

Computation Offloading and Resource Allocation in Mobile Edge Computing

by

Anuar Karzhaubayev

Submitted to the Department of Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

at the

NAZARBAYEV UNIVERSITY

December 2025

© Nazarbayev University 2025. All rights reserved.

Author
Department of Computer Science
07

Certified by
Murzabulatov Meiram
Assistant Professor
Thesis Supervisor

Certified by
Dinh-Mao Bui
Assistant Professor
Thesis Supervisor

Accepted by
Adnan Yazici
Dean, School of Engineering and Digital Sciences

Computation Offloading and Resource Allocation in Mobile Edge Computing

by

Anuar Karzhaubayev

Submitted to the Department of Computer Science
on 07, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science

Abstract

Mobile Edge Computing (MEC) allows resource-constrained devices to offload computational tasks to nearby edge servers to improve efficiency and performance. However, designing efficient task scheduling and offloading strategies in MEC is challenging due to dynamic workloads and resource constraints at the edge. Most existing approaches treat offloading, scheduling, and energy efficiency as separate optimization problems, which limits their effectiveness. This thesis proposes an energy-aware task scheduling framework for MEC with partial computation offloading using Deep Reinforcement Learning (DRL). A Deep Q-Network (DQN) based scheduler is proposed to determine whether and where to offload a task and to what extent. The task execution is split between the end device and the edge-fog nodes. The DQN takes into account system and task states to determine the best task execution schedule. The reward function of the DQN is defined using task execution latency, energy consumption, and deadline constraints. The framework is implemented and evaluated on a distributed edge-fog network. The results show that the DQN-based scheduler learns adaptive offloading behavior and performs strongly in mixed workloads, while avoiding unnecessary offloading in communication-heavy workloads. However, its advantage is workload-dependent: full offloading and oracle policies can perform better in scenarios where aggressive edge execution is naturally beneficial. The results show that DQN based task scheduling is an effective technique for MEC systems. Furthermore, the use of partial offloading improves system performance and resource utilization.

Thesis Supervisor: Murzabulatov Meiram
Title: Assistant Professor

Thesis Supervisor: Dinh-Mao Bui
Title: Assistant Professor

Contents

1	Introduction	13
1.1	Background and Motivation	13
1.1.1	Challenges of AI Models on Mobile Devices	13
1.2	Problem Statement	15
1.3	Research Aim and Objectives	16
1.4	Research Questions	17
1.5	Research Contributions	17
2	Background	19
2.1	Mobile Edge Computing	19
2.2	Computation Offloading Models	20
2.3	Partial Offloading	20
2.4	Energy Efficiency in MEC	20
2.5	Scheduling and Resource Allocation	21
2.6	Deep Reinforcement Learning Fundamentals	21
3	Related Work and Research Gap	23
3.1	Binary and Partial Offloading Approaches	23
3.2	Optimization-Based MEC Scheduling	24
3.3	DRL-Based MEC Scheduling	25
3.4	Energy-Aware Offloading Methods	26
3.5	Limitations of Existing Work	27
3.6	Research Gap and Thesis Positioning	28

4	System Model and Problem Formulation	29
4.1	Architecture Overview	30
4.2	Task and Workload Model	31
4.3	Communication Model	32
4.4	Computation Model	33
4.5	Energy Model	34
4.6	Optimization Objective	36
4.7	Constraints	36
4.8	Problem Definition	38
5	Proposed Method	39
5.1	Framework Overview	39
5.2	DRL-Based Scheduler Design	41
5.3	State Space	42
5.4	Action Space	44
5.5	Reward Function	45
5.6	Training Procedure	46
5.7	Online Decision Algorithm	48
5.8	Design Rationale	49
6	Results	51
6.1	Experimental Setup	51
6.2	Compared Policies	52
6.3	Deadline Satisfaction	53
6.4	Energy Consumption	55
6.5	Latency Performance	56
6.6	Offloading Behavior	57
6.7	Discussion	58
7	Conclusion	61
7.1	Summary of Findings	61

7.2	Answer to Research Questions	62
7.3	Contributions	63
7.4	Limitations and Future Work	64
7.5	Final Remarks	65

List of Figures

6-1	Deadline success rate across workload scenarios	54
6-2	Modeled task-level energy consumption across workload scenarios . .	55
6-3	Average latency across workload scenarios	56
6-4	Average offloading ratio across workload scenarios	57

List of Tables

6.1	Experimental parameters and DQN hyperparameters	52
-----	---	----

Chapter 1

Introduction

1.1 Background and Motivation

1.1.1 Challenges of AI Models on Mobile Devices

Thermal and Latency Challenges: High-load executions of AI models on mobile devices also pose challenges in the form of thermal throttling and latency increases in the device response time. Though local execution of models avoids the latency introduced by data transmission, it does not necessarily introduce low end-to-end response times when the mobile device is under heavy load or when executing large AI models. These drawbacks to local execution create a mismatch between the requirements of intelligent applications and the capabilities of the mobile devices [1, 2].

Mobile Edge Computing as a Solution: One solution to these problems is Mobile Edge Computing (MEC), also known as Multi-access Edge Computing (MEC). MEC extends cloud computing functions to the wireless network edge, closer to the end-users. MEC combines the computational power of edge servers with the low latency of the communication network between the mobile device and the edge server, creating a system that is able to support the demands of intelligent applications [3, 1, 4].

Offloading Mobile Applications: One way that MEC addresses the need for distributed mobile applications is through offloading portions of the applications to external infrastructure. Offloading decisions may be made through *binary offloading* strategies, wherein each mobile application is either executed entirely on the mobile device or entirely on an external (typically remote) server. This strategy, however, is often too coarse-grained for mobile applications that contain multiple different components or functions [5, 6, 4]. More fine-grained *partial offloading* strategies allow mobile applications to execute some of their functions locally on the device while offloading other functions to external servers. Examples of such strategies include offloading some machine learning model layers while executing others locally on the device [5, 7, 8, 9].

Scheduling Challenges: These types of offloading strategies introduce complexity into the system regarding scheduling decisions. Not only must decisions be made regarding which functions of a mobile application are to be offloaded, but also where they are to be offloaded, and under what current conditions in the network and in the devices involved. Consequently, scheduling decisions play a key role in the function of MEC systems [4, 5, 10, 8].

Energy Efficiency Challenges: Energy efficiency is also a consideration in MEC systems. While offloading mobile applications to external servers avoids some of the energy requirements of mobile devices, those external servers are also energy-constrained due to their distributed nature. A scheduling policy that avoids energy considerations at the mobile device may shift the energy costs of an application from the mobile device to the external servers. Therefore, it is necessary to develop scheduling policies that consider energy efficiency at both the mobile device and external server components of MEC [2, 10, 4].

Deep Reinforcement Learning Approaches: Recent mobile computing research has investigated methods based on deep reinforcement learning for mobile edge computing systems. Deep reinforcement learning strategies have been used to address a

variety of scheduling problems within MEC systems, including server selection, task partitioning, resource allocation, and deadline-aware offloading. Many of these strategies could be applied to distributed systems that use strategies of partial offloading. These factors, therefore, contribute to the motivation for the research described in this thesis [4, 5, 10, 7, 8, 9].

1.2 Problem Statement

Despite the numerous proposed solutions for MEC and computation offloading, there is still a great deal of limitation among existing approaches when it comes to supporting modern, resource-demanding applications. Many existing approaches to modeling computation offloading, for instance, do not take into account the internal structure of the applications that are to be offloaded. By ignoring these structures and their potential impact upon energy-consumption decisions, for instance, those approaches may lead to inefficient decisions regarding the energy that should be allocated to each task to complete it [5, 4, 9].

While there are a variety of potential solutions to these problems, one general class of solutions is to develop an offloading scheduler that decides whether a task is to be executed locally, partially offloaded, or entirely offloaded to an edge server based upon the requirements of that task, the resources available to the device, and the network condition between the device and the edge cloud [10, 8, 9].

These decisions are complicated by the fact that each task to be executed may have different requirements of each of these factors. In addition, the resources available at the edge may differ from device to device; they may have different processing capabilities (CPU, GPU), memory size, and network availability [3, 4, 7]. Finally, the network condition between the device and edge may also change; wireless transmissions experience variable delays due to interference with other devices on the network, mobility of those devices, signal fading, and access contention to the wireless network [5, 10, 8].

Additionally, the applications that are to be offloaded may have variable latency

constraints; many of the applications that are to be scheduled and offloaded must meet certain deadlines to remain usable themselves. Yet, minimizing the latencies associated with offloading can lead to increased energy consumption by the devices involved [10, 2, 9]. Thus, the offloading scheduler must consider each of these factors in making its decisions; the problem becomes a multi-objective problem for the scheduler to solve.

Finally, the knowledge that the scheduler has of the task to be performed and the network condition may also be uncertain; there may be a gap in the literature regarding approaches to scheduling decisions under such uncertainty [4, 10, 8, 9].

Thus, one research problem that is to be investigated in this thesis is how deep reinforcement learning can be used to develop an offloading scheduler that balances modeled task-level energy consumption with latency and deadline satisfaction while maintaining requirements for latency, adaptively to dynamic and heterogeneous edge networks.

1.3 Research Aim and Objectives

The aim of this research is to develop an energy-efficient approach for offloading computation in mobile edge computing environments by using deep reinforcement learning.

Specific objectives for this research include formulating a deep reinforcement learning-based approach to offloading computation in mobile edge computing environments for energy efficiency. More specifically, the objectives include: formulating a model of the mobile edge computing environment in which tasks may be performed on the device, partially offloaded, or offloaded to the edge; formulating the offloading decision problem as a deep reinforcement learning problem, including defining the state, action, and reward components of that problem; developing a deep reinforcement learning scheduler that adapts its decisions according to the dynamic nature of the MEC environment; and evaluating the developed deep reinforcement learning scheduler against baseline methods for offloading tasks in mobile edge computing

environments.

1.4 Research Questions

The following research questions will guide this dissertation.

First, in what ways should the problem of making decisions regarding offloading of tasks in mobile edge computing environments be modeled in order to consider both latency and energy consumption of tasks?

Second, to what extent will a deep reinforcement learning-based scheduler outperform existing methods for offloading tasks in mobile edge computing environments?

Third, how does offloading of tasks partially versus entirely to the mobile edge computing environment compare in terms of meeting latency requirements, minimizing energy consumption of mobile devices, and utilizing the resources of that edge computing environment?

Fourth, how does the learned scheduler adapt its offloading behavior across different workload classes and system conditions?

1.5 Research Contributions

The contributions of this dissertation are as follows:

- **A formal problem formulation for energy-efficient partial offloading in MEC.**

The problem of minimizing energy consumption in MEC while offloading tasks partially to the edge is formulated as a scheduling problem. The problem is formulated to minimize energy consumption and latency while taking into account the heterogeneous nature of the available resources at the edge data centers.

- **A deep reinforcement learning-based scheduling framework for adaptive decision-making.**

A deep reinforcement learning-based offloading scheduler for mobile edge computing environments is proposed. This approach learns actions (decisions of how to offload tasks) by interacting with the environment, which allows for the scheduler to adapt its strategy for offloading tasks from mobile devices to the edge in response to changes in the characteristics of mobile devices or mobile edge computing environments.

- **An energy-aware reward and state design.**

The deep reinforcement learning-based scheduler is designed to account for modeled task-level energy, latency, and deadline satisfaction. This approach includes the careful consideration of which state variables to include in the model, as well as the definition of the reward function for the reinforcement learning algorithm.

- **An evaluation of the deep reinforcement learning-based scheduling framework against baselines.**

The deep reinforcement learning algorithm that is developed in this dissertation will be evaluated against a number of baseline methods for offloading tasks in mobile edge computing environments, such as locally executing the tasks on the mobile devices, offloading the tasks entirely to the edge cloud, or employing other heuristic-based methods for offloading tasks.

Chapter 2

Background

2.1 Mobile Edge Computing

Mobile Edge Computing (MEC) or Multi-Access Edge Computing (MEC) refers to distributed computing framework where the computations and storage of data is performed closer to the end users rather than performing the tasks on remote cloud servers [3, 4]. Since the edge servers are physically closer to the end users, there is a lower round-trip delay in communicating with these servers as compared to communicating with cloud servers. This lower delay makes edge computing suitable for applications where there needs to be a real-time response to the data that is collected from the end users' devices. However, edge computing environments are typically constrained and dynamic in their capabilities. The capabilities of the edge servers can differ from device to device in terms of their processor architecture, memory, and the amount of data that must be processed. Additionally, the wireless communication between the devices and the edge server can pose issues to the data rates and the communication delay between the two [4, 1].

Due to these characteristics of edge computing, tasks related to offloading and scheduling of computation tasks become crucial to the operation of MEC systems.

2.2 Computation Offloading Models

Offloading refers to the process of offloading the computation of an app to external servers. This is often employed to increase the feasibility of an app under given constraints to processing power, battery life, or latency [11, 12]. Computing locally has the benefit of avoiding communication overhead and preserving user data privacy but is limited by the processing power of the device itself. Offloading completely to external servers allows access to more powerful computing resources but introduces a communication delay between device and server, depending on the network conditions. Binary offloading allows for the simplification of determining where to execute the app’s tasks, either locally or offloaded to external servers. This method is very common but can be too restrictive for many modern apps [11, 13].

2.3 Partial Offloading

Partial offloading allows only a portion of a task to be executed at a remote device. The remainder of the task is executed at the local device. This type of offloading is appropriate for applications where tasks can be divided into sub-tasks, such as neural networks [14, 15]. One of the main advantages of using a form of partial offloading is the adaptability of this type of offloading strategy. Because only a portion of the task needs to be offloaded to another device, there is more flexibility in how tasks can be distributed between devices. The disadvantages of this strategy include the additional complexity that is introduced in order to divide the task, and to manage the communication between devices. Additionally, because the conditions under which a task is to be offloaded may be dynamic, any decisions regarding offloading may not remain optimal over time [16, 17].

2.4 Energy Efficiency in MEC

Energy efficiency considers the energy consumption of devices and edge infrastructure. Minimizing energy consumption is especially important for mobile devices with

limited battery capacities [12].

Energy consumption relates to the computations performed by a device and the communications between devices. Offloading can reduce the energy consumed performing computations on the device but may increase the energy used for transmissions. Performing computations locally may increase the energy consumed in processing. The design of MEC systems considers the tradeoff between energy and latency. MEC systems must balance energy efficiency with quality-of-service requirements [18, 15].

2.5 Scheduling and Resource Allocation

Scheduling and resource allocation determine how the tasks are to be executed across the distributed MEC resources. The scheduling policy determines where and when the tasks are to be executed, while the resource allocation policy determines how much of the computational and communication resources are to be allocated to each task [19].

These policies are inherently multi-objective as they aim to minimize various objectives such as latency, energy consumption, throughput, and maximizing resource utilization. Furthermore, if the mobile devices are performing partial offloading of their tasks to the MEC resources, the scheduling policy must also decide how to partition the tasks and allocate them among several MEC nodes. The complexity of the scheduling policy increases even further in heterogeneous and dynamic MEC environments where adaptive scheduling policies are required [15, 20].

2.6 Deep Reinforcement Learning Fundamentals

Reinforcement learning (RL) is a framework for modeling decision-making in systems with dynamic environments and stochastic transitions between states. RL agents learn to maximize their long-term performance by interacting with their environment [21].

Deep reinforcement learning (DRL) is a subfield of RL that uses neural networks to approximate the optimal policies or value functions for a given problem. In the context of MEC, deep reinforcement learning techniques can learn policies that optimize various objectives in MEC systems while taking into account the characteristics of the workloads and the network [16, 20].

Chapter 3

Related Work and Research Gap

3.1 Binary and Partial Offloading Approaches

Offloading granularity can be discussed in terms of two different yet related concepts: binary offloading and partial offloading. Research into MEC has mostly used the concept of binary offloading for its simplicity; however, more recent surveys indicate a shift in research towards the consideration of partial offloading methods [11, 12, 13].

In methods employing binary offloading, the decision must be made as to whether the task should be executed at all on the mobile device or if it should be offloaded to the external MEC server. Such approaches often make the assumption that it is more efficient to move the entire task rather than to divide that task into its individual components and to determine where each component of the task may best be executed [11, 22, 16]. With the complexity of many mobile applications today, such approaches may not be the most efficient in relation to energy consumption and time required to execute the tasks [15, 14, 20].

Partial offloading is one technique that aims to address some of the limitations of the requirement of binary offloading decisions. Partial offloading allows for only some of a task to be executed either on the mobile device or on the remote edge server; the remainder of the task can then be executed either at the same device or at the external MEC server. The ability to make such a decision allows for more expressiveness of the offloading scheduler [14, 15, 20].

Despite these advantages of partial offloading, there are also some difficulties associated with it. Offloading tasks that are not divisible into individual components may lead to difficulties in efficiently transmitting intermediate data between mobile devices and edge servers. Furthermore, coordination between the local execution of the task and the distributed execution of that same task may create difficulties in implementing synchronization between the two processes [15, 17, 16].

The distinction between binary and partial offloading approaches has importance for the offloading scheduler. For binary offloading, the scheduler has to make a decision as to whether remote execution is to occur or if the task is to remain on the mobile device. For approaches that use partial offloading, however, the scheduler must also make decisions regarding where to perform the split between the task to be performed on the device and the task to be offloaded, how energy consumption is to be affected by the offloading of the task, and how to allocate the shared computational resources among the various workloads that may be offloaded [11, 22, 16].

3.2 Optimization-Based MEC Scheduling

One of the main approaches to MEC task scheduling is the use of optimization theory to develop methods for determining decisions to be made by the offloading scheduler. Many of these methods make use of various types of optimization theory to provide models for scheduling decisions and objectives for those scheduling decisions [11, 23, 24]. Furthermore, most optimization-based approaches enable the analysis of various trade-offs between energy consumption and service delay [11, 23, 24].

Another method that is often used in place of or in addition to optimization theory is the use of Lyapunov optimization methods. Unlike many of the optimization methods that are applied to MEC, Lyapunov methods are specifically designed to address problems that are continuously-varying in nature [25]. Therefore, these methods are often preferred over many of the optimization-based approaches for MEC scheduling problems [25].

Despite the benefits of both optimization theory and Lyapunov optimization meth-

ods for MEC scheduling problems, there are still some limitations to their application to real MEC problems. Many of the optimization methods become complex with an increasing number of devices that are to be scheduled for offloading to the MEC servers; they also often rely upon a variety of assumptions that are not often met in actual MEC deployments [11, 23, 24, 26].

3.3 DRL-Based MEC Scheduling

Deep reinforcement learning (DRL) methods have also emerged as one possible solution to the limitations of both optimization theory and Lyapunov optimization methods. In general, DRL methods have been recognized as one of the useful paradigms for computation offloading and resource scheduling in MEC [11, 27, 28, 13, 24].

In particular, deep reinforcement learning methods are beneficial for incorporating considerations of factors like channel conditions and mobility into the offloading decisions. Furthermore, since the offloading decisions in methods that consider only binary offloading may be relatively simple, deep reinforcement learning methods are particularly advantageous for enabling a task to be offloaded in a manner that also considers factors like how to split that task between mobile devices and remote edge servers [29, 30].

Despite the advantages of deep reinforcement learning methods for MEC, however, there are also some weaknesses to these methods. For example, the reward strategy that is developed for deep reinforcement learning methods often has distinct weights assigned to various factors of the offloading process, such as latency, energy consumption, deadline, and load balancing [27, 28]. Furthermore, deep reinforcement learning methods may be inefficient in their training processes, especially if the action space is large, and often use synthetic environments to train those models [27, 13]. Additionally, deep reinforcement learning models that are trained on synthetic environments may not generalize to the real world [31].

For energy-aware partial offloading in particular, many deep reinforcement learning methods may not consider aspects like the energy consumption of tasks when

they are being trained. For instance, many deep reinforcement learning methods for offloading tasks in MEC do not consider aspects like the size of the task, the deadline for the task, or the communication model between the mobile device and the MEC server [29, 30]. Furthermore, even deep reinforcement learning methods that discuss the use of concepts like edge computing and AI-enabled applications for offloading tasks are mostly limited to the scheduling of tasks rather than considering the task structures that are used for tasks like AI image recognition [32, 33].

3.4 Energy-Aware Offloading Methods

In both DRL and optimization-based methods for offloading tasks in MEC systems, energy consumption is often one of the main factors that is considered. Thus, energy-aware offloading methods have become a distinct area of research within MEC [11, 24].

Furthermore, some of the energy-aware methods also consider the energy characteristics of the MEC and edge servers to which tasks are being offloaded. For example, energy-aware offloading methods can consider the energy that is used by both edge and cloud servers, as well as the energy that is used by mobile devices' batteries, in relation to each offloaded task [34, 35]. Thus, this distinct area of research for energy-aware offloading methods is important to discuss within this research thesis.

For energy-aware offloading methods, especially if tasks are divisible, it is also possible to offload only some of the task to the MEC server; instead of offloading the entire task, only some portion of that task's energy can be consumed by the mobile device, and the remainder of that task's energy can be offloaded to the MEC server [36].

Despite the growth of energy-aware offloading methods, however, there are still some limitations to the current research on these types of methods. For example, most energy-aware offloading methods mainly consider only the energy consumption of the computational unit (CPU) and the energy consumed during uplink data transmissions [24, 11]. Furthermore, it is difficult to determine whether the energy calculations that are performed by these methods are applicable to dynamic traffic, wireless channels,

or heterogeneous access to edge servers [37, 38].

3.5 Limitations of Existing Work

While there is a variety of methods for task offloading in MEC and scheduling tasks between mobile devices and external servers, there are still some limitations to existing methods.

One limitation to many existing methods is the reliance upon simplified models for the tasks that are to be offloaded. For instance, optimization methods, energy-aware offloading methods, and deep reinforcement learning methods mainly consider tasks that are to be executed either entirely on the mobile device or entirely on the external MEC server [11, 24]. Additionally, those methods do not consider the structure of the tasks to be executed; they treat all tasks as being of equal complexity and difficulty [30, 36].

Related to the approach of treating tasks as entire units is the consideration of factors related to energy consumption. For example, energy considerations in many deep reinforcement learning and optimization methods mainly include energy consumption for computational units and uplink communications [39, 35, 24]. Waiting-related energy, memory-related energy, and energy consumed by MEC servers are often ignored in these methods [11, 24, 23].

In addition to the simplifications of the tasks themselves, many existing methods for offloading tasks to MEC servers do not consider the structure of those tasks. For example, most methods for offloading tasks do not consider using methods for managing and scheduling groups of tasks that are to be executed on mobile devices [40].

Thus, despite the variety of methods for task offloading that exist in the current literature on MEC, there is still a lack of methods that incorporate many of these elements together; specifically, in methods that incorporate realistic considerations for partial offloading, energy awareness, and dynamic traffic and traffic management methods [11, 27, 24].

3.6 Research Gap and Thesis Positioning

The existing literature on MEC already has strong foundations being established for offloading granularity, optimization-based scheduling, DRL-based scheduling, and energy-efficient scheduling. Thus, the gap to be filled by this dissertation is not in the research that has been performed on any of these individual topics. Instead, the gap is in the integration of these various topics into a solution that can be applied to structured AI workloads within dynamic MEC environments [11, 24, 27].

This dissertation is positioned at the intersection of these discussed topics and areas of research. Specifically, the focus of this dissertation is on *partial* offloading due to the complexity of structured workloads; energy efficiency as a scheduling concern rather than an overall system concern; dynamic MEC environments; deep reinforcement learning due to the dynamic nature of MEC environments; and AI-oriented workloads for which deep reinforcement learning and partial offloading are especially critical [41, 42, 43].

Chapter 4

System Model and Problem Formulation

This chapter formalizes the system model used in this thesis for energy-efficient partial offloading in a mobile edge computing (MEC) environment. The model is based on the RAPID-oriented edge offloading framework presented by Madiyev et al. [34], but it is reformulated here to match the specific scope of the thesis. In particular, the focus of the present work is not full middleware orchestration, but adaptive scheduling of structured workloads under energy and latency constraints. The chapter therefore develops a simplified but analytically consistent abstraction that supports the later optimization and learning formulation.

The chapter proceeds from system architecture to formal modeling. First, the execution environment is introduced in terms of its principal components and their roles in scheduling. The workload, communication, computation, and energy models are then defined in sequence, since together they determine the latency and energy cost of a scheduling decision. On that basis, the optimization objective, the governing constraints, and the final problem statement are presented.

4.1 Architecture Overview

Madiyev et al. [34] described an energy-efficient edge/cloud offloading architecture in which a decision engine combined system monitoring, predictive analysis, convex optimization, and learning-based scheduling to determine how tasks should be assigned across available computing nodes. In that framework, the broader middleware architecture also included components for service-level management and execution control in addition to the scheduling logic itself. Such a decomposition is suitable for full-system orchestration because it separates request handling, quality-of-service management, resource activation, and remote execution coordination.

For the purposes of the present thesis, however, not all of these architectural components need to be modeled explicitly. The central problem is narrower: how to decide whether a workload should be executed locally, remotely, or in partially offloaded form under dynamic energy and latency conditions. Accordingly, the system model was used as simplified version of RAPID-based framework and organizes it into three logical parts: the client side, the decision engine, and the server side.

The abstraction of the system that is represented in this figure includes the client-side that may execute some of the computation requested of the system; the decision engine that determines which tasks should be offloaded to which edge server nodes based upon the state of the system; and the edge server nodes that execute the tasks assigned to them. Each of these components of the system is represented in the model, but components like management and virtualized systems are not represented separately in the system. Instead, the features of the system that are represented relate to deadlines, queuing, and the available processing power of the edge server nodes. Thus, this abstraction of the system includes only those features related to the scheduling decisions that must be made, and omits those features related to the underlying system that is represented.

More specifically, the decision engine receives information regarding the tasks to be performed by the system from the client nodes. Additionally, the decision engine observes the state of the edge servers to determine which servers may perform those

tasks. Based upon these observations and tasks, the decision engine formulates a decision that includes a determination of which tasks to offload to which servers. These decisions are made continuously as tasks arrive and as the state of the edge servers changes. Thus, this architecture of the system implies the type of problem that is to be studied in this thesis: namely, a scheduling problem that occurs over time, and that considers the communication, resource, and energy costs of performing tasks on the various edge server nodes.

4.2 Task and Workload Model

Let the set of arriving tasks be denoted by

$$\mathcal{T} = \{1, 2, \dots, T\}.$$

Each task $i \in \mathcal{T}$ is characterized by the tuple

$$\mathbf{x}_i = (d_i, c_i, \delta_i, \rho_i, \kappa_i),$$

where d_i is the input data size that will be transmitted if offloading occurs, c_i is the required CPU cycles or an equivalent compute unit, δ_i is the deadline, $\rho_i \in [0, 1]$ is the offloading ratio, and κ_i denotes the task type or workload class. The offloading ratio, represented as ho_i , is the main variable that differs between the binary offloading models and the model presented in this thesis. When $ho_i = 0$, the task is to be executed entirely on the client. When $ho_i = 1$, the task is to be executed entirely on the edge. Intermediate values of ho_i indicate a fraction of the task will be executed on each of those nodes. This value is appropriate for this type of thesis due to the nature of the types of tasks that are to be discussed and scheduled.

The workload that is to be scheduled is assumed to have a structured workload. Thus, each task can be divided into two segments: a local segment and a remote segment. The software component of the system abstracts the specific partition of the software into the offloading ratio, ho_i . The remaining workload class, κ_i , represents

the different types of tasks that can exist in the system. By using these two abstract concepts, the method avoids the need to consider the partitioning of the software itself.

Finally, the tasks that are to be executed are dynamically arrive at the system. Let $\lambda(t)$ be the rate at which tasks arrive at time t . The evolution of the tasks within each monitoring window can be considered in terms of the length of the queue, the arrival rate, and the service rate of the tasks [34]. These variables will play a major role in the decisions that the offloading agent makes for the system.

4.3 Communication Model

The communication model captures the latency cost introduced when part of a task is offloaded to the edge. If task i is partially offloaded with ratio ρ_i , then the amount of data transmitted to the edge is written as

$$d_i^{\text{off}} = \rho_i d_i^*,$$

where d_i^* denotes the effective data volume associated with the remote segment. In a simple task model, d_i^* may coincide with the task input size. In a structured workload model, it may instead represent intermediate data at the partition boundary.

Let $B_i(t)$ denote the available upload bandwidth at decision time t , and let $R_i(t)$ denote the achievable upload transmission rate. The upload transmission delay is then

$$\tau_i^{\text{tx}}(t) = \frac{d_i^{\text{off}}}{R_i(t)}.$$

If the remote segment returns a result or intermediate output of size d_i^{ret} , the corresponding download delay is

$$\tau_i^{\text{rx}}(t) = \frac{d_i^{\text{ret}}}{R_i^{\downarrow}(t)},$$

where $R_i^{\downarrow}(t)$ is the achievable download transmission rate. The total communication

delay is therefore

$$\tau_i^{\text{comm}}(t) = \tau_i^{\text{tx}}(t) + \tau_i^{\text{rx}}(t).$$

This communication model is intentionally compact, but it captures the main dependency needed for scheduling: larger offloaded fractions and lower instantaneous rates lead to greater transmission delay. This dependency is especially important in partial offloading because the quality of the partition decision influences not only computation placement but also the communication overhead induced by intermediate transfers.

4.4 Computation Model

The computation model specifies how the task workload is divided between the client and the edge server once an offloading decision has been made. Let $f_i^\ell(t)$ denote the available local computation rate at the client, and let $f_{ij}^e(t)$ denote the effective computation rate allocated to task i on server j at time t .

The local computation demand is given by

$$c_i^\ell = (1 - \rho_i)c_i,$$

while the remote computation demand is

$$c_i^e = \rho_i c_i.$$

The corresponding execution delays are

$$\tau_i^\ell(t) = \frac{(1 - \rho_i)c_i}{f_i^\ell(t)}, \quad \tau_{ij}^e(t) = \frac{\rho_i c_i}{f_{ij}^e(t)}.$$

Let

$$\mathcal{S} = \{1, 2, \dots, S\}$$

denote the set of available edge servers. A server-selection variable

$$x_{ij} \in \{0, 1\}$$

is introduced such that $x_{ij} = 1$ if task i is assigned to server j , and $x_{ij} = 0$ otherwise. For each offloaded task, exactly one server is selected, so that

$$\sum_{j \in \mathcal{S}} x_{ij} = 1 \quad \text{whenever } \rho_i > 0.$$

If the local and remote segments are sequentially dependent, the task completion time on server j can be written as

$$\tau_{ij}^{\text{tot}}(t) = \tau_i^\ell(t) + \tau_i^{\text{comm}}(t) + \tau_{ij}^e(t) + \tau_{ij}^q(t),$$

where $\tau_{ij}^q(t)$ denotes the queueing delay on server j . This formulation reflects the fact that end-to-end task latency depends not only on local and remote execution times, but also on server-side contention and intermediate communication. It is therefore suitable for the thesis setting, in which the scheduler must respond to time-varying edge load rather than assuming immediate remote service.

4.5 Energy Model

Due to the research topic of energy-efficient task scheduling, it is necessary to consider the energy consumption of both the client and server. This consideration of energy consumption is also used by Madiyev et al. [34] in their approach, although not with regard to the energy consumption of mobile devices.

For the client side, the local computation energy of task i is modeled as

$$E_i^\ell = \alpha_i(1 - \rho_i)c_i,$$

where α_i is a coefficient reflecting the energy efficiency of local computation. The

uplink transmission energy is

$$E_i^{\text{tx}}(t) = p_i^{\text{tx}}(t) \tau_i^{\text{tx}}(t),$$

where $p_i^{\text{tx}}(t)$ is the transmission power used by the client. The client-side energy consumption is therefore

$$E_i^{\text{client}}(t) = E_i^\ell + E_i^{\text{tx}}(t).$$

For the server side, let P_j^{idle} denote the idle power of server j , and let $P_j^{\text{dyn}}(w)$ denote its utilization-dependent power component over a monitoring window w . If server j remains active during a window of duration Δw , its energy consumption is approximated by

$$E_j^{\text{server}}(w) = \left(P_j^{\text{idle}} + P_j^{\text{dyn}}(w) \right) \Delta w.$$

This form separates baseline power from workload-dependent consumption and is sufficient for the scheduling formulation used here.

To connect the server-side model to a task-level optimization problem, let $E_{ij}^{\text{edge}}(t)$ denote the effective edge-side energy cost attributable to executing the remote portion of task i on server j at time t . The end-to-end energy associated with task i when assigned to server j is then written as

$$E_{ij}^{\text{tot}}(t) = E_i^{\text{client}}(t) + x_{ij} E_{ij}^{\text{edge}}(t).$$

Over a scheduling horizon, the corresponding aggregate system energy can be expressed as

$$E^{\text{sys}} = \sum_{i \in \mathcal{T}} \sum_{j \in \mathcal{S}} E_{ij}^{\text{tot}}(t).$$

This thesis adopts this broader energy view because device-side savings alone are not sufficient to characterize energy-efficient scheduling. A decision that reduces client energy but increases the use of high-power or congested edge nodes may be undesirable from the system perspective. The model therefore treats energy as an end-to-end scheduling criterion rather than only a local battery metric.

4.6 Optimization Objective

Given the preceding communication, computation, and energy models, the scheduling objective is to balance latency performance and energy efficiency under dynamic conditions. For a task i assigned to server j , define the weighted cost

$$J_{ij}(t) = \beta \tau_{ij}^{\text{tot}}(t) + (1 - \beta) E_{ij}^{\text{tot}}(t),$$

where $\beta \in [0, 1]$ controls the relative importance of delay and energy.

The per-task objective extends naturally to the task stream:

$$\min_{\rho_i, x_{ij}, a_t} \sum_{i \in \mathcal{T}} \sum_{j \in \mathcal{S}} J_{ij}(t),$$

where a_t denotes the scheduling action selected by the decision engine at time t . In this formulation, the action implicitly includes the offloading ratio and the server-selection decision.

An equivalent constrained form is to minimize total energy while requiring that latency constraints be satisfied:

$$\min \sum_{i \in \mathcal{T}} \sum_{j \in \mathcal{S}} E_{ij}^{\text{tot}}(t) \quad \text{s.t.} \quad \tau_{ij}^{\text{tot}}(t) \leq \delta_i, \forall i, j \text{ with } x_{ij} = 1.$$

This formulation is consistent with the thesis objective. It does not treat energy and latency as unrelated metrics; instead, it makes explicit that scheduling decisions should minimize energy while preserving service quality. It also extends the earlier RAPID-style energy-aware scheduling view from node-activation logic to the finer-grained problem of per-task partial offloading and server selection [34].

4.7 Constraints

The optimization problem is subject to resource and service constraints arising from partial offloading and heterogeneous edge execution.

First, the offloading ratio must remain within feasible bounds:

$$0 \leq \rho_i \leq 1, \quad \forall i \in \mathcal{T}.$$

Second, each offloaded task must be assigned to exactly one server:

$$\sum_{j \in \mathcal{S}} x_{ij} = 1, \quad \forall i \text{ such that } \rho_i > 0.$$

Third, the server-side resource capacities must not be exceeded. Let C_j^{cpu} and C_j^{mem} denote the available CPU and memory capacity of server j , respectively. Then

$$\sum_{i \in \mathcal{T}} x_{ij} \rho_i c_i \leq C_j^{\text{cpu}}, \quad \forall j \in \mathcal{S},$$

and

$$\sum_{i \in \mathcal{T}} x_{ij} m_i \leq C_j^{\text{mem}}, \quad \forall j \in \mathcal{S},$$

where m_i denotes the memory requirement of task i .

Fourth, deadline constraints must be respected:

$$\tau_{ij}^{\text{tot}}(t) \leq \delta_i, \quad \forall i, j \text{ with } x_{ij} = 1.$$

Finally, if the number of active servers is itself controlled by the decision engine, let

$$y_j(t) \in \{0, 1\}$$

indicate whether server j is active at time t . Then

$$x_{ij} \leq y_j(t), \quad \forall i, j,$$

and

$$\sum_{j \in \mathcal{S}} y_j(t) \leq S.$$

These constraints ensure that a task can only be assigned to an active server and preserve the possibility of energy-aware server activation, which remains conceptually consistent with the RAPID-based scheduling perspective [34].

4.8 Problem Definition

The problem addressed in this thesis can now be stated as follows.

Given a stream of structured computation tasks generated by resource-constrained clients, a heterogeneous set of edge servers with time-varying load and power characteristics, and dynamic communication conditions, determine for each task whether it should be executed locally, remotely, or in partially offloaded form; if partially offloaded, determine the offloading ratio and the target server; and select these actions so as to minimize the long-term energy–latency cost while satisfying resource and deadline constraints.

This problem is difficult for several reasons. The decision space contains both continuous and discrete variables, since the offloading ratio is continuous while server selection is discrete. The system state is dynamic because server load, queueing delay, communication rate, and resource availability evolve over time. The optimization target is multi-criteria, because energy reduction must be balanced against latency and service feasibility. Most importantly, the problem is sequential: each scheduling action affects subsequent server states and therefore influences future decisions.

For these reasons, the problem is not treated as a one-shot static optimization problem. Instead, it is framed as a sequential scheduling problem in a dynamic MEC environment. The decision engine must therefore maintain a state representation of the current system and repeatedly select actions that produce a favorable long-term tradeoff between energy efficiency and latency. This formal problem statement provides the basis for the solution approach developed in the following chapter.

Chapter 5

Proposed Method

This chapter presents the proposed method for adaptive scheduling of partially offloadable workloads in a mobile edge computing (MEC) environment. Building on the system model and problem formulation of Chapter 4, the chapter defines a deep reinforcement learning (DRL)-based scheduler that selects offloading actions under dynamic communication, computation, and energy conditions. The central idea is to treat MEC scheduling as a sequential decision problem in which each offloading decision affects not only the immediate latency and energy cost of the current task, but also the future state of the edge system through queueing, resource occupancy, and server utilization.

The objective of the proposed method is not to minimize a single metric in isolation, but to learn a policy that balances latency, energy consumption, and deadline satisfaction under dynamic system conditions. This formulation reflects practical MEC requirements, where strict energy minimization may lead to poor responsiveness, and latency optimization alone may result in inefficient resource usage.

5.1 Framework Overview

The proposed framework consists of three interacting parts: the client side, the DRL-based scheduler, and the edge execution layer. The client side generates computation requests associated with structured workloads that may be executed fully on-device,

fully on the edge, or partially split between the two. The scheduler acts as the central decision engine. It collects the current system state, selects an offloading action for each incoming task, and forwards the remote segment to the chosen edge server when offloading is used. The edge execution layer consists of one or more heterogeneous servers that process offloaded task segments and return the corresponding outputs.

The framework operates in a closed loop. When a new task arrives, the client or front-end runtime provides task metadata to the scheduler. This metadata includes workload-related attributes such as task size, computational demand, deadline, and workload class. At the same time, the scheduler observes the current execution context, including local device status, communication condition, and edge-side resource availability. Based on this observation, the scheduler selects an action that determines how the task should be executed. The selected action may correspond to full local execution, full edge execution, or a partial offloading decision that specifies both a partition choice and a destination server.

Once the decision is applied to the task, the task can be executed. The local segment of the task can be executed on the client system, and the remote segment of the task (if any) is transmitted to and executed on the selected edge node. Following execution of the task on the selected node(s), feedback signals can be received by the scheduler. Such feedback signals may indicate, for instance, the latency of execution of the task on the edge node, the energy consumption estimates of executing the task on that edge node, whether the task met its deadlines upon execution, and the state of the resources of the edge node following execution. This feedback is used in two ways: it may be used to form reward signals for training the policy to schedule tasks, and it may be used to update the state representation that the scheduler employs to make scheduling decisions for tasks executed after reception of the feedback signals.

The importance of this architecture is that the task scheduling decisions are not made in isolation from the execution of the tasks. More specifically, the scheduler is not assumed to exist within an environment that is static and unchanging with each decision made by the scheduler. Rather, the scheduler interacts with an environment that changes in response to the decisions made by the scheduler. This interaction

between the scheduler and its environment is one of the main reasons that the model of decision-making that is employed in this thesis is of a sequential decision model.

5.2 DRL-Based Scheduler Design

The scheduler is modeled as a DRL agent interacting with the MEC environment. At each time step t , the agent observes the current environment state s_t , takes an action a_t , and receives a reward r_t along with the next environment state s_{t+1} . The agent aims to learn a policy π that maximizes its expected discounted reward

$$\max_{\pi} E_{\pi} \left[\sum_{t=0}^{\infty} \gamma^t r_t \right],$$

where $\gamma \in [0, 1)$ is the discount factor.

This formulation is appropriate because MEC scheduling has all the main characteristics of a sequential control problem. The environment is dynamic, since wireless rate, server load, queue length, and available memory evolve over time. The consequences of an action are delayed, because an offloading decision affects not only the current task outcome but also the future availability of edge resources. The decision process is also coupled across tasks: assigning one task to a highly loaded server may increase subsequent queueing delays for later tasks, while keeping too much computation local may preserve edge capacity at the expense of device energy or deadline performance.

The role of the scheduler is therefore not to solve an isolated per-task optimization problem, but to learn a decision policy that performs well over a stream of tasks under changing conditions. This is the key reason why DRL is preferred over fixed rules or static optimization-based scheduling in the proposed method. Rule-based methods can be simple to implement, but they usually cannot adapt well to the interaction between heterogeneous workloads, variable communication cost, and fluctuating edge resources. Static optimization methods offer useful structure, but they become increasingly difficult to solve online when partial offloading, server selection,

and dynamic state evolution must be handled jointly. By contrast, a learned policy can directly map observed state to action and can therefore support repeated online scheduling with bounded runtime overhead after training.

The proposed scheduler is designed at the level of per-task decision making, but its behavior is shaped by long-term effects through the reward design. In this way, the method preserves the practical simplicity of single-step action selection at runtime while still optimizing a long-term scheduling objective during training.

5.3 State Space

The state representation must be sufficiently informative to support good scheduling decisions, while remaining compact enough for stable learning. In the proposed method, the state at decision step t is written as

$$s_t = \left[s_t^{\text{task}}, s_t^{\text{device}}, s_t^{\text{net}}, s_t^{\text{edge}} \right],$$

where the four components describe the current task, the local device condition, the communication condition, and the edge-side resource condition, respectively.

The task-related component s_t^{task} captures the attributes of the task currently being scheduled. A compact representation includes normalized task size, computational demand, deadline, and workload class:

$$s_t^{\text{task}} = \left[\hat{d}_t, \hat{c}_t, \hat{\delta}_t, \hat{\kappa}_t \right].$$

These variables are included because the desirability of offloading depends strongly on how much computation is required, how much data must be transmitted, how urgent the task is, and what workload type it belongs to. For structured workloads, the class indicator also helps distinguish tasks with different computation-to-communication profiles or different feasible partitioning behavior.

The device-side component s_t^{device} describes the current local execution condition. This may include normalized available local compute rate, current local queue length,

battery or energy budget indicator, and possibly current device load:

$$s_t^{\text{device}} = \left[\hat{f}_t^\ell, \hat{q}_t^\ell, \hat{b}_t \right].$$

These variables are needed because the benefit of offloading depends not only on the task itself but also on whether the client is resource-constrained at the time the decision is made.

The communication component s_t^{net} summarizes the current transmission environment, for example through normalized uplink and downlink rates:

$$s_t^{\text{net}} = \left[\hat{R}_t^\uparrow, \hat{R}_t^\downarrow \right].$$

These variables are particularly important for partial offloading, because the cost of transmitting the remote segment depends directly on the instantaneous communication condition.

The edge-side component s_t^{edge} contains the current status of the available edge servers. For each server $j \in \mathcal{S}$, the state includes normalized CPU availability, memory availability, queue length, and possibly a power or utilization indicator:

$$s_{t,j}^{\text{edge}} = \left[\hat{f}_{t,j}^e, \hat{m}_{t,j}, \hat{q}_{t,j}^e, \hat{u}_{t,j} \right].$$

The full edge-side state is then the concatenation over all candidate servers:

$$s_t^{\text{edge}} = \left[s_{t,1}^{\text{edge}}, s_{t,2}^{\text{edge}}, \dots, s_{t,S}^{\text{edge}} \right].$$

The main design rationale behind this state space is that each component corresponds to a quantity that materially affects the energy–latency tradeoff. Task features determine the computational and transmission burden, device features determine the local execution cost, network features determine the communication cost, and edge features determine the remote execution feasibility and congestion level. At the same time, it also excludes those quantities that are redundant or difficult to estimate. The

goal is to not encode every detail of the process of offloading, but rather to provide the scheduler with enough information to make a decision regarding which of the available actions to take.

5.4 Action Space

The action space for this model must include actions that allow for the system to partially offload the model's work to another system. For this method, each action will consist of two components:

$$\mathcal{P} = \{\rho^{(0)}, \rho^{(1)}, \dots, \rho^{(K)}\}$$

denote a finite set of admissible offloading ratios, where the case of ($\rho(0)=0$) indicates that the task is to be executed entirely on the mobile device itself, and the case of ($\rho(K)=1$) indicates that the task is to be executed entirely on the remote cloud server. Intermediate values of ($\rho(k)$) indicate a candidate level of offloading of the task to perform. Using a finite set of offloading ratios allows for modeling of the possibility of offloading only some of the task to the remote server.

Let $\mathcal{S} = \{1, 2, \dots, S\}$ denote the set of candidate edge servers. Then the action at step t is defined as

$$a_t = (\rho_t, j_t),$$

where $\rho_t \in \mathcal{P}$ is the selected offloading ratio and $j_t \in \mathcal{S} \cup \{0\}$ is the execution target. The special case $j_t = 0$ denotes local-only execution and is valid only when $\rho_t = 0$. For $\rho_t > 0$, the server index j_t identifies the edge node to which the remote segment is sent.

Equivalently, the full action space may be written as

$$\mathcal{A} = \{(0, 0)\} \cup \{(\rho^{(k)}, j) \mid \rho^{(k)} \in \mathcal{P} \setminus \{0\}, j \in \mathcal{S}\}.$$

The action encoding is expressive enough to cover the three dimensions of the decisions

that must be made according to the thesis. However, the scheduler is only required to choose among a finite number of options to ensure that the system is trainable.

While the notion of offloading is continuous, only a finite number of partition choices can be made in practice for any application. Thus, discretizing this notion into a finite number of choices for the model is both more realistic and more convenient for implementation.

5.5 Reward Function

The reward function is formulated with the goal of aligning the learning of the policy with the goal of the thesis. Since the scheduler is to be designed to balance energy consumption, completion time, and quality of service, the reward function is to penalize the agent for actions that lead to high energy consumption, high completion time, or low quality of service.

Let L_t represent the latency of the task that is scheduled at time step t , and let E_t represent the total energy consumption of that task and its scheduled edge server. Let $v_t \in \{0, 1\}$ indicate deadline violation, where $v_t = 1$ if $L_t > \delta_t$ and $v_t = 0$ otherwise. A suitable reward is

$$r_t = - \left(\omega_L \hat{L}_t + \omega_E \hat{E}_t + \omega_V v_t \right),$$

where $\hat{L}_t$ and $\hat{E}_t$ are normalized latency and energy terms, and ω_L , ω_E , and ω_V are nonnegative weighting coefficients.

This form of reward also indirectly rewards the agent for the outcomes it produces. Actions that lead to lower latencies and lower energy costs will have higher rewards than those that do not. Furthermore, the deadline penalty ensures that the agent is not rewarded for minimizing energy costs alone, which may lead to all actions being taken locally or to servers that are not necessarily the best candidates for those actions. Similarly, minimizing latency alone may lead to actions that are energy-intensive.

Finally, it is also possible to include penalties for taking actions that are either infeasible or that lead to poor quality outcomes, such as assigning a request to a

server that does not have enough memory to handle the requested tasks. In that case, an additional penalty term $\omega_I I_t$ may be added, where I_t indicates an infeasible or corrected action:

$$r_t = - \left(\omega_L \hat{L}_t + \omega_E \hat{E}_t + \omega_V v_t + \omega_I I_t \right).$$

Such a definition of the term is useful in cases in which the runtime system includes functionality to mask or correct for actions taken by the agent.

The form of the reward function is intentionally simple. The function is provided as a means of representing the optimization problem that is to be solved, rather than as an attempt to represent the problem itself. The use of these terms and coefficients is a means of creating a function that is easily interpretable and tunable by the developer of the system, which is especially important in the context of an MSc level project.

An important aspect of the creation of this reward function is that it does not force the agent to minimize the amount of energy that is consumed by the system. Rather, the agent is permitted to consume more energy if that action will lead to a reduction in latency or deadline violations. Thus, the system will be programmed to account for such a scenario, and to increase either the offloading of tasks to the cloud or the local computation of tasks based upon which provides the best balance between energy consumption and latency.

5.6 Training Procedure

The DRL scheduler is trained through repeated interaction with an environment that simulates or emulates task arrivals, communication variability, and edge-resource dynamics. Each training episode consists of a sequence of task-scheduling decisions. At each step, the agent observes the current state, selects an action, applies that action to the environment, and then receives the next state and reward. The resulting transition

$$(s_t, a_t, r_t, s_{t+1})$$

forms the basic training signal.

During training, the agent must explore the action space sufficiently to learn which offloading and server-selection decisions perform well under different runtime conditions. Exploration is therefore used in the early stage of training, after which the policy gradually becomes more exploitative. The precise exploration mechanism depends on the chosen DRL algorithm, but conceptually it serves the same purpose: to avoid premature convergence to a poor scheduling strategy.

In this work, a value-based deep reinforcement learning approach is used, specifically a Deep Q-Network (DQN). The Q-network estimates the expected cumulative reward for each action given the current state, and is trained using temporal-difference learning with experience replay and a target network. These mechanisms improve training stability and allow the agent to learn effective scheduling strategies from diverse interaction samples.

Model selection is based on validation performance rather than training reward alone. Since MEC scheduling is sensitive to workload mix and system dynamics, the trained policy should be evaluated on held-out episodes or validation scenarios that differ from the training traces. A policy checkpoint may be selected according to a combination of average reward, energy consumption, mean latency, and deadline satisfaction. This is preferable to selecting the final training iteration uncritically, since late-stage training may overfit to a particular environment pattern or exhibit unstable behavior.

Training is stopped when one of the usual conditions is met: the validation performance stabilizes, improvement becomes negligible over a sustained period, or a predefined training budget is exhausted. The resulting policy is then fixed and transferred to the online scheduling phase. In this thesis, the training procedure is treated as an offline or semi-offline stage, while runtime deployment uses the learned policy for decision making without full retraining during normal operation.

The training process relies on a simulated environment and therefore reflects modeled system behavior rather than direct hardware measurements. However the systems characteristics copies computational capabilities of real physical setup (Jetson

Nano A01, and B02).

5.7 Online Decision Algorithm

After training, the learned policy is used as the runtime scheduler. When a new task arrives, the scheduler first constructs the current state vector from the available task, device, network, and edge-side information. This state is then passed to the trained DRL policy, which outputs the scheduling action to be applied for that task.

Operationally, the online decision flow is as follows. At time t , an incoming task is characterized by its workload metadata and deadline. The scheduler simultaneously collects the current device condition, communication rates, and edge-node resource status. From this information it forms the state s_t . The trained policy then selects

$$a_t = (\rho_t, j_t),$$

which determines the offloading ratio and, when offloading is used, the target edge server. If $\rho_t = 0$, the task is executed entirely on the client. Otherwise, the task is partitioned according to the selected ratio, the remote segment is sent to server j_t , and the local and remote segments are executed according to the runtime design.

Once execution completes, the system measures or estimates the observed latency, energy cost, and deadline status. These values are used to update monitoring statistics and, if online logging is enabled, to record the behavior of the policy for later analysis. The next scheduling decision then proceeds using the updated system state. In this way, the online scheduler retains the sequential nature of the original control problem even though learning itself has already been completed.

The main deployment advantage of this design is that runtime inference is lightweight compared with solving a fresh optimization problem for each task. The online scheduler needs only to evaluate the learned policy on the current state and then dispatch the task accordingly. This makes the method suitable for repeated low-latency decision making in dynamic MEC environments. At the same time, the decision process

remains grounded in the same state, action, and reward structure used during training, which ensures methodological consistency between the learned policy and its runtime use.

5.8 Design Rationale

The proposed design reflects practical considerations of MEC systems, where scheduling decisions must balance multiple competing objectives. In particular, latency and deadline satisfaction are often more critical than strict energy minimization, especially for delay-sensitive applications. Consequently, the reward function is structured to penalize deadline violations and excessive latency more strongly, while still accounting for energy consumption.

The discretized action space enables tractable learning while preserving the ability to model partial offloading decisions. This is consistent with realistic systems, where partitioning is often constrained to specific computation stages or model boundaries. The state representation is designed to capture the key factors influencing scheduling decisions without introducing excessive dimensionality, thereby supporting stable training.

Overall, the method prioritizes adaptability and robustness over strict optimality, aiming to learn scheduling strategies that generalize across varying workload conditions and system states.

The DQN algorithm was selected because the scheduling action space in this setup is discrete. The scheduler chooses from a finite set of offloading ratios, $\rho \in \{0, 0.25, 0.5, 0.75, 1.0\}$, and a finite set of action space in the face of edge servers. Since the agent can estimate the expected value of each discrete action and select the action with the highest value DQN seems to be preferable choice. DQN plays the role of lightweight reward based learner that adapts to the dynamic environment and applicable to the partial-offloading behavior of presented environment.

At the same time, DQN has its limitations, as its performance is dependent to the discretization of action space, reward design and stability of training. More

recent reinforcement learning algorithms such as Double DQN, PPO, SAC could be better suited for large-scale MEC environments, multiple simultaneous clients decision. However, they require extensive hyperparameter tuning, longer training time and more resources.

More recent reinforcement learning algorithms, including double DQN, dueling DQN, PPO, SAC, TD3, and multi-agent DRL variants, may be better suited for larger-scale MEC environments, continuous offloading ratios, or multiple simultaneous clients. These methods are therefore treated as important extensions for future work rather than as direct replacements within the current thesis scope. DQN was suitable in terms of training time and complexity of the environment.

Chapter 6

Results

The performance of the proposed method is compared against existing policies in a variety of MEC scenarios. The experiments are conducted in four scenarios: mixed, communication-heavy, balanced-heavy and compute-heavy. The results are presented in terms of four different metrics: the latency of task completion, the energy consumption required to complete each task, the success rate with which tasks are completed within a deadline, and the ratio of tasks that are offloaded to MEC servers.

6.1 Experimental Setup

Experiments were conducted in a dynamic MEC simulation environment with heterogeneous edge servers, stochastic tasks, fluctuating network conditions. The simulated setup consisted of 20 Jetson nano edge nodes, consisting of 11 B02 type nodes, and 9 A01-type nodes. Each node is modeled with their real characteristics for the future real-world experiments.

Four workload scenarios were evaluated. Mixed scenario samples tasks from other workload classes. The communication-heavy scenarios has larger input sizes and lower computational complexity. The balanced-heavy scenario uses medium input size and computational demand. Final scenario compute-heavy uses smaller input sizes but higher computational demand. Task arrival followed Poisson distribution.

The DQN agent was trained for 300 episodes with 200 tasks per episode. The

learning rate was set to 10^{-4} , the discount factor to $\gamma = 0.99$, the initial exploration rate to $\epsilon = 1.0$, the final exploration rate to $\epsilon = 0.2$, and the epsilon decay factor to 0.997. The action space consists of local execution and partial or full offloading actions using offloading ratios $\rho \in \{0.25, 0.5, 0.75, 1.0\}$ over the top-5 candidate servers.

Table 6.1: Experimental parameters and DQN hyperparameters

Parameter	Value
Workload scenarios	Mixed, communication-heavy, balanced-heavy, compute-heavy
Edge nodes	20 Jetson-class nodes: 11 B02-type, 9 A01-type
Tasks per evaluation run	300
Random seeds	20
Training episodes	300
Tasks per training episode	200
Learning rate	10^{-4}
Discount factor	$\gamma = 0.99$
Initial exploration rate	$\epsilon = 1.0$
Final exploration rate	$\epsilon = 0.2$
Epsilon decay	0.997
Offloading ratios	$\{0, 0.25, 0.5, 0.75, 1.0\}$
DQN candidate servers	Top-5 ranked servers
Reported uncertainty	95% confidence interval across seeds

6.2 Compared Policies

Evaluation was conducted in comparison of DQN with other baseline algorithms, and additional two oracle algorithms. Baseline algorithms included Local, Full Offloading, Random, Least Loaded, Heuristic and two Oracle algorithms - Oracle Top-5 and Oracle Full. Local computing processes all tasks locally without offloading any of them to remote servers, while Full Offloading was executing all the tasks on the edge servers. Random’s decision was stochastically choosing between all of the available servers. Least Loaded decided to always choose the most free server for offloading. DQN is our proposed method of Reinforcement Learning algorithm. Oracle Top 5 proposed 5 the best decision servers for the client, while Oracle Full was evaluating all available alternatives for offloading. This algorithm was chosen as an upper bound algorithm, which had privileged access to the simulator objective.

The proposed DQN policy is compared with the following policies:

- **Local-only:** executes every task locally, corresponding to $\rho = 0$.
- **Random offloading:** randomly selects an available edge server among available ones.
- **Least-loaded server:** analyzes workload of all servers and chose the most free server among all of them.
- **Heuristic energy-delay:** selects the offloading ratio based on workload and deadline class, then chooses a server using a weighted energy-delay score.
- **Full offload best server:** always uses full offloading, $\rho = 1$, and selects the server with the lowest estimated latency.
- **DQN partial offloading:** uses the trained DQN policy to select an offloading ratio and target server from the top-5 candidate action set.
- **Oracle Top-5:** proposes 5 best decision servers for client and makes the best decision that has the best reward-like objective.
- **Oracle Full:** exhaustively evaluates all available servers and all offloading ratios, providing a stronger offline upper-bound reference.

The oracle policies optimize the joint reward-like objective used for evaluation. They are therefore expected to perform best with respect to this objective, but not necessarily to minimize every individual metric independently. For example, a policy that minimizes energy alone may not be optimal for deadline satisfaction or latency.

6.3 Deadline Satisfaction

As illustrated in Figure 6-1, the oracle policies provide the upper-bound reference in most cases, while the DQN policy shows adaptive but not uniformly dominant behavior.

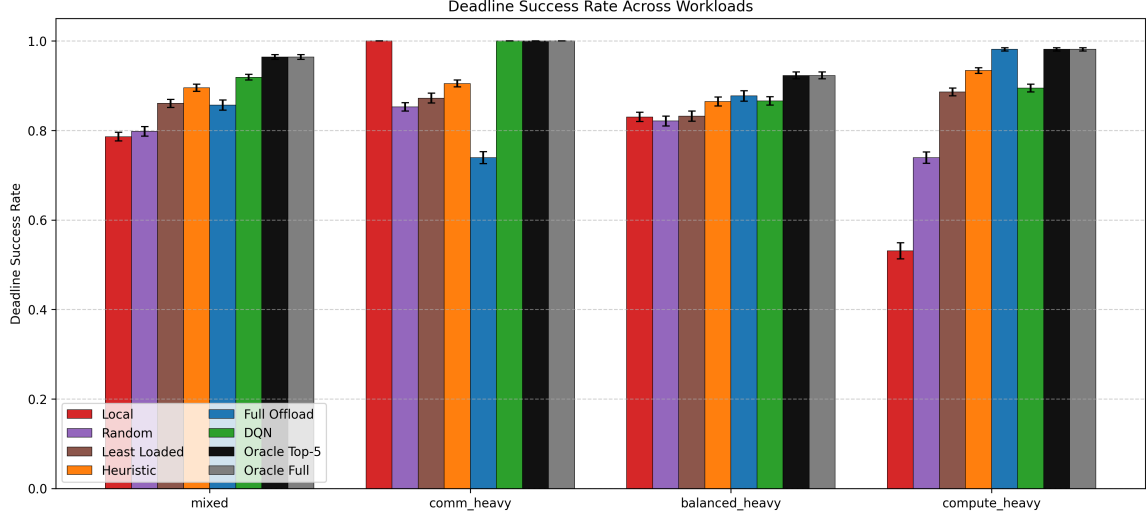


Figure 6-1: Deadline success rate across workload scenarios

In the mixed scenario, DQN achieves a deadline success rate of approximately 0.92, outperforming all non-oracle baselines, while the oracle policies reach approximately 0.97. This shows that DQN improves deadline satisfaction in mixed workloads, although it remains below the exhaustive upper bound algorithm in face of oracle ones.

In the communication-heavy scenario, both local-only execution and DQN achieve a deadline success rate of 1.0. This is expected because these tasks have relatively low computational demand, making local execution effective and avoiding unnecessary transmission overhead. Full offloading performs worse, with a success rate of about 0.75.

In the balanced-heavy scenario, DQN reaches approximately 0.87, outperforming local-only, random, and least-loaded scheduling, but remaining below full offloading and the oracle policies, which reach approximately 0.92. This suggests that balanced-heavy workloads are more challenging because communication and computation costs are both significant.

In the compute-heavy scenario, DQN improves deadline success from approximately 0.53 under local execution to approximately 0.90. However, full offloading and the oracle policies perform better, reaching approximately 0.98, indicating that compute-heavy workloads strongly benefit from edge execution.

Overall, DQN improves deadline satisfaction in several scenarios, especially mixed

and compute-heavy workloads, but it does not dominate full offloading when aggressive edge execution is naturally favorable.

6.4 Energy Consumption

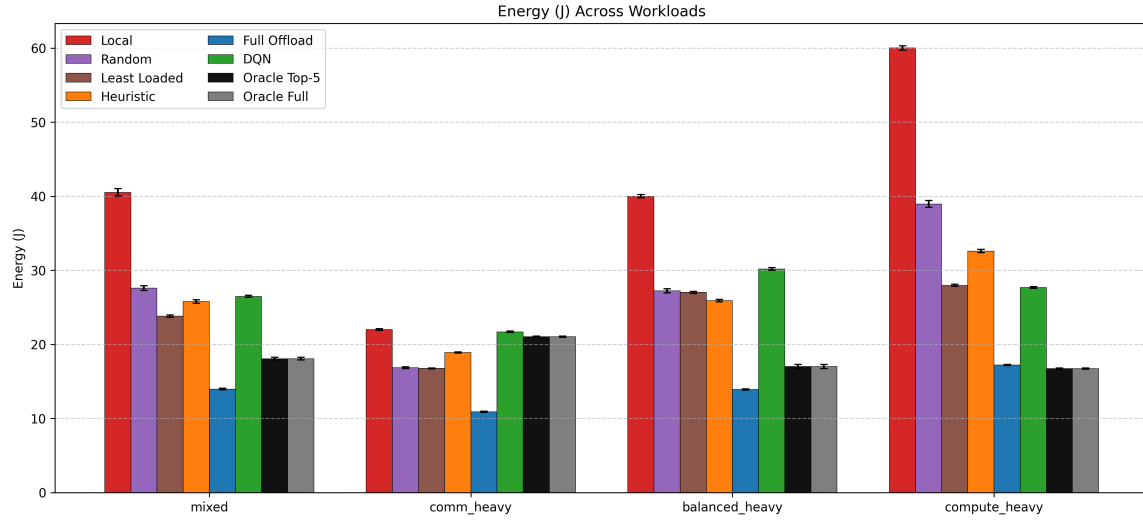


Figure 6-2: Modeled task-level energy consumption across workload scenarios

Figure 6-2 depicts the energy consumption results for the different offloading strategies and different algorithms. The plotted energy included local computational energy, client transmission energy, and edge execution energy - that involves all energy consumptions across the environment related to the simulation.

In all scenarios, local execution demonstrated highest value of energy consumption. However, Full Offloading demonstrated almost lowest values of energy consumption across all of the scenarios. The DQN policy achieves a good balance between energy efficiency and other system performance metrics. These results illustrated as an evidence of the energy-latency trade-off rather than as proof that DQN minimizes energy. DQN reduces energy relative to local-only execution, but it is not the energy-optimal policy in all scenarios. This is consistent with the reward design, which balances energy against latency and deadline penalties rather than minimizing energy alone.

6.5 Latency Performance

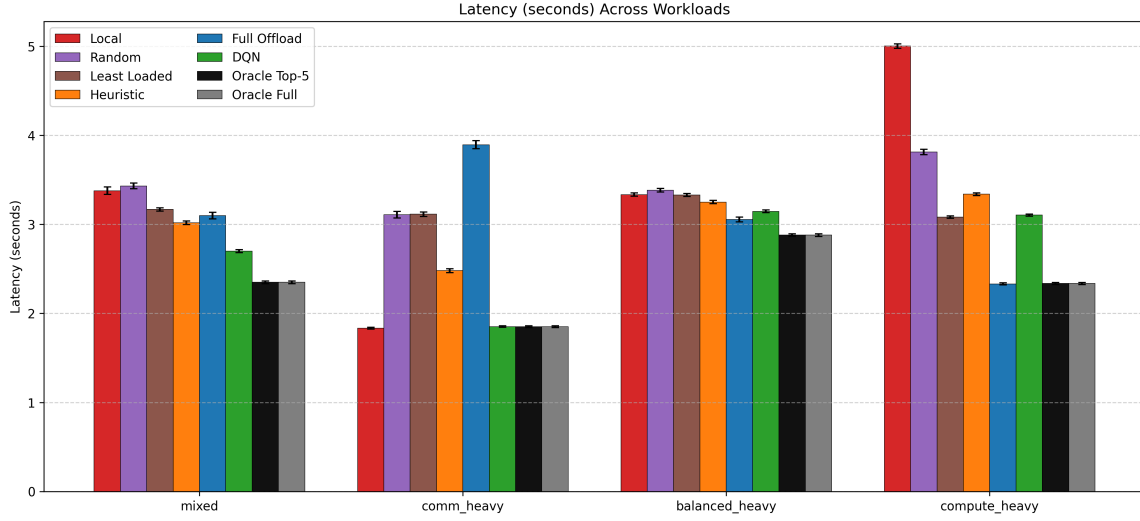


Figure 6-3: Average latency across workload scenarios

Figure 6-3 presents the average latency achieved in each scenario. The oracle policies achieve the lowest or near-lowest latency in most scenarios because they perform exhaustive action evaluation. DQN improves latency relative to several non-oracle baselines, especially in the mixed scenario, but it is not uniformly the fastest policy.

In the mixed scenario, DQN achieves an average latency of approximately 2.70 seconds. This improves over local-only execution (3.38 seconds), random offloading (3.43 seconds), least-loaded scheduling (3.17 seconds), heuristic scheduling (3.02 seconds), and full offloading (3.10 seconds). The oracle policies achieve approximately 2.35 seconds. Thus, in mixed workloads, DQN provides the best latency among deployable non-oracle policies while remaining below the oracle upper bound.

In communication-heavy scenario, local algorithm achieved about 1.8 seconds, while DQN almost reached the same latency as other two oracle algorithms of about 1.85. Full offloading demonstrated the longest latency because communication overhead outweighs the benefits of edge computation.

In the balanced-heavy scenario DQN achieved about 3.15 seconds. It outperformed local-only, random, least-loaded, and heuristic policies, but underperformed

in comparison to full offloading and the oracle policies. The oracle policies achieve approximately 2.88 seconds, showing that exhaustive search identifies more efficient offloading actions in this case.

In the compute-heavy scenario, DQN achieves an average latency of approximately 3.15 seconds, while full offloading and the oracle policies achieve approximately 2.3 seconds.

Overall, DQN improves latency in the mixed scenario and reduces local-execution latency in compute-heavy workloads. However, the results also show that full offloading is a strong baseline when computation dominates communication overhead.

6.6 Offloading Behavior

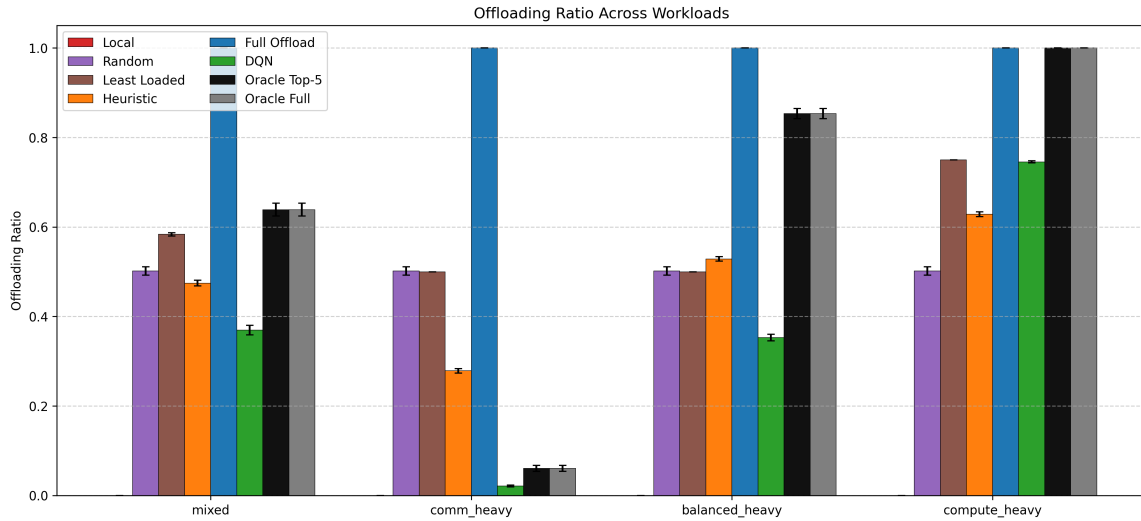


Figure 6-4: Average offloading ratio across workload scenarios

Figure 6-4 shows the average offloading ratio selected by each policy. This metric helps indicate whether a policy follows a fixed execution strategy or adapts to workload conditions.

The DQN algorithm selects different offloading ratios regarding the scenario. In the communication-heavy scenario, it uses less offloading, and prefers local computing more than transporting task to the distant Edge Server around 0.02. In mixed and balanced-heavy scenarios, DQN has about 0.37 and 0.35. It showed that distribution

between clients and edge servers is better than relying only on local execution or on fully offloaded strategy.

In the compute-heavy scenario, DQN increases the offloading ratio to approximately 0.745, reflecting the benefit of edge execution for computation-intensive workloads. However, the oracle and full-offloading policies use ratios close to 1.0, which explains their stronger latency and deadline performance in this scenario.

Overall, the results show that DQN adapts its offloading behavior across workload classes, although it remains more conservative than the oracle policies in scenarios where near-full offloading is beneficial.

6.7 Discussion

The results show that the DQN-based scheduler learns adaptive offloading behavior, but its advantage is workload-dependent. In the mixed workload, DQN achieves the strongest non-oracle performance in several metrics. In communication-heavy workloads, it learns to avoid unnecessary offloading. However, in compute-heavy and balanced-heavy scenarios, full offloading and oracle policies can outperform DQN because aggressive edge execution is more beneficial under those conditions.

The results also show the trade-offs between energy efficiency and latency. The results show that offloading the workload to the edge minimizes the energy consumption of mobile devices but increases the communication delay. On the other hand, executing the workload on the mobile device leads to lower latency but increases the energy consumption of the device. When the workload is too compute-intensive for the device, it will fail to execute the workload. The DQN policy provides a good balance between energy consumption and latency.

The oracle baselines are important for interpreting these results. Oracle Top-5 provides the fairest upper-bound comparison because it uses the same candidate action space as DQN. Oracle Full provides a stronger theoretical upper bound because it evaluates all servers. Since the oracle policies use exhaustive simulator access, they are not deployable online schedulers. They are included to show how far the learned

policy is from the best action under the simulator objective.

The statistical reporting was strengthened by increasing the evaluation to 20 seeds and by reporting 95% confidence intervals. This addresses stochastic variation in task generation, network state, and queue evolution. However, the confidence intervals quantify variability within the simulator and should not be interpreted as deployment-level statistical validation.

Chapter 7

Conclusion

7.1 Summary of Findings

This thesis investigates the problem of energy-efficient scheduling for partial computation offloading in Mobile Edge Computing (MEC) environments. With the increasing demand for computation-intensive applications to be executed on mobile devices, it is becoming necessary to offload some of the computation to remote servers. However, offloading all of the computation for a mobile device to a remote server introduces communication overhead, which can be undesirable. Partial offloading allows mobile devices to offload only portions of the required computations to remote MEC servers while performing the remaining portions locally on the mobile device.

The problem of offloading computations to MEC servers was modeled as a sequential decision problem whose solution could be found with a Deep Q-Network (DQN). The DQN was trained to maximize a defined reward function that reflects the trade-off between the latency and energy consumption of executing the computations on the mobile device versus offloading them to the remote MEC servers. The DQN-based scheduling policy was evaluated through simulations of various types of mobile device workloads. The results of the simulations show that the DQN-based policy achieves lower latency and higher success rates for meeting the deadlines for the tasks assigned to the mobile devices relative to local-only and heuristic-based strategies. Furthermore, the DQN exhibits adaptive behavior that shifts between strategies for

minimizing energy consumption, offloading a significant portion of the computations to the MEC servers in scenarios in which the communications between the mobile device and MEC servers is significant relative to the computational requirements of the mobile device, and offloading a smaller portion of the computations to the MEC servers in scenarios in which the computations performed by the mobile device are significant relative to the communication requirements of the mobile device.

The results of the simulations indicate that the DQN-based policy does not minimize the energy consumption of mobile devices. Instead, the energy consumption increases for mobile devices when the scheduling policy determines it will allow for lower latency and higher success rates in meeting deadlines for mobile device tasks. Thus, the policy is based on a definition of quality of service that considers energy consumption as only one of several factors in determining how resources should be allocated between local execution of mobile device tasks versus offloading those tasks to MEC servers.

7.2 Answer to Research Questions

RQ1: How can partial offloading be modeled while considering both latency and energy?

This thesis modeled partial offloading as a sequential scheduling problem in which each task is described by input size, computational demand, deadline, workload class, offloading ratio, and server assignment. The model includes local computation delay, communication delay, edge execution delay, queueing delay, and modeled task-level energy. This formulation allows local execution, full offloading, and partial offloading to be represented within a unified action space.

RQ2: To what extent does the DQN-based scheduler outperform baseline methods?

The DQN-based scheduler outperforms several baselines in the mixed workload scenario and improves substantially over local execution in compute-heavy workloads. It also learns to avoid unnecessary offloading in communication-heavy workloads.

However, the results do not show universal dominance over all baselines in all scenarios. Full offloading performs better in compute-heavy and some balanced-heavy cases, while oracle policies provide the upper-bound reference. Therefore, the main finding is that DQN provides an adaptive learned trade-off rather than a universally optimal policy.

RQ3: How does partial offloading compare with full local and full edge execution?

The results show that neither full local execution nor full offloading is uniformly best. Local execution is effective for communication-heavy workloads because it avoids transmission overhead. Full offloading is effective for compute-heavy workloads because edge execution reduces computation delay and modeled energy. Partial offloading is most useful in mixed workloads, where the scheduler must balance computation cost and communication overhead. The DQN policy captures this behavior by selecting low offloading ratios for communication-heavy workloads and higher offloading ratios for compute-heavy workloads.

RQ4: How does the learned scheduler adapt its offloading behavior across different workload classes and system conditions?

The DQN scheduler adapts its average offloading ratio across workload classes. It selects almost local execution in the communication-heavy scenario, intermediate offloading in the mixed and balanced-heavy scenarios, and higher offloading in the compute-heavy scenario. This indicates that the learned policy captures the relationship between communication cost and computation cost. Oracle policy demonstrates that the DQN scheduler does not fully meet optimal solution, however demonstrates good adaptability for the dynamic environment.

7.3 Contributions

This thesis makes a contribution to the field of MEC scheduling by providing an integrated approach to energy-aware partial offloading. The problem is formulated and a deep Q-network based scheduling algorithm is provided which considers the

concept of partial offloading as one of the primary variables to consider within the algorithm’s design.

Additionally, the algorithm is compared with several existing algorithms to highlight its advantages. Beyond the proposed algorithm itself, the work also provides insight into how the presented algorithm can be applied to real-world scenarios.

7.4 Limitations and Future Work

Despite the promising results, several limitations remain. First, the experiments were performed on a simulated environment using a modeled energy function. The reported energy values represent modeled task-level energy rather than direct hardware-level power measurements. Future work should validate the energy model using real client and edge devices.

Second, the action space was fixed into finite set on offloading servers. This makes DQN suitable for current environment, but it is limited on granularity of the offloading decisions. Future work should extend simulated environment for the continuous reinforcement learning method that would fit dynamic decisions.

Third, future work should compare DQN with more advanced, complex algorithms, such as double DQN, PPO, SAC, in order to compare stability and scalability of the models.

Fourth, the environment could be also modified for the near real world conditions with addition of different simultaneous client devices, interference of the network and also congestions. It would be challenge algorithm and measure its scalability in dynamic environment.

Finally, future work should include a detailed real-time latency study. Such a study should separate scheduler decision latency, transmission latency, queueing delay, local execution time, edge execution time, and return latency. This would help determine whether the proposed method is suitable for dynamic, high-traffic edge deployments.

7.5 Final Remarks

Overall, the research presents a viable and effective approach to offloading computations adaptively in MEC systems. By using reinforcement learning to adaptively formulate policies for computation offloading, the system improves its performance. Thus, even within the limitations of the presented experiment, this approach can be utilized to enable efficient MEC systems.

Bibliography

- [1] J. Huang, F. Liu, and J. Zhang, “Multi-Dimensional QoS Evaluation and Optimization of Mobile Edge Computing for IoT: A Survey,” *Chinese Journal of Electronics*, vol. 33, no. 4, pp. 859–874, 2024.
- [2] L. Cao, T. Huo, S. Li *et al.*, “Cost Optimization in Edge Computing: A Survey,” *Artificial Intelligence Review*, vol. 57, p. 312, 2024.
- [3] ETSI, “Mobile-Edge Computing – Introductory Technical White Paper,” European Telecommunications Standards Institute, Sophia Antipolis, France, Tech. Rep., 2014.
- [4] A. A. Ismail, N. E. Khalifa, and R. A. El-Khoribi, “A Survey on Resource Scheduling Approaches in Multi-Access Edge Computing Environment: A Deep Reinforcement Learning Study,” *Cluster Computing*, vol. 28, p. 184, 2025.
- [5] J. Yan, S. Bi, and Y.-J. A. Zhang, “Offloading and Resource Allocation With General Task Graph in Mobile Edge Computing: A Deep Reinforcement Learning Approach,” *IEEE Transactions on Wireless Communications*, vol. 19, no. 8, pp. 5404–5419, 2020.
- [6] T. Alfakih, M. M. Hassan, A. Gumaei, C. Savaglio, and G. Fortino, “Task Offloading and Resource Allocation for Mobile Edge Computing by Deep Reinforcement Learning Based on SARSA,” *IEEE Access*, vol. 8, pp. 54 074–54 084, 2020.
- [7] X. Yu, S. Zhou, and B. Wei, “Dependent Task Offloading and Resource Allocation via Deep Reinforcement Learning for Extended Reality in Mobile Edge Networks,” *Electronics*, vol. 13, no. 13, p. 2528, 2024.
- [8] Y. Li, Y. Liu, X. Liu, Y. Xie, G. Han, T. Qiu, and P. Wu, “Mobility-Aware Partial Task Offloading and Resource Allocation Based on Deep Reinforcement Learning for Mobile Edge Computing,” *IEEE Transactions on Vehicular Technology*, vol. 74, no. 4, pp. 6459–6472, 2025.
- [9] E. Mustafa, J. Shuja, F. Rehman, A. Namoun, M. Ali, and A. Alourani, “Deep Reinforcement Learning With Dueling DQN for Partial Computation Offloading and Resource Allocation in Mobile Edge Computing,” *IEEE Access*, vol. 13, pp. 94 319–94 335, 2025.

- [10] L. Ale, N. Zhang, X. Fang, X. Chen, S. Wu, and L. Li, "Delay-Aware and Energy-Efficient Computation Offloading in Mobile Edge Computing Using Deep Reinforcement Learning," *IEEE Transactions on Cognitive Communications and Networking*, vol. 7, no. 3, pp. 881–892, 2021.
- [11] C. Feng, P. Han, X. Zhang, B. Yang, Y. Liu, and L. Guo, "Computation offloading in mobile edge computing networks: A survey," *Journal of Network and Computer Applications*, vol. 202, p. 103366, 2022.
- [12] J. Huang, F. Liu, and J. Zhang, "Multi-dimensional qos evaluation and optimization of mobile edge computing for iot: A survey," *Chinese Journal of Electronics*, vol. 33, no. 4, pp. 859–874, 2024.
- [13] A. A. Ismail, N. E. Khalifa, and R. A. El-Khoribi, "A survey on resource scheduling approaches in multi-access edge computing environment: A deep reinforcement learning study," *Cluster Computing*, vol. 28, p. 184, 2025.
- [14] S. Liu, Y. Yu, L. Guo, P. L. Yeoh, B. Vucetic, and Y. Li, "Adaptive delay-energy balanced partial offloading strategy in mobile edge computing networks," *Digital Communications and Networks*, vol. 9, no. 6, pp. 1310–1318, 2023.
- [15] L. Yu, H. Xu, Y. Zeng, and J. Deng, "Delay-aware resource allocation for partial computation offloading in mobile edge cloud computing," *Pervasive and Mobile Computing*, vol. 105, p. 101996, 2024.
- [16] J. Zhang, S. Ma, Z. Yan, and J. Huang, "Joint dnn partitioning and task offloading in mobile edge computing via deep reinforcement learning," *Journal of Cloud Computing*, vol. 12, p. 116, 2023.
- [17] J. Wang, H. Xu, R. Wang, and L. Wang, "Joint multi-user dnn partitioning and task offloading in mobile edge computing," *Ad Hoc Networks*, vol. 144, p. 103156, 2023.
- [18] L. Ale, N. Zhang, X. Fang, X. Chen, S. Wu, and L. Li, "Delay-aware and energy-efficient computation offloading in mobile edge computing using deep reinforcement learning," *IEEE Transactions on Cognitive Communications and Networking*, vol. 7, no. 3, pp. 881–892, 2021.
- [19] A. A. Ismail, N. E. Khalifa, and R. A. El-Khoribi, "A survey on resource scheduling approaches in multi-access edge computing environment: A deep reinforcement learning study," *Cluster Computing*, vol. 28, p. 184, 2025.
- [20] Y. Li, Y. Liu, X. Liu, Y. Xie, G. Han, T. Qiu, and P. Wu, "Mobility-aware partial task offloading and resource allocation based on deep reinforcement learning for mobile edge computing," *IEEE Transactions on Vehicular Technology*, vol. 74, no. 4, pp. 6459–6472, 2025.

- [21] M. Vidyasagar, “A tutorial introduction to reinforcement learning,” *SICE Journal of Control, Measurement, and System Integration*, vol. 16, no. 1, pp. 172–191, 2023.
- [22] M. Xie, J. Ye, G. Zhang, and X. Ni, “Deep reinforcement learning-based computation offloading and distributed edge service caching for mobile edge computing,” *Computer Networks*, vol. 250, p. 110564, 2024.
- [23] M. Y. Akhlaqi and Z. B. M. Hanapi, “Task offloading paradigm in mobile edge computing-current issues, adopted approaches, and future directions,” *Journal of Network and Computer Applications*, vol. 212, p. 103568, 2023.
- [24] S. Dong, J. Tang, K. Abbas, R. Hou, J. Kamruzzaman, L. Rutkowski, and R. Buyya, “Task offloading strategies for mobile edge computing: A survey,” *Computer Networks*, vol. 254, p. 110791, 2024.
- [25] X. Zhao, M. Li, and P. Yuan, “An online energy-saving offloading algorithm in mobile edge computing with lyapunov optimization,” *Ad Hoc Networks*, vol. 163, p. 103580, 2024.
- [26] R. Latip, J. Aminu, Z. M. Hanafi, S. Kamarudin, and D. Gabi, “Metaheuristic task offloading approaches for minimization of energy consumption on edge computing: a systematic review,” *Discover Internet of Things*, vol. 4, p. 35, 2024.
- [27] P. Peng, W. Lin, W. Wu, H. Zhang, S. Peng, Q. Wu, and K. Li, “A survey on computation offloading in edge systems: From the perspective of deep reinforcement learning approaches,” *Computer Science Review*, vol. 53, p. 100656, 2024.
- [28] Z. Luo and X. Dai, “Reinforcement learning-based computation offloading in edge computing: Principles, methods, challenges,” *Alexandria Engineering Journal*, 2024.
- [29] W. Shi, J. Wang, S. Liu, X. Liang, and F. Lyu, “Task offloading decision-making algorithm for vehicular edge computing: A deep-reinforcement-learning-based approach,” *Sensors*, vol. 23, no. 17, p. 7595, 2023.
- [30] L. Yu, H. Xu, Y. Zeng, and J. Deng, “Delay-aware resource allocation for partial computation offloading in mobile edge cloud computing,” *Pervasive and Mobile Computing*, vol. 105, p. 101996, 2024.
- [31] P. Khoshvaght, A. Haider, A. M. Rahmani, S. Rajabi, F. S. Gharehchopogh, J. Lansky, and M. Hosseinzadeh, “A self-supervised deep reinforcement learning for zero-shot task scheduling in mobile edge computing environments,” *Ad Hoc Networks*, vol. 178, p. 103977, 2025.
- [32] M. Xie, J. Ye, G. Zhang, and X. Ni, “Deep reinforcement learning-based computation offloading and distributed edge service caching for mobile edge computing,” *Computer Networks*, vol. 250, p. 110564, 2024.

- [33] Y. Chen, K. Peng, and C. Ling, “Copsa: a computation offloading strategy based on ppo algorithm and self-attention mechanism in mec-empowered smart factories,” *Journal of Cloud Computing*, vol. 13, p. 153, 2024.
- [34] A. Madiyev, D. Bulegenov, A. Karzhaubayev, M. Murzabulatov, and D.-M. Bui, “Energy-efficient offloading framework for mobile edge/cloud computing based on convex optimization and deep q-network,” *The Journal of Supercomputing*, vol. 81, no. 11, p. 1182, 2025.
- [35] H. He, C. Zhou, F. Huang, H. Shen, and S. Li, “Energy-efficient task offloading in wireless-powered mec: A dynamic and cooperative approach,” *Mathematics*, vol. 12, no. 15, p. 2326, 2024.
- [36] Y. Long, Z. Wang, S. Lan, R. Zhang, and K. Xu, “Energy-latency tradeoff for task offloading and resource allocation in vehicular edge computing,” *Computer Networks*, vol. 258, p. 111026, 2025.
- [37] L. Huang and Q. Yu, “Mobility-aware and energy-efficient offloading for mobile edge computing in cellular networks,” *Ad Hoc Networks*, vol. 158, p. 103472, 2024.
- [38] B. Tolba, M. Elsabrouty, and E. Shaaban, “Delay-energy-aware joint multi-cell association, service caching, and task offloading in hybrid-task heterogeneous edge computing networks,” *Computer Networks*, vol. 262, p. 111231, 2025.
- [39] M. Kumar, A. Kishor, P. K. Singh, and K. Dubey, “Deadline-aware cost and energy efficient offloading in mobile edge computing,” *IEEE Transactions on Sustainable Computing*, vol. 9, no. 5, pp. 778–789, 2024.
- [40] J. Liu, Y. Mi, X. Zhang, and X. Li, “Task graph offloading via deep reinforcement learning in mobile edge computing,” *Future Generation Computer Systems*, vol. 158, pp. 545–555, 2024.
- [41] R. Singh and S. S. Gill, “Edge ai: A survey,” *Internet of Things and Cyber-Physical Systems*, vol. 3, pp. 71–92, 2023.
- [42] X. Wang, Z. Tang, J. Guo, T. Meng, C. Wang, T. Wang, and W. Jia, “A comprehensive survey on on-device ai models,” *ACM Computing Surveys*, vol. 57, no. 9, pp. 1–44, 2025.
- [43] Z. Peng, Y. Qiu *et al.*, “Model caching and application offloading for mobile edge computing with ai online inference,” *IEEE Transactions on Services Computing*, 2025.