- We propose the first DRL algorithm that selects an arbitrary number of actions for combinatorial decisions. This improves the convergence and execution speed of the DRL algorithms by minimizing the state space and depth of decision.

- we adopt the transformer neural network to handle ASO inputs and arbitrary action selection.
- We perform a numerical comparison using an online resource allocation problem to evaluate the convergence and complexity of sequential action selection versus coalition action selection.

This work is organized as follows. Section 3.2 reviews related work. The problem description is presented in 3.3. The proposed approaches are described in Section 3.4 and then evaluated in Section 3.5. Conclusions and extensions of the work are presented in Section 3.6.

3.2 Related Works

The review of existing techniques for dealing with ASO is presented in Section 2.1.3 and Table 2.1. In this chapter, the transformer neural network is selected as a state-of-the-art method for handling ASO inputs for the DRL algorithm with coalition action selection.

In this section, we review related work only in terms of coalition action selection and arbitrary action selection. The term *sequential action selection* refers to selecting one action at a time, while the term *coalition action selection* refers to selecting a combination of actions in each time step.

Most existing DRL algorithms employ a sequential action selection approach, in which actions are selected one at a time (Atashbar and Shi, 2022). The curse of dimensionality makes large Markov decision processes (MDPs) intractable without a guarantee of convergence (Gu, 2003) for such sequential action selection approaches. Gu (2003) did an in-depth analysis of existing state aggregation and macro-action approaches to reduce state space in large MDPs. Delarue et al. (2020) explicitly formulated the action selection problem as a mixed-integer optimization problem and used an optimization solver to find the optimal or near-optimal action for the capacitated vehicle routing problem (CVRP). This combinatorial action selection is not suitable for the online combinatorial resource allocation problem for three reasons. First, it accepts only a predefined number of *nearest_M*, inputs at a time in the CVRP. If the number of cities is greater than *nearest_M*, it considers *nearest_M* by distance. There is no reasonable way to choose a fixed number of traffic demands in our online resource allocation problem because they have a complex feature vector as presented in Section 3.3. Second, it generates a fixed number of actions, whereas the number of actions to be selected in the online combinatorial resource allocation problem is arbitrary in number, depending on the resource constraint. Third, it is not convenient to maximize long-term rewards with bootstrapping. There exist other approaches with binary decisions in resource allocation (Huang

et al., 2018, 2019b), but they work with a fixed size of inputs and outputs. Similarly to the limitation discussed for Delarue et al. (2020), they are also not convenient for long-term reward maximization. Another work by Yao et al. (2021) has proposed an approach that utilizes reversible actions to modify a current solution for combinatorial optimization problems. These actions involve flipping or swapping vertex labels and are encoded using a graph neural network to represent state-action pairs. They have mentioned that permutation invariance is a drawback of their approach. He et al. (2016b) proposed a combinatorial action selection approach for recommendation systems. The action space is designed to be a fixed set of combinations of the tasks to be recommended.

In NLP, sequence-to-sequence modeling neural networks, such as the pointer neural network (Vinyals et al., 2015) and the transformer neural network (Vaswani et al., 2017), have shown significant advances. They are also applied to combinatorial optimization problems such as CVRP (Peng et al., 2020). However, even though the encoder part of these sequence-modeling neural networks is processed in parallel, the decoder part is still sequential and gives the output sequentially.

3.3 Problem Description

The description of the problem is customized from the multiagent learning (MAL) approach for online distributed resource allocation in a network of computing clusters by Zhang et al. (2009). In MAL, there are 16 clusters as seen in Figure 3.1 with different numbers of processing units in each. Tasks with different resource requirements arrive to be processed in a computing cluster from an external environment or are routed internally from neighboring clusters. The offloading of tasks to neighboring clusters aims to maximize global utility by efficiently using resources to process tasks before their deadline. MAL focused on whether to allocate each task locally or forward it to one of the neighboring clusters, assuming a limit on the number of tasks that can be transferred due to the limited capacity of the communication links. That is, the number of tasks that can be offloaded are bound by the preset limit. Our work is complementary to MAL, where MAL decides whether tasks are processed locally or offloaded to their neighboring clusters, and our work uses a DRL algorithm to make a combinatorial decision on which of the tasks use the link given the resource constraint and which of them are deferred to the next time step. We customized the problem description of the work by Zhang et al. (2009) to consider the bandwidth constraint of a link connecting two clusters as a constraint rather than setting a limit on the number of tasks that can be offloaded. Therefore, the resource allocation in MAL is a computational resource and the resource allocation in our work is a communication resource known as traffic demand [1] (Almasan et al., 2022). Traffic demands are requests for bandwidth resources

[1] Traffic demand is a communication resource required to transfer a task.

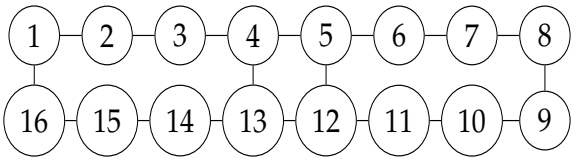


FIGURE 3.1: The network of clusters from Zhang et al. (2009)

to transfer tasks from a source cluster to a neighbor cluster.

Traffic demands are indicated by the identifier k, which ranges from 1 to the total number of traffic demands. To make the identifiers continuous from 1 to the number of traffic demands, the identifier is updated at every time step because new traffic demands can be generated and others can expire. Traffic demand is described by a feature vector that comprises the utility of allocating traffic demand V_k, the bandwidth demand of traffic demand D_k in Optical Data Units (*ODUK*) as used by Almasan et al. (2022), the time length the bandwidth is needed for traffic demand L_k in time steps, and the maximum allowable waiting time W_k in time steps before it is allocated. Furthermore, the time step at which a traffic demand is generated, indicated by G_k, is recorded for every traffic demand. A set of traffic demands are denoted by $\{k\}$, and hence the set of their V_k, D_k, L_k, and W_k are denoted by $\{V_k\}$, $\{D_k\}$, $\{L_k\}$, and $\{W_k\}$ respectively. At each time step, the total number of traffic demands generated is between 0 and k_{max}. The communication links have a resource constraint of B (in *ODUK*).

3.3.1 Formulation of the Problem

The problem is to make an online combinatorial resource allocation decision to maximize utility in the long term, as shown in the objective function in Equation (3.1a).

$$\max_{X} \quad \sum_{t=1}^{T} \sum_{k=1}^{|\{k\}|} V_{kt} \cdot X_{kt} \cdot G_{kt} \tag{3.1a}$$

$$\text{s.t.} \quad \sum_{k=1}^{|\{k\}|} D_k \cdot X_{kt} \cdot G_{kt} \leq B_t \quad \forall t \in T \tag{3.1b}$$

$$\sum_{t=1}^{T} X_{kt} \cdot G_{kt} \in \{0, L_k\} \quad \forall k \in \{k\} \tag{3.1c}$$

where $|\{k\}|$ is the number of traffic demands, B_t [2] is the link bandwidth constraint at time step t, G_{kt} binary indicator where $G_{kt} = 1$ if $G_k \leq t \leq G_k$+W_k+L_k or $G_{kt} = 0$ otherwise, $V_{kt} = \frac{V_k}{L_k}$ is an indicator that utility is for the entire length of L_k, and X_{kt} is the binary decision variable for traffic demands where $X_{kt} = 1$ if the traffic demand k is allocated at time step t and 0 otherwise. Equation (3.1c) ensures that a traffic demand

[2]The value of B_t can be less than or equal to the value of B because previous traffic demands can continue using the link for L_k steps.

is accepted for the entire length of L_k or rejected. Note that our work is designed to be complementary to MAL. It assumes that clusters have decided which tasks to allocate locally, which tasks must be forwarded to the neighboring clusters, and to which neighboring cluster. Because the task routing in the MAL decides for a one-hop distance at a time, the combinatorial resource allocation of the tasks that need to pass through a given link is decided independently of the decision on other links. Therefore, the objective function is from the perspective of a single link.

3.3.2 Deployment of the Models

In MAL, the models are deployed as distributed agents in the clusters. In our case, because every link is shared by two clusters, the agent for each link is deployed in either of the clusters at both ends of the link that has a higher number of processing units.

Notation	Description
k	Identifier for a traffic demand
V_k	Utility of traffic demand k
D_k	Bandwidth demand of traffic demand k
W_k	Maximum waiting time before allocation of traffic demand k
G_k	The time step traffic demand k is generated on
L_k	The length in time steps traffic demand k requires to use the resource
U_k	Unique distinguisher of traffic demand in the state s
B	Bandwidth contract of a communication link between two clusters
X_t	Decision variable of allocation of the traffic demands at time t
r	reward or accepting traffic demand k
$\bar{r}$	Total reward of the allocated traffic demands
T	Total time steps
t	Current time step
β	Soft weight update for the target network
e	Current episode
π	The policy of the DRL
θ	DNN parameters of the primary network
θ'	DNN parameters of the target network

TABLE 3.1: List of notations for Chapter 3

3.4 Formulating the Deep Reinforcement Learning with Coalition Action Selection

A CO problem can be formally defined as a triplet consisting of a set of CO problem instance $\{I\}$, a CO instance to a solution space mapping function S, and an objective function f that maps the solutions in $S(I)$ to real values. This definition is described

by Oren et al. (2021) and can be used to model the DRL algorithms with sequential and coalition action selection.

3.4.1 Modeling Sequential Action Selection in Combinatorial Optimization

Oren et al. (2021) modeled a sequential action selection process for an instance I using an MDP (Puterman, 1994) of T steps. At each time t, the state s_t corresponds to a partial solution and the action $a_t \in A_s$ corresponds to a feasible extension of s_t. A reward $r_{t+1} = r(s_t, a_t) = f(s_{t+1}) - f(s_t)$, transition probability $p(s_{t+1}|s_t, a_t)$, and an action distribution set by a policy $\pi(a_t|s_t)$ are also defined. This leads to a distribution of trajectories $\rho = (s_t, a_t, r_{t+1})$ for $t = 0, \ldots, T-1$, where $p(\rho) = p(s_0) \prod_{t=0}^{T-1} \pi(a_t|s_t) p(s_{t+1}|s_t, a_t)$. The Q function is defined as $Q(s_t, a_t) = \mathbb{E}\rho \left[\sum_{i=0}^{T-1} r(s_i, a_i) \middle| s_0 = s_t, a_0 = a_t \right]$. The agent's objective is to find an optimal policy $\pi^*(a|s) = \text{argmax}_a \, Q(s, a)$.

3.4.2 Modeling the Coalition Action Selection for Combinatorial and Online Resource Allocation Problems

The description of the coalition action selection problem makes minor changes to the sequential one. Partial solutions are only a subset of those in the sequential formulation. An illustrative example is provided in Table 3.2. At each time t, the state s_t corresponds to a partial solution. A feasible coalition of actions $\{a_t\} \subseteq A_s$ extends the partial solution to another partial solution. The reward of the coalition is $\bar{r}_{t+1} = r(s_t, \{a_t\}) = f(s_{t+1}) - f(s_t)$, which can be the sum of the rewards of individual actions depending on the objective function. The transition probability $p(s_{t+1}|s_t, \{a_t\})$ and an action distribution set by a policy $\pi(\{a_t\}|s_t)$ are also defined. This leads to a distribution of trajectories $\rho = (s_t, \{a_t\}, r_{t+1})$ for $t = 0, \ldots, T-1$, where $p(\rho) = p(s_0) \prod_{t=0}^{T-1} \pi(\{a_t\}|s_t) p(s_{t+1}|s_t, \{a_t\})$. The Q function is defined as $Q(s_t, \{a_t\}) = \mathbb{E}\rho \left[\sum_{i=0}^{T-1} \bar{r}(s_i, \{a_i\}) \middle| s_0 = s_t, \{a_0\} = \{a_t\} \right]$. The agent's objective is to find an optimal policy $\pi^*(a|s) = \text{argmax}_a \, Q(s, a)$. Note that the Q-function is computed by the cumulative reward but the execution and training are run in parallel for the elements of the partial solution to output the Q-value for their best coalition.

3.4.3 The Depth of Decision and the Size of the State Space of Coalition and Sequential Action Selection

DRL-based CO can be illustrated by a decision tree where the root node represents the instance and the child nodes represent the sub-problems after taking an action. An example of the structure of the decision tree is shown in Table 3.2 with an example of a 0-1 knapsack problem, $I = \{i1, i2, i3\}$ with corresponding weights $\{10, 15, 30\}$, utilities

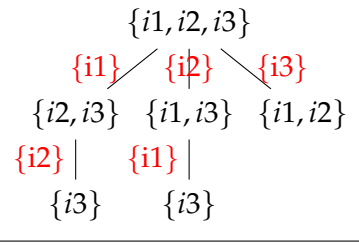
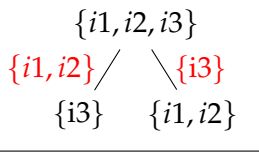
Sequential action selection	Coalition action selection
	

TABLE 3.2: Simple illustration of the depth of decision and state space of sequential and coalition action selection using 0-1 knapsack problem

$\{80, 60, 100\}$, and a knapsack capacity of 30. In the trees, the selected elements are represented by red labels at the edges. The leaf nodes represent the terminal states when the knapsack cannot add any more elements. The sequential action selection technique selects only one element at a time. On the contrary, the coalition action selection approach can select any feasible coalition of elements whose sum of weights does not exceed the capacity. In the DRL algorithm, the coalition is formed from the elements with the highest Q-values. The sequential action selection has a depth of 2 and a state space size of 5, while the coalition action selection has a depth of 1 and a state space size of 3.

3.4.4 Deep Reinforcement Learning with Coalition Action Selection

The DRL algorithm, deployed as link agents for each communication link, must make online decisions for the online combinatorial resource allocation problem shown with the objective function in Equation (3.1a). Figure 3.2 illustrates the interaction between the resource allocation environment and the DRL agent with coalition action selection. At each time step, the DRL algorithm uses a policy π, which is a transformer neural network, that takes a state s containing information about unallocated traffic demands and the resource constraint feature vector B_L [3] for the next L time step as input and outputs corresponding Q-values for the traffic demands in parallel. Then, a coalition of traffic demands is selected based on the order of the Q-values of the traffic demands, considering the resource constraint for the sum of their demands. A reward is computed from the sum of V_k of the selected traffic demands. B_L is updated after every step because once a traffic demand is allocated, it uses the link until L_k expires. Traffic demands that are not accepted at the current step remain available for decision in subsequent steps as long as their W_k has not expired. The dotted lines indicate that a copy is stored in the replay memory for training. The state, action, and reward of the DRL algorithm are as follows.

[3] B_L is the future state of B_t for L time steps, where L is the maximum possible value of L_k, because previous traffic demands occupy the link for L_k time steps.

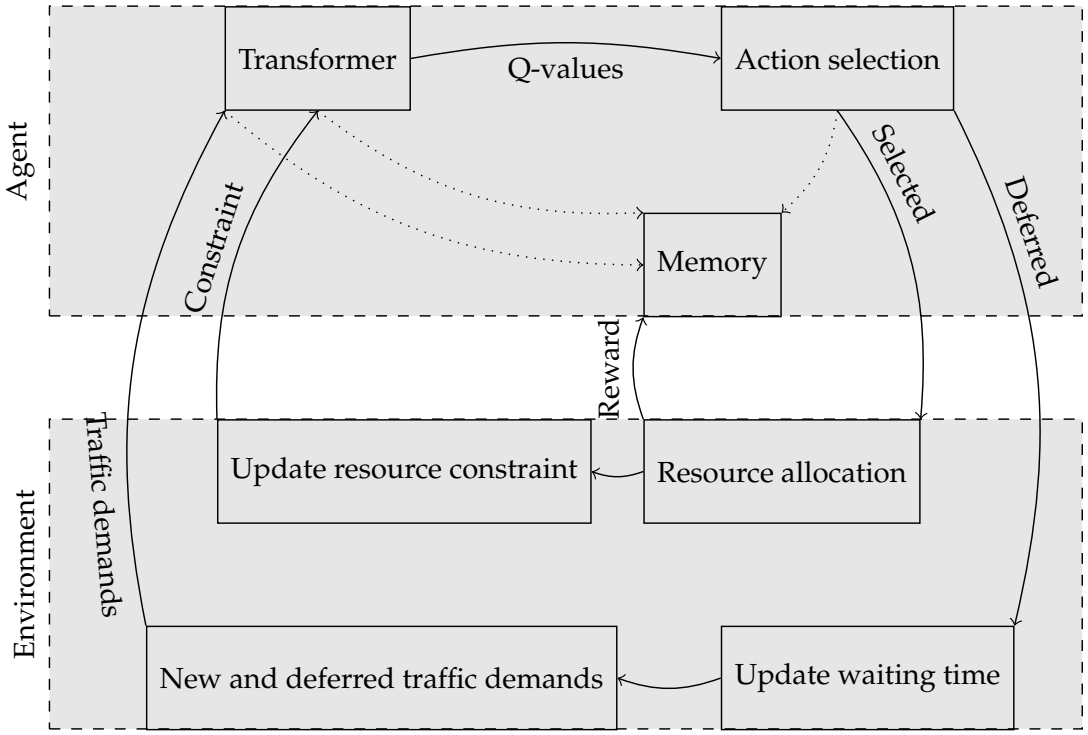


FIGURE 3.2: The interaction diagram between the transformer-based DRL agent with coalition action selection and the online combinatorial resource allocation environment

State: A state s includes the set of feature vectors $[U_k, V_k, D_k, W_k, L_k]$ of the set of traffic demands $\{k\}$ that are waiting to be allocated and the bandwidth constraint vector of the link B_L. Since U_k is used only in the selection of coalition actions, its explanation is available in the description of the action. The state is variable in size because the number of traffic demands can be arbitrarily large. To handle this arbitrary length input in the DRL algorithm, we used the encoder part of the transformer neural network (Vaswani et al., 2017) to process the ASO input simultaneously as follows.

State Using Transformer

The transformer neural network, which is also known as the transformer, is the state-of-the-art neural network-based ASO transformation as explained in Section 2.1.3. The transformer is used in a variety of NLP tasks, but it is also applied to CO problems (Peng et al., 2020). It is made up of an encoder layer and a decoder layer. The encoder processes the input sequence in parallel. The decoder generates the output sequence one by one by processing the encoder output and the previous output of the decoder. The encoder and decoder have blocks of feedforward and attention layers. The attention layer aggregates information about other elements using attention weights. If the order of the input is important, the positional encoder (Vaswani et al., 2017), but we do not need it in our ASO input because the ASO input assumes independent tasks. The transformer allows the input elements (traffic demands, in our case) to compute the complex relationship among themselves in parallel using an attention neural network.

They can handle arbitrary lengths of input. Therefore, we used the encoder part of the transformer network to simultaneously output the Q value of the input elements from their hidden state. In NLP, the input sequence is embedded in a dictionary. However, since the input is numerical data in CO, the embedding part is replaced by a feedforward neural network (Peng et al., 2020).

Therefore, the state transformation is applied by directly entering the set of feature vectors $[U_k, V_k, D_k, W_k, L_k]$ of traffic demands $\{k\}$ and the resource constraint B_L into the encoder as s= $\{B_L, \{k\}\}$. B_L is padded with B to make its feature vector equal to the length of the feature vector of a traffic demand.

Action: The action space $\{a\}$ is the set of traffic demands $\{k\}$. The Q-values of the traffic demands are outputted from the last layer of the encoder-only transformer. Then, the traffic demands with higher demand than the capacity of the link are masked, and the action selection starts for the rest in decreasing order of their Q-value until the resource constraint is exhausted. If a traffic demand has a higher D_L than the resource constraint, the algorithm skips it and checks the next one.

Distinguisher in Coalition Action Selection

DRL outputs the same Q-value for the same input. Therefore, it will be challenging for coalition selection to learn to put the same traffic demands on different coalitions. The U_k, which is 1 by default for all $\{k\}$, is included to distinguish the same traffic demands in the coalition action selection. If two or more traffic demands have the same feature vector, they are distinguished by indexing them with increasing U_k. The U_k is updated at each time step.

Reward: The set of selected actions receives a joint reward $\bar{r}$, which is the sum of V_k of the selected traffic demands.

Next State: Taking a set of actions $\{a\}$ in state s of the problem transforms the state into a new state s' with reward $\bar{r}$. The selected traffic demands are removed from the set of unallocated demands. The unallocated traffic demands and newly generated traffic demands form the traffic demands of the next state. The resource constraint vector B_L is also updated by subtracting the traffic demands $\{D_k\}$ of the selected actions and also by releasing the occupied resource of traffic demands whose L_k has expired.

Our DRL algorithm with coalition action selection is in Algorithm 1. First, it initializes hyperparameters. Then it runs for E episodes and T steps for every episode. Training is performed at the end of the episode (lines 36 to 42). For the iterations of the steps, it starts by generating initial traffic demands k_{max}, as seen in line 4. After the first step, it generates any number of traffic demands between 0 and k_{max} per step, as seen in line 31. Lines 9 to 14 show exploration and exploitation. Traffic demands are randomly shuffled for exploration. When exploiting, the traffic demands and the resource constraint are fed to the transformer to simultaneously output Q-values for all traffic demands. Then

they are sorted so that they are selected accordingly, as seen in lines 18 to 27. Action selection is carried out by iterating on the sorted $\{k\}$ using the index a. In line 19, it checks if the demand D_a of the traffic demand at index a of $\{k\}$ is not greater than the resource constraint. If not, it appends the identifier of the traffic demand to the selected list and adds its utility to the reward, as seen in lines 20 and 21. Then update the resource constraint because traffic demand will use the resource for L_a time steps. This process continues until no more traffic demand can be accepted. The reward is used to update the Q-value of the selected traffic demands in the training. During training, only the selected actions update their Q-values with the target Q-value computed from the shared reward and the best Q-value of the next state, as seen in line 41. Because each action can be explored with different coalitions of actions at different time steps, the DRL algorithm gradually finds the optimal coalition using their Q-values during constrained action selection.

For better training efficiency, we use double Q-learning (Van Hasselt et al., 2016) and prioritized experience replay (Schaul et al., 2016). We use the decaying exploration-exploitation probability ϵ which starts at 1 and decays by subtracting $\frac{\epsilon}{5000}$ in every episode.

3.5 Experimental Evaluation

In this section, we experimentally evaluate whether coalition action selection, which benefits from reducing the depth of the decision, the state space, and the action space, yields superior performance and lower complexity than sequential selection. We implemented our algorithm for the online resource allocation problem to assess the validity of the hypothesis. First, we introduce the offline optimal.

3.5.1 Offline Optimal using Integer Programming

To evaluate the performance of the proposed algorithms, we used integer programming (IP) as an offline optimal. We compared the utility of the coalition and sequential action selection methods over an episode as percentages with the offline optimal. IP is unrealistic for online resource allocation, as it assumes that all information about demand is available in advance. Therefore, we store the traffic demands generated during the time steps of the episode and run the IP at the end for the objective function shown in Equation (3.1a).

Algorithm 1 Transformer Neural Network-based DRL Algorithm with Coalition Action Selection

Initialize parameters: primary transformer parameters θ) , target transformer parameters $\theta' = \theta$), discount factor $\gamma = 0.99$, ϵ-greedy $\epsilon = 1$, replay memory M = [], minibatch b = [], start and maximum episode ($e = 1$, $E = 50,000$), number of time steps $T = 10$

while $e \leq E$ **do**
  $\epsilon = \epsilon - \frac{\epsilon}{5000}$
  Time step $t = 1$
  Initialize the set of traffic demands $\{k\}$ with k_{max} number of random initial traffic demands
  Initialize B for L_{max} time steps B_L:$B_l = B$ for $0 \leq l \leq L_{max}$
  Pad B_L with B to make it same length with the feature vector of k
  while $t \leq T$ **do**
    Compute state $s = \{\{k\}, B_L\}$
    if rand $\leq \epsilon$ **then**
      Shuffle $\{k\}$ randomly and form new lists V, D , L, W
    else
      Get Q-value $Q = Q\left(s \mid \theta\right)$ in parallel for s
      Sort $\{k\}$ in descending Q and form new lists V, D , L, W
    end if
    Reward r = 0
    Selected = []
    a = 0
    while $a \leq |\{k\}|$ **do**
      if $D_a \leq B_t$ **then**
        Selected = append(selected,a)
        r = r + V_a
        for i = 0 to L_a **do**
          $B_i = B_i - D_a$
        end for
      end if
      $a = a$+ 1
    end while
    Exclude the selected traffic demands from $\{k\}$
    Decrement $\{L_k\}$ and $\{W_k\}$ of the traffic demands
    Free occupied resources from B_L for all $L_k \leq 0$
    Generate new traffic demands of size between 0 and k_{max}, and append to $\{k\}$
    Compute next state s'
    Store the experience (s, $selected$, r, s') to M
    Increment t
  end while
  Sample a minibatch of (s,$selected$,r,s') from M to b
  Get Q-value $Q' = Q\left(s' \mid \theta'\right)$ for the traffic demands at s'
  Mask the Q-values of the infeasible ($W_k > B$) traffic demands from Q′ and find maximum Q-value $maxQ_i$ = max(Q'_i) $\forall i \in b$
  Compute target Q-values $y_i = r_i$+$\gamma maxQ_i$ $\forall i \in b$
  Get current Q-values $currQ = Q\left(s \mid \theta\right)$ for the traffic demands
  Update the DQN by minimizing $Loss = \frac{1}{|b|} \sum_{i \in b} \left(y_i - currQ_i\left(s_i\left(selected_i\right)\right)\right)^2$
  Update the targets: $\theta' \leftarrow \theta$
  Increment e
end while

3.5.2 Benchmark Selection

Arbitrary action selection is understudied in DRL. As described in related work, existing work overlooks the challenge of arbitrary action selection in DRL by assuming that only one task arrives at a time. Although some techniques select a fixed number of outputs at a time, as discussed in related work for CVRP (Delarue et al., 2020) and resource allocation (Huang et al., 2018, 2019b), they are not suitable for comparison for the mentioned reasons. Therefore, we chose a sequential action selection approach as in Zhang et al. (2009) and the offline optimal with IP as the lower bound and the upper bound benchmarks, respectively.

3.5.3 Experimental Setup

Because our work is complementary to MAL as explained in Section 3.3, we generate a random number of traffic demands in the range of 0 and k_{max} from a uniform distribution at every time step to resemble the traffic demands that arrive at a cluster externally or offloaded from neighboring clusters. First, we run the experiment with a k_{max} value of 10 to analyze convergence and complexity, and then run the experiment for $k_{max} \in \{2, 5, 10, 15, 20\}$, without changing other settings, to evaluate performance with various traffic demand arrival rates. The feature vectors of the traffic demands are generated from a uniform distribution between 1 and $V_{max} = 5$, $L_{max} = 1$, and $W_{max} = 3$, inclusive. Similarly to the work by Almasan et al. (2022), we consider three *ODUK* types for the values of $\{D_k\}$ with $\{ODU2, ODU3, \text{and } ODU4\}$, whose bandwidth requirements are expressed in terms of multiples of *ODU0* signals. Therefore, the values of $\{D_k\}$ are selected as a random choice of {8,32,64} *ODU0* units, and the resource constraint B is 100 (*ODU0*). The time step T is 10.

The hyperparameters of our DRL algorithm are configured as follows. The coalition action selection has five inputs including the distinguisher, and the sequential action selection has four inputs. The numerical inputs are embedded in a single-layer neural network with 8 outputs. This is followed by 6 blocks of feedforward neural network and multihead attention layers. The feedforward neural networks have 32 neurons each. We used a multihead attention value of 8. The final feedforward neural network layer has one output, which will be the Q-value of the corresponding traffic demand at the input. Note that the transformer processes the traffic demands in parallel to compute the Q-values. We used a discount factor of 0.99, a learning rate of 0.001, a replay memory of size 10,000 which stores the transitions in a first-in-first-out order, and a minibatch size of 64. The mentioned number of encoder blocks and their number of neurons are selected because they led to superior convergence after exhaustive trial-and-error experiments with various choices. We ran the experiments for 40 runs. The experiments are implemented with Pytorch. To ensure reproducibility, the experiment

environment is initialized with a seed value of 0. The episodes are independent. All experiments are initialized with 50,000 episodes and run for 24 hours.

3.5.4 Experimental Comparison

We conducted experiments to evaluate convergence and performance as follows.

3.5.4.1 Visualization

We choose a line graph to present the performance, where the x-axis represents the episodes and the y-axis represents the total reward of an episode. The total reward for an episode is the sum of the rewards gained in each of the steps of the episode. Because the number of episodes is very large, the visualization appears dense, as seen in Figure 3.3 (A) when the y-axis is plotted for each episode individually. To smooth the lines, we plot the results for an average of a moving window of 50 and 1000 episodes, as seen in Figure3.3 (B) and Figure 3.3 (C) respectively. The plot of an average moving window of 50 in a given episode is the average reward of the recent 50 episodes. If the current episode number is less than 50, it averages for all of them. The subplot with a moving window of 1000 visualizes the difference between the performance of the two algorithms.

Note that the results are plotted with a 95% confidence interval of 40 runs. That is, the rewards on the y-axis are averaged over 40 experiments, and the shaded area represents the 95% confidence interval.

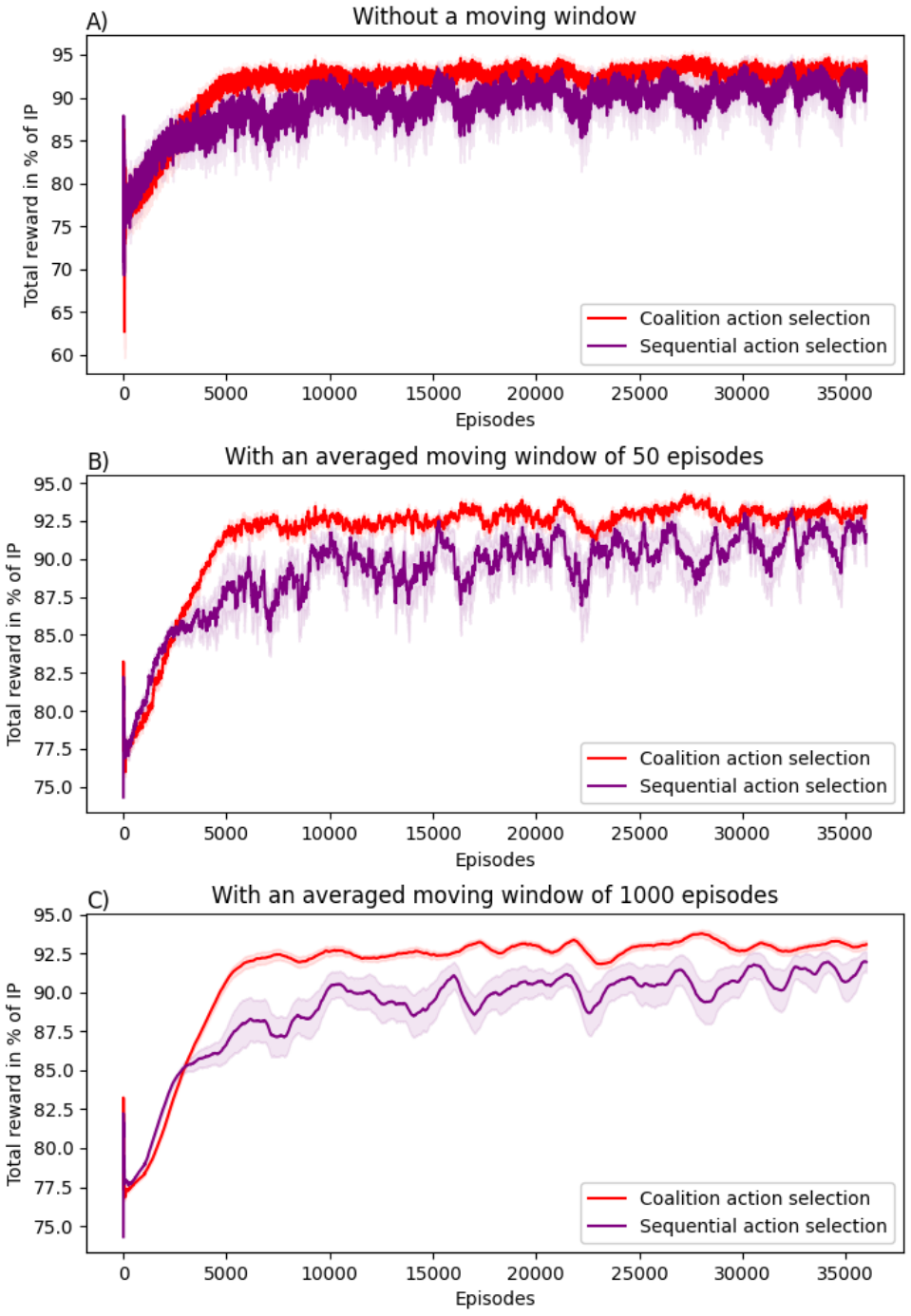


FIGURE 3.3: The visualization of the performance: A) without a moving window; B) with a moving window of 50; C) with a moving window of 1000

3.5.4.2 Convergence

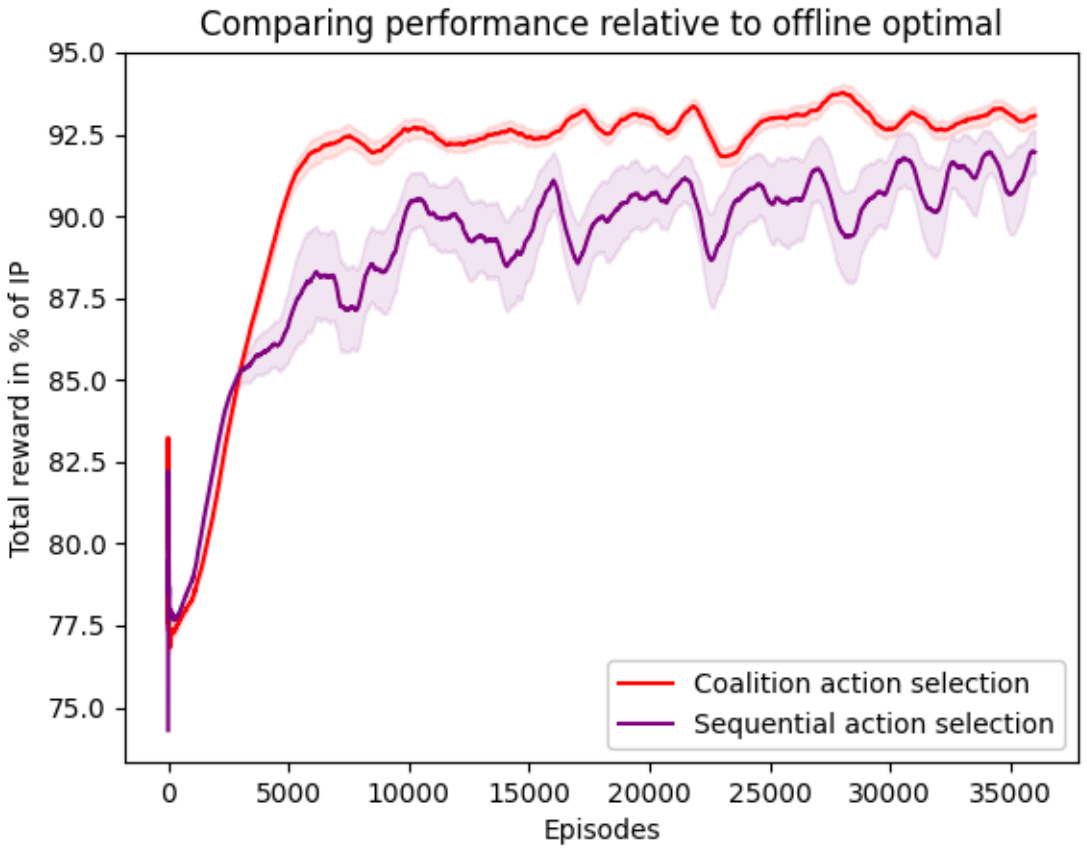


FIGURE 3.4: Performance of the coalition action selection and the sequential action selections with episodic training (the DRL algorithm is trained at the end of each episode).

First, we evaluated the convergence of the performance of the two algorithms and their complexity. To evaluate performance, we compared the performance of coalition action selection and sequential action selection as a percentage of the offline optimal in each episode. To smooth the curves, the results in Figure 3.4 are plotted for an averaged moving window of 1000 episodes. Figure 3.4 shows that the coalition action selection approach converges faster and is superior with an average gap of 2% over sequential action selection. Because the 40 runs end with varying numbers of episodes during the 24-hour training period, we normalize them to the minimum number of episodes to ensure that the 95% confidence interval is computed over an equal number of episodes for Figures 3.4.

Training of the DRL algorithm for the result in Figure 3.4 is performed at the end of each episode. We also explore the performance of the algorithms when trained at every step of the episodes, as seen in Figure 3.5, which performed lower due to overfitting. Training in a stepwise manner leads to overfitting to the data collected from the early

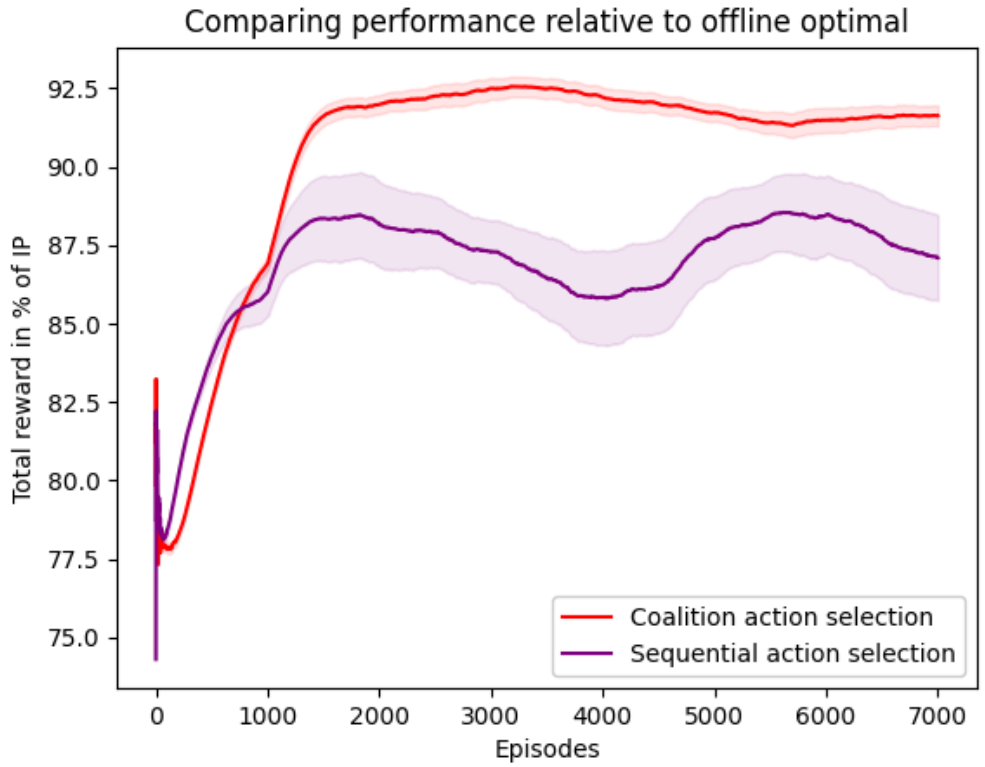


FIGURE 3.5: Performance of the coalition action selection and the sequential action selection with step-wise training (the DRL algorithm is trained at every step).

episodes of the experiment, while training at the end of each episode provided better generalization.

3.5.4.3 Comparison of Complexity of Coalition Action Selection and Sequential Action Selection

To compare the computational cost of executing the sequential action selection and coalition action selection approaches, we present the number of executions (or iterations of traffic demands) and the CPU time as shown in Figure 3.6. The number of executions and the CPU time of the box plots are generated from the number of episodes in Figures 3.4. The number of executions is a count of the number of computing Q-values for the traffic demands. The CPU time is the sum of the fraction of seconds each algorithm spent running the DRL algorithm to select actions. The time the algorithm spends training is not considered. As seen in Figure 3.6 (A), the median number of executions for episodes of the coalition action selection approach is 114 and the number of executions for 50% of the episodes ranges between 110 and 117, while episodes of sequential action selection have a median of 479 executions, and the number of executions for 50% of them ranges between 455 and 499. The sequential action selection

performs more executions because it selects one action at a time, resulting in some traffic demands to be executed again in the next decision. On the other hand, coalition action selection has a smaller number of executions because it selects multiple actions after a single parallel execution. Figure 3.6 (B) shows that the CPU time spent executing the DRL in sequential action selection and coalition action selection is proportional to the number of executions in Figure 3.6 (A). The number of executions and CPU time is averaged over the 40 runs episodic before being used for the box plots.

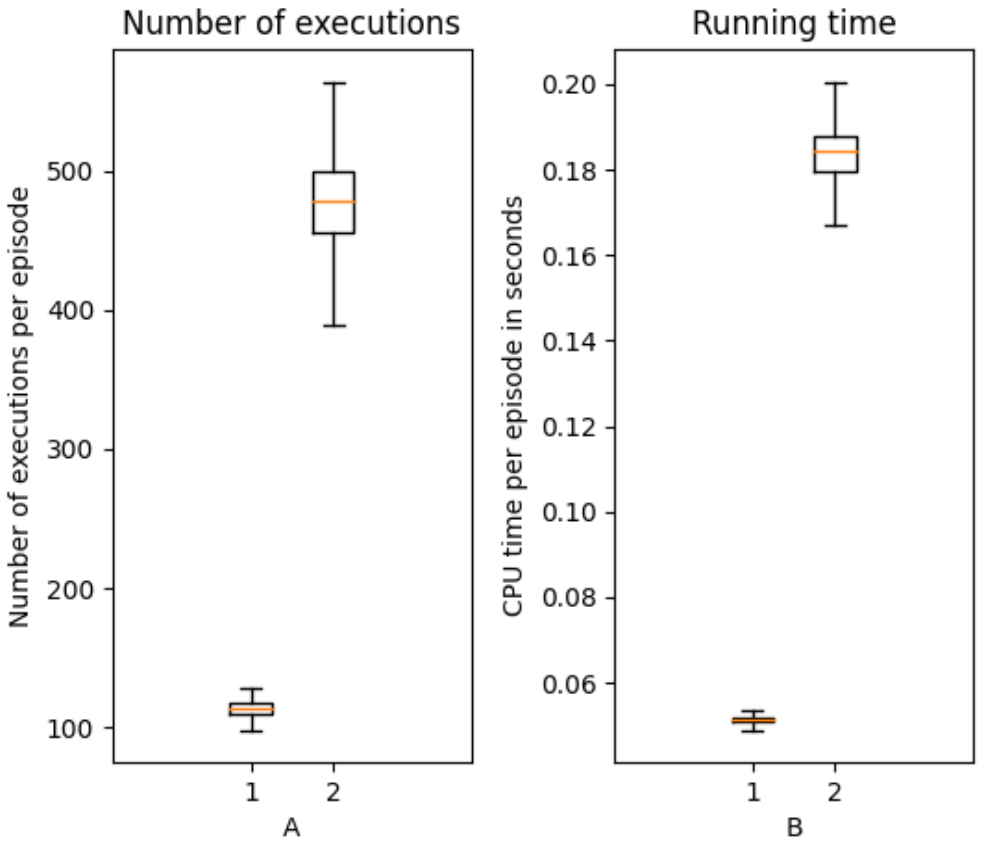


FIGURE 3.6: A. The complexity using the number of executions for (1) coalition action selection and (2) sequential action selection. B. The execution in CPU time for (1) coalition action selection and (2) sequential action selection

3.5.4.4 Comparison of Performance and Execution Complexity with Various Problem Sizes

Finally, we repeat the experiment for traffic demand arrival rates of k_{max} = 2, k_{max} = 5, k_{max} = 15, and k_{max} = 20 to compare the algorithms with various sizes of the online combinatorial resource allocation problem. Note that the experiment discussed above is for k_{max} = 10. The results are plotted for the maximum convergence of the best run of 40 runs for a moving window of 5000 episodes, as seen in Figure 3.7. For example,

the maximum convergence for the experiment with $k_{max} = 10$ shown in Figure 3.4 with a moving window of 1000 episodes is 93.777% in episode number 28,036. However, Figure 3.4 is plotted for the average of 40 runs, while Figure 3.7 is plotted for the best run of the 40 runs. The reason is that we were unable to compute an average run of 40 runs for k_{max} values of 15 and 20 because some runs failed due to limitations of the academic license of the Gurobi optimizer[4]. Almost all runs failed for k_{max} greater than 20. Note that, with a maximum arrival rate k_{max} of 20, a maximum waiting time of W_{max} of 3 for each traffic demand, and for the time steps per episode T of 10, there are $2^{20*3*10}$ combinations of solutions per episode in the worst case for the Gurobi optimizer. The license does not affect the DRL algorithm, but the offline optimal.

For episodic training, the coalition action selection approach has shown superior performance, with a convergence of around 95% to offline optimal in the best runs of different problem sizes, but the performance of sequential action selection algorithms decreases, up to 92%, as the size of the problem increases. Note that the two algorithms perform almost the same for very small arrival rates because all traffic demands can be accepted without exhausting the resource constraint. For $k_{max} = 10$, Figure 3.4 shows that the coalition action selection converges faster, but Figure 3.7 shows that they are almost the same because it is plotted only for the maximum convergence. The figure also shows that the performance gap between coalition action selection and sequential action selection increases in stepwise training due to overfitting, as discussed above.

To compare computational complexity, we present the number of executions for each corresponding run that led to the peak convergence in performance for episodic training in Figure 3.7. Unlike Figure 3.6, which plotted the number of executions as box plots in the run episodes, Figure 3.7 plots the mean number of executions in the run episodes because we have to plot a scalar value for every k_{max} on the x-axis. The result shows that the coalition action selection has a lower complexity than the sequential action selection, in terms of the number of executions and CPU time.

[4] https://www.gurobi.com/

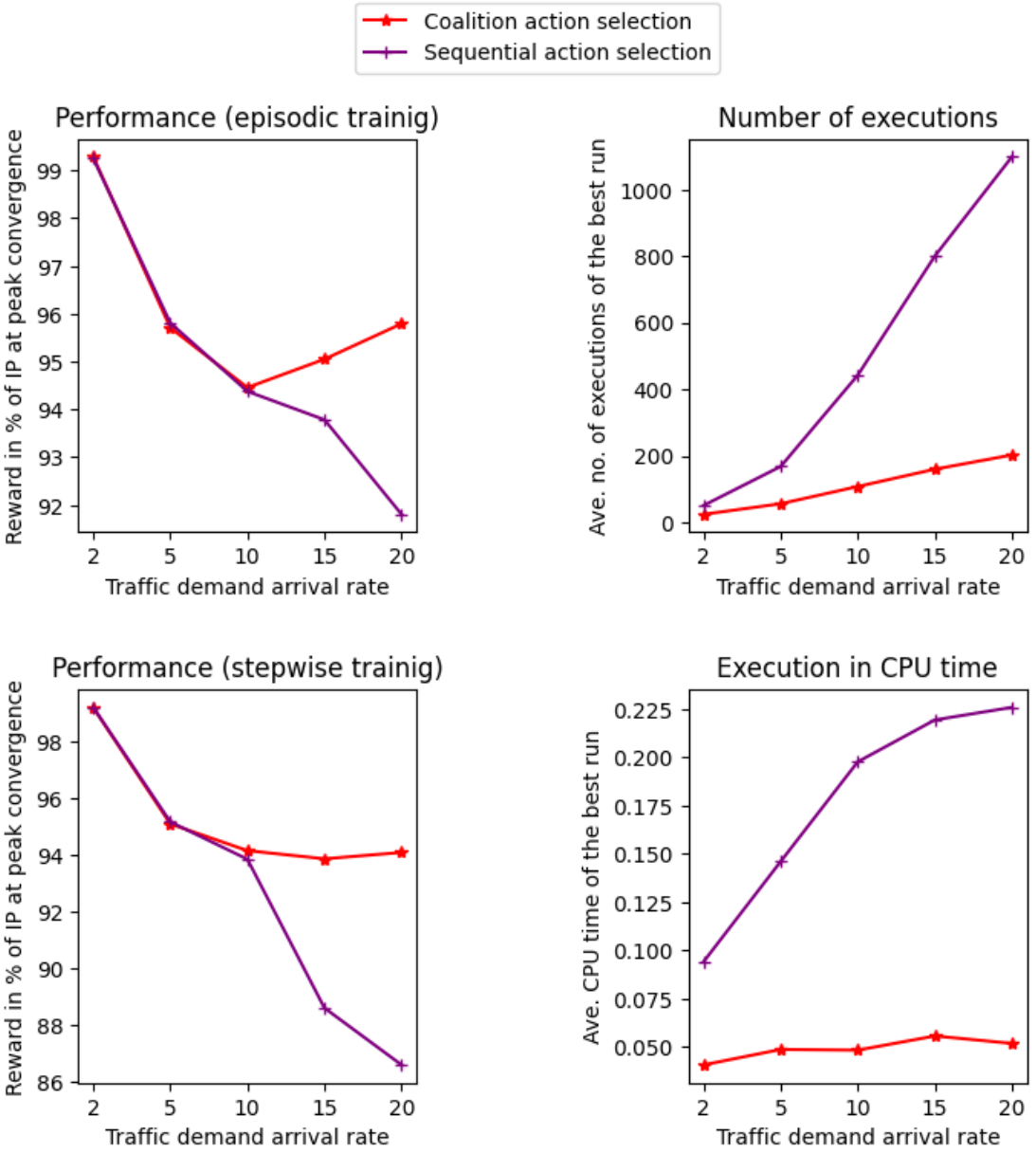


FIGURE 3.7: The comparison of the coalition action selection and sequential action selection in terms of performance, number of executions, and running time in CPU time for different numbers of traffic demand arrival rates. The complexity plotted in terms of the number of executions and CPU time are averaged over the episodes of the best-run experiment that led to maximum values in the performance shown for the episodic training.

3.6 Conclusion

We propose DRL with coalition action selection for online combinatorial resource allocation with arbitrary action space and experimentally demonstrate that it gives better convergence and lower complexity than sequential action selection. DRL with coalition

action provides better performance than sequential action selection because it minimizes the number of iterations, the state space, and the uncertainty of the problem by producing the Q-values for all elements in the input in parallel, unlike sequential action selection which makes longer iterations by selecting one action at a time and updating the state. The actions of the coalition action selection are coordinated by sharing their reward in the training.

The coalition action selection technique is extended to the combinatorial action selection of the master agent in the CCM_MADRL algorithm presented in Chapter 5.

Chapter 4

Deep Reinforcement Learning with Computationally Efficient Stationary ASO Input Transformation for Online Combinatorial Resource Allocation

4.1 Introduction

Although, as described in Chapter 3, neural network-based input transformation techniques are more expressive because the original input is directly fed to the policy, learning the fixed-size vector from the input with an arbitrary length increases computational complexity and slows the convergence. This is because the policy's neural network learns both the input transformation and the policy. A review of neural network-based and stationary ASO input transformation techniques is provided in Section 2.1.3. GNN and attention-based transformers are popular neural network-based for dealing with ASO input. Their suitability for handling ASO input is analyzed in Table 2.1. GNN is efficient when there is a topological relationship between the inputs. The transformer is efficient if there is a contextual relationship between the inputs. The transformer can be considered a special case of GNN with a fully connected graph (Wu et al., 2021b; Veličković, 2023). Therefore, the transformer is more suitable than GNN for handling ASO in resource allocation for set inputs, such as in combinatorial optimization problems that do not have a graph-structured relationship. However, although transformers are efficient in NLP, they are not as such for numeric ASO inputs for the following reasons.

- Slows the convergence because the DRL has to learn both the policy and the transformation.
- The execution speed is slow because there is a huge dependence on parallel execution, where every element of the input has to compute a value using the attention mechanism for every other element at every layer of the transformer.
- When the ASO inputs are independent, there is no complex relationship to learn. The transformer incurs computational complexity, making dependent parallel computations over a series of layers for independent ASO inputs.

On the other hand, because they can transform various input sets into the same context vector, the current stationary ASO input transformation strategies are less expressive and can cause ambiguity in the DRL algorithm. This motivated us to develop computationally efficient stationary ASO input transformations to reduce computational cost and improve efficiency. The transformation of the ASO input and the learning of the policy are separated. First, the ASO input is transformed into a fixed-size and permutation-invariant vector using stationary mathematical equations. Then, the per-action DQN processes the items independently but shares the transformed vector as a common state among them.

The main contributions of this work are two-fold:

- We propose a novel computationally efficient stationary ASO input transformation that transforms a set of inputs into a fixed-size vector, which reduces the learning complexity of DRL algorithms. We call it stationary because an ASO input is always transformed into the same vector, unlike neural network-based transformations, which transform the input into different outputs as their weight changes.
- We conducted a numerical comparison with the transformer-based approach in Chapter 3 to evaluate the convergence speed of the proposed stationary ASO input transformation.

This chapter is organized as follows. The proposed approaches are described in Section 4.2 and then evaluated in Section 4.4. Conclusions and future work are presented in Section 4.5.

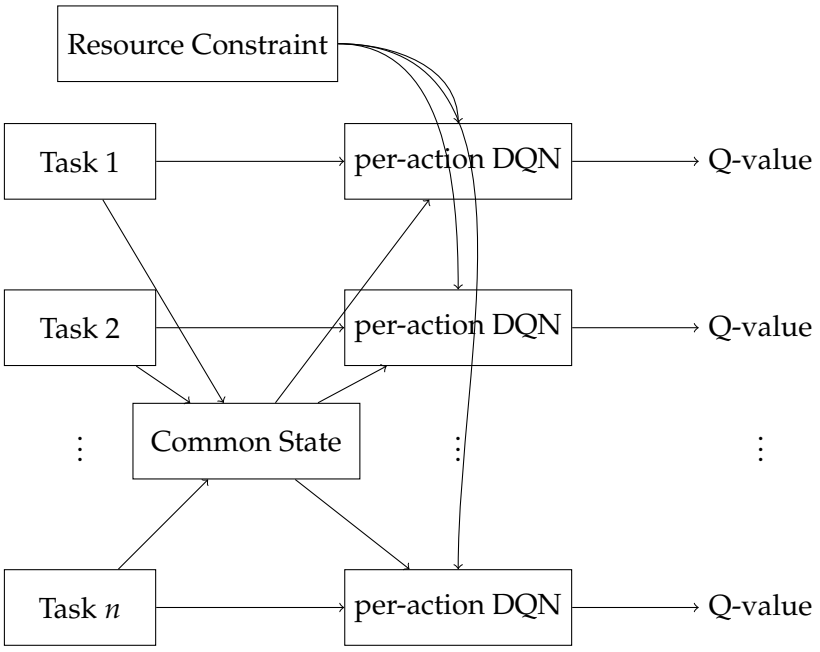


FIGURE 4.1: The architecture of the per-action DQN with stationary ASO input transformation (ASO Per-action DQN)

4.2 The Proposed Stationary ASO Input Transformation for the Deep Reinforcement Learning with Coalition Action Selection for Online Resource Allocation

Because the benchmark for the proposed stationary ASO input transformation is the transformer-based approach in Chapter 3, the same description of the problem formulation is also used for the online resource allocation. The architecture of the DRL with coalition action selection using per-action DQN and stationary ASO input transformation is shown in Figure 4.1. First, a common state is computed for traffic demands. Traffic demands are then processed by the same per-action DQN model independently but have a common state computed by the stationary ASO input transformation. Finally, a coalition action selection technique selects traffic demands in descending order of their Q values until the resource constraint is reached.

The interaction diagram of the DRL algorithm with the coalition action selection that uses per-action DQN and stationary ASO input transformation with the resource allocation environment is presented in Figure 4.2. It is similar to the interaction diagram of the transformer-based benchmark in Figure 3.2, except that the transformer network of the DRL agent is replaced by the architecture of the per-action DQN with stationary ASO input transformation in Figure 4.1. Because the way they transform the ASO input is different, they also store their experiences differently. The transformer stores the original tasks as states in the replay memory, but the per-action DQN with ASO transformation computes a common state from the tasks and stores it in the replay

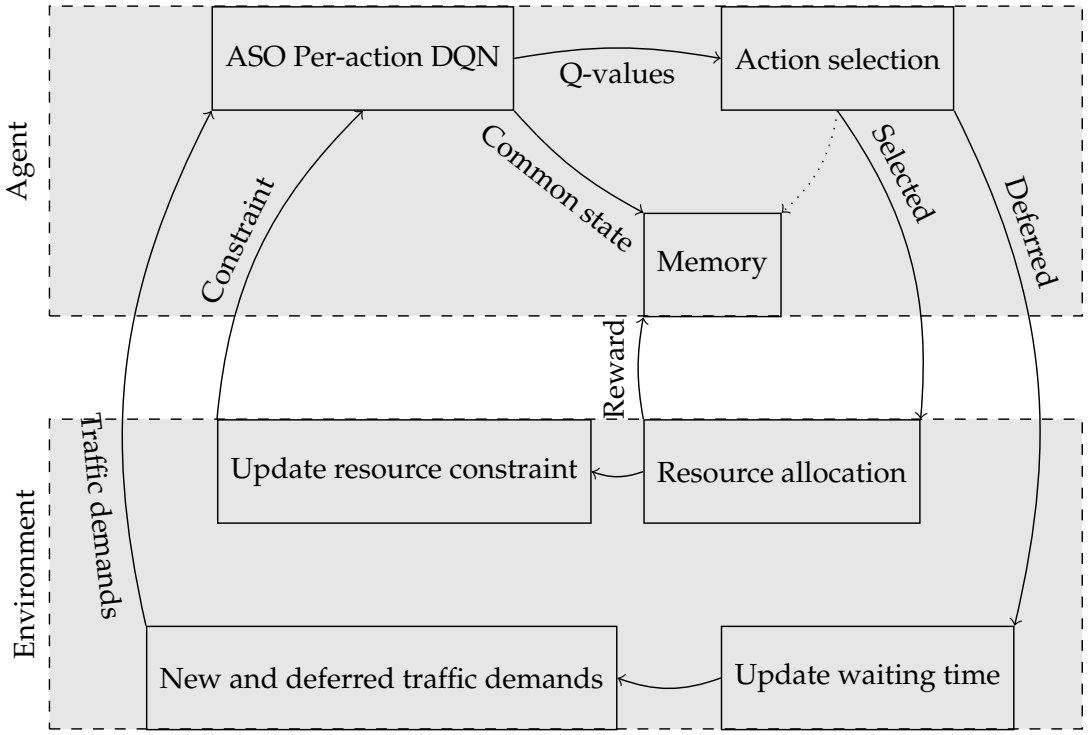


FIGURE 4.2: The interaction diagram between the per-action DQN and stationary ASO input transformation-based DRL agent with coalition action selection and the online combinatorial resource allocation environment

memory. The tasks in the replay memory of the transformer will be processed by the transformer at every training to be converted to a fixed vector, whereas the per-action DQN with stationary ASO input transformation computes the common state once and reuses it. Compared to neural network-based ASO input transformation techniques, which store all original input in replay memory, the DRL algorithm with stationary ASO input transformation stores only the selected tasks, the common state vector, and the resource constraint in replay memory.

The state, action, and reward transitions of the DRL algorithm are described in the following.

State: The state s includes the set of feature vectors $[U_k, V_k, D_k, W_k, L_k]$ of the traffic demands $\{k\}$ that are waiting to be allocated and the bandwidth constraint vector of the link B_L. Since U_k is used only in the selection of coalition actions, its explanation is available in the description of the action.

The state information is variable in size because the number of traffic demands can be arbitrarily large. However, the standard neural network of a DRL algorithm has a fixed number of inputs. The arbitrary-length state representation must, therefore, be mapped to a fixed-size state representation using a transformation mechanism. In this section, we describe the mapping of the state to a fixed vector using the proposed stationary

ASO input transformation function, which will be used as input to the per-action-DQN with stationary ASO input transformation to learn the policy.

4.2.1 State Representation Using the Proposed Stationary ASO Input Transformation

The proposed stationary ASO input transformation function is a vector of size 5 as shown in Equation (4.1).

$$ST(A) = [\,\eta(A),\ \mu(A),\ \sigma(A),\ \sigma(A + A \circ A),\ \prod_{i|A_i \neq 0} A_i] \tag{4.1}$$

where A is an ASO input. $A \circ A$, η, μ, σ are the Hadamard-product, size, mean, and variance of the ASO input respectively. $\prod_{i \neq 0} A_i$ computes the product of the ASO input (ignoring zeros). The theoretical derivation of the stationary ASO input transformation function is provided in the supplementary material.

Equation (4.1) gives the same result for any permutation of the input A. Therefore, it maps a set of scalar values to a fixed-size vector. However, the feature vectors of the traffic demands are vectors. Therefore, the equation must be applied element-wise, i.e. $ST(V_k, D_k, L_k, W_k) = [ST(\{V_k\}), ST(\{D_k\}), ST(\{L_k\}), ST(\{W_k\})]$ to the feature vectors of traffic demands. Note that U_k is not included in the state, but will be included in actions.

Using the element-wise transformation can cause collisions of different inputs to the same output because it does not keep the correspondence between elements of the same feature vector. The element-wise transformation cannot distinguish a set of two traffic demands whose feature vectors, $[U_k, V_k, D_k, W_k, L_k]$, are $\{[1,1,3,4,2], [1,2,5,2,4]\}$ from other traffic demands with feature vectors $\{[1,1,3,2,2], [1,2,5,4,4]\}$. To address this, before applying the element-wise transformation, the traffic demands are sorted in lexicographical order of their feature vectors. That is, $\{k\} = \{[U_k, V_k, D_k, W_k, L_k]\}$ will transform into $\{k'\} = \{[U_k^{'}, V_k^{'}, D_k^{'}, W_k^{'}, L_k^{'}]\} = LexicographicOrder(\{k\}) = LexicographicOrder(\{[U_k, V_k, D_k, W_k, L_k]\})$. After the lexicographic order, every traffic demand in $\{k'\}$ is updated by Equation(4.2) after which it will undergo element-wise transformation using Equation (4.1).

$$E_k^{'} = E_k^{'} + \frac{k}{K} E_k^{'} \qquad \forall E_k^{'} \in [V_k^{'}, D_k^{'}, W_k^{'}, L_k^{'}] \tag{4.2}$$

where k is the identifier of the traffic demand (or index) after the lexical order and $K = |\{k\}|$. The lexicographical order helps to avoid different permutations of the same set of traffic demands, whereas the multiplication by the index distinguishes elements of the feature vector from elements of other feature vectors.

The state of the DRL algorithm also includes the bandwidth limit of the link B for the next L_{max} time steps, where L_{max} is the maximum possible L_k that a traffic demand can have. This traffic demand constraint in the time steps L_{max} is denoted by a bandwidth constraint vector B_L. Initially, all elements of the vector B_L are assigned with the value of B. Therefore, by combining ST and B_L, we obtain the state in a time step using Equation (4.3).

$$s = ST(V_k, D_k, L_k, W_k) = [B_L, ST(E')] \tag{4.3}$$

where E' is Equation (4.2).

Action: The action space $\{a\}$ is the set of traffic demands $\{k\}$. For the per-action DQN (He et al., 2015), each of the feature vectors of the traffic demands will be embedded in the state s and fed into the DQN, giving an output Q-value for each of the traffic demands, that is, the state and the action are inputs into the per-action DQN, $Q[a] = Q(s, a \mid \theta)$. After computing the Q-values of the traffic demands, the action selector sorts the actions by their Q-values and starts to select the traffic demands starting from the highest Q-value until there is no more traffic demand that can be accepted with the remaining capacity of the link in the current time step. If a traffic demand has a higher D_L than the resource constraint of the link, the algorithm skips it and checks the next one.

Distinguisher in Coalition Action Selection

DRL outputs the same Q value for the same input. Therefore, it will be challenging for coalition selection to learn to put the same traffic demands on different coalitions. The U_k, which is 1 by default for all $\{k\}$, is included to distinguish the same traffic demands in the coalition action selection. If two or more traffic demands have the same feature vector, they are distinguished by indexing them with increasing U_k. The U_k is updated at each time step.

Reward: The set of selected actions receives a joint reward $\bar{r}$, which is the sum of V_k of the selected traffic demands.

Next state: Taking a set of actions $\{a\}$ in state s of the problem transforms the state into a new state s' with reward $\bar{r}$. The selected traffic demands are removed from the set of unallocated traffic demands. The unallocated traffic demands and newly generated traffic demands form the traffic demands of the next state. The resource constraint vector B_L is also updated by subtracting the traffic demands $\{D_k\}$ of the selected actions and also by releasing the occupied resource of traffic demands whose L_k has expired. Finally, it is applied to the stationary ASO input transformation.

Action selection	Best case	Worst case
coalition	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Sequential	$\mathcal{O}(n)$	$\mathcal{O}(nx)$

TABLE 4.1: Complexity of calling stationary ASO input transformation

The DRL algorithm with the coalition action selection using the stationary ASO input transformation is in Algorithm 2. The difference with Algorithm 1 is in the computation of the Q-values. Algorithm 2 computes the Q-values using per-action DQN as $Q(s, k \mid \theta)$ whereas Algorithm 1 computes it using the transformer as $Q(s \mid \theta)$. Note that the state s is different in the two algorithms.

For better training efficiency, we used double Q-learning (Van Hasselt et al., 2016) and prioritized experience replay (Schaul et al., 2016). We use the decaying exploration-exploitation probability ϵ that starts at 1 and decays by ϵ - $\epsilon/5000$ in every episode to 0 after which the DRL takes advantage of its trained model.

4.2.2 Complexity Analysis of the Stationary ASO Input Transformation and the Execution of the Per-action DQN with Stationary ASO Input Transformation

In this section, we analyze the best-case and worst-case complexity of the coalition and sequential action selection in terms of both computing the state and executing the per-action DQN. The analysis of complexity between the coalition action selection and the sequential on the transformer is not included, as it had similar differences but higher complexity than per-action DQN as the attention weights are computed at every layer of the neural network. We compared the complexity in terms of the number of executions of the model, not the complexity of the model itself, since it is similar between sequential and simultaneous.

Using n to denote the T time steps of an episode and x to denote the number of unallocated traffic demands $|\{k\}|$ at a time step, we analyzed the complexity of computing the state and the complexity of executing (iterating) the DQN over the actions. Note that the complexity of computing a state using Equation (4.1) also depends on the number of traffic demands, but we focus only on how many times the equation that computes the state is called. The structure of the per-action DQN algorithm is also the same in both algorithms, but only the number of times it is called varies.

4.2.2.1 Complexity of Calling the Stationary Input Transformation

The complexity of calling the stationary ASO input transformation function for the selection of coalition actions and sequential action selection is shown in Table 4.1. The

Algorithm 2 Per-action DQN for DRL with coalition action selection using stationary ASO input transformation

Initialize parameters: primary DQN network θ , target DQN network $\theta' = \theta$, discount factor $\gamma = 0.99$, ϵ-greedy $\epsilon = 1$, replay memory M = [], minibatch b = [], start and maximum episode ($e = 1$, $E = 50000$)

while $e \leq E$ **do**
  $\epsilon = \epsilon - \frac{\epsilon}{5000}$
  Time step $t = 1$
  Initialize the set of traffic demands $\{k\}$ with k_{max} number of random initial traffic demands
  Initialize B for L_{max} time steps B_L:$B_l = B$ for $0 \leq l \leq L_{max}$
  while $t \leq T$ **do**
    Compute state $s = [ST(.), B_L]$
    Q = []
    if rand $\leq \epsilon$ **then**
      Shuffle $\{k\}$ randomly and form new lists V, D , L, W
    else
      for each traffic demand k whose $D_k \leq B_t$ **do**
        Get Q-value $Q_k = Q(s, k \mid \theta)$
        $Append(Q,\ Q_k)$
      end for
      Sort $\{k\}$ in descending Q and form new lists V, D , L, W
    end if
    Reward r = 0
    Selected = []
    $a = 0$
    while $a \leq |\{k\}|$ **do**
      if $D_a \leq B_t$ **then**
        Selected = append(selected,a)
        r = r + V_a
        for i = 0 to L_a **do**
          $B_i = B_i - D_a$
        end for
      end if
      $a = a+ 1$
    end while
    Exclude the selected traffic demands from $\{k\}$
    Decrement $\{L_k\}$ and $\{W_k\}$ of the traffic demands
    Free occupied resources from B_L for all $L_k \leq 0$
    Generate new traffic demands of size between 0 and k_{max}, and append to $\{k\}$
    Compute next state s'
    Store the experience (s, $selected$, r, s') to M
    Increment t
  end while
  Sample a minibatch of (s,$selected$,r,s') from M to b
  Get Q-value $Q' = Q\,(s', k \mid \theta')$ for the feasible traffic demands ($k : B \geq W_k$) at s'
  Find maximum Q-value $maxQ_i = \max(Q'_i)\ \forall i \in b$
  Compute target Q-values $y_i = r_i + \gamma maxQ_i\ \forall i \in b$
  Get current Q-values $currQ = Q\,(s, selected \mid \theta)$ for the traffic demands
  Update the DQN by minimizing $Loss = \frac{1}{|b|} \sum_{i \in b} (y_i - currQ_i)^2$
  Update the targets: $\theta' \leftarrow \theta$
  Increment e
end while

Action selection	Best-case	Worst-case
coalition	$\mathcal{O}(nx)$	$\mathcal{O}(nx)$
Sequential	$\mathcal{O}(nx)$	$\mathcal{O}(nx^2)$

TABLE 4.2: Complexity of executing the DQN

coalition action selection algorithm calls the stationary ASO input transformation function only once in each time step, and the best-case and worst-case complexity are the same. However, for the sequential action selection algorithm, the best case is the same as the coalition action selection if the algorithm terminates after selecting one action. The worst-case complexity of the algorithm occurs when it has enough resources to select all actions. In such a case, the algorithm calls the stationary ASO input transformation function x times in each step, that is, it computes the state once for the x traffic demands and selects one action, then computes the state for the remaining $x-1$ traffic demands and selects one action, and it continues until no traffic demand is left. Since it repeats the sequence for newly generated traffic demands at each time step, the total number of calls to the stationary ASO input transformation function will be nx.

4.2.2.2 Complexity Analysis of Executing the DQN

Table 4.2 shows the best-case and worst-case complexities of the DQN algorithms of the coalition action selection and sequential action selection. For the selection of coalition actions, both the best and the worst cases are $\mathcal{O}(nx)$ because, at each time step, the algorithm must calculate the Q value for each traffic demand. Sequential action selection also has the same complexity in its best case, where only one traffic demand is elected at each time step. The worst-case complexity of iterating the DQN algorithm in the selection of sequential actions is $x * (x+1)/2 + n$ at each time step.

4.3 Theoretical Derivation and Level of Unique Transformation of the Stationary ASO Input Transformation

This section presents the derivation and theoretical proof of the level of uniqueness of the stationary ASO input transformation function, as shown in Equation (4.1). We call the equation a *stationary ASO input transformation* because the equation always transforms an input to the same output, unlike machine learning algorithms, which transform an input to different outputs at different times depending on the values of their weights. If the input can have nonpositive numbers, it can be transformed into an ASO input of positive numbers by adding the absolute value of the minimum possible negative number in the ASO input and 1 as $A = A + \mid min(A) \mid + 1$. However, the minimum amount of resources in our resource allocation algorithm is 0.

4.3.1 Permutation-invariance of the Stationary ASO Input Transformation

The stationary ASO input transformation function has two objectives: to transform a set of ASO inputs to a fixed vector and to transform all permutations of ASO inputs to the same vector. As seen in Equation (4.1), the stationary ASO input transformation outputs a vector of size 5 regardless of the size of the ASO input A. The output of the stationary ASO input transformation is also the same, regardless of the order of the elements in the ASO input. The reason is that the five terms in the stationary ASO input transformation are permutation-invariant. That is, if we look at the terms one by one: the size of an ASO input is the same irrespective of the order of its elements; The mean and the product for an ASO input do not change by changing the order of the elements of the ASO input, because addition and multiplication are commutative; for the other terms, the fact that the subtractions and the squares are element-wise and that their summations are commutative makes them permutation invariant.

The following section presents the derivation and analysis of the stationary ASO input transformation.

4.3.2 Analysis of the Stationary ASO Input Transformation on Its Unique Input to Output Transformation

A higher-dimensional vector cannot be uniquely transformed into a lower-dimensional vector of the same domain of numbers. However, stationary ASO input transformation is proposed for resources quantified by discrete values. Therefore, it is possible to transform a higher-dimensional vector with discrete values to a lower-dimensional vector with continuous values.

The proof focuses on ASO inputs of natural number elements because discrete-valued fractional numbers can be mapped to natural numbers as follows. If an ASO input can have discrete-valued fractional numbers with intervals e and a maximum possible fractional value of f_{max}, it can be mapped to a domain of natural numbers from 0 up to $(f_{max}.b)/e_{GCM}$, where e_{GCM} is the least integer multiple of e, and b is the number of discrete values less than e_{GCM}. For example, if the range of fractional numbers is [0, 0.4, 0.8, 1.2, 1.6, 2.0, 2.4, 2.8, 3.2], then $e = 0.4$, $f_{max} = 3.2$, $e_{GCM} = 2$, and $b = 5$. Therefore, the range of discrete fractional numbers is transformed into a range of integers from 0 up to 8.

First, we introduce the Grobner basis, which we have used to prove the uniqueness.

4.3.2.1 Grobner Basis

Grobner basis (Buchberger and Kauers, 2010) is a set of multivariate nonlinear polynomial systems, called Grobner bases, that have some properties to simplify the solution for many fundamental problems. By starting to find the roots of polynomials with fewer variables, the roots of multivariate polynomials can also be derived. Any complex multivariate nonlinear polynomial can be converted to Grobner bases using Buchberger's algorithm. The Grobner basis is time-consuming to compute manually. As a result, it should be computed using software.

Grobner basis is applied to many problems of non-linear computational geometry such as theorem proving (Kutzler and Stifter, 1986). In this work, we used it to see if the proposed stationary ASO input transformation in Equation (4.1) transforms an ASO input uniquely or if there are collisions. Here, we start with a very simple example in Example 4.1.

Example 4.1. *Question to prove: For a set of two variables $\{X,\ Y\}$, do the equations $X + Y$ and XY, give a unique output? I.e., for any two constants s and p, how many combinations of real values of X and Y are there such that $s = X + Y$ and $p = XY$?*

The two equations are transformed to a Grobner basis, so that it becomes easy to find the roots. We generate the bases in MATLAB using the following code.

Proof. Using gbasis() of MATLAB: syms X Y s p

vars=[X Y]; e=[X+Y-s, X*Y-p]; bases = gbasis(e,vars)

This outputs bases = $[-Y^2 + s * Y - p, X + Y - s]$

Now, we can see that one of the Grobner bases, $-Y^2 + s * Y - p$, is a univariate polynomial. Since the univariate polynomial has a degree of 2, it has two roots. Due to the fact that $X + Y$ and XY are operations invariant with permutation, substituting the roots of Y of the first base into the second basis to find the roots of X gives the permutations of the roots of Y. Therefore, the two roots are the solutions of the two simultaneous equations that we are trying to solve. This shows that there is only one combination of real values of inputs X and Y for any output constants s and p using the summation and multiplication operations. □

4.3.2.2 Derivation of the Proposed Equation

The proof starts by defining some definitions and lemmas as follows.

Definition 4.1. The set of full prime factors (F_{full}) of a natural number is the set of prime factors of the number including their repetitions and 1.

Definition 4.2. An N partition of the set of prime factors of a number (G_N) is an instance of placing the set of the full prime factors (F_{full}) of the number into an N non-empty subsets, where every member of F_{full} is assigned to only one of the subsets, and $N = \eta(A)$ is the size of the ASO input A that we want to create from the prime factors. An example for 36 is given in Example 4.2.

Definition 4.3. The set of N partitions of the set of full prime factors of a number (S_{G_N}) is the set of all G_N of the number.

Definition 4.4. A *product identity vector* is a vector that contains at most one arbitrary natural number in its elements, and all the other elements have 1.

Definition 4.5. The set of *product collisions*of an ASO input A or its product $\prod(A)$, are the set of other vectors that give the same product of vectors as $\prod(A)$.

Now, let us see how we come up with the stationary ASO input transformation function.

To transform an ASO input A of elements of natural numbers of size $\eta(A) = 2$ using only the sum of its elements, there are always collisions $ceil(s-1)/2$, where s is the sum of the numbers. This is because every natural number x from 1 to $s-1$ has another number $y = s - x$ where $x + y = s$, and half of them are their permutations. Therefore, using the sum as a stationary ASO input transformation results in a very large number of collisions. Similarly, transform the ASO input using only the product of its elements; the number of collisions depends on the number of prime factors of its product. We define the following lemmas and set a theorem for the number of collisions.

Lemma 4.6. *If an ASO input A of size $\eta(A) \geq 2$ with natural number elements is not a product identity vector, it has a non-empty set of product collisionsthat includes product identity vector, which is computed using the product of the ASO input $\prod(A)$ as the natural number element of product identity vector.*

Lemma 4.7. *The S_{G_N} of an ASO input A is the same as the S_{G_N} of the product of the ASO input $\prod(A)$, because the set of full prime factors of the number of their product (F_{full}) can be generated from (F_{full}) of the elements in the ASO input, excluding repeated ones.*

Lemma 4.8. *A set of product collisionsof an ASO input A having a product of $\prod(A)$, can be computed by replacing each partition in S_{G_N} of the ASO input with its product.*

Theorem 4.9. *For an ASO input A of size $\eta(A) \geq 2$ with natural number elements, or its product using Lemma 4.7, the number of other ASO inputs of natural number elements that give the same product is at most equal to the size of the set $\mid S_{G_n} \mid + 1$. The 1 is added for the product identity vector in Lemma 4.6.*

Theorem 4.9 is easy to prove because the set of natural numbers that can have a given number as their product is found only from the members of the factors of the number.

In other words, any element of a set of natural numbers is a member of the factors of the product of the set. Theorem 4.9 is used to count the collisions of sets of the same size.

By Theorem 4.9, if we want to use only the product of the elements of the ASO input as a stationary ASO input transformation function, the number of *product collisions* of an ASO input A of elements of natural numbers with other ASO inputs of equal size is at most $\mid S_{G_n} \mid$+1. Therefore, if the elements of an ASO input A of any size $\eta(A)$ contains 1 and the first $N-1$ complete prime numbers without repetition, where $N = \eta(A)$, its only *product collision* is the *product identity vector*, because these elements of the ASO input are the only set of S_{G_N}. A *product identity vector* of size $\eta(A) = 2$ can be uniquely represented by its product $\prod(A)$ if and only if the integer in the *product identity vector* is a prime number because its S_{G_N} is the *product identity vector* itself.

Theorem 4.9 shows that the *product collisions*of an ASO input A of elements of natural numbers can only be generated from the S_{G_N} and the *product identity vector*. To prove the uniqueness of the stationary ASO input transformation, we only need to focus on how to uniquely represent the *product collisions* of the ASO input A, because other ASO inputs of the same size will not have the same product.

The product of the ASO input $\prod(A)$ is not enough to uniquely represent an ASO input of size $\eta(A) = 2$ if it is not a *product identity vector* with a prime number because there can be other $\mid S_{G_n} \mid + 1$ vectors of the same size and product.

Theorem 4.10. *An ASO input A of size $\eta(A) = 2$ with elements of natural numbers can be uniquely represented using the product $\prod(A)$ and the sum of $\sum(A)$ of its elements because the product collisions of a number have a unique sum for $\eta(A) = 2$ except for its permutation.*

Proof. To prove Theorem 4.10, if $s = X + Y$ is the sum and $p = XY$ is the product, $p = X(s - X)$ = -X^2+sX-p is a polynomial of degree 2. A polynomial of degree 2 has at most 2 roots. If we compute Y using the roots of X, we find the permutations of X, because the variables are permutation invariant to the equations and the same second-degree polynomial can be generated using Y. Therefore, the ASO input containing the roots is the only ASO input that gives s and p. It can also be proved using the Grobner basis as seen in Example 4.1. □

The mean of the ASO input can be used in place of the sum because it plays the same role except for ASO inputs of different sizes of all 1 element. ASO inputs of different sizes are distinguished by their size in the stationary ASO input transformation.

If the size of the ASO input is $\eta(A) = 3$, the product $\prod(A)$ and the sum of $\sum(A)$ of the ASO input are not enough to uniquely represent it because some of its *product collisions*can have the same sum as seen in Example 4.2.

Example 4.2. *ASO input A = [1, 6, 6] has a product of 36. The full prime factors of 36 are {1,2,2,3,3}. The S_{G_N} of 36 includes { [{1},{2},{2,3,3}],[{1},{3},{2,2,3}], [{1},{2,2},{3,3}], [{1},{2,3},{2,3}], [{2},{2},{3,3,1}], [{2},{3},{2,3,1}], and [{3},{3},{2,2}]}. By changing each partition to their product, this gives the set of product collisions of {[1,2,18],[1,3,12],[1,4,9],[1,6,6],[2,2,9],[2,3,6],[3,3,4], and [1,1,36]}. It can be seen that vector A = [1, 6, 6] and vector B = [2, 2, 9] have the same sum $\sum(A) = \sum(B) = 13$.*

We have to find a method to discriminate the *product collisions*that have the same sum.

Lemma 4.11. *The difference of the elements of an ASO input from their mean D=d_1, ..., d_i, ..., d_N is unique for any vector because every element is subtracted by a constant value. However, the sum of $\sum(D)$ is always zero.*

For $\eta(A) = 3$, we can avoid the *product collisions*of the ASO input that have the same sum by computing $\sum(D^2)$, where D^2 is D^2=d_1^2, ..., d_i^2, ..., d_N^2. $\sum(D^2)$ is the variance of the ASO input.

Theorem 4.12. *The product collisionsof an ASO input A of size $\eta(A) = 3$ with natural number elements can be uniquely represented by their variances if they have the same mean.*

Including variance as the third term next to the product and sum can uniquely represent an ASO input A of size $\eta(A) = 3$ with natural number elements. We have provided two methods below to find an ASO input that has the same mean and variance as another ASO input. It is possible to find ASO inputs of elements of a natural number of size $\eta(A) = 3$ with the same mean and variance, but we did not find ASO inputs with the same product, mean, and variance.

Proof. We proved Theorem 4.12 using the Grobner basis by generating the bases as follows. syms X Y Z s p m

$e1 = X * Y * Z; e2 = (1/5) * (X + Y + Z); e3 = (1/5) * ((X - e2)^2 + (Y - e2)^2 + (Z - e2)^2); vars = [XYZ]; e = [e1 - p, e2 - s, e3 - m]; bases = gbasis(e, vars)$. This outputs the bases as: bases = $[(2 * Z^3)/5 - 2 * s * Z^2 + ((18 * s^2)/5 - m) * Z - (2 * p)/5, (2 * Y^2)/5 + (2 * Y * Z)/5 - 2 * Y * s + (2 * Z^2)/5 - 2 * Z * s + (18 * s^2)/5 - m, X/5 + Y/5 + Z/5 - s]$. The first term of the three bases has a univariate polynomial of degree 3 meaning that it has at most 3 real-valued roots. By substituting the three roots on the other bases, we find the permutations of the three roots because the product, mean, and variance are permutation-invariant operations. Therefore, this proves that the three terms uniquely transform an ASO input of three variables even for real numbers. □

We also test it using a brute force algorithm that generates all unique vectors of size $\eta(A) = 3$ as discussed at the end of this section.

For any ASO input A, we can use the following two methods to find if there is another ASO input with the same mean and variance.

Method 1: Sort D in increasing order and compute D^2. Then if rearranging D^2 in reverse order gives a different permutation, compute the roots of D^2 and negate them starting from the left until their sum is zero. Then add the mean of the ASO input A to them to find another ASO input B with the same mean and variance. Check if B has natural number elements.

Method 2: Describe D by another set of numbers O^2=O_1^2, ..., O_i^2, ..., O_N^2 where $\sum(O) = 0$ and $\sum(O^2)$=$\sum(D^2)$. Compute the roots of O^2 and find the other ASO input B as used in *Method 1*.

Example 4.3. *To see an example of one of the methods, let A be A= [6, 3, 6]. The mean is 5. D = [1, -2, 1]. After sorting and squaring, D^2 = [1, 1, 4]. To find the other number that gives such D, we find the roots of D^2 as D = [1, 1, 2]. Since the sum of the elements of D below must be zero, start negating them starting from the left until they give 0 sum. D = [-1, -1, 2]. After adding the mean, we find the other ASO input with the same variance A= [4, 4, 7]. However, A and B are not product collisions.*

We can find *product collisions*of an ASO input of size $\eta(A) = 4 \geq 4$ that has the same mean and variance using the two methods. To discriminate the ASO inputs that can be generated by the two methods, we use the variance of the sum of the ASO inputs and its Hadamard-product.

Theorem 4.13. *Any product collision of an ASO input A of size $\eta(A) = 4$ with natural number elements can be uniquely represented by the variance of the element-wise sum of the ASO input with its Hadamard-product if it has the same mean and variance.*

Proof. Theorem 4.13 is proved by the Grobner basis. We extended the Grobner basis used to prove Theorem 3 by adding a fourth variable W, a fourth constant d, and the expanded form of $\sigma(A + A \circ A)$ as seen at the end of this section. The Grobner basis gives as bases, with one of them having a single variable polynomial of degree 4, which leads to the conclusion that the polynomial has 4 roots, which correspond to the size of the ASO input $\eta(A) = 4$. The other solutions by substitution in the other bases are permutations of the roots because the terms in the stationary ASO input transformation are permutations invariant to the variables. Therefore, the stationary ASO input transformation can be unique even for ASO inputs of real numbers for $\eta(A) = 4$. □

We also experimentally tested the uniqueness of Theorem 4.13 and Theorem 4.12 experimentally as follows. We generated all possible unique vectors of size $\eta(A) = N$ and maximum integer K using the brute force algorithm in Algorithm 3 and found no collision for $K = 100$ and $\eta(A) = 4$. To test it experimentally, we transformed all the

ASO inputs generated by Algorithm 3 into another ASO input using Equation (4.1) and check if there is the same output for different input vectors. We repeated the experiment for $K = 50$ and $\eta(A) = 5$, $K = 10$, and $\eta(A) = 10$ and did not find any collision. We did not try it for large K, because the brute-force algorithm continues to run for more than a day. We also check Theorem 4.12 for $K = 100$ and $\eta(A) = 3$ using only the product, mean, and variance as a stationary ASO input transformation function, and no collision is found.

Although it is not our requirement, because we assume positive discrete fractional numbers in our resource allocation algorithms, we check the stationary ASO input transformation for a mixture of positive and negative integers by experimenting with brute force for between $K = -10$ and $k = 10$ of size $\eta(A) = 4$ without transforming them into positive numbers. But we do not try it for real numbers because we do not have a brute-force algorithm to generate unique vectors of real numbers.

This proof shows the level of unique transformation. It does not prove that the stationary ASO input transformation is unique for any size of A and for any real number. For example, for $\eta(A) \geq 5$, the Grobner basis has no base with leading monomials having single variables, which leads to the conclusion that the stationary ASO input transformation is not unique for ASO inputs of real numbers when $\eta(A) \geq 5$. Nevertheless, this cannot be a contradiction to our algorithm, as the stationary ASO input transformation is only for natural numbers. We could not find the possible number of integer roots using the Grobner basis because the constants in the simultaneous equations are generic.

The 5 terms of Equation (4.1) are shown in more detail below. As can be seen, all terms play their own discriminatory role in the stationary ASO input transformation because none of them can be generated from the other terms. That is, none of the four terms is redundant in the stationary ASO input transformation.

Algorithm 3 Brute-force unique vectors generator

```
1: Assign K and N
2: U=[]
3: for n_1=1:k do
4:    for n_i=n_{i-1}:k do
5:       for n_N=n_{N-1}:k do
6:          Append(u, [n_1,..., n_i ,..., n_N])
7:       end for
8:    end for
9: end for
```

$$\prod(A) = \prod_{i=1}^{N} A_i$$

$$\eta(A) = N$$

$$\mu(A) = \frac{1}{N}\sum_{i=1}^{N} A_i$$

$$\sigma(A) = \frac{1}{N}\sum_{i=1}^{N}(A_i - \mu(A))^2 = \frac{1}{N}\sum_{i=1}^{N} A_i^2 - \mu(A)^2$$

$$\begin{aligned}\sigma(A + A \circ A) &= \frac{1}{N}\sum_{i=1}^{N}[(A_i + A_i^2) - \mu(A + A \circ A)]^2 \\ &= \frac{1}{N}\sum_{i=1}^{N}[(A_i + A_i^2) - \frac{1}{N}\sum_{k=1}^{N}(A_k + A_k^2)]^2 \\ &= \frac{2}{N}\sum_{i=1}^{N} A_i^3 + \frac{1}{N}\sum_{i=1}^{N} A_i^4 - \mu(A)^4 - 2\mu(A)^3 \\ &- 2\sigma(A)\mu(A)^2 - 2\sigma(A)\mu(A) - \sigma(A)^2 + \sigma(A)\end{aligned}$$

Note that the expansion of $\sigma(A + A \circ A)$ is simplified by continuing the expansion of $\sigma(A)$ by replacing A with $A + A \circ A$. Expanding it on its own also gives the same final terms.

4.4 Experimental Evaluation

The objective of this experiment is to evaluate whether the stationary ASO numerical input transformation function leads to superior convergence than the transformer-based transformation, as it allows the DRL algorithm to focus only on learning the policy. We compared the stationary ASO input transformation function with the transformer-based coalition and sequential action selections presented in Chapter 3. We also used the same problem of online resource allocation and offline optimal.

4.4.1 Experimental Setup

Both DQNs for coalition and sequential action selection of the per-action DQN have the same neural network structure except that the coalition action selection has an input layer of 27 neurons while the sequential has 26. This is because the coalition has a distinguisher index to distinguish similar elements in action selection. They have 2 hidden layers with 64 and 32 neurons, respectively, with ReLU activation functions, and an output neuron with a linear activation function. The number of inputs is determined by a combination of the lengths of the resource constraint vector, the stationary ASO input transformation vector, and the feature vector of the embedded action. The $\eta(I)$ of equation (4.1) is considered only once, not element-wise, but another term, $1/Max$, is included to avoid collision in normalization. We normalize the transformed vector to be in the range of [0,1] by dividing it by the maximum number in the vector. The hidden layers and their neurons in neural networks are decided by trial and error.

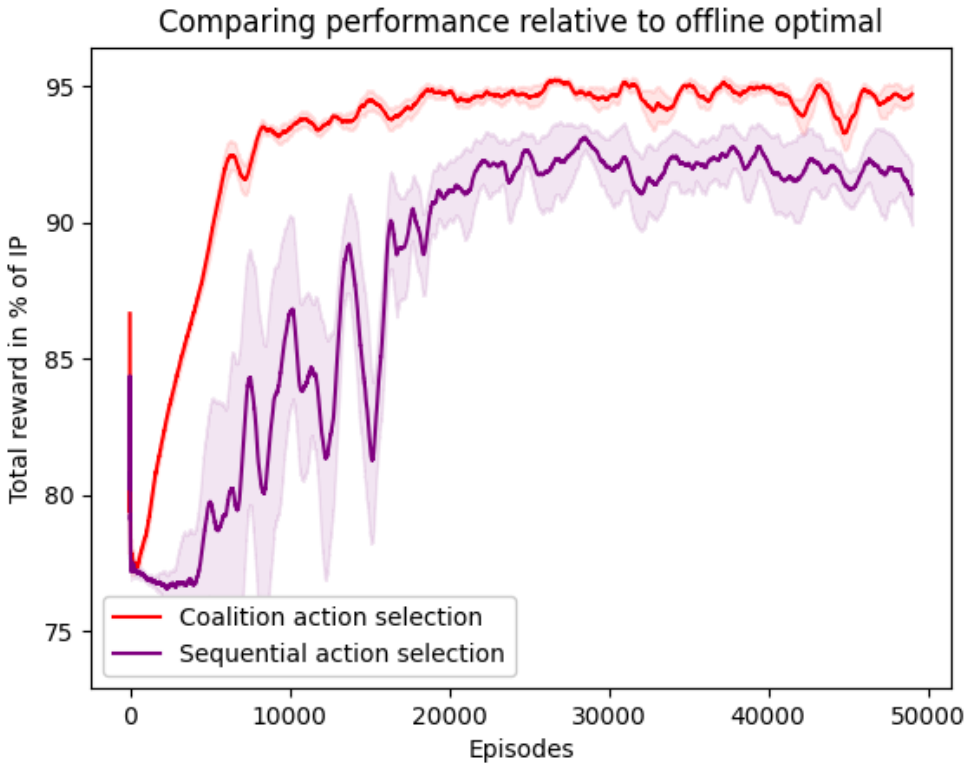


FIGURE 4.3: Convergence of the stationary ASO input transformation-based DRL algorithm on the coalition action selection and the sequential action selection using end-of-episode training

We set the bandwidth vector length B_L to 1 so that the IP is faster. We used a replay memory of size 10000 and a minibatch size of 64. The experiment was run for 40 runs. The experiments are implemented with Pytorch.

4.4.2 Experimental Comparison

First, we show the convergence of the DRL with a computationally efficient stationary ASO input transformation. Next, we did an exhaustive experiment for the transformer-based benchmark algorithm to find its best hyperparameter setting. Finally, we compare both coalition action selection and sequential action selection with their transformer-based counterparts to show the advantage of the stationary ASO input transformation.

4.4.2.1 Convergence

Using the same experimental setup as the experiment in Chapter 3, we evaluated the performance and complexity of the proposed algorithm. We use IP as an offline optimal and compare coalition action selection and sequential action selection by plotting

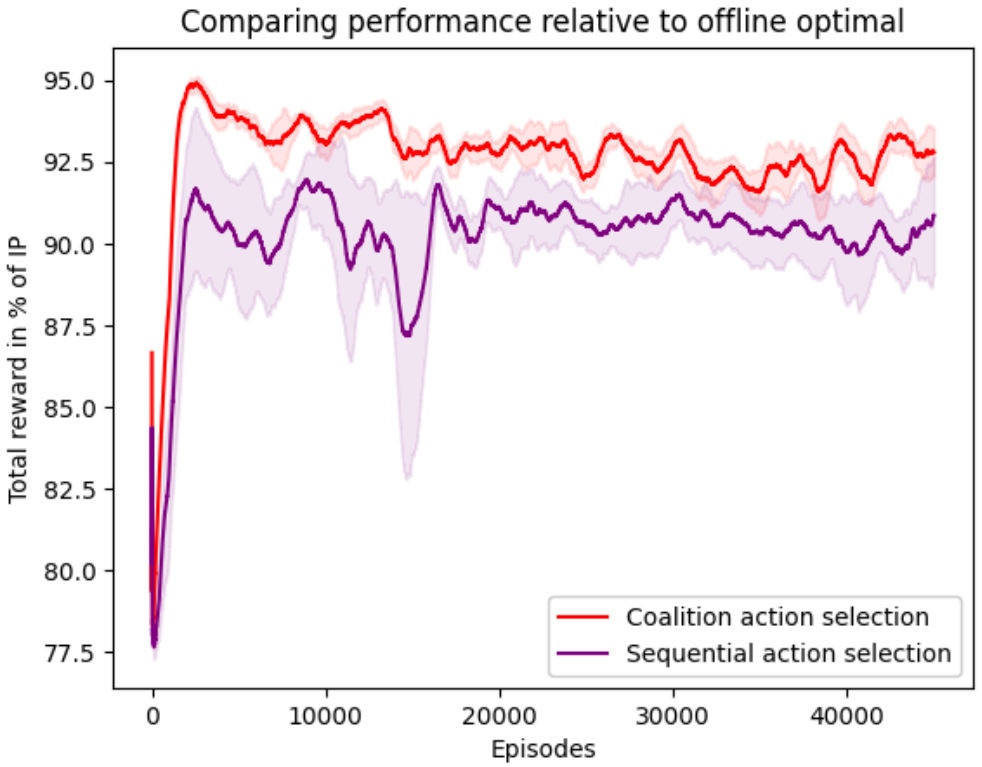


FIGURE 4.4: Convergence of the stationary ASO input transformation-based DRL algorithm on the coalition action selection and the sequential action selection using step-by-step training

the result of the two approaches as a percentage of the offline optimal. All experiments were averaged over 40 runs. To smooth the curves, the results in Figure 4.3 are plotted for the averaged moving window of 1000 episodes. As seen in Figure 4.3 for episodic training, the coalition action selection approach converges to 95.234% at episode number 26621. It has outperformed sequential action selection with an average gap of 2.8%. We also experimented to show the impact of step-by-step training, as shown in Figure 4.4 where performance is reduced due to overfitting. However, overfitting due to stepwise training is worse in the transformer-based DRL in Chapter 3 than in the stationary ASO input transformation because the transformer is more complex than the per-action DQN and is more susceptible to overfitting.

4.4.2.2 Exhaustive Exploration of the Performance of the Transformer

The objective of the experiment in Chapter 3 was to compare the coalition action selection and sequential action selection approaches. For this reason, it does not require an exhaustive experiment because both the coalition action selection and the sequential action selection have the same transformer and hyperparameters, and hence the coalition

action selection always provides superior performance than the sequential action selection. However, transformer-based and stationary ASO input transformation-based DRL algorithms have different types of neural network. Therefore, before using the transformer-based DRL algorithm as a benchmark for the stationary ASO input transformation, an exhaustive experiment of the transformer-based algorithm is required so that it is compared at its best performance.

Number of layers	Neurons in each layer	Peak value	Index of peak value
2	32	93.167%	38789
2	64	93.203%	38788
2	128	93.364%	38008
2	256	93.347%	38792
4	32	93.669%	27517
4	64	93.545%	27655
4	128	93.402%	27518
4	256	93.478%	27439
6	32	93.777%	28036
6	64	93.452%	27621
6	128	93.739%	27209
6	256	93.444%	27207
6	512	93.425%	27494
8	32	92.975%	21823
6	16	93.537%	27666
6	8	93.107%	17410

TABLE 4.3: Performance of the transformer-based DRL for various numbers of hidden layers and neurons

Table 4.3 presents the maximum performance of coalition action selection at different values of the number of hidden layers and neurons of the transformer-based DRL. It shows that the transformer with six hidden layers and 32 neurons each in each layer performed better than other values with a maximum value of 93.777% of the IP-based offline optimal. The maximum value is the value computed after the averaged moving window and the average of the 40 runs. The index value indicates the episode number where the peak value is recorded. Choosing the number of layers and neurons to be 6 and 32 respectively based on the result in Table 4.3, we ran the experiment for various multihead attention values as seen in Table 4.4, which also shows that multi-headed attention of 8 yielded better performance. These are the hyperparameters used in Chapter 3. Note that multihead attention values are factors of the size of the input embedding layer. This is because the attention head has to divide the input into equal parts and process them differently using different self-attention layers, and finally combine them into a fixed-size vector.

The experiments in Table 4.3 and Table 4.4 are based on an average of 40 runs of the experiment. Due to time and resource constraints, we run the following experiments with an average of 10 runs to do more experiments to explore different learning rates,

Number of attention heads	1	2	4	8
Peak value	93.664%	93.764%	93.631%	93.777%
Index of peak value	28030	27932	28043	28036

TABLE 4.4: Performance of the transformer-based DRL for various attention heads

Learning rate	0.1	0.001	0.0001
Peak value	93.389%	93.777%	92.590%
Index of peak value	17279	28036	27620

TABLE 4.5: Performance of the transformer-based DRL for various learning rates

batch sizes, and embedding layer sizes. As such, we examine the algorithm for different learning rate values, as seen in Table 4.5 which shows that the learning rate of 0.001 is better. The result for the learning rate of 0.001 is taken from Table 4.3. We also run for various batch sizes, as seen in Table 4.6.

Size of minibatch	32	64	128
Peak value	93.714%	93.777%	93.333%
Index of peak value	24540	28036	21862

TABLE 4.6: Performance of the transformer-based DRL for various minibatch sizes

The results in Chapter 3 are plotted after an exhaustive experiment of the table above for a learning rate of 0.001, a batch size of 64, a hidden layer block of 6 with 32 neurons each, and an attention head of 8. The embedding layer was selected to be 8 neurons, a close guess to the number of inputs 5. Now, we vary the embedding layer as presented in Table 4.7, which gives another maximum value for an embedding layer of 16, with 6 hidden layers of 16 neurons in each.

Embedding size	No. of neurons in hidden layers	Peak value	Index of peak value
8	8	93.559%	27666
16	16	94.421%	27658
16	32	93.962%	27215
16	64	94.035%	27134
16	128	93.940%	27369
32	32	93.481%	27147
32	64	93.525%	9595
32	128	93.809%	27265
64	64	93.063%	9778
64	128	93.261 %	16819

TABLE 4.7: Performance of the transformer-based DRL for various embedding sizes

4.4.2.3 Comparison of Performance with the Transformer-based Benchmark

From the exhaustive experiments in Section 4.4.2.2, we found that the maximum convergence of the DRL with the transformer-based state transformation is 94.421%. The

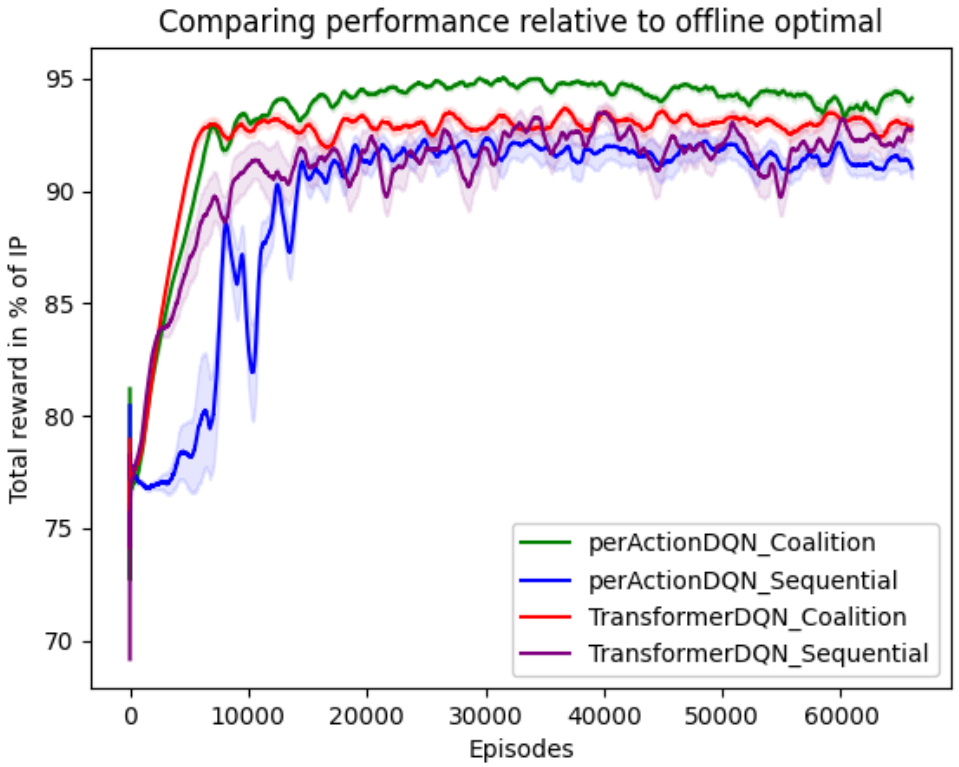


FIGURE 4.5: Comparison of the performance of the stationary ASO input transformation-based and transformer neural network-based transformation on both the coalition action selection and the sequential action selection approaches

hyperparameter values that achieved maximum performance are an embedding layer of 16, 6 hidden layers of 16 neurons each, a multihead number of 8, a learning rate of 0.001, and a minibatch size of 64. The maximum value is still less than the maximum value achieved with the stationary ASO input transformation shown in Figure 4.3, which is 95.234%.

Note that all experiments so far were recorded from an experiment that ran for 24 hours. To explore the performance of the transformer-based transformation and the stationary ASO input transformation with a longer experiment, we carried out the experiments together for 60 hours, as seen in Figure 4.5 using their best hyperparameters. This is plotted with an average of 40 runs.

The maximum values for the experiments in the result of Figure 4.5 are in Table 4.8. Note that the maximum score for the transformer-based DRL is 93.685% but it was 94.421% in the exhaustive experiment for the same hyperparameter setting. Although the result in Table 4.7 was recorded using an average of 10 runs and Figure 4.5 is for 40, the results cannot be the same even with the same number of runs. For example, the stationary ASO input transformation recorded a maximum value of 95.234% in Figure 4.3 but 95.034% in Figure 4.5 for the same hyperparameter setting.

Action selection	Coalition	Sequential
Transformer	(93.685%, 36685)	(93.504%, 40077)
ASO per-action DQN	(95.034%, 31577)	(92.350%, 30063)

TABLE 4.8: Maximum values and their corresponding episode number for the transformer-based and stationary ASO input transformation-based DRL algorithms on the coalition action selection and sequential action selection

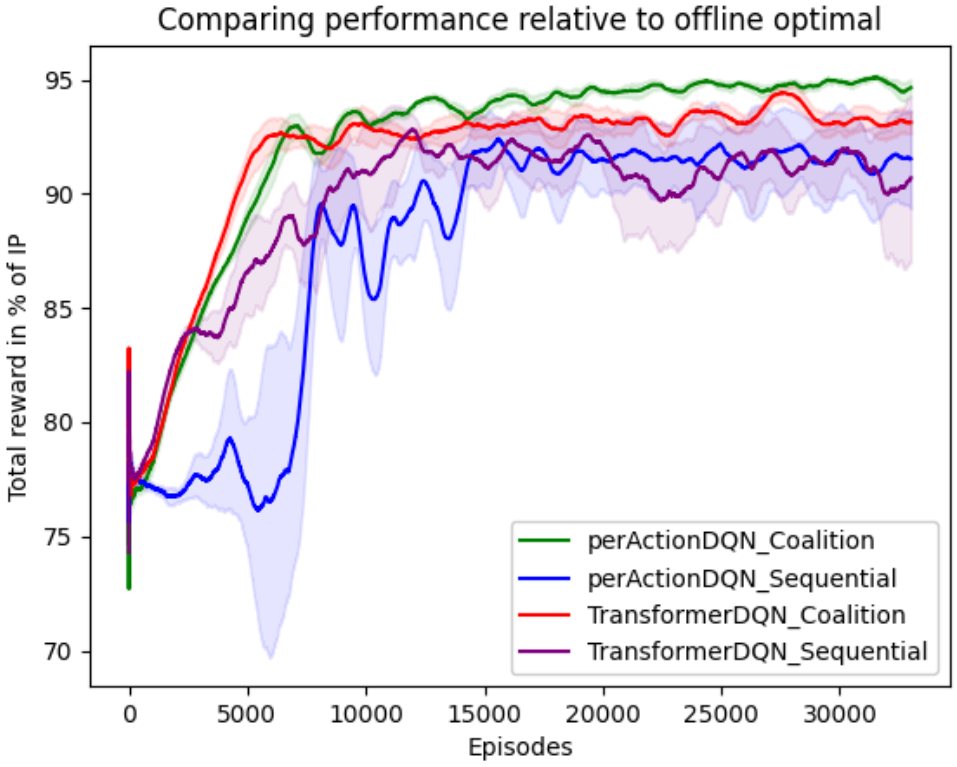


FIGURE 4.6: The performance of using stationary ASO transformation and neural network-based transformation on both coalition action selection and sequential action selection: averaged on 10 runs

We plotted the average of the first 10 runs of the experiment Figure 4.5 with the results that scored 94.421% in Table 4.7 to compare the best score with the same number of runs as seen in Figure 4.6.

4.4.2.4 Comparison of Complexity of Stationary ASO Input Transformation and Transformer-based Transformation on the Coalition Action Selection

To compare the computational cost of running the DRL algorithms using the proposed stationary ASO input transformation and the transformer-based benchmark, we analyzed the CPU time consumed by both transformation techniques for the coalition action selection as shown in Figure 4.7. The CPU times of the box plot are generated from the 66001 episodes in Figures 4.5. As described in Section 3.5.4.3, the CPU time

is the sum of the fraction of seconds that each algorithm spends running the DRL algorithm to select actions. The number of executions and CPU time is averaged over the 40 runs episodic before being used for the box plot. Because the objective is to compare the transformer-based transformation and the stationary ASO input transformation, we compared them only on the coalition action selection. The difference in their complexity increases proportionally with the sequential action selection as seen in Section 3.5.4.3 for the transformer-based state transformation.

Figure 4.7, presents the complexity using CPU time. The CPU time considered the action selection that includes the execution of the neural networks and state computations. As seen in box 1, the DRL algorithm with coalition action selection took a median CPU time of 0.031030813704273897 seconds to run an episode; 25% of the episodes executed the DRL algorithm in less than 0.03007814525408321 seconds; 75% of the episodes took less than 0.03197447307420589 seconds to select the actions. On the other hand, the DRL with coalition action selection took a median of 0.05276393976528197 seconds, with 25% of the episodes taking less than 0.05213756428352098 seconds and 75% of them taking less than 0.05720445671867083 seconds when the transformer is used as the state transformation technique. Using stationary ASO input transformation provided faster execution times for two reasons. First, the neural network is simple compared to the complex transformer. Second, it computes a common state once and executes only the feasible traffic demands, unlike the transformer-based transformation, which executes all the traffic demands to produce intermediate values for the attention layers at every block of the transformer.

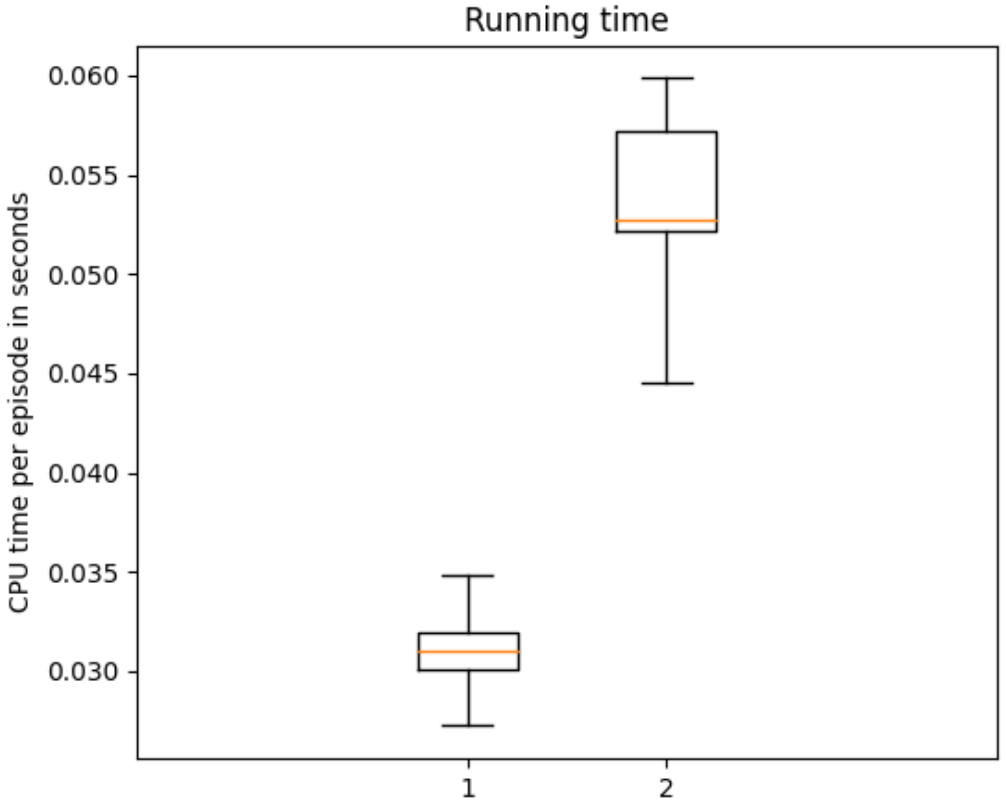


FIGURE 4.7: The execution in CPU time for (1) Per-action DQN with stationary ASO input transformation-based coalition action selection and (2) Transformer-based station transformation-based coalition action selection.

4.4.2.5 Comparison of Performance and Complexity of Stationary ASO Input Transformation and Transformer-based Transformation with Various Traffic Demand Arrival Rates

In the same scenario as in Section 3.5.4.4, we also run the experiment for various arrival rates as seen in Figure 4.8. We used the same hyperparameters as for the experiment used in Figure 4.5. That is, the experiment is run for another 4 scenarios with traffic demand arrival rates of k_{max} = 2, k_{max} = 5, k_{max} = 15, and k_{max} = 20, with 10 runs in each experiment scenario due to time and resource constraints. Note that the result in Section 3.5.4.4 was taken from 40 runs in each arrival rate. For k_{max} = 10, we take the first 10 runs of the result in Figure 4.5. Then we take the maximum value of the 10 runs of each arrival rate, as discussed in Section 3.5.4.4 using an averaged moving window of 5000 episodes. The CPU time is the average CPU time for the episodes of the run of the experiment that outputted the maximum performance in each experiment. The experiment was carried out for 60 hours.

The result in Figure 4.8 shows that the DRL with the proposed computationally efficient stationary ASO input transformation has a lower complexity than the transformer-based benchmark in both the coalition action selection and sequential action selection counterparts. Performance is also superior. However, the proposed state transformation is proposed to be unique up to 4 variables. As a consequence, the transformer-based DRL can perform better at higher arrival rates because it is not affected by size. The proposed state transformation can be customized to accommodate large arrival rates.

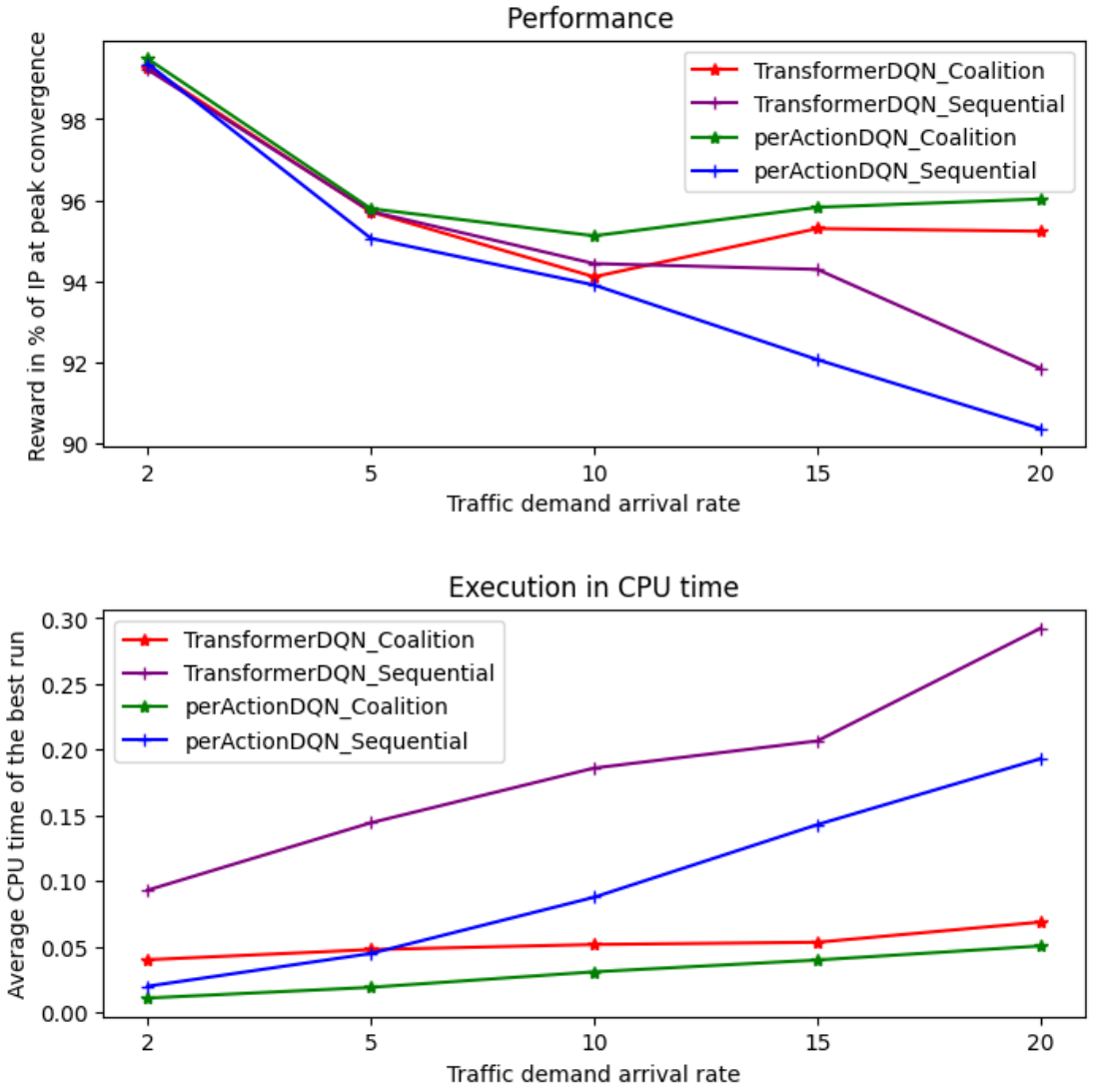


FIGURE 4.8: The performance and complexity for per-action DQN with stationary ASO input transformation and transformer-based state transformation with varying task arrival rates. The CPU time is averaged over the episodes of the best run that led to peak convergence in the plotted performance.

4.4.2.6 Evaluation with Statistical Tests

It is obvious from the numerical evaluation presented with tables and figures with 95% confidence interval that there is a significant difference between the performance of the DRL algorithms. Moreover, there is no overlap between the 95% confidence intervals of the stationary ASO state transformation-based coalition action selection algorithm and the transformer neural network-based coalition action selection algorithm. Nevertheless, we performed statistical tests to enhance the assessment. In this section, the performances of stationary ASO state transformation and transformer-based state transformation techniques are evaluated with a statistical difference test. In Chapter 5 the proposed CCM_MADRL_MEC is evaluated in the same way against the benchmark and heuristic algorithms.

A statistical difference test provides a principled way to compare the central performance of two algorithms with two hypotheses: the *null hypothesis*, which suggests that there is no significant difference between the algorithms, and the *alternative hypothesis*, which suggests that there is a significant difference in performance (Colas et al., 2019). The hypotheses are determined on the basis of the P-value calculated from the sample data. If the P-value is small (usually less than 0.05), the null hypothesis is rejected. Colas et al. (2019) has summarized the types of statistical tests for DRL algorithms, including their assumptions. Parametric tests, such as the t-test and ANOVA (Analysis of Variance), compare the means of performance of the algorithms by making certain assumptions about their distributions. ANOVA compares three or more algorithms. Nonparametric tests, such as the Wilcoxon signed rank test, the Kruskal-Wallis test, the Mann-Whitney U test, and the ranked t-test, do not have assumptions about the type of distribution of the data. Instead of assuming a certain distribution of the data, they rely on ranks and ordering of the data. Therefore, they compare medians rather than means.

In our case, the assumptions of the parametric test are not met. As seen in Figure 4.5, performance starts low in the early episodes and stabilizes after a few episodes. This is not normally distributed as depicted using histograms in Figure 4.9.

The histogram in Figure 4.9 is plotted from the experiments episodes excluding the moving average window described in Section 3.5.4.1, which was only used to smooth the lines. As seen in the histogram, the performance in the episodes of the DRL algorithms is not normally distributed because the DRL algorithms are training and improving performance across episodes. The results of each episode are averaged (episode-wise) over 40 runs of the experiment. We also plot another histogram using the experimental runs as a data sample as seen in Figure 4.10, where the histograms are plotted from a sample size of 40, which are the maximum performances of each experimental run using an averaged moving window of 1000 episodes. We used the averaged moving window because episodes generate different traffic demands. Therefore, if we

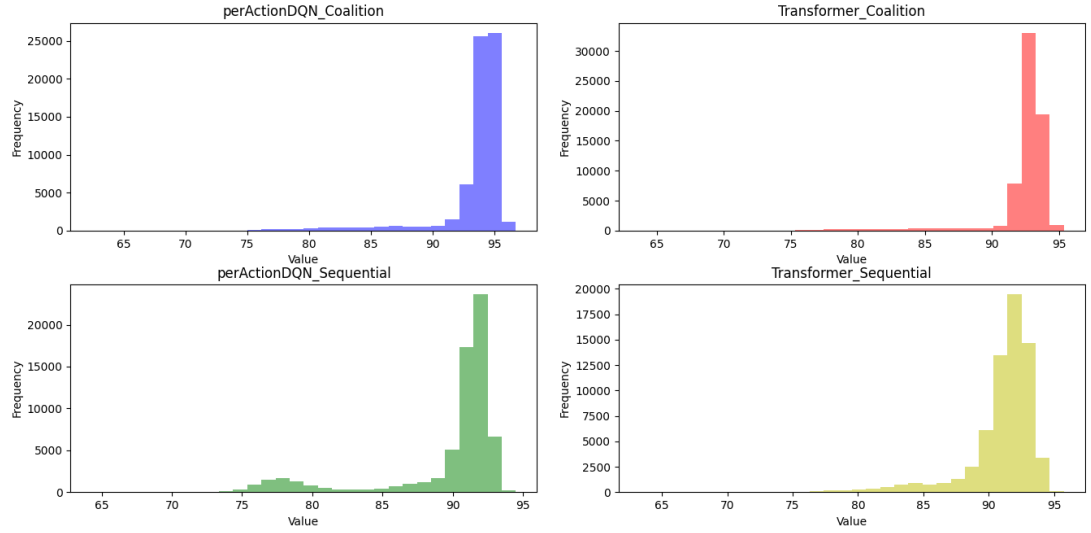


FIGURE 4.9: Histogram for the performance of the algorithms shown in Figure 4.5 using the episodes as sample

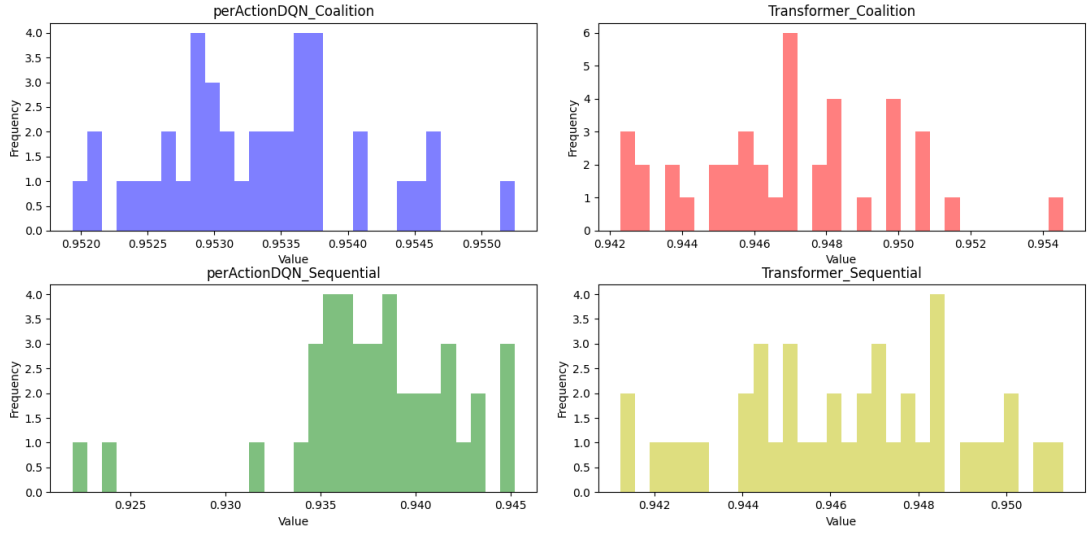


FIGURE 4.10: Histogram for the performance of the algorithms shown in Figure 4.5 using the runs of the experiment as sample

compute the maximum value episode-wise, the performance of all algorithms can be 100% to the offline optimal if the demands generated in an episode are smaller than the resource constraint. It can be seen that the distribution is better using the runs of the experiment as a sample than using the episodes as a sample. However, it is still not normally distributed.

Due to their assumptions, it is challenging to decide which of the statistical tests is suitable for evaluating the performance of the DRL algorithms. Therefore, we performed multiple statistical tests. Our algorithm requires a comparison between two groups. The stationary ASO state transformation-based coalition action selection is compared with the transformer neural network-based counterpart. The sequential action selection is also compared to the corresponding sequential action selection. However, we

applied ANOVA to test whether there is a difference in four of them. Then, the t-test is used to compare the algorithms in pairs. The t-test assumes equal variance and equal sample size. The results of the experiment are equal in size, but we also performed the Welch t-test, which does not assume equal variance, and found the same values. Furthermore, because the t-test is parametric and assumes a normal distribution, which is not satisfied in our case, the statistical test is repeated with nonparametric tests.

ANOVA rejected the null hypothesis with an F-statistic of 16108.681982760563 and a P-value of 0.0. F-statistic showed that there is substantial variation between the means. Note that ANOVA does not make pairwise comparisons, but only shows that there is a difference in the performance of the four algorithms. A statistical test that compares the coalition action selection of the transformer-based DRL with the stationary ASO transformation-based counterpart is necessary. The t-test for the experimental result plotted in Figure 4.5 is presented in Table 4.9.

Action selection	T-statistic	P-value
Coalition	67.45770406333097	0.0
Sequential	67.04944126775521	0.0

TABLE 4.9: T-test between the transformer-based and stationary ASO input transformation-based DRL algorithms on the coalition action selection and sequential action selection using the episodic rewards as data sample. The values are a comparison of coalition with coalition and sequential with sequential.

Table 4.9 shows that both coalition action selection and sequential action selection have shown significant differences in transformer-based and stationary ASO transformation-based algorithms with P-values of 0 and t-values of 67, showing significant differences.

The statistical evaluation of the DRL algorithms with the runs of the experiment is as follows. ANOVA provided an F-statistic of 175.12542120891644 and a P-value of 1.0103450012748946e-49. The t-test is presented in Table 4.10. Both tests show a P-value of very close to zero, which means that there is a significant difference in the performance of the algorithms.

Action selection	T-statistic	P-value
Coalition	13.966806712556847	6.343021425947807e-23
Sequential	10.10873811117661	7.800580831393717e-16

TABLE 4.10: T-test between the transformer-based and stationary ASO input transformation-based DRL algorithms on the coalition action selection and sequential action selection using the experimental runs as data sample

We repeat the statistical test with the nonparametric test as follows. Wilcoxon Mann-Whitney rank sum test is an example of a nonparametric test for comparing the medians of two groups of data samples. Although nonparametric tests do not have assumptions about the type of distribution, the Wilcoxon Mann-Whitney rank sum test assumes the same shape of the spread between the two groups of data samples. This assumption is met in Figure 4.9 but not in Figure 4.10 but we apply it to both.

Action selection	U-statistic	P-value
Coalition	3568512337.0	0.0
Sequential	2601506569.5	0.0

TABLE 4.11: Wilcoxon Mann-Whitney rank sum test between the transformer-based and stationary ASO input transformation-based DRL algorithms on the coalition action selection and sequential action selection using the episodic rewards as data samples.

Action selection	U-statistic	P-value
Coalition	1563.0	2.1817427332690098e-13
Sequential	1543.0	9.018635937375004e-13

TABLE 4.12: Wilcoxon Mann-Whitney rank sum test between the transformer-based and stationary ASO input transformation-based DRL algorithms on the coalition action selection and sequential action selection using the experimental runs as data samples.

Tables 4.11 and 4.12 show that both coalition action selection and sequential action selection have shown significant differences in transformer-based and stationary ASO transformation-based algorithms.

In summary, multiple statistical tests demonstrate that DRL algorithms have significant differences.

4.5 Conclusion

We have demonstrated that the stationary ASO input transformation offers faster convergence than the transformer-based transformation on ASO input with numerical data. This is because it minimizes the learning complexity of the DRL by allowing it to focus only on learning the policy. Although using an attention neural network for ASO numerical input transformation is more representative of the transformation, it slows the training because it has to learn both the transformation of ASO numerical input and the policy. The attention neural network is efficient for NLP, but stationary ASO input transformation converges faster for numerical data. The proposed stationary ASO numerical input transformation has a limited scope in terms of size and is aimed at demonstrating that the stationary input transformation is better than a transformer for the ASO input with numerical data. We provide proof of its unique transformation for up to 4 inputs, but a more unique stationary ASO input transformation is essential to avoid the collision of transformations for large input sets. However, the convergence is better than that of the transformer up to an arrival rate of 20 as seen in the results. As explained in Section 3.5.4.4, the academic license in Gurobi optimizer, which is used to compute the offline optimal, did not allow us to run the experiment for arrival rates greater than 20.

We plan to integrate our work with the complementary algorithms by Zhang et al. (2009) and Liu et al. (2021), coordinating them as a multi-agent heterogeneous DRL where node agents and link agents coordinate to make efficient decisions.

Chapter 5

Combinatorial Client-Master Multiagent Deep Reinforcement Learning for Task Offloading in Mobile Edge Computing

5.1 Introduction

Recently, there has been an explosion of mobile applications that perform computation-intensive tasks, such as video streaming, data mining, virtual reality, augmented reality, image processing, video processing, face recognition, and online gaming (Zhang et al., 2013; Chen et al., 2023; Birhanu Engidayehu et al., 2022; Kan et al., 2018). However, user devices (UDs), such as tablets and smartphones, have a limited ability to perform the computation tasks of these applications. Mobile Cloud Computing (MCC) has been considered the key technology to improve the quality of experience (QoE) of UDs by offloading their computation tasks to the cloud (Mahenge et al., 2022). One of the challenges of MCC is latency caused by the distance of MCC servers from UDs (Sajnani et al., 2018). Mobile edge computing (MEC) has emerged as a promising technology for addressing the challenges of MCC and the increasing computing demands of UDs. By providing the MCC service, such as computing and storage, on the edge of the network, MEC serves the computation-intensive tasks of the UDs in their vicinity. MEC is an essential component of the Internet of Things (IoT) and 5G architecture that improves the computing experience of UDs by reducing latency and energy consumption (Vhora and Gandhi, 2020), especially in scenarios where real-time processing is critical.

Task offloading in MEC has become an attractive solution to meet the diverse computing needs of UDs (Islam et al., 2021). By distributing computational tasks between

the UDs and the MEC servers, MEC optimizes resource utilization, reduces data transmission overhead, and improves energy efficiency for the UDs. This results in a more seamless and responsive QoE, which is crucial to the success of IoT applications that rely on timely data processing and interaction.

The various types of continuous and discrete resource constraints on UDs and MEC servers pose significant challenges in the design of an efficient task-offloading strategy. UDs have many limitations, such as finite battery life and limited computational capabilities (Kan et al., 2018; Yan et al., 2018), as well as quality of service (QoS) requirements, such as response time or throughput, and QoE constraints, such as energy consumption and long-term QoS satisfaction. Similarly, MEC servers and the wireless network also come with storage capacity and number of channels, respectively, as constraints. Taking into account the resource constraints of both UDs and MEC servers is crucial to the effective operation of task-offloading methods, especially during periods of high demand.

Although a great number of task-offloading algorithms have been proposed, they focus only on the core objective of MEC technology, namely minimizing latency and energy consumption from the perspective of UDs, assuming the availability of enough storage resources on the server. The decision entities of the algorithms are either the UDs or the MEC servers. A comprehensive survey on task offloading by Islam et al. (2021) has presented task offloading strategies in MEC from different perspectives, including the computational model, the decision-making entity, and the algorithm paradigm. Many algorithms have considered the wireless communication resource and the computing resource of the server. For example, the insufficient computing resource of the MEC server can be alleviated by using MEC-MCC collaboration or collaboration among multiple MEC servers (Chen et al., 2023). Many DRL-based task offloading algorithms (Nguyen et al., 2023; Jiang et al., 2023) also considered communication channels in their state and action spaces. However, the storage constraint on the server is overlooked in existing task-offloading techniques in general and DRL-based algorithms in particular. There is no DRL-based task-offloading algorithm that considers the storage constraint on the server. Server storage can be overwhelmed when multiple UDs offload their tasks.

Furthermore, many existing task-offloading algorithms use traditional convex optimization methods for single-agent task-offloading scenarios (Sadatdiynov et al., 2023). Despite their advantages over convex optimization methods, most existing DRL algorithms are also single-agent algorithms that solve a single objective function and have many limitations with different action spaces and constraints. DRL techniques, such as DQN, have yielded encouraging results by modeling the task-offloading problem as MDP with DNN for the function approximation (Liu et al., 2022). However, due to the curse of dimensionality, DQN is insufficient for learning with large discrete action spaces (Dulac-Arnold et al., 2015) and a combination of continuous and discrete

action spaces (Zhang et al., 2020). Although DDPG-based task-offloading algorithms can handle continuous action spaces, the representation of discrete and continuous action spaces still poses a challenge (Zhang et al., 2020; Jiang et al., 2023). Moreover, a single agent DRL is not suitable for formulating the various constraints in the UDs, wireless communication, and server. MADRL has the potential to address these limitations by facilitating intelligent task offloading in MEC networks. Despite the advances of MADRL in task offloading, such as cooperative offloading decisions (Nguyen et al., 2023) and mixed continuous and discrete action spaces (Zhang et al., 2020; Jiang et al., 2023), most existing MADRL-based algorithms still formulate their reward functions, action selection, and constraints from the point of view of UDs.

If the server's storage capacity is exceeded, failure to take into account the storage constraint causes some tasks to be abandoned before processing begins. This leads to inefficient performance for three reasons: first, some tasks are dropped; second, the dropping of tasks happens according to their arrival time while a combinatorial decision could improve the performance; and third, the decisions are made using only actor agents that run at the UDs with only local information using an actor agent to coordinate the training only, while coordinated agents at the UDs and the servers could have made better performance using combinatorial decision. There is no existing algorithm that addresses this. Existing MARL algorithms apply a penalty as feedback to learn the QoS constraints of the UDs, but do not consider the resource constraint at the server that affects the volume of UDs that their request can be served on. To address this limitation, MADRL must deploy heterogeneous agents in the UDs and on the servers to make combinatorial decisions in a client-master configuration so that an optimal combination of tasks should be offloaded considering the QoS requirements of the tasks, the resources at the UDs and servers, the communication resource, and the storage constraint of the server. In addition to selecting an optimal combination of tasks, there should be communication between the agents at the UDs and the servers in a client-master setting about their decisions to avoid dropping of tasks, unlike existing approaches where either the UDs or the servers are the decision-making entities.

The main contributions of this work are fivefold as follows.

- By combining the advantages of policy gradient and value function, we proposed a novel combinatorial client-master MADRL (CCM_MADRL) algorithm for task offloading in MEC (CCM_MADRL_MEC) with continuous-valued constraints on the UDs and discrete constraints such as the server storage constraints and number of communication channels on the wireless network. Client agents are deployed at the UDs to decide their resource allocation, and a master agent is deployed at the server to make combinatorial decisions based on the actions of the clients. The constraints of the UDs are considered as a penalty in the reward of the client agents, whereas the channel and storage constraints are considered in

the combinatorial decision of the master agent. The master agent is adapted from the coalition action selection algorithm in Chapter 3 and the per-action DQN.

- By avoiding the number of channels from the state and action space, and considering it as a constraint in the combinatorial action selection, we reduced the dimensionality of the DRL algorithm of the client agents.
- This is the first DRL-based task offloading algorithm to consider combinations of continuous and discrete resource constraints on the UDs, the communication channel, and the server.
- We develop different heuristic benchmarking methodologies for the proposed algorithm and perform an exhaustive numerical analysis to determine the efficacy of the proposed algorithm.
- We conducted extensive experiments on the learning rates for the client and the master agents and evaluated them in different experimental settings.

This chapter is organized as follows. Section 5.2 reviews the related work, and Section 5.3 follows the description of the system model. The formulation of the problem, the CCM_MADRL algorithm, and the numerical evaluation are presented in Section 5.4, Section 5.5, and Section 5.6 respectively. Section 5.7 concludes the CCM_MADRL_MEC.

5.2 Related Works

This section reviews the related work from two perspectives: task offloading and combining the policy gradient and value function in DRL.

5.2.1 Task Offloading Algorithms

Existing MADRL-based task offloading algorithms (Nguyen et al., 2023; Zhang et al., 2020; Jiang et al., 2023) use homogeneous agents where only UDs or edge servers make decisions. Nguyen et al. (2023) assumed there is enough resources on the MEC server. Zhang et al. (2020) considered the available computational resource on an MEC server should be shared with users and that a common completion time is computed on the server, but did not consider the different arrival times of tasks on the server. Xiong et al. (2023) considered the computational resource constraint and the number of computing units on the server. They also compute different start and end times for the tasks. However, their action spaces are limited to prioritizing tasks and making offloading decisions accordingly. They did not consider the allocation of resources and power. The communication resource constraint is also not considered.

There exist DRL algorithms that have considered the storage constraint on the MEC server for service placement problems (Lu et al., 2022). Some works also consider a three-stage task offloading decision that includes processing tasks locally in UDs, offloading to the MEC server, and offloading to MCC (Chen et al., 2023). However, existing DRL algorithms do not consider discrete resource constraints, such as the storage capacity on the MEC server and the number of communication channels. The number of channels is included in the action space by Nguyen et al. (2023) and Jiang et al. (2023) but it is not necessary to include it, as it worsens the dimensionality. The dimension of the action space in the task offloading can be reduced by restricting each channel to be used by only one UD at a time. A channel can be reused by multiple UDs one after the other. Because the channels are equal, it does not matter which channel a UD uses. Our proposed technique for this approach is described in detail in Section 5.5.2. However, none of the DRL-based task-offloading algorithms considered the server storage constraint.

5.2.2 Combining Policy Gradient and Value Function in Deep Reinforcement Learning Algorithms

Many MADRL algorithms are homogeneous agents that can be classified as a policy gradient or a value function. As discussed in Section 2.2.2.3, policy gradient approaches are good at directly optimizing policies to maximize cumulative rewards, but they may converge to local optima. On the other hand, value function techniques have the potential to converge to optimum action values, but they can encounter difficulties in high-dimensional or continuous action spaces. DDPG finds a compromise by combining both the policy gradient and a value function. However, the latter serves only to provide feedback for policy training and is useless after the training is completed.

DRL algorithms with a mix of policy gradient and value function have been used in existing works for purposes different from the objective of our proposed algorithm. Dulac-Arnold et al. (2015) proposed a combination of the policy gradient and the value function to cope with the dimensionality of large discrete action spaces. The actor produces a continuous-value action, and the critic selects discrete actions from the K action spaces closest to the continuous action. Xiong et al. (2023) proposed a two-step decision in which the actor prioritizes the tasks and selects the best, and then the critic makes a decision on whether local processing or MEC processing is performed on the task selected by the actor agent, but both the actor and the critic are policy optimization techniques. A hybrid actor-critic MADRL is proposed by Zhang et al. (2020) and Jiang et al. (2023) to make coordinated decisions about discrete and continuous actions of UDs only. That is, the actors decide the continuous actions such as computational resource and power allocation, whereas the critic decides the discrete actions such as channel selection and server selection. However, none of the DRL algorithms with

mixed policy gradient and value function techniques is used to select a combinatorial or arbitrary number of actions.

In summary, existing DRL algorithms consider homogeneous agents with homogeneous constraints. No DRL algorithm considers the various discrete and continuous constraints in the UDs and on the server, including storage constraints. In addition, there is no MADRL algorithm for arbitrary combinatorial action selection. In this work, the benefits of both the value function and the policy gradient are leveraged to design robust MADRL that mitigates converging to local optimal while at the same time reducing the dimensionality caused by the combination of continuous and discrete action spaces. The server makes a combinatorial decision as to which of the clients should offload their tasks.

5.3 System Model

This section considers MEC for task offloading, which mainly includes a base station (BS), UDs, tasks, energy harvesting, and wireless networks. We consider a multi-user MEC scenario shown in Figure 5.1. In this scenario, there is a single wireless BS equipped with an MEC server that provides a computing and storage service and an SDN controller that controls communication between UDs and BS. BS serves a set of $N = \{1, 2, 3, \ldots, N\}$ UDs. A single UD in the set N is denoted by n. For local processing, we consider that each UD n has a minimum and maximum computational resource allocation budgets denoted by f_n^{min} and f_n^{max}, respectively, in gigahertz (GHz) cycles per second, where $f_n^{min} = f^{min}$ for all UDs, but they have different f_n^{max} uniformly generated with $f_n^{max} | f^{min} \leq f_n^{max} \leq f^{max}$. Similarly, to offload its task to the server, we consider that each UD has a minimum and maximum transmission power allocation threshold denoted by p_n^{min} and p_n^{max}, respectively, in dBm, where $p_n^{min} = p^{min}$ for all UDs and their maximum transmission powers are generated uniformly from $p_n^{max} | p^{min} \leq p_n^{max} \leq p^{max}$. Furthermore, we consider the UDs to have a minimum battery threshold $b_n^{min} = b^{min}$ and maximum battery capacities in the range of $b_n^{max} | b^{min} \leq b_n^{max} \leq b^{max}$ respectively in Megajoules (MJ). The BS has multiple constraints and characteristics, such as the server storage constraint z_e in bits and the number of processing units on the server U_e, each having an equal processing capacity of f_e in gigahertz cycles per second. The communication network has a bandwidth of W in megahertz that is equally divided between K channels. The terms commonly used for the mathematical equations in this chapter are presented in Table 5.1.

We consider a task-offloading problem for T time steps of τ_{max} length each. It is assumed that each UD n generates one task at each time step. If the processing of a task is not completed in τ_{max}, it is discarded before the next time step starts. The detailed

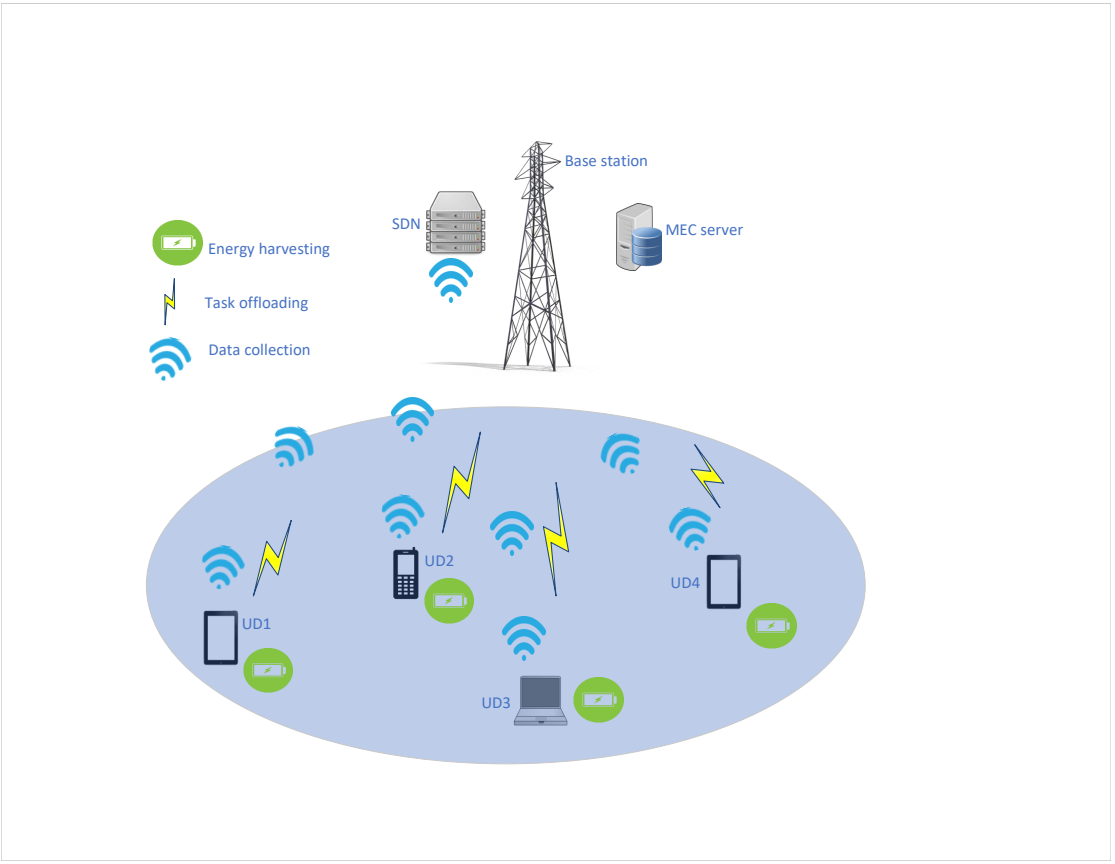


FIGURE 5.1: Network model

operational model, the task model, the processing model and the energy harvesting are described in the following sections.

5.3.1 Operational and Deployment Model

The CCM_MADRL_MEC involves heterogeneous DRL agents, called client agents, and a master agent. Client agents are used to decide the allocation of resources for the UDs. The master agent is used to make combinatorial decisions for the UDs that propose their tasks to be considered in the combinatorial decision. The operational model of the CCM_MADRL_MEC is presented in Section 5.5.2 for the two-step execution of the client agents and the master agent to produce the actions. Client agents are trained centrally on the server but can be executed in a decentralized setting in the UDs or centrally on the server. Therefore, the setting of the proposed MADRL is centralized training. However, it can be deployed as a decentralized execution as in Nguyen et al. (2023) or as a centralized execution as in Xiong et al. (2023).

The task-offloading model is designed by combining the merits of different existing works. Since Nguyen et al. (2023) has used a data set from Huawei Technologies, we

adapted it ignoring the blockchain part. However, Nguyen et al. (2023) does not consider the energy harvesting technique of the MEC system. Therefore, we consider the energy harvesting process in Zhang et al. (2020). Nguyen et al. (2023) considers concurrent processing on the server with independent completion time. Since the server cannot have as many processors as the number of tasks, we specify a limited number of processing units similar to the work of Xiong et al. (2023). The work of Zhang et al. (2020) is also not efficient in computing the completion time of individual tasks, since it uses the sum of the maximum completion time of the tasks and the common processing latency for the completion of all tasks. For this reason, we take advantage of the efficient computation of the completion times of tasks by Xiong et al. (2023), which computes the completion time of tasks on the server based on the completion time of other tasks scheduled before them. Similarly, we assume that the server processes the tasks in the order of their arrival on the server. The arrival times of the tasks are determined by their offloading times. In the following, we present the task and computing model for the task offloading problem.

5.3.2 Task Model

The task model is based on the setting in the work of Nguyen et al. (2023) on a data set from Huawei Telecom. At each time step, each UD n generates a task denoted by its notation as n[1] that is represented by characteristics such as the size of the task z_n in bytes, the number of CPU cycles per bit required to process the task c_n, and the maximum deadline τ_n to which task processing is expected to finish.

Before processing the task, there are three decision variables: a binary decision of whether to process it locally or offload it to the MEC server x_n, a local resource allocation f_n, and a transmission power allocation p_n, which are described in detail in Section 5.5.2. So, we make a binary decision $X = \{x_n | n \in N\}$ to describe the processing mode, as seen in Equation (5.1).

$$x_n = \begin{cases} 1, & \text{MEC processing} \\ 0, & \text{Local processing} \end{cases} \tag{5.1}$$

Next, we present the local and MEC models.

5.3.3 Local Processing

A task is processed locally if one of the following happens as presented in Section 5.5.2: if the UD decides to process the task locally; if a UD proposes the task to the master

[1]Because a UD has one task at a time step, we use n to denote both the UD and its task to reduce the number of notations

Notation	Description
N	Set of UDs
T	Number of time steps
τ_{max}	Maximum length of a time step
n	A UD or a task of the UD
T_n	Total latency of processing task n
x_n	A binary indicator of local processing or offloading for the task n
E_n	Energy consumption of task n
E_{loc_n}	Energy consumption of task n in local processing
E_{off_n}	Energy consumption of offloading task n
T_{loc_n}	Computation time for the local processing of the task n
T_{off_n}	Offloading time of task n
λ_1 and λ_2	weight coefficients of T_n and E_n
b_n	Battery level of US n
p_n	Transmission power allocation of the UD n
z_n	Size of task n
z_e	Storage capacity of the server
U_e	Number of processing units in the server
L_n	Cost of processing task n
R_n	Reward of processing task n
f_n	Resource allocation for local processing of the task n
C_n	Number of CPU cycles to process one bit of task n
α_ϕ	Learning rage of the master agent
α_θ	Learning rage of client agent
β	Soft target update
J	Joules

TABLE 5.1: List of notations for Chapter 5

agent and the master agent decides that the task should be processed locally. Then, the UD processes the task using its local computational resource assigned to its task, which is restricted within its own resource allocation budget as $f_n | f_n^{min} \leq f_n \leq f_n^{max}$. The local computing latency to process the task is computed as follows:

$$T_{\text{loc}_n} = \frac{z_n c_n}{f_n} \tag{5.2}$$

The energy consumption in the local processing mode is calculated based on the size of the task and the allocation of resources for processing the task, as shown in Equation (5.3).

$$E_{\text{loc}_n} = \kappa z_n c_n (f_n)^2 \tag{5.3}$$

where κ is energy consumption coefficient (Zhang et al., 2020).

5.3.4 MEC Processing

In this mode, the task is transferred to the MEC server to be processed by one of the processing units U_e of the server. The decision happens when the UD proposes the task to be processed and the master agent approves it. To be processed on the server, the task needs transmission resources, which is a function of transmission power p_n and other network parameters such as bandwidth and channel gain. The transmission power p_n is decided by the UD from its transmission power budget $p_n | p_n^{min} \leq p_n \leq p_n^{max}$ as discussed in Section 5.5.2. Then, the data transmission rate d_n in a single channel of the wireless network is calculated using Shannon's capacity as

$$d_n = \frac{W}{K} \log_2 (1 + p_n g_n) \tag{5.4}$$

where W is a constant coefficient which is the bandwidth of the wireless network, K is the number of channels, p_n is the transmit power of UD n, and $g_n = h_n / \sigma^2$ is the normalized channel gain of the uplink channel between UD n and the BS, with channel gain h_n and the background noise variance σ^2. We did not consider interference between multiple UDs because we assume that a channel is used by one task at a time.

Once the data transmission rate is determined, the transmission time T_{off_n} and energy consumption E_{off_n} of offloading task n to the server can be computed as:

$$T_{\text{off}_n} = \frac{z_n}{d_n} \tag{5.5}$$

$$E_{\text{off}_n} = p_n T_{\text{off}_n} \tag{5.6}$$

Like many works on task offloading (Wang et al., 2022; Nguyen et al., 2023; Zhang et al., 2020), we assume that the communication resource required to return the information about the processed task to the UD is negligible because only analytical information is returned, not the entire task.

Note that the energy consumption in task offloading is computed only for the UDs as they are battery-powered. The energy consumption on the server is not a concern of the task offloading problem. However, the latency of processing the tasks on the server matters because the tasks have deadline constraints. Therefore, the total latency of processing a task on the server is determined by the transmission time, the earliest availability of the processing unit on the server, and the time required to process the task on the server. The processing time of task n in one of the processing units in the server is computed as:

$$T_{\text{ser}_n} = \frac{z_n c_n}{f_e} \tag{5.7}$$

However, the processing of the task on the server does not start as soon as the task has arrived at the server. The processing units on the server process one task at a time. The tasks that are offloaded to the server are processed in the order of their arrival at the server, which is determined by T_{off_n}. Tasks are assigned to the earliest free processing unit. Therefore, the start of processing the task n depends on the earliest availability of a processing unit, which is determined by the number of processing units on the server U_e, and T_{ser_n} and T_{off_n} of other tasks that are shorter T_{off_n} than that of task n. Accordingly, the total latency of the offloading task n to the MEC server T_{MEC_n} is computed as:

$$T_{\text{MEC}_n} = T_{\text{ser}_n} + Max(T_{\text{off}_n}, T_{\text{ear}_n}) \tag{5.8}$$

where T_{ear_n} is the estimated availability time of the first available processing unit U_e of the server after the arrival of task n and $Max(.)$ ensures that task processing starts when a free processing unit is found after task offloading is completed.

The T_{ear_n} is calculated based on the completion time of other accepted tasks on the server with the earliest offloading time than that of task n. T_{ear_n} is reset to 0 at the beginning of each time step. Therefore, for the first U_e tasks that are offloaded to the server, their T_{ear_n} is 0 because all processing units are available. For the rest of the offloaded tasks, the computation of their T_{ear_n} depends on the completion time in the processing units. This estimate is adapted from the work of Xiong et al. (2023). However, unlike Xiong et al. (2023) which has used it in the state space of the DRL algorithm, in this work, T_{ear_n} is only used by the processing units in the server to schedule the processing of the tasks and compute the latency of the task that is used to calculate the reward. The client agents in CCM_MADRL_MEC make decisions using only the local state. The master agent agent also uses only the combined states and actions of the clients.

5.3.5 Energy Harvesting

The energy harvesting process is adapted from the work of Zhang et al. (2020). For simplicity, we assume that the UDs harvest e_n energy at the beginning of each time interval. Initially, each UD is full with a maximum battery capacity of b_n^{max}. In this work, we assume that only the energy consumption of local computation and transmission power affects the energy consumption and ignore others for simplicity. Therefore, the level of the battery in the next time interval depends on both the energy consumption and the harvesting, which evolves according to the following equation in the T time steps.

$$b_n(t+1) = Min(Max(b_n(t) - E_n(t) + e_n(t), 0), b_n^{max}) \tag{5.9}$$

where E_n is the energy consumption calculated based on E_{loc_n} and E_{off_n} as described in Section 5.4.1 and $Min(.)$ and $Max(.)$ ensure that the level of the battery cannot be negative and does not exceed the maximum capacity.

5.4 Problem Formulation

5.4.1 Processing Cost

For the task processing, the total processing latency and energy costs are equal to

$$T_n = (1 - x_n)T_{\text{loc}_n} + x_n T_{\text{MEC}_n} \tag{5.10}$$

$$E_n = (1 - x_n)E_{\text{loc}_n} + x_n E_{\text{off}_n} \tag{5.11}$$

Considering that the cost of processing a task is collectively determined by its energy consumption and latency, the cost function for processing a task is specified as follows:

$$L_n = \lambda_1 T_n + \lambda_2 E_n \tag{5.12}$$

where λ_1 and λ_2 are weight coefficients for the latency and energy consumption.

5.4.2 System Cost Formulation

The CCM_MADRL_MEC is aimed to solve the optimization problem that can be formulated as the cost minimization for all UDs and T time steps while meeting the different constraints in the UDs and the server as follows:

$$\underset{\{x_n, p_n, f_n\}}{\text{minimize}} \quad \sum_{t}^{T} \sum_{n \in N} L_n^t \tag{5.13a}$$

$$\text{subject to} \quad x_n \in \{0, 1\}, \quad \forall n \in N \tag{5.13b}$$

$$p_n^{\min} \leq p_n \leq p_n^{\max}, \quad \forall n \in N \tag{5.13c}$$

$$T_n \leq \tau_n, \quad \forall n \in N \tag{5.13d}$$

$$b_n \geq b_n^{min}, \quad \forall n \in N \tag{5.13e}$$

$$f_n^{\min} \leq f_n \leq f_n^{\max}, \quad \forall n \in N \tag{5.13f}$$

$$\sum_{n \in N} x_n \leq K \tag{5.13g}$$

$$\sum_{n \in N} x_n z_n \leq z_e \tag{5.13h}$$

where each constraint is explained as follows:

- Equation (5.13b) implies that a task is processed locally or uploaded to the MEC server.
- Equation (5.13c) indicates that the transmission power should be between $p_n^{\min}$ and the maximum value $p_n^{\max}$.
- Equation (5.13d) implies that the processing time of each task cannot exceed its processing deadline.
- Equation (5.13e) guarantees that the battery level should not exceed the low battery level.
- Equation (5.13f) ensures that the local computational resource allocated to each task should be in the preset minimum and maximum values.
- Equation (5.13g) and ensure that the number of offloaded tasks does not exceed the number of transmission channels by ensuring that only one task uses a channel. It is used if and only if it is necessary to use only one channel for one user as used by Kan et al. (2018).
- Equation (5.13h) guarantees that the sum of the sizes of the off-loaded tasks does not exceed the storage capacity of the server.

5.5 Combinatorial Client-Master MADRL Algorithm for Task Offloading in MEC

To solve the optimization problem of the cost minimization in Equation (5.13a), we convert the optimization problem into a reward maximization problem and apply our proposed combinatorial client-master MADRL (CCM_MADRL) algorithm for MEC with server resource constraints (CCM_MADRL_MEC). The states, client and master actions, and the formulation of the reward function are presented as follows.

5.5.1 State

The state $S(t)$ of the MEC environment includes the set of states of the UDs as shown in Equation (5.14):

$$S(t) = \{S_n(t)\}, \quad \forall n \in N \tag{5.14}$$

Constant values such as the number of channels K, the number of processors on the server U_e, the processing capacity f_e, the energy harvesting rate, and the storage capacity z_e of the server are excluded from the state space. The state of a UD $S_n(t)$ is characterized by five components: task state $S_n^{\text{task}}(t)$, channel gain state $S_n^{\text{channel gain}}(t)$, power transmission state $S_n^{\text{pow}}(t)$, battery state $S_n^{\text{battery}}(t)$, $S_n^{\text{deadline}}(t)$ and local resource allocation state $S_n^{\text{res}}(t)$ as defined in Equation (5.15). The energy harvesting rate of a UD can be included in the state space if it is dynamic.

$$S_n(t) = \{S_n^{\text{task}}(t), S_n^{\text{gain}}(t), S_n^{\text{pow}}(t), S_n^{\text{res}}(t), S_n^{\text{battery}}(t)\} \tag{5.15}$$

where each component is explained as follows.

- Task state of a UD: $S_n^{\text{task}}(t) = [z_n(t), c_n(t), \tau_n(t)]$. $z_n(t)$ represents the task size and $c_n(t)$ represents the CPU cycle required to compute the task, and $\tau_n(t)$ is the time required to process the task.
- Normalized channel gain state: $S_n^{\text{gain}}(t)$ which is $g_n = h_n/\sigma^2$ for every UD n. The channel gain h_n is impacted by many factors, including distance. For simplicity, we assume that the UDs are stationary and have a stationary normalized channel gain depending on their distance from the BS. The variance of background noise σ^2 is also constant.
- Power transmission state: $S_n^{\text{pow}}(t)$ is specified as $p_n(t)$, which represents the transmission power allocated to the UD and its value is in the range of $[p_{\min}, p_{\max}]$.

- Local resource allocation state: $S_n^{\text{res}}(t) = f_n(t)$, where $f_n(t)$ is the local computational resource allocated to UD n.
- Battery level: $S_n^{\text{battery}}(t)$ as described in Equation (5.9).

5.5.2 Action

At the beginning of each time step, the UDs make decisions about their resource allocations using client agents. Then, the SDN controller collects information about the state and action of the UDs and makes the following decision using a master agent: 1) for the UDs that decide to make a local allocation, the server does not interfere. 2) if the number of UDs that propose to offload their tasks is greater than the number of channels or if the sum of the size of their tasks is greater than the storage capacity of the server, the server makes a combinatorial decision on which of the requests of the UDs to approve and which of them to reject. 3) If the proposed requests are less than the constraints, the server accepts all of them. Finally, channels are assigned to the accepted UDs, and eventually the transmission of the tasks to the server and processing starts.

Existing DRL-based task offloading algorithms, such as the work of Nguyen et al. (2023) and Jiang et al. (2023), included the number of channels in their state and action spaces. However, the channels have equal transmission capacity from the perspective of a UD as seen in Equation (5.4). If we restrict that a channel is used by only one UD at a time, it does not matter which channel a UD uses. Therefore, the inclusion of channels in the state and action space incurs a dimensionality problem without playing any significant role. We excluded channel information from the state and action spaces of the DRL agents of the UDs and considered them as a constraint in the combinatorial action selection of the server. If the number of UDs is greater than the number of channels, the master agent makes the combinatorial decision, and then the SDN controller assigns one UD to one channel. There is a similar work by Kan et al. (2018) even if it is not a DRL algorithm. Note that a channel is used by one UD at a time. However, a channel can be reused by multiple UDs one after the other. In such a case, only the storage capacity of the server becomes the constraint in combinatorial action selection.

The action for continuous-valued power and computational resource allocation and the action for combinatorial decision for the storage and number of communication channels are processed in a two-step execution, as described below. First, the client agents decide the computational resource and power allocation and whether their task should be processed locally or proposed to the master agent for consideration of offloading to the server. For client agents which decide to process their tasks locally, the decision will be final. And for those which proposed their tasks for the master agent to decide, the master considers them in its combinatorial decision. The actions of the client agents and the master agent are as follows.

5.5.2.1 Client Actions

After observing the state of the system, each UD produces three actions using the policy gradient: the decision about the mode of task processing $x_{c,n}(t)$, the allocation of transmission power $p_{c,n}(t)$, and the local allocation of computational resources $f_{c,n}(t)$, which are all continuous value actions between [0, 1] inclusive. The action space can be expressed as:

$$A_c(t) = x_{c,n}(t), p_{c,n}(t), f_{c,n}(t) \leftarrow \theta_n(S_n(t)), \quad \forall n \in N \tag{5.16}$$

where $S_n(t)$ is the state of UD n as described in Equation (5.15) and θ_n is the parametrized policy of the UD. The corresponding usage of each component is as follows:

- Task offloading decision $x_{c,n}(t)$: If $x_{c,n}(t) < 0.5$, the binary decision in Equation (5.1) becomes 0, and the task is processed locally. Otherwise, the task is proposed to the master agent to be considered for the combinatorial decision with other tasks from other UDs.
- Transmission power allocation $p_{c,n}(t)$: Decide on the transmission power p_n using Equation (5.17).
- Local allocation of computational resources $f_{c,n}(t)$: Decide on the computational resource f_n using Equation (5.18).

Based on the outputs of the client agent, p_n and f_n are determined as:

$$p_n = Max(p_n^{min}, p_{c,n}(t)p_n^{max}) \tag{5.17}$$

$$f_n = Max(f_n^{min}, f_{c,n}(t)f_n^{max}) \tag{5.18}$$

5.5.2.2 Master Action

For the client actions with $x_{c,n}(t) \geq 0.5$, the master agent takes the combinations of states and actions of the client agents and provides a binary output for the combinatorial decision on which of them should be allocated locally and which of them should be accepted for processing by the MEC server.

$$A_m(t) = x_{m,n}(t) \leftarrow \phi(S, A, S_c, A_c), \quad \text{for all } n \in N \text{ such that } A_{c,n} \geq 0.5 \tag{5.19}$$

where S and A are the set of states and actions of all client agents, and S_c and A_c are the set of states and actions of the client agents whose $A_{c,n} \geq 0.5$ and ϕ is the parametrized policy of the master agent.

The master agent is a modification of the critic in classical MADDPG algorithms by leveraging the coalition action selection of Chapter 3. Unlike the coalition action selection presented in Chapter 3, which uses the transformer neural network, the master agent uses the per-action DQN as introduced in Chapter 4 excluding the stationary transformation function. The stationary transformation function is excluded for the simplicity of benchmarking-with and customization of MADDPG as in the following sections.

5.5.3 System Reward Function

The system reward function must represent the objective function and the constraints. To compute the reward, we use the negative of the objective function, and we compute a penalty function for the time and energy constraints of the users. The constraints of the server storage capacity and communication channels are already considered in the action selection and do not need to be included as a penalty. Zhang et al. (2020) included a drop-off penalty in their reward for running out of batteries. In ours, we start the penalty from a preset minimum battery threshold.

$$L'_n = \lambda_1 Min((\tau_n - T_n), 0) + \lambda_2 Min((b_n - b_n^{min}), 0) \tag{5.20}$$

Since the design of our system formulation is based on the cost minimization problem in Equation (5.13a), our system reward function is equal to the negative of the system cost function and the penalty function. Thus, we can formulate the reward function of the system as follows.

$$\begin{aligned} \bar{r}(S(t), A(t)) &= -\frac{1}{|N|} \sum_{n \in N} (r(S_n(t), A_n(t)) - r'(S_n(t), A_n(t))) \\ &= -\frac{1}{|N|} \sum_{n \in N} (L_n(t) - L'_n(t)) \end{aligned} \tag{5.21}$$

5.5.4 Long Term Maximization

The power and resource allocation decisions made in the current step by the UDs affect their operational life in the next time steps by affecting energy consumption. Therefore, the DRL must consider immediate reward and long-term return using the Bellman equation as shown in Equation (5.22)

$$Q(S, A|\phi) = (1 - \alpha_\phi) Q(S, A) + \alpha_\phi (R(S, A) + \gamma \sum_n P(S'|S, A) \max_{S'_n, A'_n} Q(S', A', S'_n, A'_n|\phi')) \tag{5.22}$$

where α_ϕ is the learning rate, γ is the discount factor, ϕ is the policy of the critic, ϕ' is the policy of the target critic to benefit from DDQN as discussed in Section 2.2.3, $S = \{S_1, ..., S_N\}$ and $S' = \{S'_1, ..., S'_N\}$ are combined current and next states, $A = \{A_1, ..., A_N\}$ and $A' = \{A_1, ..., A'_N\}$ are combined current and next actions of the client agents, and S'_n and A'_n are the corresponding states and actions of the agents. The role of the four parameters in Q' is presented in the following section.

5.5.5 A Master Agent with Per-client DQN

DQN has the potential to converge to the optimal value because Q-learning is theoretically proven to converge with a probability of 1 (Watkins and Dayan, 1992). Therefore, the critic in MADDPG knows the best combination of actions that leads to the best reward. However, existing critics in MADDPG are only used to give feedback to actors when training. The convergence of the actors is not guaranteed. In our work, the CCM_MADRL algorithm can produce better convergence because the master is used not only to give feedback to the clients but also to choose the best combination of actions from the proposed actions. In MADDPG, there is only a single Q-value for the combined state and action pair of all actors, which is calculated as:

$$Q(S, A) = (1 - \alpha_\phi)Q(S, A|\phi) + \alpha_\phi(R(S, A) + \gamma Q(S', A'|\phi')) \tag{5.23}$$

If we want to customize the critic to select actions, it should be able to provide a Q-value per client agent. One possible solution is the decentralized actor-critic, where each actor has its critic which uses the state and action of the actor to output the Q-value. However, using the Q values of a decentralized actor-critic for making action selections will lead to unstable learning because they use different weights for the DNNs. In other words, the Q values are provided by different DNNs.

The master agent with per-client DQN in the CCM_MADRL algorithm adapts the concept of per-action DQN (He et al., 2015). It is also applied to a single agent DRL with coalition action selection for online combinatorial resource allocation problems with an arbitrary action space in Chapter 4. In this work, the state S_n and the action A_n of each agent are appended to the combined state and the action of the actors to calculate the relative Q value in the combination of the state and the action as:

$$Q(S, A, S_n, A_n|\phi), \quad \forall n \in N \tag{5.24}$$

Applying the per-client DQN to MADDPG without changing the training function will not change the performance of MADDPG, because the per-client critic will have the same Q value for all S_n and action A_n. However, in the CCM_MADRL algorithm, the combined rewards are given only to selected clients in the task-offloading problem, as

seen in Algorithm 6 so that they will have different Q values to distinguish them in action selection. Therefore, the usual Bellman equation for MADDPG is customized to consider the relative Q values of the clients, as seen in Equations (5.22) and (5.24). Therefore, CCM_MADRL is a customized version of MADDPG with a per-client DQN. The third and fourth parameters are included to determine the relative Q-value of an agent.

5.5.6 Combinatorial Action Selection

If the server resource constraint is not met, the proposed CCM_MADRL algorithm is not different from the MADDPG because all clients will be selected and all will receive the combined reward. Therefore, their relative Q-values will be the same. When the demands of the clients exceed the resource constraints, the CCM_MADRL algorithm has to select the best combination for the tasks of the client agents. The reward will also be shared with the set of selected clients. This leads to different Q-values in Equation (5.22).

In addition to combinatorial action selection, the master agent with per-client DQN has three advantages: 1) reduction of dimensionality, 2) handling arbitrary action space, and 3) avoiding non-stationarity. Clients decide their resource requirements and whether to make a local allocation or propose their tasks to the master agent for a combinatorial decision. Then, only the clients that choose to offload their tasks to the server are considered in the action space of the master agent. The consideration of only the proposed clients reduces the dimension of the action space and enables the selection of an arbitrary number of clients in an approach similar to the coalition action selection in Chapter 4. In training, the master agent learns with the combined reward of all agents and applies it to the actions it accepted for processing on the server. Note that the reward is computed from all clients, including those processed locally, but the master applies it only to those selected for server processing. This helps the server distinguish which of the clients were accepted, but it does not affect the clients that chose to process locally because they receive the feedback at training as classical MADDPG. In other words, even though the master applies the reward to the selected clients, the same feedback is given to all clients. The feedback is found from the Q value of one of the selected actions because they are trained with the same target as shown in Equation (5.22). If no client agent has offloaded its task, the master agent is trained with a zero-valued space holder as a fake client agent to learn the combined reward of the local allocation so that it can use it to provide feedback. In other words, for the UDs that choose to make a local allocation, the master agent only provides feedback. For the UDs that proposed to offload their task to the server, the master agent provides feedback and also makes decisions on the proposed actions.

Regarding avoiding non-stationarity, MADDPG with a centralized critic is not subject to non-stationarity because the reward is given to the centralized critic. Note that, unlike the classical critic network, the master agent in CCM_MADRL_MEC participates in the action selection and employs exploration and exploitation, which is one of the challenges causing non-stationarity in MADRL. Exploration actions become challenging, as the reward received depends on the decision of others. This results in agents collecting non-stationary rewards to their replay memory at different times for the same states and actions. Two-step MARL algorithms where both agents use a reward for training all agents, such as the work in Xiong et al. (2023) are more susceptible to non-stationarity if they use a replay memory. Xiong et al. (2023) excluded the use of replay memory and trained their algorithm as a series of epochs on the data. However, this is not a convenient way to train DRL algorithms when the data set is not available and must be collected using interaction with the environment. Furthermore, it cannot take advantage of the prioritized experience replay (Schaul et al., 2016) and long-term returns of the Q values of the next states, since it only learns from immediate rewards. The CCM_MADRL_MEC is not prone to non-stationarity caused by exploration and exploitation because the reward is used only by the master, and the clients receive the best feedback from the master agent no matter whether exploration or exploration is used in the clients and the master. Client agents do not use rewards. They are trained only on the basis of the feedback from the master agent. This centralized use of reward makes non-stationarity not a concern in our CCM_MADRL_MEC.

Figure 5.2 shows the interaction diagram of the CCM_MADRL algorithm and the MEC system. Client agents represent the policies of the UDs. The master agent represents the policy on the MEC server. The environment represents the allocation of resources in the UDs and on the server. After a client produces its output, it does the following as mentioned in Section 5.5.2: if $x_{c,n} < 0$ assigns $x_n = 0$ and starts the local allocation. Otherwise, it forwards $x_{c,n}, p_n, f_n$ to the master for the combinatorial decision. Then, the master agent produces the binary decision and applies it to the UDs and the server. Finally, a shared reward is computed and provided to the master agent to train its value function. Client agents are also trained using a TD error computed by the master agent as feedback.

Like the actors in classical DDPG algorithms, the client agents in CCM_MADRL_MEC output continuous actions, but x_n should be a binary decision. Similarly to the work of Dulac-Arnold et al. (2015), we can use the continuous output of the client agents. Then, those below 0 will be assigned 0 in the binary decision of x_n. For values greater than 0, the binary decision is decided by the master agent.

Since the master agent in the CCM_MADRL_MEC algorithm has two functions: providing feedback for training the clients, similar to the MADDPG, and participating in the combinatorial action selection of the clients, it follows different procedures for both.

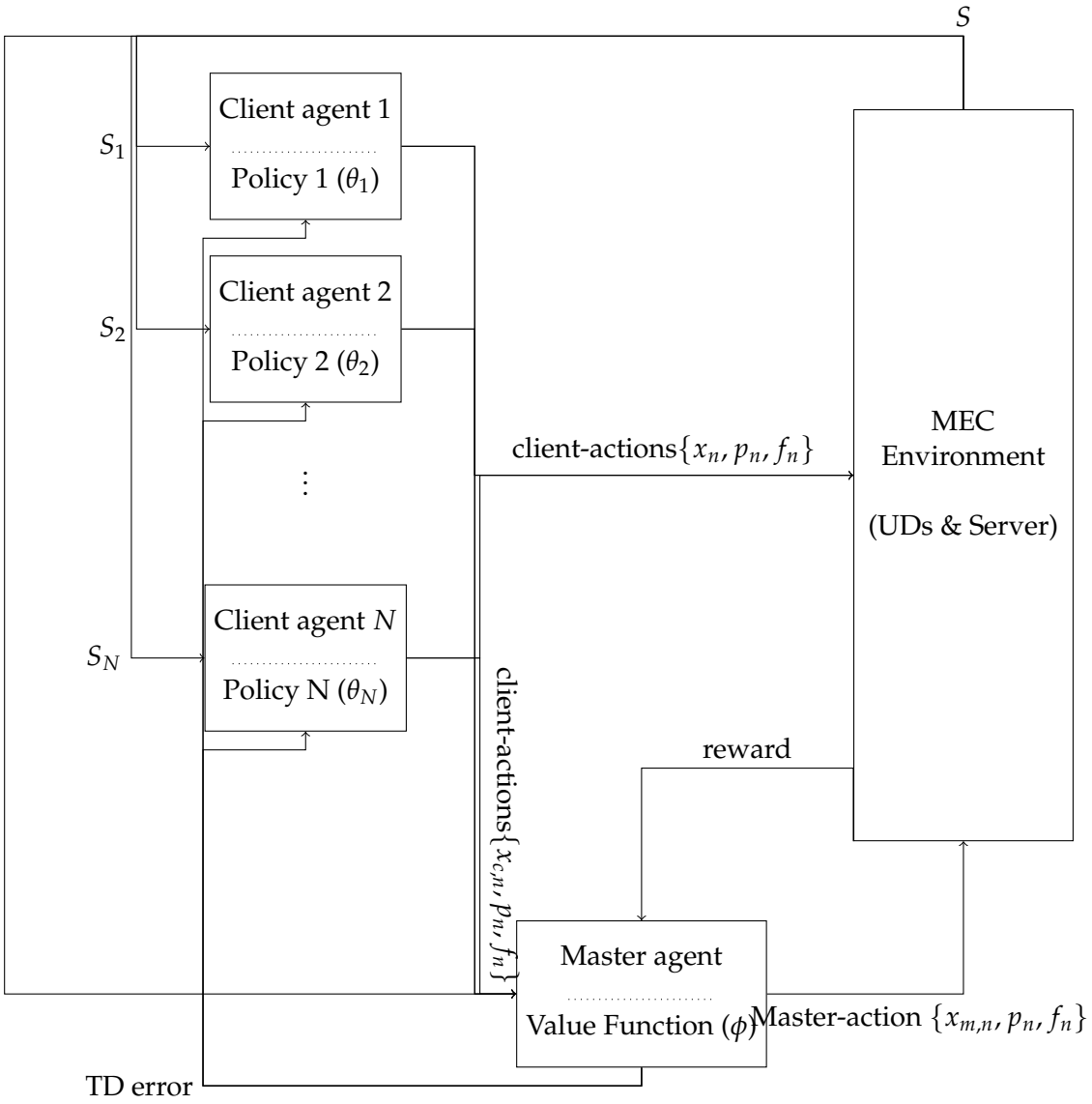


FIGURE 5.2: The interaction diagram of the agents and the MEC environment. Client agents output their actions $\{x_n, p_n, f_n\}$. Clients with $\{x_{c,n} < 0.5\}$ start local processing; and the others propose their tasks to the master agent, which makes the combinatorial decision on which of the proposed tasks should be offloaded and which of them should be designated for local processing

Therefore, the algorithm is presented below in three parts: a main algorithm, an action selection algorithm, and a training algorithm in the following sections.

5.5.7 Algorithms

Since the master agent in the CCM_MADRL_MEC algorithm has two functions: providing feedback for training the clients, similar to the MADDPG, and participating in the combinatorial action selection of the clients, it follows different procedures for both.

For clarity of presentation, the algorithm is provided in three parts: a main algorithm, an action selection algorithm, and a training algorithm in the following sections.

5.5.7.1 Main Algorithm

The main algorithm runs the action selection algorithm, the training algorithm, the training environment, and the evaluation environment. The main algorithm starts by configuring all initial parameters and iterates for $Max\_Episodes$ iterations. In each episode, it runs for T time steps, as seen in lines 7 to 12 of Algorithm 4. For each step, the action selection algorithm is called, the rewards are computed, and then the experience is recorded to replay memory. When the iteration over the steps is complete, the training is called and the trained policies of the client agents and the master agent are evaluated as seen in lines 14 to 21. Evaluation is carried out for each episode using the same evaluation environment to determine the improvement after each call to the training algorithm. The evaluation episodes are seeded into their index so that they are reproduced for each training episode.

Algorithm 4 CCM_MADRL main algorithm

1: Initialize $Max\_Episodes$ = 2000, $Min\_Epsilon$ = 0.01, $Max\_Epsilon$ = 1, γ = 0.99
2: Initialize client agents $\theta_n \quad \forall n \in N$ and the master agent ϕ with random weights
3: Initialize target client agents $\theta'_n \leftarrow \theta_n$ and the target master agent $\phi' \leftarrow \phi, \quad \forall n \in N$
4: Initialize replay memory RM
5: **for** episode $= 1$ to $Max\_Episodes$ **do**
6: Reset environment and get initial state $S_n\left(t=1\right), \quad \forall n \in N$
7: **for** $t = 1$ to T **do**
8: Go to Algorithm 5 using evaluation = False flag to select client and master actions
9: Execute actions and observe total reward $\bar{r}\left(t\right)$ and next state $S_n\left(t+1\right), \quad \forall n \in N$
10: Store transition $\left(S_n\left(t\right), A_n\left(t\right), \bar{r}\left(t\right), S_n\left(t+1\right)\right), \quad \forall n \in N$ in to RM
11: Update the state $S_n\left(t\right) \leftarrow S_n\left(t+1\right), \quad \forall n \in N$
12: **end for**
13: Go to Algorithm 6 for training
14: **for** EvalEpisode in EvalEpisodes **do**
15: Reset and seed episode to EvalEpisode and find state $S\left(t=1\right)$
16: **for** $t = 1$ to T **do**
17: Go to Algorithm 5 using evaluation = True flag to select client and master actions
18: Execute actions and observe total reward $\bar{r}\left(t\right)$ and next state $S_n\left(t+1\right), \forall n \in N$
19: Update the state $S_n\left(t\right) \leftarrow S_n\left(t+1\right), \quad \forall n \in N$
20: **end for**
21: **end for**
22: **end for**

5.5.7.2 Action Selection Algorithm

Because the action selection algorithm has to use exploration in the training environment, we present the computation of ϵ, which is used for ϵ-greedy to determine whether to explore a new action or to exploit the learned knowledge in the master agent and to scale the noise in the client agents as seen in Algorithm 5. In each episode, ϵ is updated using Equation (5.25).

$$\epsilon = Min\_Epsilon + (Max\_Epsilon - Min\_Epsilon) \cdot e^{-\frac{episode}{Max\_Episodes}} \tag{5.25}$$

where $Min\_Epsilon$ and $Max\_Epsilon$ are the minimum and maximum values of the decaying epsilon, $episode$, is the current episode, and $Max\_Episodes$ is the maximum number of episodes.

The action selection algorithm applies the exploration of actions for the client agents and the master agent as follows. Note that the evaluation flag is used to indicate whether the actions are running for the training environment or for the evaluation environment. In the evaluation, no exploration is needed. For client agents, the exploration is performed by adding noise to the actual output of the client agents, as seen in lines 5 to 8 of the Algorithm 5. After adding noise to the actual action, the values are clipped to [-1,1] so that they are within the activation function of the client agents, Tanh in this case. All actions, explored or exploited, are scaled to be between [0,1] before applying to compute the resource allocation in Section 5.5.2.

The master agent follows ϵ-greedy for exploration and exploitation, as seen in lines 41 to 47. First, a random number is generated as seen in line 11 to decide whether to explore or exploit. If the number is less than ϵ, the master agent shuffles the proposed actions as seen in line 45 and follows the combinatorial action selection procedure described below. Otherwise, the master agent computes the Q value based on the states and actions of the proposed actions and appends the Q value along with the identifiers of the tasks n to Qs and $Index$ and follows the combinatorial action selection procedure.

After the actions of the client agents are provided, the master agent follows one of the following three procedures in the exploitation mode, as described in Section 5.5.2. If all client agents decide to process their tasks locally, as in line 18, the master agent does not intervene. Line 20 computes the Q values of the proposed tasks using per-client DQN, and appends them to Qs and $Index$ along with their identifiers n, to be considered in the combinatorial decision. If the number of proposed tasks or the sum of their sizes is less than the number of sub-channels and the storage constraint on the server, the server accepts all of them as seen in lines 24 to 26. If the number of proposed tasks is greater than the number of sub-channels or if the sum of their size is greater than the storage capacity of the server, the master agent uses the Q-values computed using the states and actions of the client agents that proposed to offload their tasks to make decisions.

It starts to approve the proposed actions of the clients with the highest Q-values until the number of sub-channels or the storage constraint is met. The remaining agents are designated to process their tasks locally. The algorithm is provided in Algorithm 5. The procedure of exploitation is provided similarly to the procedure of exploitation except that the proposed actions are shuffled randomly rather than getting sorted by their Q-values.

5.5.7.3 Training Algorithm

The algorithm for training the client agents and the master agent is provided in Algorithm 6. Because the structure of the master agent is different from the MADDPG critic as seen in Section 5.5.5, Algorithm 6 is significantly different from existing MADDPG training algorithms in that: It generates multiple Q-values rather than one combined Q-value, because the master agent has to make a combinatorial decision using the relative Q-values of the clients as seen in lines 7,17, and 33; The client agents are trained by computing the highest Q value from the tasks offloaded to the server as seen in lines 13 and 38; If all tasks are allocated locally by the client agents, the master agent uses *all_zeros* as a placeholder to hold the combined Q-value as seen in lines 11, 22, and 35. Note that after the master agent decides which of the tasks should be processed in the server as described in Section 5.5.2, the reward is applied only to the selected tasks during training. That is, the reward is computed at the system level using Equation (5.21) but it is used only by the tasks that are offloaded to the server when training the algorithm so that the tasks are distinguished by their Q-values in the action selection algorithm.

The notation of the client and master actions is changed in the training algorithm due to the subscript i for the minibatch which is used to iterate over the entries of the minibatch of size M. Unlike MADDPG, which computes the Q-values of the minibatch as a batch, the Q-values of the minibatch in the CCM_MADRL algorithm are computed individually because they are processed conditionally as seen with many if clauses in the algorithm. The training algorithm starts by selecting a minibatch of size M from the replay memory. Each entry in the minibatch includes the combined state S and action A of all client agents S, the set of binary actions of the tasks by the master agent A^{mas}, the total reward of the tasks $\bar{r}$, the combined next state S', and a flag that indicates whether the episode was ended or not, *done*.

The master agent is trained according to lines 2 through 28. Line 2 computes the target action for every client and every entry in the minibatch using their next state. The target action is used to compute the target Q-value using the master agent. Then, lines 3 and 4 concatenate the target actions of the client agents because the master agent accepts a combined state and action of all clients as input as seen in line 7. Lines 5 and 6 check if the client agents have decided to process the tasks locally or propose them

Algorithm 5 The action selection algorithm for the client agents and the master agent

1: Input: state S_n for each client agent n, $\{K, z_e\}$, and Evaluation flag
2: Output: client actions A_n for each n with x_n decided by collaboration with the master agent
3: Get action $A_n \leftarrow \pi_n(S_n, \theta_n), \quad \forall n \in N$
4: **if** Evaluation == False **then**
5: Compute ϵ using Equation (5.25)
6: *noise* = random($|N|$ by $|A_n|$)*ϵ
7: $A_n = A_n + noise_n, \quad \forall n \in N$
8: Clip A_n to [-1, 1]
9: **end if**
10: Scale A_n to [0, 1] using $\frac{A_n}{2} + 0.5, \quad \forall n \in N$
11: Generate a *random* number
12: Qs = [], $Index$ = []
13: **if** $random < \epsilon$ or Evaluation == True **then**
14: $S = \{S_n\}$, $A = \{A_n\}, \quad \forall n \in N$
15: **for** $n \in N$ **do**
16: Get $x_{c,n}$ from A_n as described in Section 5.5.2
17: **if** $x_{c,n} < 0.5$ **then**
18: $x_n = 0$
19: **else**
20: $Append(Qs, Q(S, A, S_n, A_n, \phi))$, $Append(Index, n)$
21: **end if**
22: **end for**
23: **if** $Length(Index) \leq K$ and $Sum(z_n \; \forall n \in N \; and \; x_{c,n} \geq 0.5) \leq z_e$ **then**
24: **for** $\forall n \in N \; and \; x_{c,n} \geq 0.5$ **do**
25: $x_n = 1$
26: **end for**
27: **else**
28: Sort Qs, and adjust $Index$ accordingly
29: $TotalSizeOfAccepted = 0$
30: **while** $Length(Index) > K$ **do**
31: $n = Pop(Index)$
32: $x_n = 1$
33: **if** $TotalSizeOfAccepted + z_n \leq z_e$ **then**
34: $TotalSizeOfAccepted = TotalSizeOfAccepted + z_n$
35: **else**
36: $x_n = 0$
37: **end if**
38: **end while**
39: **end if**
40: **else**
41: Collect the index n of the tasks with $x_{c,n} \geq 0.5 \; \forall n \in N$ to $Index$
42: **if** $Length(Index) \leq K$ and $Sum(z_n \; \forall n \in N \; and \; x_{c,n} \geq 0.5) \leq z_e$ **then**
43: Execute lines 24 to 26
44: **else**
45: shuffle $Index$ in to random order
46: Execute lines 29 to 38
47: **end if**
48: **end if**

Algorithm 6 The training algorithm for the client agents and the master agent

Sample a random minibatch of transitions $(S, A, A^{mas}, \bar{r}, S', done)$ of size M from RM
Set target actions $A'_{i,n} \leftarrow \pi_n\left(S'_{i,n}, \theta'_n\right), \quad \forall n \in N$, and for $i = 1$ to M
$S'_i = \{S'_{i,n}\}, \quad \forall n \in N$, and for $i = 1$ to M
$A'_i = \{A'_{i,n}\}, \quad \forall n \in N$, and for $i = 1$ to M
Get $x_{i,n}$ from $A'_{i,n}$ as described in Section 5.5.2, $\quad \forall n \in N$, and for $i = 1$ to M
if $x'_{i,n} \geq 0.5, \quad \forall x'_{i,n} \in a'_{i,n}, \quad \forall n \in N$, and for $i = 1$ to M **then**
  $Append\left(Q'_N, Q\left(S'_i, A'_i, S'_{i,n}, A'_{i,n}, \phi'\right)\right)$
end if
$Q'_i = Q'_N$, for $i = 1$ to M
if $Length\left(Q'_i\right)$ is 0 for any i **then**
  $nextQ_i = Q\left(S'_i, A'_i, all\_zeros, all\_zeros, \phi'\right)$
else
  $nextQ_i = Max\left(Q'_i\right)$
end if
y = [], Qs = []
if $A^{mas}_{i,n} = 1 \quad \exists n \in N \quad \exists i \in M$ **then**
  $Append\left(Qs, Q\left(S_i, A_i, S_{i,n}, A_{i,n}, \phi\right)\right)$
  targetQ = $\bar{r}_i + \gamma nextQ_i * (1 - done_i)$
  append(y, targetQ)
end if
if $A^{mas}_{i,n} = 0\,, \quad \forall n \in N \quad \exists i \in M$ **then**
  $Append\left(Qs, Q\left(S_i, A_i, all\_zeros, all\_zeros, \phi\right)\right)$
  targetQ = $\bar{r}_i + \gamma nextQ_i * (1 - done_i)$
  $append\left(y, targetQ\right)$
end if
Compute the TD error: $\delta = \frac{1}{Length(y)} \sum_{j=1}^{Length(y)} \left(y_j - Qs_j\right)^2$
Update parameters of master agent ϕ: $\phi \leftarrow \phi + \alpha_\phi \cdot \nabla_\phi \delta$
Update target master network $\phi' \leftarrow \phi$
for each client n **do**
  Q^N_i = [] for $i = 1$ to M, $tarQ$ = []
  Set new actions $A^{new}_{i,n} \leftarrow \pi_n\left(S_{i,n}, \theta_n\right), \quad \forall n \in N$, and for $i = 1$ to M
  $A^{new}_i = \{A^{new}_{i,n}\}, \quad \forall n \in N$, and for $i = 1$ to M
  $Append\left(Q^N_i, Q\left(S_i, A^{new}_i, S_{i,n}, A^{new}_{i,n}, \phi\right)\right), \quad \forall n \in N$ with $A^{mas}_{i,n} = 1$, and for $i = 1$ to M
  if $Length\left(Q^N_i\right)$ is 0 for any i **then**
    $Qloc = Q\left(S_i, A^{new}_i, all\_zeros, all\_zeros), \phi\right)$
    $Append\left(tarQ, Qloc\right)$
  else
    $Append\left(tarQ, Max\left(Q^N_i\right)\right))$
  end if
  Compute the gradient for the client:
  $\nabla_{\phi_n} J(\phi_n) \leftarrow -\frac{1}{Length(tarQ)} \sum_{j=1}^{Length(tarQ)} \nabla_{\phi_n} tarQ_j$
  Update the client parameters θ_n:
    $\theta_n \leftarrow \theta_n - \alpha_\theta \nabla_{\theta_n} J\left(\theta_n\right)$
  Update target client networks $\theta'_n \leftarrow \theta_n$
end for

to the master agent. For each task that was proposed to the master, a relative Q-value is computed on line 7 and the maximum Q-value will be computed in line 13. Line 9 concatenates the Q-values of the offloaded tasks in the same entry of a minibatch. If no task was offloaded, a Q-value will be computed using a placeholder to train the master agent so that it is used to give feedback like the classical MADDPG. Then, the combined reward is provided to the actions selected to offload their tasks as seen in lines 16 to 20. The current and target Q-values in lines 17 and 17 are used to compute the TD error in line 26. As seen in lines 21 to 25, if all client agents, at any entry in the minibatch, decide to process their tasks locally, the master agent concatenates the state and action of all agents and adds *all_zeros* as a placeholder to learn the Q value when all tasks are processed locally, and is only used to provide feedback in training the client agents as seen in line 35.

The training of client agents is seen from lines 29 to 44. They are trained similarly to the training of actors in classical MADDPG except that the feedback is computed differently as seen in lines 33 to 39, because the Q value is provided for the client agents that offloaded their task to the server. Therefore, if one or more clients were offloaded their task, the feedback for training the clients is computed from the Q value of one of the offloaded tasks as they are trained with the same rewards. The maximum Q value of the offloaded tasks is considered for consistency. The calculation of the maximum Q-value is the same as that of the training for the master agent.

We used DDQN (Van Hasselt et al., 2016) and prioritized experience replay (Schaul et al., 2016) for better training efficiency.

5.6 Experimental Evaluation

We perform exhaustive experiments as follows. First, we examine the convergence of CCM_MADRL_MEC with different learning rates. Second, we compare our algorithm with other benchmarks and heuristic algorithms. The experiment aims to evaluate the importance of applying a master agent to MADRL for combinatorial decisions. We did not evaluate the contribution of CCM_MADRL in terms of dimensionality reduction, which considers the number of channels in the combinatorial action selection, rather than including them in the state and action spaces of the actors. This is because it is difficult for the benchmark algorithms to converge using a continuous auction space that is discretized to the number of channels in the action selection, let alone to compare with CCM_MADRL. Instead, we applied combinatorial action selection using heuristic ordering to the benchmark algorithms to achieve fairness in the comparison.

5.6.1 Benchmark Algorithms

The main benchmark for our algorithm is MADDPG for two reasons. First, most existing task-offloading algorithms use DDPG and MADDPG. Second, our algorithm is an extension of MADDPG to heterogeneous combinatorial client-master MADRL agents of the policy gradient and value function techniques that cooperate to make a combination of continous-valued and combinatorial decisions in a distributed setting. However, we also developed different heuristic benchmarks to show the impact on different possible orders of accepting tasks on the MEC server. The heuristic algorithms differ from the proposed CCM_MADRL_MEC in that, instead of training a master agent to make combinatorial decisions about the clients using their Q-values, a stationary algorithm is used to decide on which of the tasks to approve for processing in the MEC server based on some ordering mechanism. The heuristic-based MADDPG algorithms are discussed below.

- MADDPG: This is MADDPG where the UDs will use actor agents to make decisions. If $x_{c,n} < 0$, the UD assigns its task locally. Otherwise, it forwards its proposed action to the MEC server. The SDN controller in the BS will only cooperate in assigning the UD tasks to different channels considering the constraints. If the number of tasks proposed to be offloaded by the UDs is not greater than the number of channels, and if the sum of their size is less than or equal to the storage capacity of the server, all of them are accepted to the MEC server. The SDN controller assigns each UD to a different channel and the UDs begin to offload their tasks. Otherwise, if the proposed tasks are greater than the number of channels or the sum of their size is greater than the storage capacity of the server, the SDN assigns the UDs to the channels in the order of their offloading time. Tasks that are not assigned to any channel are discarded. The discarded tasks are assigned τ_{max} to T_{MEC_n} as a penalty to discourage similar actions in the future. This penalty is applied because only the actors are in charge of task-offloading decisions in MADDPG. Because the penalty can be unfair for benchmark comparison, the following heuristics are developed to have equivalent combinatorial decisions with the CCM_MADRL_MEC for the tasks that are not accepted. That is, return the unaccepted tasks to be processed by the UDs rather than dropping them.

- MADDPG with the shortest offloading time first heuristic: This is similar to MADDPG, with the distinction that, after the combinatorial decision, the tasks that are not assigned to the channels or storage are designated for local processing within the UDs.

- MADDPG with deadline/size first heuristic: This is based on the heuristic that offloading tasks with critical deadlines and larger sizes to the MEC server improves

overall latency and power consumption. If $x_{c,n} < 0$ the UD allocates its task locally. Otherwise, it proposes its task to the MEC server. Then, if the number of offloaded tasks is not greater than the channels and if the sum of their size is less than or equal to the storage capacity of the server, all of them are accepted to the MEC server. Otherwise, the SDN controller assigns the tasks to the channels in the order computed by deadline/size first until either the channel constraint or the storage constraint is met. The rest are assigned to the UDs.

Even if the way tasks are accepted by the MEC server differs between the benchmarks and the CCM_MADRL_MEC, the order of processing of the accepted tasks is always in the order of arrival at the MEC, which is computed using Equation (5.5)

5.6.2 Experimental Settings

The experimental setting is provided in Table 5.2. We use a configuration and data set similar to Nguyen et al. (2023) because they use a data set from Huawei Telecom. However, because we introduced the server storage constraint, the size of the tasks is customized by increasing their size and decreasing the number of cycles per bit required to process a task similar to the configuration in Zhang et al. (2020). The typical storage capacity of modern servers is GB and TB. However, because we chose a small experimental setting of 50 UDs due to computational resources, we considered a storage constraint of 400 MB so that the task offloading problem is combinatorial to the server. The energy harvesting setting is adapted from Zhang et al. (2020). We used a seed of 37 for the reproducibility of the simulation environment. Evaluation episodes are seeded with their index. We used a discount factor of 0.99.

5.6.3 Training Environment and Evaluation Environment

When using continuous-valued actions, such as MADDPG, for resource allocation, it is not an efficient decision to evaluate the performance of the algorithms while they are being trained, in the same environment in which they are being trained, and the actions that are a sum of the output of the actor and exploration noise. First, the number of tasks generated in the training environment episodes and their resource requirements are independent and different. As a consequence, it is not convenient to visualize whether the performance is showing an upward trajectory. In contrast, evaluating performance using a separate evaluating environment in each episode can show an upward trajectory of performance because the same evaluation environment is used for every episode of the training. Second, decaying exploration noise can make the algorithm look like it is improving. This is because the exploration noise can round most of the actions of the actor to 0 and 1 in the early stages of the training. Although the

Parameter	Value
Number of UDs ($\|N\|$)	50
Number of channels (K)	10
Time constraint (τ_n)	[0.1-0.9] s
Bandwidth (W)	40 MHz
Maximum power (P^{max})	24 dBm
Minimum power (P^{min})	1 dBm
Maximum power (p_n^{max}) of a UD	[P^{min} - P^{max}] dBm
Minimum power (p_n^{min}) of a UD	P^{min} dBm
Maximum local computing resource (f_{max})	1.5 GHz
Minimum local computing resource (f_{min})	0.4 GHz
Maximum local computing resource (f_n^{max}) of a UD	[f_{min}-f_{max}] GHz
Maximum battery capacity (b_{max}) of a UD	3.2 MJ
Minimum battery threshold (b_{min}) of a UD	0.5 MJ
Initial battery level (b_{max}) of a UD	[b_{min}-b_{max}] MJ
Normalized uplink channel gain (g_n)	[5-14] dB
MEC computing resource (f_e)	4 GHz
Task size (z_n)	[1-50] MB
Server storage capacity (z_e)	400 MB
CPU cycles (c_n)	[300 - 737.5] cycles
Number of processing units (U_e) on the server	8
Energy coefficient (κ) in a UD	5×10^{-27}
Weight coefficients (λ_1, λ_2)	(0.5, 0.5), (1, 5), (1, 1000)
Energy harvesting (e_n) of a UD	0.001 J

TABLE 5.2: Experimental parameters of CCM_MADRL_MEC

actions of the actor can be between 0 and 1 due to the activation functions and scaling, the exploration noise is large and causes most of the actions to be rounded to 0 and 1, leading to either no resource allocation or full resource allocation, which is not efficient. As the noise decays to zero, most of the actions will be between 0 and 1, allowing fractional resource allocation with more efficiency than binary allocation. This effect occurs regardless of whether the actor is learning or not.

Therefore, we evaluated the performance using a separate environment using only the actions outputted by the algorithm that do not include the exploration noise.

5.6.4 Hyperparameter Selection and Convergence

The selection of hyperparameters is challenging in DRL. Unlike certain well-defined algorithms, there is no one-size-fits-all rule for selecting the appropriate hyperparameters in DRL. One of the most influential parameters in hyperparameter tuning is the learning rate (Gulde et al., 2020). In this experiment, we selected some of the hyperparameters based on experience in the preceding chapters: replay memory 10000, batch size 64, target soft update 1, discount factor 0.99. The training algorithm is run at the

end of each episode for two reasons. The first reason is to provide better generalization by avoiding overfitting, as observed in the experiments of Chapter 3 and Chapter 4. The second is to run many episodes and collect many data by increasing the distance of the training interval. The parameters of the DNNs are selected by trial and error. The DNN of the client agents has 7 inputs, representing the state of a UD, two hidden layers of 64 and 32 neurons each with a ReLu activation function, and 3 outputs with a Tanh activation function representing the action space. The actor agents in the benchmark and heuristic algorithms also have the same hyperparameters as the clients of CCM_MADRL_MEC. However, because the master agent uses per-client DQN, it has 510 inputs, which includes 10 for the combined input of the states and actions of the 50 agents, and an additional 10 for the per-client input, whereas the critic networks of the benchmark and heuristic algorithms have only 500 inputs as they are classical MADDPG algorithms. Apart from the input, the master agent and the critic networks of the benchmark and heuristic algorithms have the same other hyperparameters: two hidden layers with 512 and 128 neurons each followed by a single output with linear activation function. ADAM optimizer is used in all the experiments in this thesis.

As shown in Equation (5.22), the learning rate in DRL is a hyperparameter that regulates the speed with which the weights of the DNNs are updated based on the feedback from the environment. It defines how much the agent's Q-values should be updated based on fresh information collected from the environment. Because it is a combination of the policy gradient and the value function, selecting the appropriate learning rate is more challenging than the other parameters in MADDPG and our algorithm. Therefore, we conducted extensive experiments to find the best combination of learning rates for the client agents and the master agent as follows.

First, we filter a combination of 16 learning rates by comparing their impacts on the DNN weights of the client agents and the master agent. The DNNs are initialized with the same weights using the 23 seed on the PyTorch so that the performance becomes only an effect of the learning rates. A seed of 23 is also used in the NumPy package of Python for the random number generators of exploration noise and exploration and exploitation probability so that all learning rates follow the same exploration and exploitation sequences.

Figure 5.3 shows the performance using the evaluation environment. Because the results were saved in an Excel file before plotting them, we found that {Client learning rate, master learning rate} of {0.1, 0.0001}, {0.01, 0.0001}, {0.001, 0.001}, {0.0001, 0.001}, and {0.001, 0.0001} converged up to -34. The performance with a learning rate of 0.1 (blue line) for the master agent is overlapped with the performance of the master agent's learning rate of 0.01 (yellow line). As presented in Algorithm 4, the evaluation environment is run after each training episode. We also evaluated the performance of the CCM_MADRL_MEC in the training environment, as seen in Figure 5.4. The combination of {Client learning rate, master learning rate} of {0.1, 0.0001}, {0.01, 0.0001},

and {0.001, 0.0001} converged up to -38 on the training environment while the others are below -52. However, as explained in Section 5.6.3, the training environment cannot be used as evidence that the algorithm is training correctly and converging.

Next, we further refined the best learning rates based on the results demonstrated in Figure 5.3 and Figure 5.4 as follows. The combinations of learning rates of {0.1, 0.0001} and {0.01, 0.0001} are the same and do not show steady performance, but we took the latter along with other combinations of learning rates that have shown a steady upward trajectory in the results shown above such as {0.001, 0.001}, {0.0001, 0.001}, and {0.001, 0.0001} to make further comparisons with multiple experiments and benchmark algorithms in the next section.

As discussed at the beginning of this section, the experiments for the learning rates are run for the same initialization of DNN weights and the same exploration and exploitation sequences, so that only the learning rate impacts the training of the algorithms. The learning rate of {0.0001, 0.001 } for client and master, respectively, has shown stable and steady learning, but learning rates such as {0.01, 0.0001} have shown constant performance by default in the evaluation environment. The reason they seem to show an increase in performance in the training environment is due to exploration noise, as explained in Section 5.6.3. Therefore, to make a general comparison, we performed another experiment for {0.01, 0.0001}, {0.001, 0.001}, {0.0001, 0.001}, and {0.001, 0.0001} using multiple runs with different initialization weights of DNN and a different evaluation environment as discussed in Section 5.6.5. Their corresponding results are shown in Figure 5.6, Figure 5.7, Figure 5.5, and Figure 5.8. It can be seen that the performance of the learning rates with {0.01, 0.0001} and {0.001, 0.001} have declined for the experiments with many runs with different initializations of the weights, as seen in Figure 5.6, and Figure 5.7 respectively. The reason they have shown competitive performance for the single-run experiment in Figure 5.3, and Figure 5.4, but declined in Figure 5.6 and Figure 5.7, could be due to the default weights of the DNN that could be close to convergence with fewer training episodes. Note that the learning rates in Figure 5.3 are compared for the same initialization of the weights of the neural networks with a seed of 23. Convergence is relatively better with learning rates of {0.001, 0.0001}, as seen in Figure 5.8. However, we can see from all experiments that the learning rates of {0.0001, 0.001} for client and master, respectively, have led to better performance as shown in Figure 5.5. Therefore, the learning rate of {0.0001, 0.001} is selected as the best combination of learning rates to proceed to the rest of the experiments.

5.6.5 Generalizability

Convergence can be affected by the initialization of the DNN weights, the exploration noise that is added to the actions of the client agents, and the sequence of exploration and exploration in the master agent. Therefore, we conducted multiple experiments

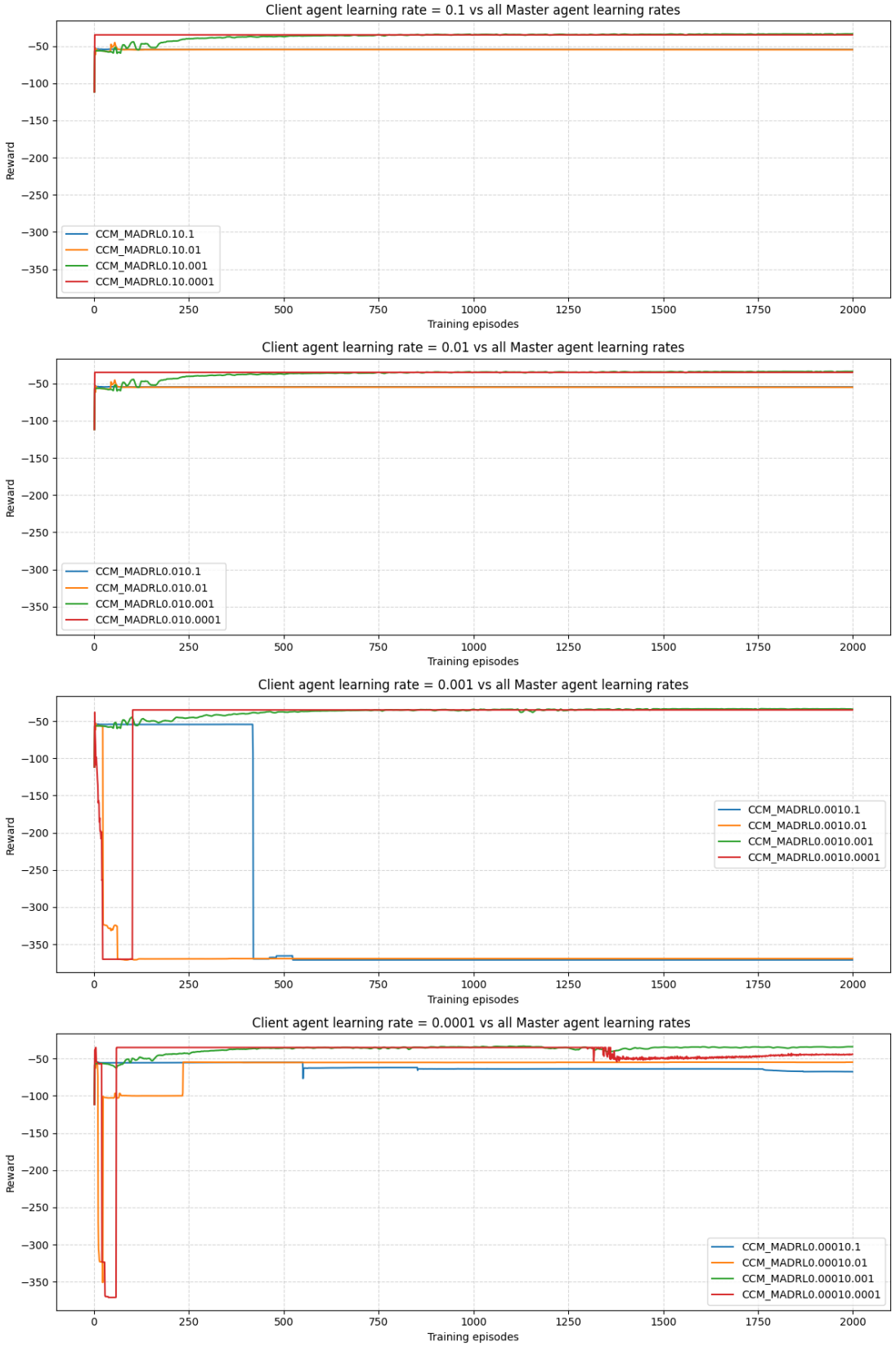


FIGURE 5.3: Performance of the CCM_MADRL using the evaluation environment for different combinations of client agent and master agent learning rates

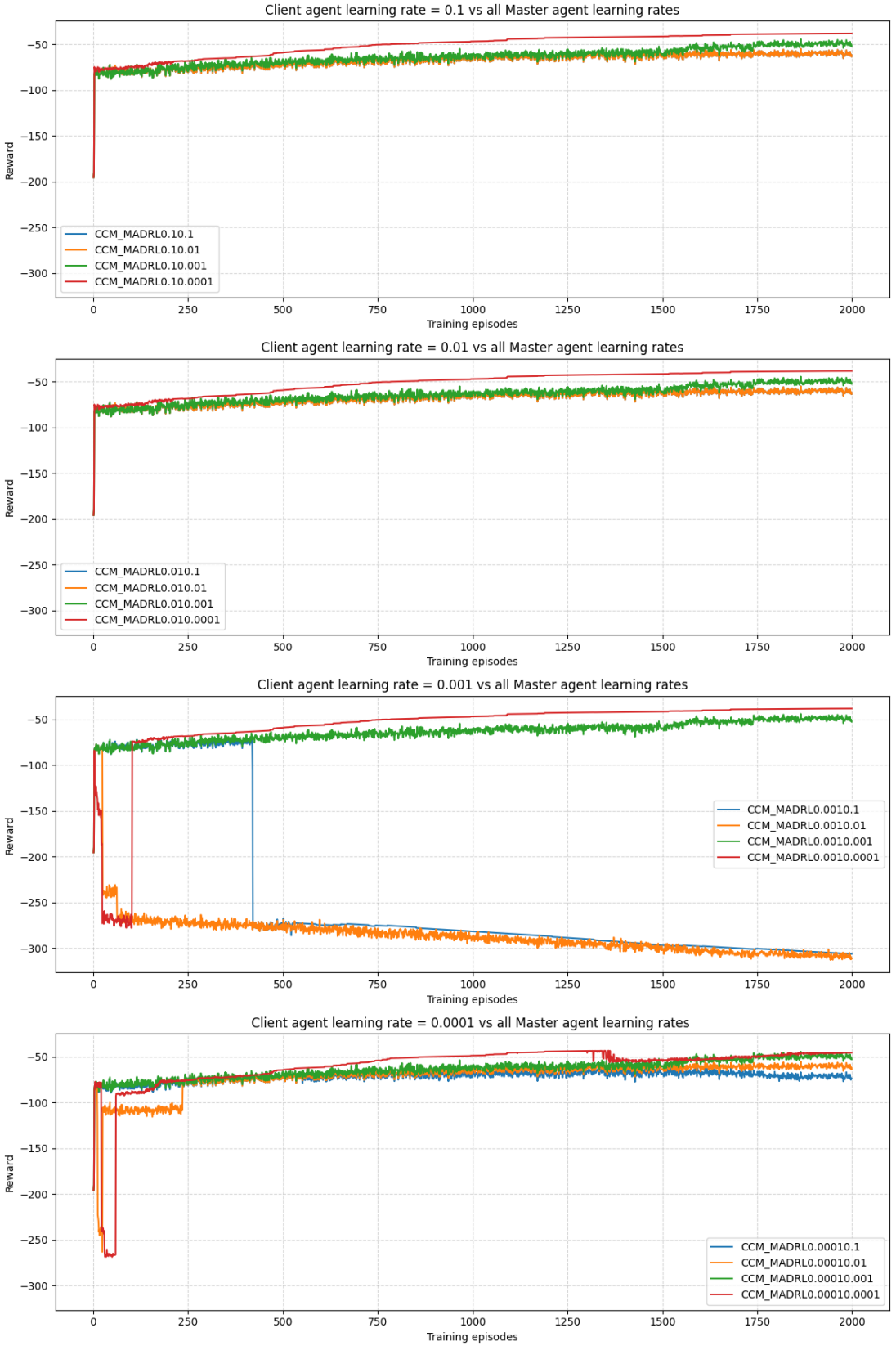


FIGURE 5.4: Performance of the CCM_MADRL algorithm on the training environment for different combinations of client agent and master agent learning rates

for each algorithm using different initializations of the weights, exploration noise, and different exploration and exploitation sequences for each run of the experiment and plotted the result with a 95% confidence interval. The importance of using the evaluation environment is described in Section 5.6.3. Furthermore, we evaluated the experiment with multiple evaluation episodes for a more general evaluation. Therefore, the experiment is run for 2000 training episodes and 50 other evaluation episodes that are performed at every training episode. The results in the figures that used a separate evaluation environment are an average of 50 evaluation episodes that are evaluated in each training episode of the environment to see how the training has improved the convergence in each episode.

5.6.6 Experimental Results Using the Evaluation Environment

Figure 5.5 shows the evaluation for 10 steps per episode using the evaluation environment. The CCM_MADRL algorithm has performed better than the other algorithms because once the clients choose their action, the master agent also makes a combinatorial decision on the action of the clients. This two-step decision provides another option for convergence in case the client agents stick to local optimal. That is, using a combination of advantages of the policy gradient and the value function to make decisions, the CCM_MADRL mitigates the challenge of sticking to the local optimal. In contrast, the actors in the heuristic and benchmark algorithms receive only feedback from the critic. They took full responsibility for adjusting their actions.

The second subplot in Figure 5.5 (B) shows the percentage of tasks whose deadline is exceeded. It shows that about 10% of the tasks exceeded their deadline at the best convergence of the CCM_MADRL. The percentage of tasks that exceed their deadline is inversely proportional to the reward. The higher the reward, the lesser the number of times the deadline has been exceeded. The last subplot in 5.5 (C) shows the percentage of UDs that exceed the minimum battery threshold. It can be seen that no UD has exceeded its minimum battery level because the experimental setting was configured for 10 steps per episode.

We also carried out the same experiment with learning rates of $\{0.01, 0.0001\}$ for the clients and the master, respectively, as shown in Figure 5.6 to support our analysis in Section 5.6.4. All algorithms showed poor performance because they only use actor agents to make decisions, but the CCM_MADRL algorithm has the advantage over the others using its master agent. The same effect is observed with learning rates of $\{0.001, 0.001\}$ as seen in Figure 5.7. The result for learning rates of $\{0.001, 0.0001\}$ has shown relatively better performance than seen in Figure 5.8 but is not comparable to learning rates of $\{0.0001, 0.001\}$.

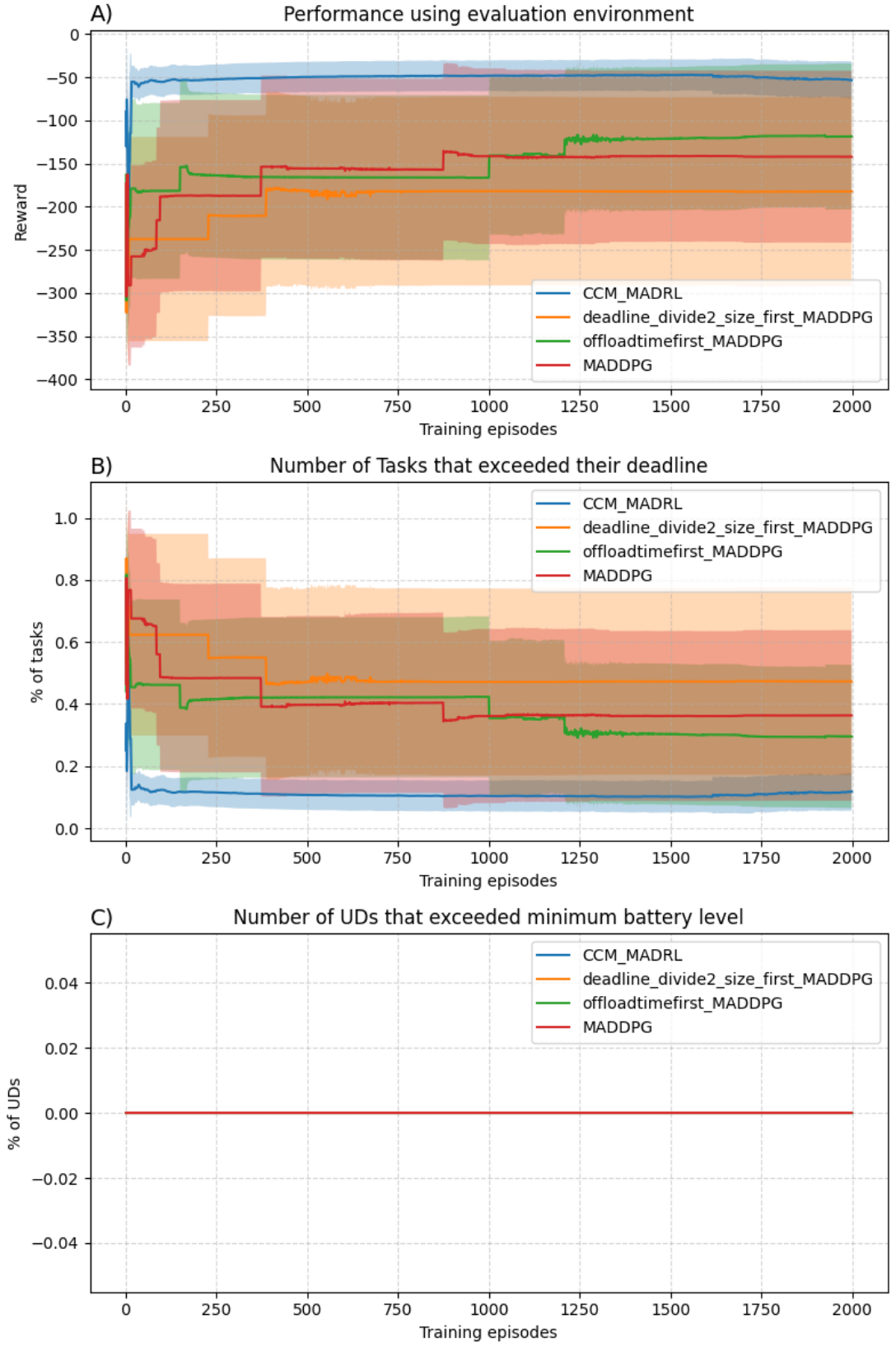


FIGURE 5.5: Comparison of the CCM_MADRL with the heuristic and MADDPG algorithms with a learning rate of 0.0001 and 0.001 for the clients and master agents respectively

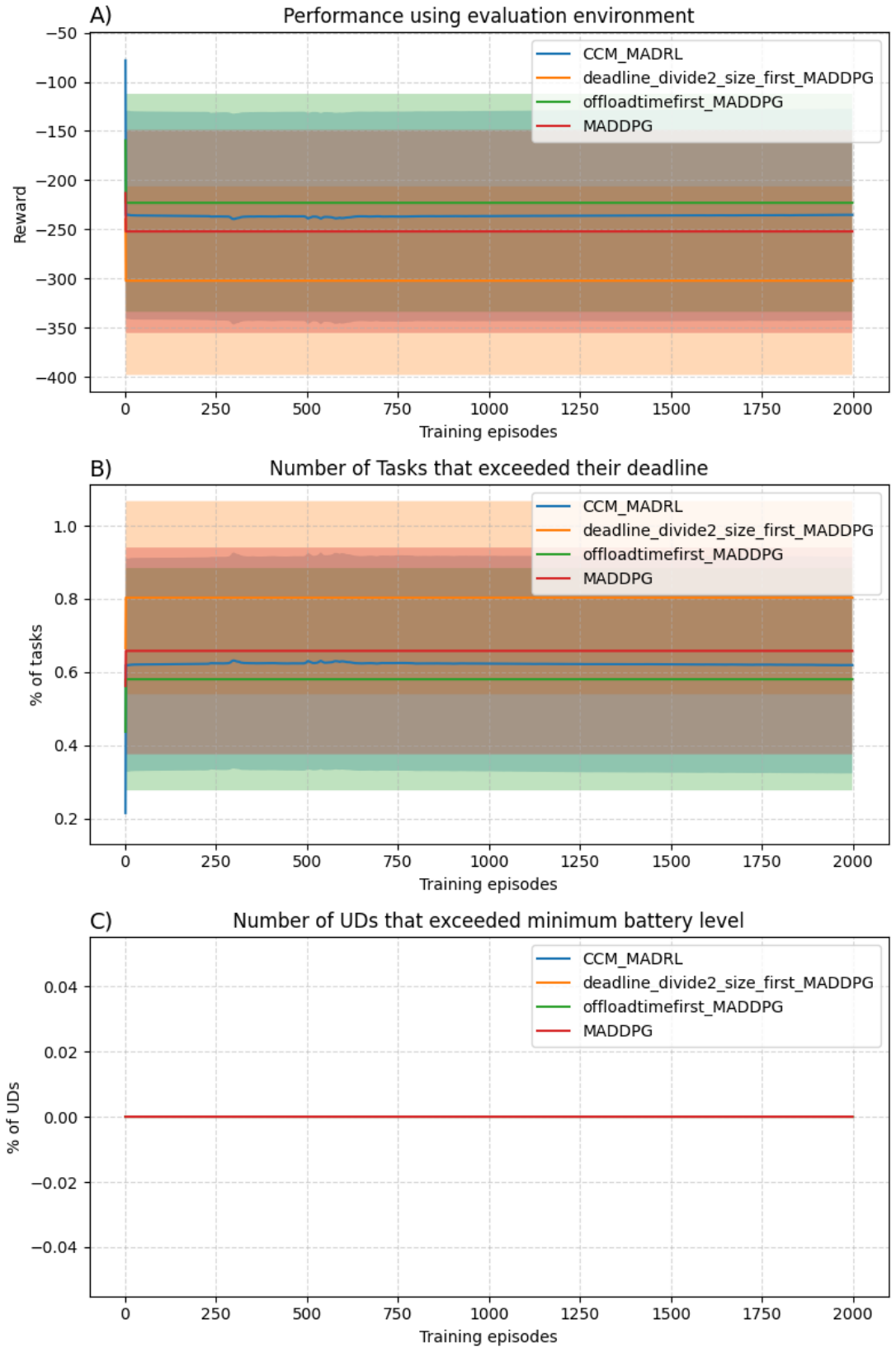


FIGURE 5.6: Comparison of the CCM_MADRL with the heuristic and MADDPG algorithms with learning rates of 0.01 and 0.0001 for the clients and master agents respectively

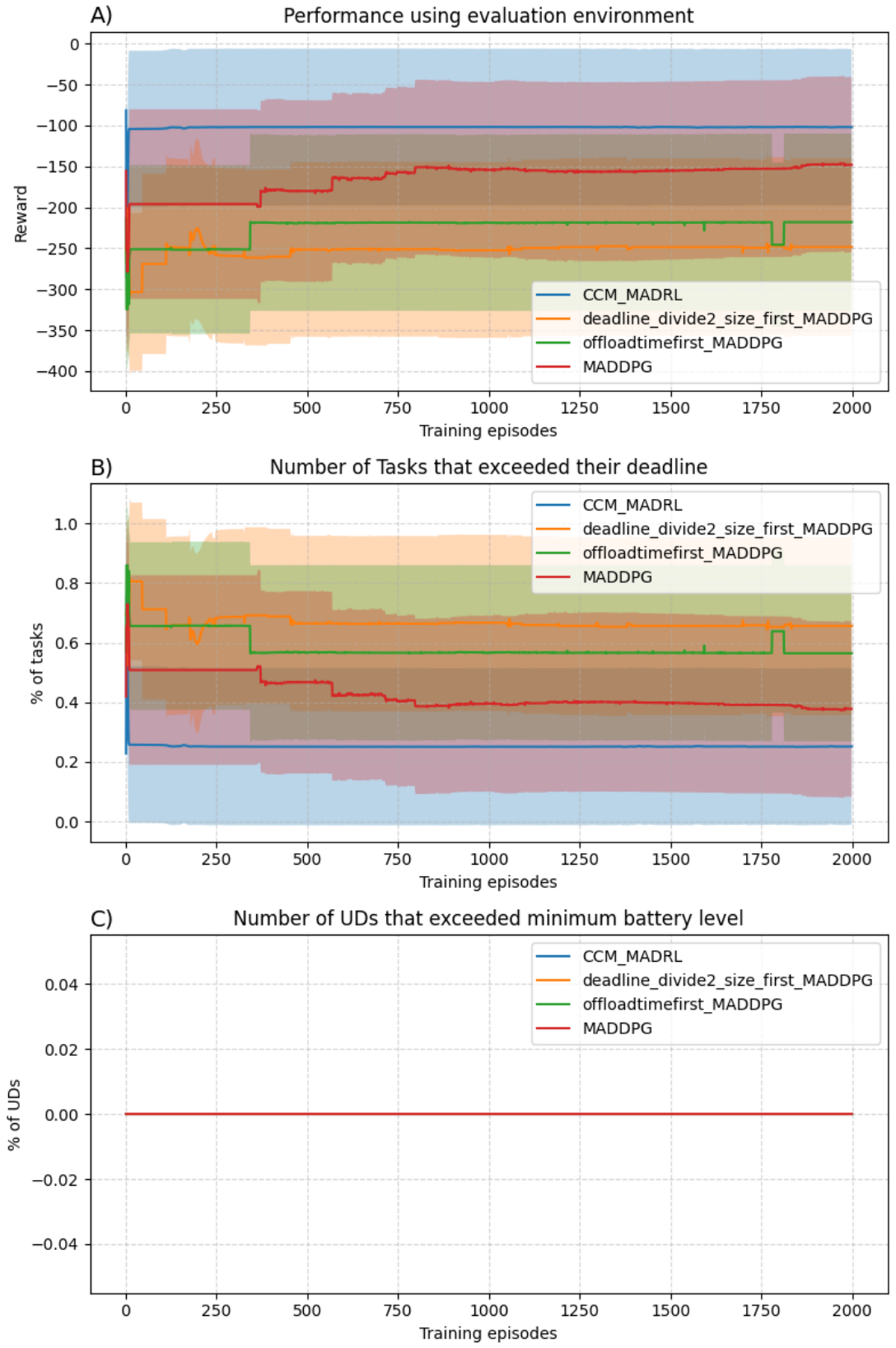


FIGURE 5.7: Comparison of the CCM_MADRL with the heuristic and MADDPG algorithms with learning rates of 0.001 and 0.001 for the clients and master agents respectively

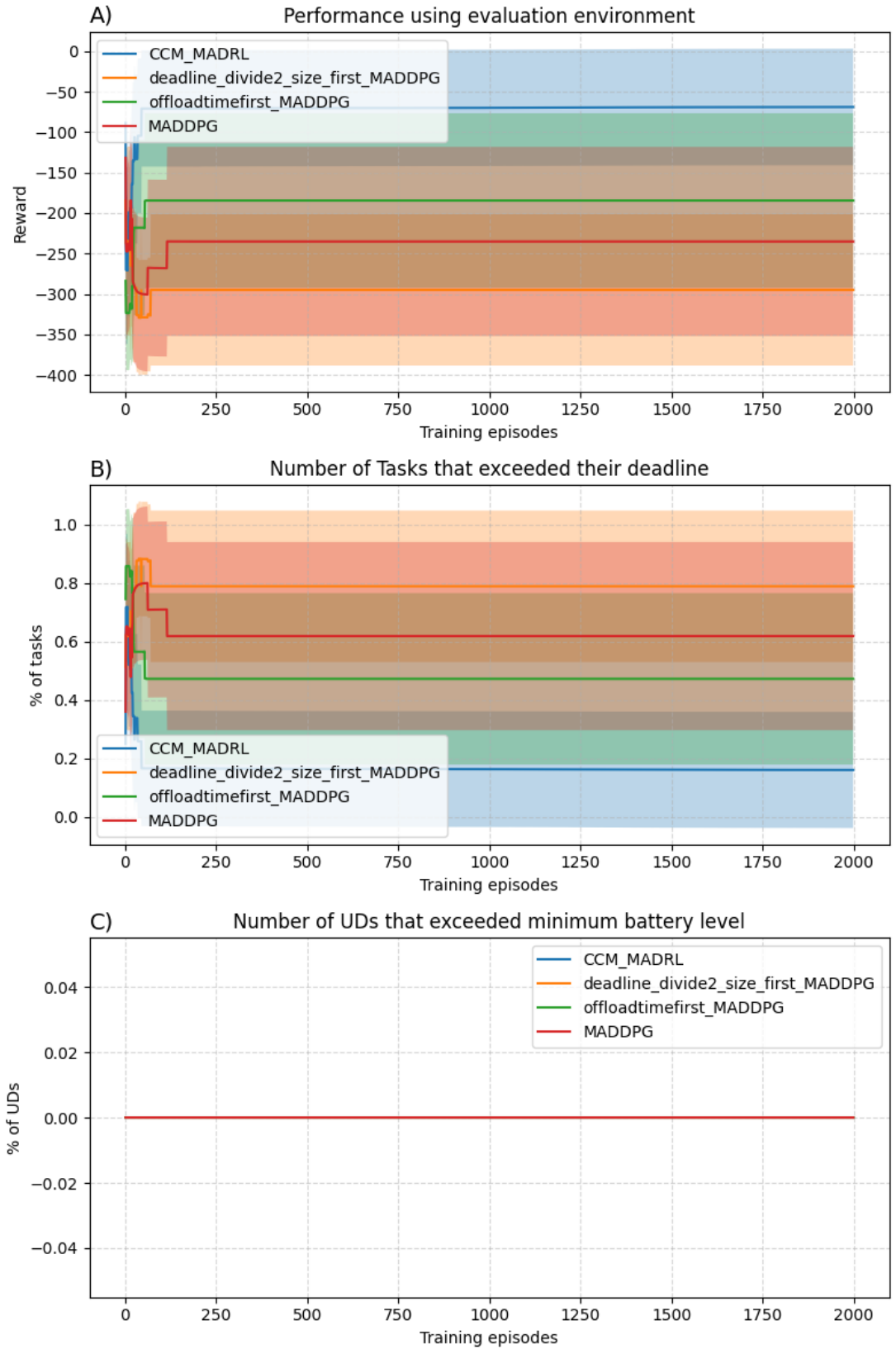


FIGURE 5.8: Comparison of the CCM_MADRL with the heuristic and MADDPG algorithms with learning rates of 0.001 and 0.0001 for the clients and master agents respectively

To see the impact on the battery level, we changed the experimental setting and ran it for 100 steps per episode. We also changed b_{max} to $b_{min} + 1J$ so that some UDs have a small battery capacity. Note that no UD has exceeded the battery threshold in Figure 5.5 (C) because the number of steps per episode is not long enough to run the battery below the minimum battery threshold.

The CCM_MADRL_MEC performs better than the heuristic and MADDPG algorithms, as shown in Figure 5.9. Figure 5.9 (C) shows that CCM_MADRL_MEC has more UDs running below the minimum battery threshold than the benchmark and heuristic algorithms. However, the number of UDs that run below the minimum battery level is not included in the cost function. The penalty for running below the battery threshold is computed by subtracting the amount of joules it falls below the threshold. Therefore, it is affected by the scales given to the deadline penalty and the energy penalty. We use $\lambda_1 = \lambda_2 = 0.5$ as weight coefficients for latency and energy consumption costs in the reward function. These coefficients can be varied to give higher priority to the critical one. The CCM_MADRL_MEC is trained using the reward computed from the cost function. CCM_MADRL_MEC has outperformed the benchmark and heuristic algorithms for the reward function in which they are trained. The reason why CCM_MADRL_MEC is showing more UDs exceeding their battery threshold than the benchmark and heuristic algorithms is that the training focuses on minimizing the sum of latency and energy consumption costs rather than the number of UDs exceeding their deadline. This can happen when many UDs exceed their battery threshold with small values than when fewer UDs exceed their threshold with large values. The experiment was run on Iridis [2], an HPC cluster at the University of Southampton. The experiment for the result in Figure 5.9 is carried out with 10 runs for each algorithm for 60 hours. All of the experiments for heuristic and benchmark algorithms were finished earlier, but one of the runs for the CCM_MADRL_MEC ran out of time at episode number 1911. For convenience in plotting with the 95% confidence interval, all runs of the CCM_MADRL_MEC are cut after episode 1911 as seen in Figure 5.9. Note that the benchmark and heuristic algorithms have only one Q value in the critic for a combination of state and actions of the actors. On the other hand, the number of Q values to train in the CCM_MADRL_MEC is equal to the number of offloaded tasks or 1 if all of them are allocated locally.

We also experimented with $\lambda_1 = 1$ and $\lambda_2 = 5$ because Figure 5.9 (B) looks like the exact inverse of Figure 5.9 (B). The scales $\lambda_1 = \lambda_2 = 0.5$ have not provided enough balance for energy consumption, which is a very low number, and time consumption, which is relatively higher. The result is plotted in Figure 5.10 with a 95% confidence interval of 40 runs, unlike previous experiments, which are plotted with a 95% confidence interval of 10 runs. However, the plots do not show a significant difference between 10 runs and 40 runs, other than the shaded range of the confidence interval. Now, the subplot (B) and subplot (A) do not look exactly inverse to each other. However,

[2] https://www.southampton.ac.uk/isolutions/staff/iridis.page

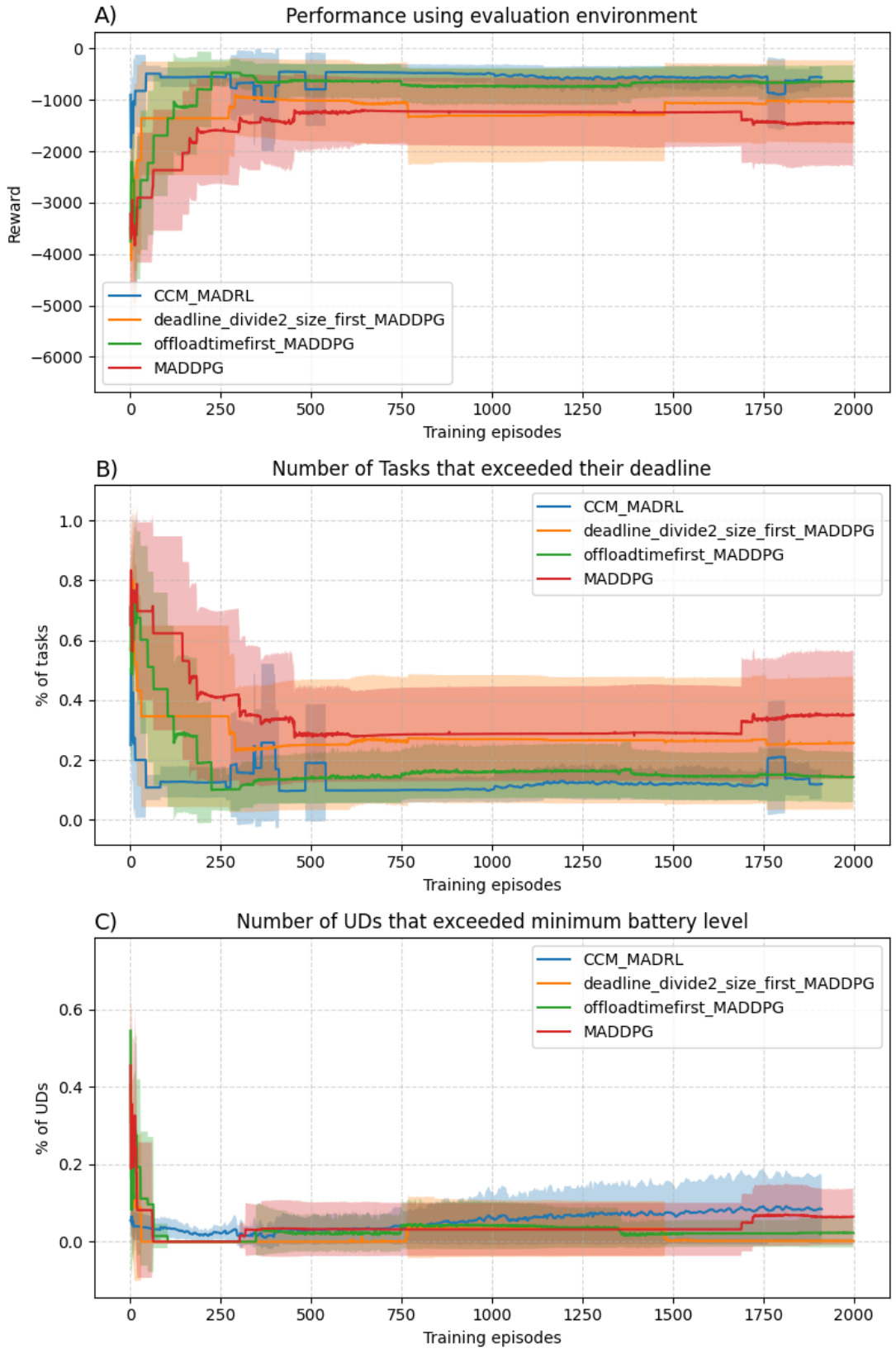


FIGURE 5.9: Comparison of the CCM_MADRL with the heuristic and MADDPG algorithms with learning rates of 0.0001 and 0.001 for the clients and master agents respectively, with 100 steps per episode, and $b_{max} = b_{min} + 1J$

the subplot (C) shows that CCM_MADRL_MEC still has more UDs running below the minimum battery threshold than the benchmark and heuristic algorithms. Nevertheless, CCM_MADRL_MEC has performed better than the other algorithms in the combined performance and shows more advantage in the number of tasks whose deadlines exceeded. Using a scalar reward for multiobjective functions does not represent the underlying problem (Vamplew et al., 2022), but is not within the scope of this work. Hayes et al. (2022) has studied the essence and techniques of multiobjective DRL.

Since scaling with $\lambda_1 = 1$ and $\lambda_2 = 5$ does not yet balance latency and energy consumption costs in the combined reward, we continue the experiment with $\lambda_1 = 1$ and $\lambda_2 = 1000$ as seen in Figure 5.11. Because energy consumption is scaled to a value of 1000, it led to the percentage of UDs that exceeded the battery threshold to zero. CCM_MADRL_MEC still has shown performance in the combined reward and the percentage of UDs exceeding their battery threshold. Note that it seems that the performances of the algorithms look overlapped, as seen in Figure 5.11 (A). But this is due to the magnitude of the total reward, which is in millions due to the effect of the value of λ_2. The maximum value at convergence of the algorithms is; CCM_MADRL_MEC=-3207.2385251393675, MADDPG with deadline/size first heuristic = -4063.309111026686, MADDPG with the shortest offloading time first heuristic = -3895.1804818457117, and MADDPG=-4567.158552095975.

It can be seen that the benchmark and heuristic algorithms demonstrated a closer performance to the CCM_MADRL_MEC on 100 steps per episode than in 10 steps per episode. This difference is caused by overfitting, as observed in Chapter 3 and Chapter 4. Training after 10 steps leads to overfitting to the first episodes, whereas training after every 100 episodes provides more generalization. The benchmark and heuristic algorithms are impacted by overfitting more than CCM_MADRL_MEC because they use only their actors to select action, while the CCM_MADRL_MEC uses the advantage of both clients and master to mitigate overfitting and sticking to local optimal. It can also be seen from Figure 5.9 that the shortest offloading time first benchmark performs close to the CCM_MADRL_MEC algorithm. This is because the energy consumption is negligible and the latency has a higher priority due to their scales, making the shortest offloading time first benchmark competitive.

5.6.7 Evaluation with Statistical Tests

We perform statistical tests in the same way as in Section 4.4.2.6. First, we plot the histogram using the episodes and runs of the experiment as follows. The statistical test is conducted for the last experiment with $\lambda_1 = 1$ and $\lambda_2 = 1000$ as plotted in Figure 5.11 because it visually looks like the algorithms' performance overlaps.

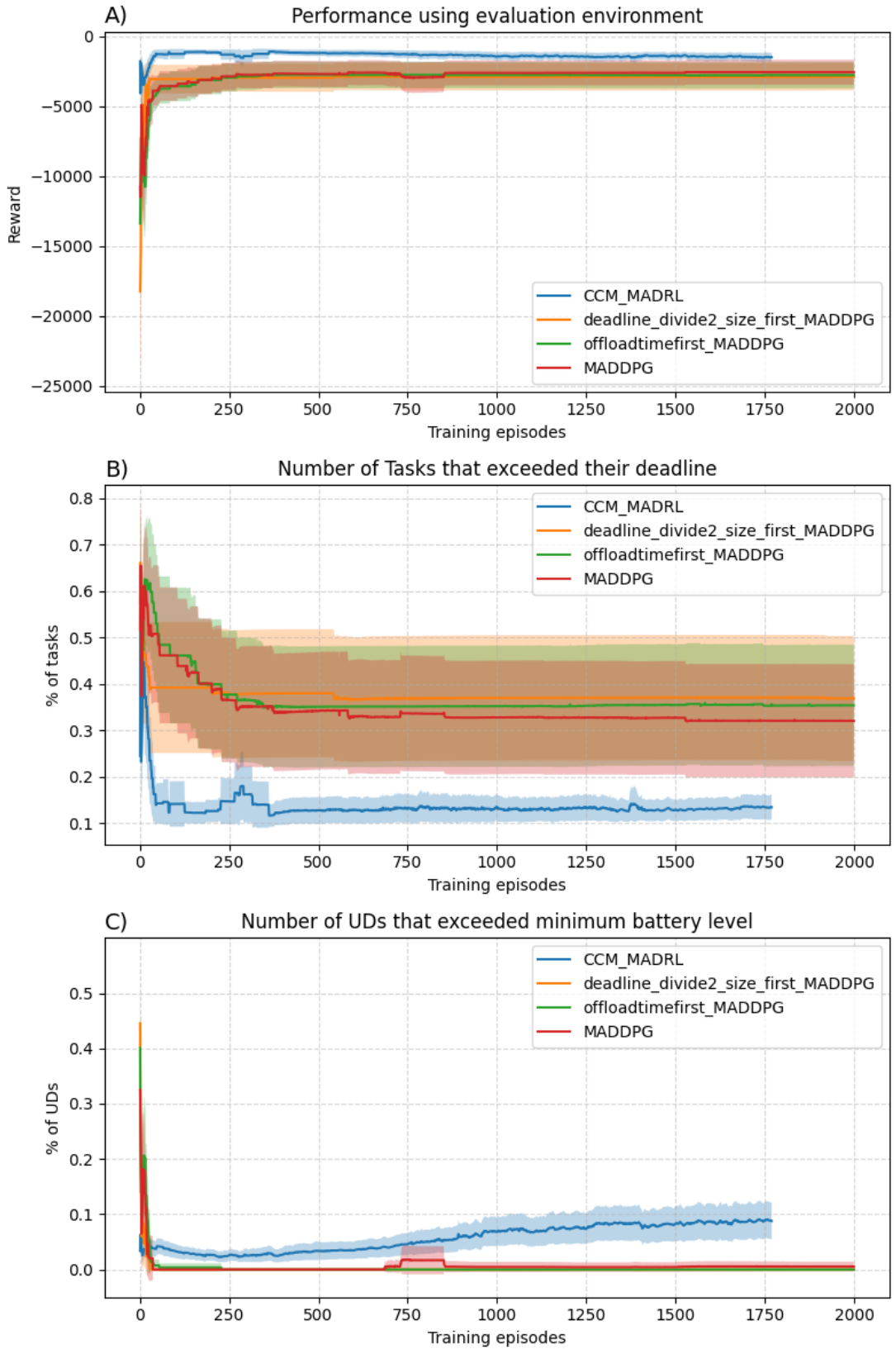


FIGURE 5.10: Comparison of the CCM_MADRL with the heuristic and MADDPG algorithms with learning rates of 0.0001 and 0.001 for the clients and master respectively, $\lambda_1 = 1$ and $\lambda_2 = 5$, with 100 steps per episode, and $b_{max} = b_{min} + 1J$

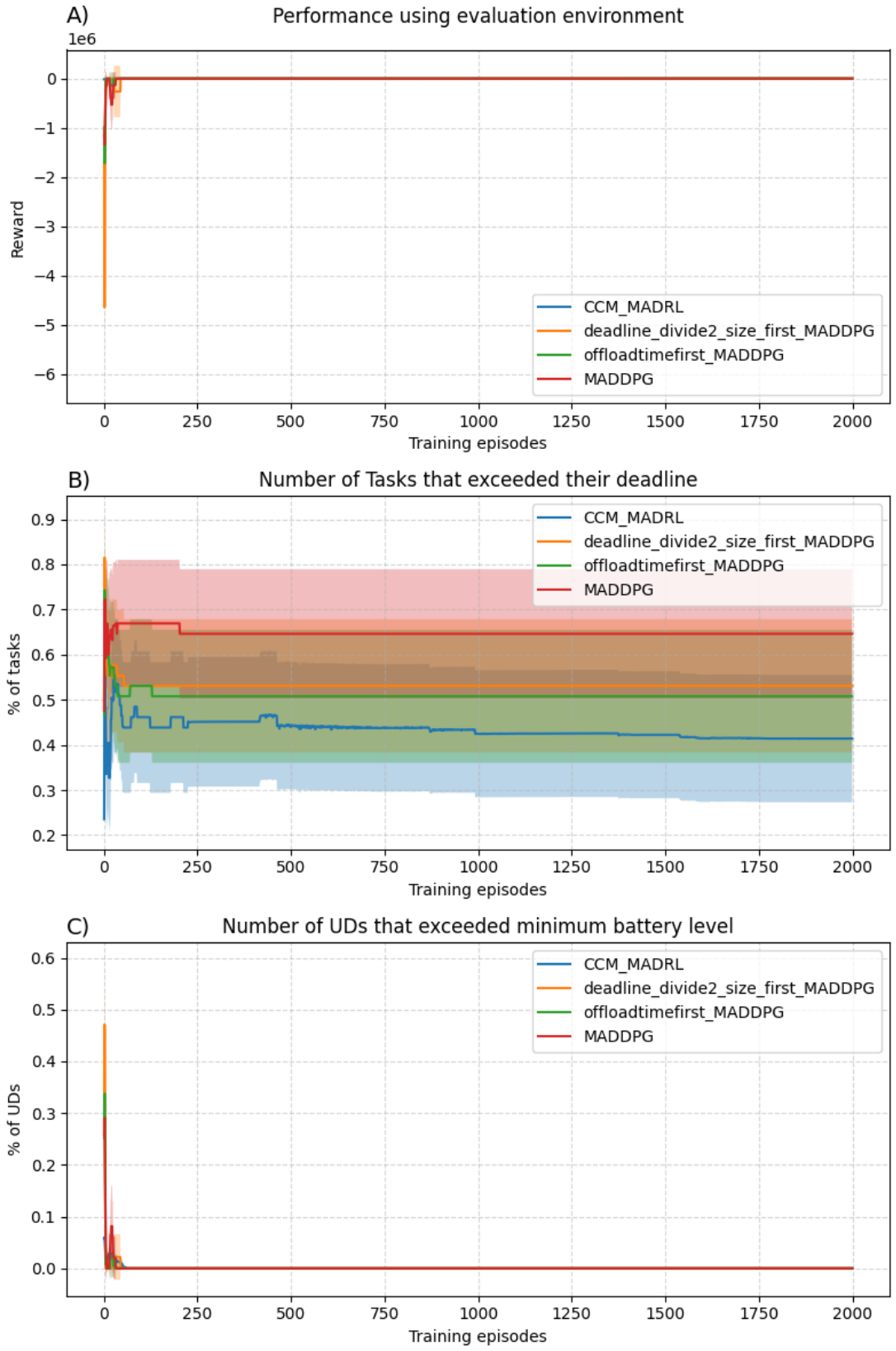


FIGURE 5.11: Comparison of the CCM_MADRL with the heuristic and MADDPG algorithms with learning rates of 0.0001 and 0.001 for the clients and master respectively, $\lambda_1 = 1$ and $\lambda_2 = 1000$, with 100 steps per episode, and $b_{max} = b_{min} + 1J$

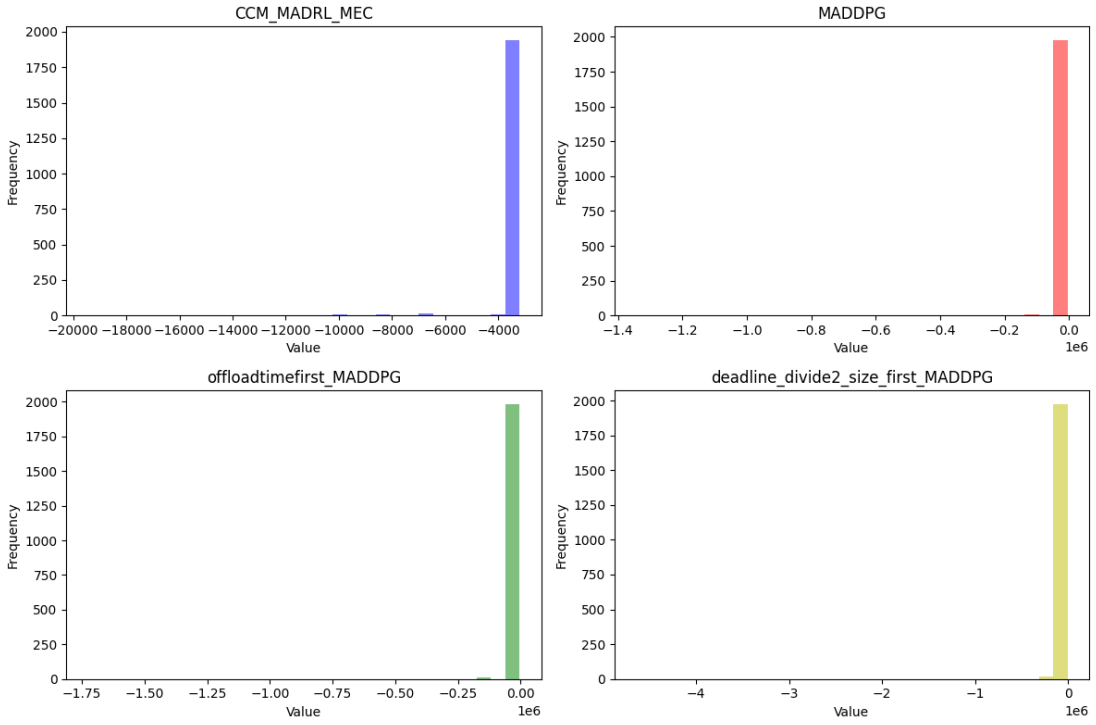


FIGURE 5.12: Histogram for the performance of the algorithms shown in Figure 5.11 using the episodes as data sample

As seen in Figure 5.12, the performance of the DRL in episodes is not normally distributed for the same reason as discussed in Section 4.4.2.6. Figure 5.13 shows that the distribution is better when the experimental runs are used as a data sample. Unlike the description in Section 4.4.2.6, the maximum values of the experimental runs are not computed by averaging over a moving window, because in this chapter a different evaluation environment is used, which is the same for all episodes.

The statistical test with ANOVA recorded an F-statistic of 4.192263096768374 and a P-value of 0.005671193240624048 when episodes are used as a sample. Next, we performed a pairwise evaluation of CCM_MADRL with each of the benchmark and heuristic algorithms, as presented in Table 5.3. We also repeated the statistical test using the nonparametric Wilcoxon Mann-Whitney rank sum test as seen in Table 5.4.

Benchmark	t-statistic	P-value
MADDPG	4.871407961046222	1.1509819197451862e-06
Shortest offload time first MADDPG	2.800671227050114	0.005124157804480162
Deadline divide to size first MADDPG	2.7776572741066237	0.0055007795974677575

TABLE 5.3: T-test between the CCM_MADRL and each of the benchmark and heuristic algorithms using the episodic rewards as sample

As discussed in Section 4.4.2.6, we performed the statistical test on the experimental runs. The ANOVA assessment provided an F-statistic of 3.4421006126211022 and a

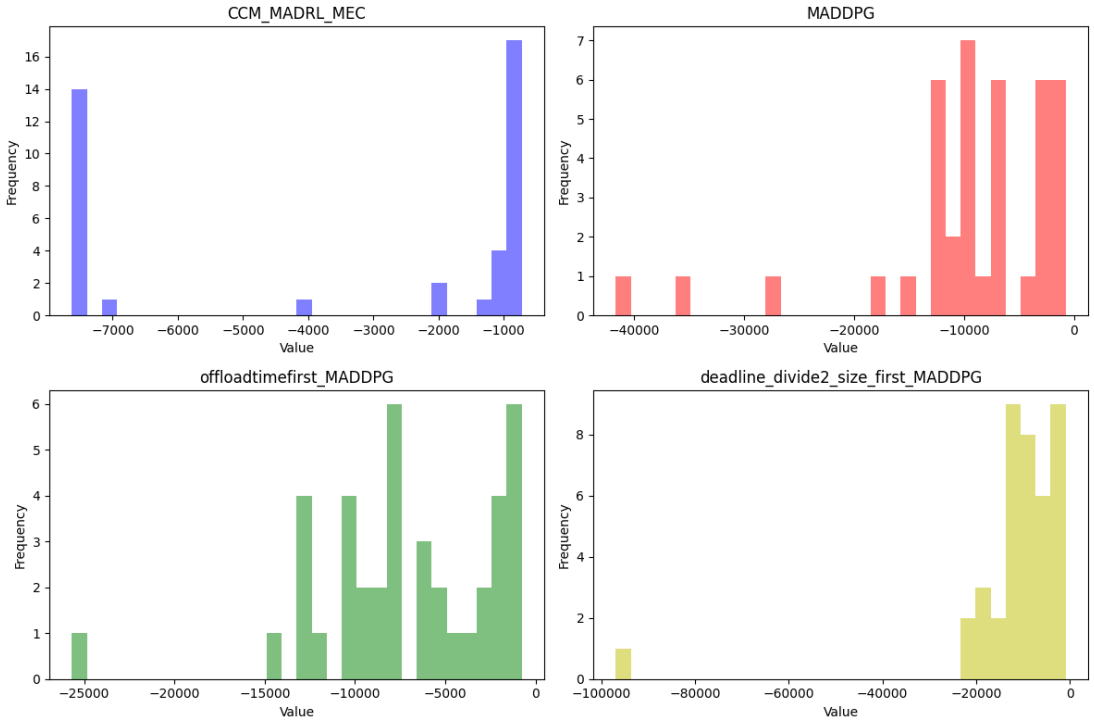


FIGURE 5.13: Histogram for the performance of the algorithms shown in Figure 5.11 using the runs of the experiment as data sample

Benchmark	U-statistic	P-value
MADDPG	3901260.0	0.0
Shortest offload time first MADDPG	3893365.0	0.0
Deadline divide to size first MADDPG	3892935.0	0.0

TABLE 5.4: Wilcoxon Mann-Whitney rank sum test between the CCM_MADRL and each of the benchmark and heuristic algorithms using the episodic rewards as sample

P-value of 0.01830702538963317. The pairwise comparison using t-test and Wilcoxon Mann-Whitney rank sum test is provided in Table 5.5 and Table 5.6.

Benchmark	t-statistic	P-value
MADDPG	2.1486133773494323	0.03476737847989825
Shortest offload time first MADDPG	2.897388989848565	0.0048820984240530545
Deadline divide to size first MADDPG	3.151444433158739	0.002305087554929515

TABLE 5.5: T-test between the CCM_MADRL and each of the benchmark and heuristic algorithms using experimental runs as data sample

All statistical evaluations showed small P-values, leading to the conclusion that the difference in the performance of the DRL algorithms is not a result of random chance.

Benchmark	U-statistic	P-value
MADDPG	1144.0	0.0008961501478583839
Shortest offload time first MADDPG	1065.0	0.010397300466515273
Deadline divide to size first MADDPG	1032.0	0.02457683153819857

TABLE 5.6: Wilcoxon Mann-Whitney rank sum test between the CCM_MADRL and each of the benchmark and heuristic algorithms using experimental runs as data sample

5.7 Conclusion

In this chapter, we propose a combinatorial client-master MADRL algorithm for task offloading in MEC. We consider the storage capacity of the server and the number of communication channels as a combinatorial constraint for the task-offloading problem. By combining the advantages of both policy gradient and value function methods to output continuous and combinatorial actions, the CCM_MADRL algorithm provides better convergence than existing homogeneous MADRL algorithms, such as MADDPG. Note that MADDPG combines a policy gradient and a value function to train its actors, but the critic does not participate in the action selection. CCM_MADRL applies the coalition action selection proposed in Chapter 3 and the per-action DQN as used in Chapter 4 to make combinatorial decisions on the actions proposed by the clients. Client agents decide on their continuous-valued resource allocations, and the master agent makes combinatorial decisions. The experiments demonstrate that CCM_MADRL_MEC outperforms the benchmark and heuristic algorithms.

In the future, we plan to extend CCM_MADRL_MEC to multi-server MEC where multiple servers cooperate to make combinatorial decisions.

Chapter 6

Conclusions

6.1 Summary of the Thesis

This thesis started by outlining the different challenges that DRL algorithms face in online combinatorial resource allocation problems and the research requirements that must be met. To move forward to the solution, a comprehensive literature review of DRL-based resource allocation algorithms and their limitations, including the types of DRL, state and action spaces, and coordination methods in MADRL is conducted. All categories of literature review are also discussed in terms of their suitability for the stated research challenges and requirements. Then, two novel approaches are proposed to address arbitrary action and state spaces caused by ASO input and output in online combinatorial resource allocation problems that brought DRL a challenge: A coalition action selection approach is proposed to address the constrained arbitrary action selection in the output, and a computationally efficient stationary input transformation is proposed for ASO inputs. Furthermore, the action selection approach is extended to MADRL to make efficient combinatorial action selection in task-offloading problems, considering various resource constraints. CCM_MADRL_MEC combines the advantage of policy gradient and value functions to make efficient decisions. The CCM_MADRL_MEC has multiple contributions in terms of dimensionality reduction, non-stationarity, and the ability to work with an arbitrary number of agents.

The results of the experiment support the assertions made in the proposed methodologies and algorithms. Coalition action selection is evaluated using an online resource allocation problem with an arbitrary number of task arrivals. It has outperformed the conventional sequential action selection approach in terms of proximity to an offline optimal solution, speed of convergence, and execution costs. It retained close to the offline optimal performance on the online combinatorial resource allocation problem for varying arrival rates, whereas the performance of the sequential approach declines as the arrival rate increases. The coalition action selection is implemented using the

encoder of the transformer neural network. The computationally efficient stationary ASO input transformation is also evaluated for the same problem as the coalition action selection. The stationary ASO input transformation has shown better performance and lower execution complexity than the transformer-based ASO input transformation. Finally, the CCM_MADRL_MEC is evaluated in a task-offloading problem with various types of constraints. The experiment shows that by taking advantage of the policy gradient, value functions, coalition action selection, and per-action DQN, the CCM_MADRL algorithm guarantees superior convergence to the MADDPG and complementary heuristic methodologies.

6.2 Advancement of the State-of-the-Art and Achieved Requirements

This research has advanced the applicability of DRL algorithms for online combinatorial resource allocation problems in many ways, as follows.

- **Distributed solution:** The CCM_MADRL_MEC enables a distributed solution where every entity in the task offloading participates in the decision. The UDs decide their resource allocation and the server makes combinatorial decisions about the actions of the UDs. This makes it the first DRL-based work on task offloading to consider storage constraints. Additionally, the distributed solution allowed for consideration of heterogeneous constraints on the UDs and the server. Continuous-valued resource constraints of UDs are considered in the reward function, whereas discrete-valued constraints such as storage and the number of communication channels are considered in the combinatorial decision.
- **Dimensionality reduction:** CCM_MADRL_MEC minimizes the dimension of the state and action space of the DRL-based task offloading algorithms by considering the number of channels as a constraint in the combinatorial decision. The coalition action selection also minimizes the state space and depth of exploration by selecting multiple actions at a time, which also minimizes the complexity. Moreover, the proposed stationary state transformation minimizes execution complexity.
- **Arbitrary action space and selection:** Existing DRL-based resource allocation algorithms assume that one task arrives at a time. Zhang et al. (2009) considered an arbitrary number of tasks and proposed a sequential action selection approach using Q-learning. Q-learning does not suffer from the challenges of DRL but at the same time misses the advantage of deep learning. The coalition action selection advanced the state-of-the-art in DRL-based resource allocation to consider

the arrival of an arbitrary number of tasks and select an arbitrary number of actions based on the resource allocation.

- **Coordination and Convergence:** Existing MADDPG-based task offloading algorithms converge suboptimally because they use only policy gradient-based agents to make decisions without knowing the decisions of other agents. They use the value function-based critic network only for training. CCM_MADRL_MEC uses both the policy gradient and value function methods to make coordinated decisions. The combinatorial decision approach minimizes the drawback of policy gradient agents being trapped in local optima because the master agent can select the best combination.

The proposed methods and algorithms have addressed the research requirements described in Section 1.2. The coalition action selection approach presented in Chapter 3 addresses Requirement R_I, Requirement R_{III}, and Requirement R_V. The requirement to handle the ASO input of Requirement R_{II} is solved by the stationary ASO input transformation proposed in Chapter 4. The stationary ASO input transformation also minimizes the computational cost in the online execution of Requirement R_{III} and the generalizability to any size and any permutation of Requirement R_V. The CCM_MADRL_MEC in Chapter 5 combines many advantages of existing DRL methods and the methods proposed in Chapter 3 and Chapter 4 to addressed many of the research requirements. Using the advantage of the policy gradient and the value function, it becomes a contribution to meet the dimensionality reduction of Requirement R_{III}, and the coexistence of heterogeneous agents (client and master) in Requirement R_{IV}. Clients decide on their resource requirements and a continuous-valued action proposal to decide the local allocation or propose to the server. The master agent applies a combinatorial decision to decide which tasks should be accepted and assigned to the channels. This reduces the dimension of the DNNs. The per-client DQN used by the master agent also achieves the need for an arbitrary action space of Requirement R_{II}. Furthermore, as discussed in Section 6.4, CCM_MADRL_MEC meets Requirement R_{II}. We did not experiment to prove this because the coalition action selection and the stationary input transformation are supported by an experiment in Chapter 3 and Chapter 4 for arbitrary arrival of tasks, which is evident that it also works for an arbitrary number of clients.

6.3 Limitations of the Research

This research has the following limitations:

- As described in Section 3.5.4.4, The experiment for the coalition action selection and the state transformation has computational resource limitations in extending for varying numbers of task arrivals beyond 20 because we used the Gurobi optimizer with an academic license for computing the offline optimal.

- The stationary ASO input transformation in Chapter 4 is theoretically proved to have a unique transformation for up to an input size of four. The purpose of the research is to show the importance of stationary ASO input transformation over transformer-based transformation in terms of computational costs. However, for large sizes of ASO inputs, the stationary transformation needs to be proportionally customized to be unique.

- The problem description in Chapter 3 is customized from Zhang et al. (2009). As discussed in Section 3.3, we consider the algorithm to be complementary to the work of Zhang et al. (2009) where their work decides which of the tasks to send to which node, and our algorithm decides how the tasks should use the communication resource by deciding which of the tasks should be allocated and which of them should be deferred. we chose their problem description because recent works assume one task arrives at a time. We did not implement their algorithm to show the integration between the algorithms. Instead, we generate the arrival of tasks from a uniform distribution to resemble the decision of their algorithm. This simplification ensures that our experiment is not affected by the convergence of their work.

- The real storage capacities of the server are in GB and TB. However, due to computational resource limits, the CCM_MADRL_MEC is simulated on 50 UDs running their client agents, which do not meet the capacity of a real server. To make the server storage capacity critical for combinatorial decisions to the tasks of the UDs, we assumed a server capacity of 400 MB. Note that with 50 UDs, a total of 51 DRL agents are training simultaneously, 50 of which are client agents and one master agent.

- As with any existing hyperparameter tuning technique in DRL, the learning rates are selected by trial and error as described in Section 5.6.4. We have noticed experimentally that some learning rates led to a horizontal line in the performance evaluation using the evaluation environment. This shows that the weights of the DNNs stop updating after some level of training. We included a code that detects whether the weight of the DNNs is updating or not. We noticed that those learning rates giving a straight line have stopped updating the weight of the DNN of the actor and client agents at different levels of the training, starting from the fifth episode whereas the algorithm with client agents learning rates of 0.0001 and master agent learning rate of 0.001 continued to update the weights of the DNNs until the last episode, showing stable training. This phenomenon

occurred exclusively on the client agents, but not on the master agent. According to the literature, DNNs can stop updating their weights while training due to the vanishing gradient that occurs in policy gradient algorithms because of the type of activation functions they use but this does not happen to value function methods as they use the ReLu activation function. However, the reason why it happens for large learning rates but not for small learning rates is understudied, and we do not analyze it because it is not within the scope of this research.

6.4 Research Extensions, Future Direction, and Applications

Research Extension: The chapters in this research are related to each other. The coalition action selection of Chapter 3 and the application of the per-action DQN for combinatorial action selection in Chapter 4 are applied to the CCM_MADRL_MEC algorithm in Chapter 5 to select an arbitrary number of tasks proposed by the clients until the resource constraint on the server is met. This work can be extended as follows.

- By replacing the concatenated states S and actions A in Equation (5.22) and Equation (5.24) with the transformer as in Chapter 3 or the stationary transformation in Chapter 4, the CCM_MADRL_MEC algorithm can be extended to accept ASO client agents.
- As discussed in Section 5.6.6 for the effects of the values of λ_1 and λ_2 on the experimental results, we evaluated our algorithm using a scalar reward which is a combination of latency and energy consumption. The CCM_MADRL_MEC can be evaluated with a vectorized reward (Vamplew et al., 2022) of latency and energy consumption instead of a scalar reward to benefit from the multiobjective DRL (Hayes et al., 2022). We did not experiment with a vectorized reward because it is not within the scope of the research requirements of the thesis, and multi-objective DRL has already been studied. However, applying the multiobjective DRL with vectorized reward can give a better performance from the task offloading perspective, but it has the same effect on the proposed and the benchmark algorithms.

Research Directions: Although this research addresses the challenges of DRL algorithms in resource allocation algorithms, there are many open research directions as follows.

- The proposed stationary ASO input transformation opens the direction for large ASO inputs.

- While coalition action selection is the first work for an arbitrary action space and arbitrary action selection, it is designed for independent actions. This opens a research direction on how to apply arbitrary action selection for dependent actions such as an ordered output of cities for the capacitated vehicle routing problem.
- CCM_MADRL_MEC opens a research direction on how to apply cooperative combinatorial decisions with multiple master agents.
- As explained in Section 6.3, the observation in stopping the weights of the learning rates opens a research direction on the effect of learning rates and the vanishing gradient.

Application Areas: The main application areas of this research are online combinatorial resource allocation and task-offloading problems. However, they can be customized for other combinatorial problems. The coalition action selection can be customized to the capacitated vehicle routing problem. The client-master cooperation in CCM_MADRL_MEC can also be extended to bargaining mechanisms.

References

Paul Almasan, José Suárez-Varela, Krzysztof Rusek, Pere Barlet-Ros, and Albert Cabellos-Aparicio. Deep reinforcement learning meets graph neural networks: Exploring a routing optimization use case. *Computer Communications*, 196:184–194, 2022.

Tohid Atashbar and Rui Aruhan Shi. Deep reinforcement learning: Emerging trends in macroeconomics and future prospects. *IMF Working Papers*, 2022(259), 2022.

Andrew G Barto, Richard S Sutton, and Charles W Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, pages 834–846, 1983.

Marc G Bellemare, Will Dabney, and Rémi Munos. A distributional perspective on reinforcement learning. *arXiv preprint arXiv:1707.06887*, 2017.

Irwan Bello, Hieu Pham, Quoc V Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940*, 2016.

Lilian Besson and Emilie Kaufmann. Multi-player bandits revisited. In *Algorithmic Learning Theory*, pages 56–92. PMLR, 2018.

Seble Birhanu Engidayehu, Tahira Mahboob, and Min Young Chung. Deep reinforcement learning-based task offloading and resource allocation in mec-enabled wireless networks. In *2022 27th Asia Pacific Conference on Communications (APCC)*, pages 226–230, 2022.

Guillaume Bono, Jilles Steeve Dibangoye, Laëtitia Matignon, Florian Pereyron, and Olivier Simonin. Cooperative multi-agent policy gradient. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 459–476. Springer, 2018.

Robert N Boute, Joren Gijsbrechts, Willem Van Jaarsveld, and Nathalie Vanvuchelen. Deep reinforcement learning for inventory control: A roadmap. *European Journal of Operational Research*, 298(2):401–412, 2022.

Bruno Buchberger and Manuel Kauers. Groebner basis. *Scholarpedia*, 5(10):7763, 2010.

Lucian Buşoniu, Robert Babuška, and Bart De Schutter. *Multi-agent reinforcement learning: An overview*, pages 183–221. Springer, 2010.

Georgios Chalkiadakis and Craig Boutilier. Coordination in multiagent reinforcement learning: A bayesian approach. In *Proceedings of the second International joint Conference on Autonomous Agents and Multiagent Systems*, pages 709–716, 2003.

Yash Chandak, Georgios Theocharous, James Kostas, Scott Jordan, and Philip Thomas. Learning action representations for reinforcement learning. In *International Conference on Machine Learning*, pages 941–950. PMLR, 2019.

Guang Chen, Yueyun Chen, Zhiyuan Mai, Conghui Hao, Meijie Yang, and Liping Du. Incentive-based distributed resource allocation for task offloading and collaborative computing in mec-enabled networks. *IEEE Internet of Things Journal*, 10(10):9077–9091, 2023.

Haoqiang Chen, Yadong Liu, Zongtan Zhou, Dewen Hu, and Ming Zhang. Gama: Graph attention multi-agent reinforcement learning algorithm for cooperation. *Applied Intelligence*, 50(12):4195–4205, 2020.

Xiaoliang Chen, Jiannan Guo, Zuqing Zhu, Roberto Proietti, Alberto Castro, and S. J. B. Yoo. Deep-rmsa: A deep-reinforcement-learning routing, modulation and spectrum assignment agent for elastic optical networks. In *2018 Optical Fiber Communications Conference and Exposition (OFC)*, pages 1–3, 2018.

Mariusz Cichoń. Evolution of longevity through optimal resource allocation. *Proceedings of the Royal Society of London. Series B: Biological Sciences*, 264(1386):1383–1388, 1997.

Cédric Colas, Olivier Sigaud, and Pierre-Yves Oudeyer. A hitchhiker's guide to statistical comparisons of reinforcement learning algorithms. *arXiv preprint arXiv:1904.06979*, 2019.

Jingjing Cui, Yuanwei Liu, and Arumugam Nallanathan. Multi-agent reinforcement learning-based resource allocation for uav networks. *IEEE Transactions on Wireless Communications*, 19(2):729–743, 2019.

Silver David, Singh Satinder, Precup Doina, and S. Sutton Richard. Reward is enough. *Artificial Intelligence*, 299:103535, 2021. ISSN 0004-3702.

Peter Dayan and Yael Niv. Reinforcement learning: the good, the bad and the ugly. *Current Opinion in Neurobiology*, 18(2):185–196, 2008.

Arthur Delarue, Ross Anderson, and Christian Tjandraatmadja. Reinforcement learning with combinatorial actions: An application to vehicle routing. *Advances in Neural Information Processing Systems*, 33:609–620, 2020.

Gabriel Dulac-Arnold, Richard Evans, Hado van Hasselt, Peter Sunehag, Timothy Lillicrap, Jonathan Hunt, Timothy Mann, Theophane Weber, Thomas Degris, and Ben Coppin. Deep reinforcement learning in large discrete action spaces. *arXiv preprint arXiv:1512.07679*, 2015.

Harrison Edwards and Amos Storkey. Towards a neural statistician. 2017.

S. Rasoul Etesami. Online job scheduling on a single machine with general cost functions. In *2021 60th IEEE Conference on Decision and Control (CDC)*, pages 6690–6695, 2021.

Jakob Foerster, Richard Y Chen, Maruan Al-Shedivat, Shimon Whiteson, Pieter Abbeel, and Igor Mordatch. Learning with opponent-learning awareness. In *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*, pages 122–130, 2018a.

Jakob Foerster, Gregory Farquhar, Triantafyllos Afouras, Nantas Nardelli, and Shimon Whiteson. Counterfactual multi-agent policy gradients. In *Thirty-second AAAI Conference on Artificial Intelligence*, page 2974–2982, 2018b.

Tesfay Zemuy Gebrekidan, Sebastian Stein, and Timothy J. Norman. Deep reinforcement learning with coalition action selection for online combinatorial resource allocation with arbitrary action space. In *Proceedings of the 23rd International Conference on Autonomous Agents and Multiagent Systems*, page 660–668, Richland, SC, 2024a. International Foundation for Autonomous Agents and Multiagent Systems. ISBN 9798400704864.

Tesfay Zemuy Gebrekidan, Sebastian Stein, and Timothy J. Norman. Combinatorial client-master multiagent deep reinforcement learning for task offloading in mobile edge computing. In *Proceedings of the 23rd International Conference on Autonomous Agents and Multiagent Systems*, page 2273–2275, Richland, SC, 2024b. International Foundation for Autonomous Agents and Multiagent Systems. ISBN 9798400704864.

Ivo Grondman, Lucian Busoniu, Gabriel AD Lopes, and Robert Babuska. A survey of actor-critic reinforcement learning: Standard and natural policy gradients. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 42(6): 1291–1307, 2012.

Yilan Gu. Solving large markov decision processes (depth paper), 2003.

Ralf Gulde, Marc Tuscher, Akos Csiszar, Oliver Riedel, and Alexander Verl. Deep reinforcement learning using cyclical learning rates. In *2020 Third International Conference on Artificial Intelligence for Industries (AI4I)*, pages 32–35. IEEE, 2020.

Miao Guo, Bin Xin, Jie Chen, and Yipeng Wang. Multi-agent coalition formation by an efficient genetic algorithm with heuristic initialization and repair strategy. *Swarm and Evolutionary Computation*, 55:100686, 2020.

Conor F Hayes, Roxana Rădulescu, Eugenio Bargiacchi, Johan Källström, Matthew Macfarlane, Mathieu Reymond, Timothy Verstraeten, Luisa M Zintgraf, Richard Dazeley, Fredrik Heintz, et al. A practical guide to multi-objective reinforcement learning and planning. *Autonomous Agents and Multi-Agent Systems*, 36(1):26, 2022.

He He, Jordan Boyd-Graber, Kevin Kwok, and Hal Daumé III. Opponent modeling in deep reinforcement learning. In *International Conference on Machine Learning*, pages 1804–1813, 2016a.

Ji He, Jianshu Chen, Xiaodong He, Jianfeng Gao, Lihong Li, Li Deng, and Mari Ostendorf. Deep reinforcement learning with a natural language action space. *arXiv preprint arXiv:1511.04636*, 2015.

Ji He, Mari Ostendorf, Xiaodong He, Jianshu Chen, Jianfeng Gao, Lihong Li, and Li Deng. Deep reinforcement learning with a combinatorial action space for predicting popular Reddit threads. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 1838–1848, 2016b.

Matteo Hessel, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9:1735–1780, 1997.

Yujing Hu, Qing Da, Anxiang Zeng, Yang Yu, and Yinghui Xu. Reinforcement learning to rank in e-commerce search engine: Formalization, analysis, and application. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 368–377, 2018.

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

Liang Huang, Xu Feng, Cheng Zhang, Liping Qian, and Yuan Wu. Deep reinforcement learning-based joint task offloading and bandwidth allocation for multi-user mobile edge computing. *Digital Communications and Networks*, 5(1):10 – 17, 2019a. ISSN 2352-8648. .

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

Maria Huegle, Gabriel Kalweit, Branka Mirchevska, Moritz Werling, and Joschka Boedecker. Dynamic input for deep reinforcement learning in autonomous driving. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 7566–7573. IEEE, 2019.

Ehsan Imani, Eric Graves, and Martha White. An off-policy policy gradient theorem using emphatic weightings. In *Advances in Neural Information Processing Systems*, pages 96–106, 2018.

Wacharawan Intayoad, Chayapol Kamyod, and Punnarumol Temdee. Reinforcement learning based on contextual bandits for personalized online learning recommendation systems. *Wireless Personal Communications*, pages 1–16, 2020.

Akhirul Islam, Arindam Debnath, Manojit Ghose, and Suchetana Chakraborty. A survey on task offloading in multi-access edge computing. *Journal of Systems Architecture*, 118:102225, 2021.

Tommi Jaakkola, Satinder P Singh, and Michael I Jordan. Reinforcement learning algorithm for partially observable markov decision problems. In *Advances in Neural Information Processing Systems*, pages 345–352, 1995.

Michael Janner, Qiyang Li, and Sergey Levine. Offline reinforcement learning as one big sequence modeling problem. In A. Beygelzimer, Y. Dauphin, P. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, 2021.

Wei Jiang, Daquan Feng, Yao Sun, Gang Feng, Zhenzhong Wang, and Xiang-Gen Xia. Joint computation offloading and resource allocation for d2d-assisted mobile edge computing. *IEEE Transactions on Services Computing*, 16(3):1949–1963, 2023. .

Te-Yi Kan, Yao Chiang, and Hung-Yu Wei. Task offloading and resource allocation in mobile-edge computing system. In *2018 27th Wireless and Optical Communication Conference (WOCC)*, pages 1–4, 2018. .

Jasneet Kaur, M Arif Khan, Mohsin Iftikhar, Muhammad Imran, and Qazi Emad Ul Haq. Machine learning techniques for 5g and beyond. *IEEE Access*, 9:23472–23488, 2021.

Bernhard Kutzler and Sabine Stifter. Automated geometry theorem proving using buchberger's algorithm. In *Proceedings of the fifth ACM symposium on Symbolic and algebraic computation*, pages 209–214, 1986.

Juho Lee, Yoonho Lee, Jungtaek Kim, Adam Kosiorek, Seungjin Choi, and Yee Whye Teh. Set transformer: A framework for attention-based permutation-invariant neural networks. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 3744–3753. PMLR, 09–15 Jun 2019.

Lihong Li, Wei Chu, John Langford, and Robert E Schapire. A contextual-bandit approach to personalized news article recommendation. In *Proceedings of the 19th International Conference on World wide web*, pages 661–670, 2010.

Timothy Lillicrap, Jonathan Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

Chenyi Liu, Mingwei Xu, Yuan Yang, and Nan Geng. Drl-or: Deep reinforcement learning-based online routing for multi-type service requirements. In *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications*, pages 1–10, 2021.

Feng Liu, Ruiming Tang, Xutao Li, Weinan Zhang, Yunming Ye, Haokun Chen, Huifeng Guo, and Yuzhou Zhang. Deep reinforcement learning based recommendation with explicit user-item interactions modeling. *arXiv preprint arXiv:1810.12027*, 2018.

Xiaowei Liu, Shuwen Jiang, and Yi Wu. A novel deep reinforcement learning approach for task offloading in mec systems. *Applied Sciences*, 12(21), 2022.

Angela Lopez-del Rio, Maria Martin, Alexandre Perera-Lluna, and Rabie Saidi. Effect of sequence padding on the performance of deep learning models in archaeal protein functional prediction. *Scientific reports*, 10(1):1–14, 2020.

Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, Pieter Abbeel, and Igor Mordatch. Multi-agent actor-critic for mixed cooperative-competitive environments. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pages 6382–6393, 2017.

Shuaibing Lu, Jie Wu, Jiamei Shi, Pengfan Lu, Juan Fang, and Haiming Liu. A dynamic service placement based on deep reinforcement learning in mobile edge computing. *Network*, 2(1):106–122, 2022.

Nguyen Cong Luong, Dinh Thai Hoang, Shimin Gong, Dusit Niyato, Ping Wang, Senior Member, Ying-chang Liang, and Dong In Kim. Applications of Deep Reinforcement Learning in Communications and Networking : A Survey. *IEEE Communications Surveys & Tutorials*, 21(4):3133–3174, 2019.

Xueguang Lyu, Yuchen Xiao, Brett Daley, and Christopher Amato. Contrasting centralized and decentralized critics in multi-agent reinforcement learning. In *Proceedings of the 20th International Conference on Autonomous Agents and MultiAgent Systems*, pages 844–852, 2021.

Michael Pendo John Mahenge, Chunlin Li, and Camilius A Sanga. Energy-efficient task offloading strategy in mobile edge computing for resource-intensive mobile applications. *Digital Communications and Networks*, 8(6):1048–1058, 2022.

Warwick Masson, Pravesh Ranchod, and George Konidaris. Reinforcement learning with parameterized actions. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30, pages 1934–1940, 2016.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*, pages 1928–1937, 2016.

Dmitry Mukhutdinov, Andrey Filchenkov, Anatoly Shalyto, and Valeriy Vyatkin. Multi-agent deep learning for simultaneous optimization for time and energy in distributed routing system. *Future Generation Computer Systems*, 94:587–600, 2019.

D. C. Nguyen, M. Ding, P. N. Pathirana, A. Seneviratne, J. Li, and H. Poor. Cooperative task offloading and block mining in blockchain-based edge computing with multi-agent deep reinforcement learning. *IEEE Transactions on Mobile Computing*, 22(04): 2021–2037, 2023.

T. T. Nguyen, N. D. Nguyen, and S. Nahavandi. Deep reinforcement learning for multi-agent systems: A review of challenges, solutions, and applications. *IEEE Transactions on Cybernetics*, 50(9):3826–3839, 2020.

Rui Nian, Jinfeng Liu, and Biao Huang. A review on reinforcement learning: Introduction and applications in industrial process control. *Computers & Chemical Engineering*, 139:106886, 2020a. ISSN 0098-1354.

Rui Nian, Jinfeng Liu, and Biao Huang. A review on reinforcement learning: Introduction and applications in industrial process control. *Computers and Chemical Engineering*, page 106886, 2020b.

Joel Oren, Chana Ross, Maksym Lefarov, Felix Richter, Ayal Taitler, Zohar Feldman, Dotan Di Castro, and Christian Daniel. Solo: search online, learn offline for combinatorial optimization problems. In *Proceedings of the International Symposium on Combinatorial Search*, volume 12, pages 97–105, 2021.

Kei Ota, Tomoaki Oiki, Devesh Jha, Toshisada Mariyama, and Daniel Nikovski. Can increasing input dimensionality improve deep reinforcement learning? In *International conference on machine learning*, pages 7424–7433. PMLR, 2020.

Sindhu Padakandla, KJ Prabuchandran, and Shalabh Bhatnagar. Reinforcement learning algorithm for non-stationary environments. *Applied Intelligence*, 50(11):3590–3606, 2020.

Bo Peng, Jiahai Wang, and Zizhen Zhang. A deep reinforcement learning algorithm using dynamic attention model for vehicle routing problems. In *Artificial Intelligence Algorithms and Applications: 11th International Symposium, ISICA 2019, Guangzhou, China, November 16–17, 2019, Revised Selected Papers 11*, pages 636–650. Springer, 2020.

Martin L. Puterman. Markov decision processes: Discrete stochastic dynamic programming. In *Wiley Series in Probability and Statistics*, 1994.

Charles R Qi, Hao Su, Kaichun Mo, and Leonidas J Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 652–660, 2017.

Tabish Rashid, Mikayel Samvelyan, Christian Schroeder, Gregory Farquhar, Jakob Foerster, and Shimon Whiteson. Qmix: Monotonic value function factorisation for deep multi-agent reinforcement learning. pages 4295–4304, 2018.

Gavin Adrian Rummery and Mahesan Niranjan. On-line q-learning using connectionist systems. Technical report, F-INFENG/TR, 1994.

Kuanishbay Sadatdiynov, Laizhong Cui, Lei Zhang, Joshua Zhexue Huang, Salman Salloum, and Mohammad Sultan Mahmud. A review of optimization methods for computation offloading in edge computing networks. *Digital Communications and Networks*, 9(2):450–461, 2023.

Dileep Kumar Sajnani, Abdul Rasheed Mahesar, Abdullah Lakhan, and Irfan Ali Jamali. Latency aware and service delay with task scheduling in mobile edge computing. *Communications and Network*, 10(04):127–141, 2018.

Samriddhi Sarkar, Mariana Malta, and Animesh Dutta. A survey on applications of coalition formation in multi-agent systems. 34, 02 2022.

Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009.

Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. 2016.

Yuxiang Sheng, Huawei Ma, and Wei Xia. A pointer neural network for the vehicle routing problem with task priority and limited resources. *Inf. Technol. Control.*, 49: 237–248, 2020.

David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *Proceedings of the 31st International Conference on International Conference on Machine Learning-Volume 32*, pages I–387, 2014.

Kyunghwan Son, Daewoo Kim, Wan Ju Kang, David Earl Hostallero, and Yung Yi. Qtran: Learning to factorize with transformation for cooperative multi-agent reinforcement learning. In *International Conference on Machine Learning*, pages 5887–5896. PMLR, 2019.

Sebastian Stein, Mateusz Ochal, Ioana-Adriana Moisoiu, Enrico Gerding, Raghu Ganti, Ting He, and Tom La Porta. Strategyproof reinforcement learning for online resource allocation. pages 1296–1304, 2020.

Guolin Sun, Tesfay Zemuy Gebrekidan, Gordon Owusu Boateng, Daniel Ayepah-Mensah, and Wei Jiang. Dynamic reservation and deep reinforcement learning based autonomous resource slicing for virtualized radio access networks. *IEEE Access*, 7: 45758–45772, 2019.

Esther M Sundermann, Martin J Lercher, and David Heckmann. Modeling photosynthetic resource allocation connects physiology with evolutionary environments. *Scientific Reports*, 11(1):15979, 2021.

Peter Sunehag, Guy Lever, Audrunas Gruslys, Wojciech Marian Czarnecki, Vinícius Flores Zambaldi, Max Jaderberg, Marc Lanctot, Nicolas Sonnerat, Joel Z Leibo, Karl Tuyls, et al. Value-decomposition networks for cooperative multi-agent learning based on team reward. In *AAMAS*, pages 2085–2087, 2018.

Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. *Advances in neural information processing systems*, 27, 2014.

Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

Richard S Sutton, David A McAllester, Satinder P Singh, Yishay Mansour, et al. Policy gradient methods for reinforcement learning with function approximation. In *NIPs*, volume 99, pages 1057–1063. Citeseer, 1999.

Ming Tan. Multi-agent reinforcement learning: Independent vs. cooperative agents. In *Proceedings of the tenth international conference on machine learning*, pages 330–337, 1993.

Xiaoqi Tan, Alberto Leon-Garcia, Yuan Wu, and Danny HK Tsang. Online combinatorial auctions for resource allocation with supply costs and capacity limits. *IEEE Journal on Selected Areas in Communications*, 38(4):655–668, 2020.

Ming Tang and Vincent WS Wong. Deep reinforcement learning for task offloading in mobile edge computing systems. *arXiv*, pages arXiv–2005, 2020.

Yujin Tang and David Ha. The sensory neuron as a transformer: Permutation-invariant neural networks for reinforcement learning. *Advances in Neural Information Processing Systems*, 34:22574–22587, 2021.

Arash Tavakoli, Fabio Pardo, and Petar Kormushev. Action branching architectures for deep reinforcement learning. In *Thirty-Second AAAI Conference on Artificial Intelligence*, pages 4131–4138, 2018.

Mulugeta Kassaw Tefera, Shengbing Zhang, and Zengwang Jin. Deep reinforcement learning-assisted optimization for resource allocation in downlink ofdma cooperative systems. *Entropy*, 25(3):413, 2023.

Peter Vamplew, Benjamin J Smith, Johan Källström, Gabriel Ramos, Roxana Rădulescu, Diederik M Roijers, Conor F Hayes, Fredrik Heintz, Patrick Mannion, Pieter JK Libin, et al. Scalar reward is not enough: A response to silver, singh, precup and sutton (2021). *Autonomous Agents and Multi-Agent Systems*, 36(2):41, 2022.

Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Thirtieth AAAI Conference on Artificial Intelligence*, pages 2094–2100, 2016.

Martijn van Otterlo and Marco Wiering. *Reinforcement Learning and Markov Decision Processes*, pages 3–42. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-27645-3. .

Aparna S Varde, Shuhui Ma, Mohammed Maniruzzaman, David C Brown, Elke A Rundensteiner, and Richard D SissonJR. Comparing mathematical and heuristic approaches for scientific data analysis. *AI EDAM*, 22(1):53–69, 2008.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Petar Veličković. Everything is connected: Graph neural networks. *Current Opinion in Structural Biology*, 79:102538, 2023.

Fatema Vhora and Jay Gandhi. A comprehensive survey on mobile edge computing: Challenges, tools, applications. In *2020 Fourth International Conference on Computing Methodologies and Communication (ICCMC)*, pages 49–55, 2020.

Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. *Advances in neural information processing systems*, 28, 2015.

Jun-Bo Wang, Junyuan Wang, Yongpeng Wu, Jin-Yuan Wang, Huiling Zhu, Min Lin, and Jiangzhou Wang. A machine learning framework for resource allocation assisted by cloud computing. *IEEE Network*, 32(2):144–151, 2018a. .

Suzhen Wang, Zhongbo Hu, Yongchen Deng, and Lisha Hu. Game-theory-based task offloading and resource scheduling in cloud-edge collaborative systems. *Applied Sciences*, 12(12), 2022.

Weixun Wang, Jianye Hao, Yixi Wang, and Matthew Taylor. Achieving cooperation through deep multiagent reinforcement learning in sequential prisoner's dilemmas. In *Proceedings of the First International Conference on Distributed Artificial Intelligence*, pages 1–7, 2019.

Zhi Wang, Lihua Li, Yue Xu, Hui Tian, and Shuguang Cui. Handover optimization via asynchronous multi-user deep reinforcement learning. In *2018 IEEE International Conference on Communications (ICC)*, pages 1–6. IEEE, 2018b.

Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. Dueling network architectures for deep reinforcement learning. In *International Conference on Machine Learning*, pages 1995–2003, 2016.

Christopher J.C.H. Watkins and Peter Dayan. Technical note "q-learning". *Machine Learning (cited on page 89)*, 1992.

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

Ian H Witten. An adaptive optimal controller for discrete-time markov environments. *Inf. Control.*, 34(4):286–295, 1977.

Wenbo Wu, Zhengdong Huang, Jiani Zeng, and Kuan Fan. A fast decision-making method for process planning with dynamic machining resources via deep reinforcement learning. *Journal of manufacturing systems*, 58:392–411, 2021a.

Zhanghao Wu, Paras Jain, Matthew Wright, Azalia Mirhoseini, Joseph E Gonzalez, and Ion Stoica. Representing long-range context for graph neural networks with global attention. *Advances in Neural Information Processing Systems*, 34:13266–13279, 2021b.

Jianyu Xiong, Peng Guo, Yi Wang, Xiangyin Meng, Jian Zhang, Linmao Qian, and Zhenglin Yu. Multi-agent deep reinforcement learning for task offloading in group distributed manufacturing systems. *Engineering Applications of Artificial Intelligence*, 118:105710, 2023.

Jianqiao Xu, Zhuohan Xu, and Bing Shi. Deep reinforcement learning based resource allocation strategy in cloud-edge computing system. *Frontiers in Bioengineering and Biotechnology*, 10:908056, 2022.

Jia Yan, Suzhi Bi, and Ying-Jun Angela Zhang. Optimal offloading and resource allocation in mobile-edge computing with inter-user task dependency. In *2018 IEEE Global Communications Conference (GLOBECOM)*, pages 1–8, 2018. .

Fan Yao, Renqin Cai, and Hongning Wang. Reversible action design for combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:2102.07210*, 2021.

Manzil Zaheer, Satwik Kottur, Siamak Ravanbhakhsh, Barnabás Póczos, Ruslan Salakhutdinov, and Alexander J Smola. Deep sets. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pages 3394–3404, 2017.

Chongjie Zhang, Victor R Lesser, Prashant J Shenoy, et al. A multi-agent learning approach to online distributed resource allocation. In *Ijcai*, volume 9, pages 361–366. Citeseer, 2009.

Jing Zhang, Jun Du, Yuan Shen, and Jian Wang. Dynamic computation offloading with energy harvesting devices: A hybrid-decision-based deep reinforcement learning approach. *IEEE Internet of Things Journal*, 7(10):9303–9317, 2020. .

Kaiqing Zhang, Zhuoran Yang, Han Liu, Tong Zhang, and Tamer Basar. Fully decentralized multi-agent reinforcement learning with networked agents. In *International Conference on Machine Learning*, pages 5872–5881. PMLR, 2018.

Kaiqing Zhang, Zhuoran Yang, and Tamer Başar. Multi-agent reinforcement learning: A selective overview of theories and algorithms. *Handbook of Reinforcement Learning and Control*, pages 321–384, 2021.

Weiwen Zhang, Yonggang Wen, Jun Wu, and Hui Li. Toward a unified elastic computing platform for smartphones with cloud support. *IEEE Network*, 27(5):34–40, 2013.

Yin Zhang, Rong Jin, and Zhi-Hua Zhou. Understanding bag-of-words model: a statistical framework. *International Journal of Machine Learning and Cybernetics*, 1(1-4): 43–52, 2010.

Ziyao Zhang, Liang Ma, Konstantinos Poularakis, Kin K Leung, Jeremy Tucker, and Ananthram Swami. Macs: Deep reinforcement learning based SDN controller synchronization policy design. In *2019 IEEE 27th International Conference on Network Protocols (ICNP)*, pages 1–11. IEEE, 2019.

Xiangyu Zhao, Liang Zhang, Long Xia, Zhuoye Ding, Dawei Yin, and Jiliang Tang. Deep reinforcement learning for list-wise recommendations. *arXiv preprint arXiv:1801.00209*, 2017.