MRLATO: An Adaptive Task Offloading Mechanism Based on Meta Reinforcement Learning in Edge Computing Environment

Peiying Zhang, Jiamin Liu, Maher Guizani, Jian Wang, *Senior Member, IEEE*, Neeraj Kumar, *Senior Member, IEEE*, and Lizhuang Tan

***Abstract*—Traditional cloud computing models struggle to meet the requirements of latency-sensitive applications when processing large amounts of data. As a solution, Multi-access Edge Computing (MEC) extends computing resources to the edge of the network to reduce processing delays and improve user experience. However, in dynamically changing edge computing environments, effective decision making on whether to offload tasks to edge servers remains a core challenge. For this purpose, we propose MRLATO, an adaptive task offloading mechanism based on Meta Reinforcement Learning (MRL), which exploits a large amount of a priori knowledge of different tasks to achieve fast adaptation. The task offloading process is modelled as multiple Markov Decision Processes (MDPs) and solved using a Sequence to Sequence (Seq2Seq) neural network integrating multi-head attention and recursive task sequencing. It is shown by the experimental results that the proposed method has the lowest latency in all experimental settings and the convergence efficiency is improved by 14.06% compared to the traditional Deep Reinforcement Learning (DRL) algorithm. This research fully demonstrates the significant benefits of the deep integration of MRL with the edge computing domain, providing new optimisation ideas for task offloading decisions.**

***Index Terms*—Adaptation, edge computing, meta reinforcement learning, task offloading.**

Received 11 December 2024; revised 26 January 2025 and 23 February 2025; accepted 12 April 2025. Date of publication 18 April 2025; date of current version 19 September 2025. This work was supported in part by the National Natural Science Foundation of China under Grant 62471493 and Grant 62402257, in part by the Natural Science Foundation of Shandong Province under Grant ZR2023LZH017, Grant ZR2022LZH015, Grant ZR2024MF066, and Grant 2023QF025, and in part by the Open Foundation of Key Laboratory of Computing Power Network and Information Security, Ministry of Education, Qilu University of Technology (Shandong Academy of Sciences) under Grant 2023ZD010. The review of this article was coordinated by Dr. Mostafa Fouda. *(Corresponding author: Lizhuang Tan.)*

Peiying Zhang is with the Qingdao Institute of Software, College of Computer Science and Technology, China University of Petroleum (East China), Qingdao 266580, China, also with the Shandong Key Laboratory of Intelligent Oil and Gas Industrial Software, Qingdao 266580, China, and also with the Key Laboratory of Computing Power Network and Information Security, Ministry of Education, Qilu University of Technology (Shandong Academy of Sciences), Jinan 250014, China (e-mail: zhangpeiying@upc.edu.cn).

Jiamin Liu is with the Qingdao Institute of Software, College of Computer Science and Technology, China University of Petroleum (East China), Qingdao 266580, China, and also with the Shandong Key Laboratory of Intelligent Oil and Gas Industrial Software, Qingdao 266580, China (e-mail: liujiamin@s.upc.edu.cn).

Maher Guizani is with the Computer Science and Engineering Department, University of Texas, Arlington, TX 76019 USA (e-mail: mahergzani@gmail.com).

Jian Wang is with the College of Science, China University of Petroleum (East China), Qingdao 266580, China (e-mail: wangjiannl@upc.edu.cn).

Neeraj Kumar is with the Department of Computer Science and Engineering, Thapar Institute of Engineering and Technology, Patiala 147004, India, also with the Department of Computer Science, University of Economics and Human Sciences, 01-043 Warszawa, Poland, and also with the Department of Networks and Communications, College of Computer Science and Information Technology, Imam Abdulrahman Bin Faisal University, Dammam 31441, Saudi Arabia (e-mail: neeraj.kumar@thapar.edu).

Lizhuang Tan is with the Key Laboratory of Computing Power Network and Information Security, Ministry of Education, Shandong Computer Science Center (National Supercomputer Center in Jinan), Qilu University of Technology (Shandong Academy of Sciences), Jinan 250014, China, and also with the Shandong Provincial Key Laboratory of Computing Power Internet and Service Computing, Shandong Fundamental Research Center for Computer Science, Jinan 250014, China (e-mail: tanlzh@sdas.org).

Digital Object Identifier 10.1109/TVT.2025.3562142

I. Introduction

With the development of information technology and the popularization of the Internet of Things (IoT), the number of mobile devices and smart terminals has increased dramatically, providing users with rich and diverse experiences. However, the large amount of data and computing requirements they generate pose significant challenges to traditional cloud computing models. According to [1], mobile data traffic is expected to grow at a compound annual growth rate of approximately 20% by 2029. The traffic growth in fields such as augmented reality (AR), online gaming, and online video is more significant. However, existing connected devices have limitations in communication bandwidth, storage space, and processing capabilities, which contrasts sharply with the growing demand for emerging applications. Despite recent advances in hardware technology, mobile computing platforms still struggle when handling applications that require massive data generation, real-time processing and storage, as well as high computational intensity. To overcome these challenges, the industry has begun exploring strategies to migrate computing tasks from local devices to centralized cloud environments to address the issue of insufficient local device performance [2]. In conventional cloud computing scenarios, tasks can be offloaded from user equipment (UE) to cloud servers with more powerful computing capabilities. When dealing with massive data, the delay in data transmission on account of the long distance between cloud servers and UE and bandwidth limitations make it unable to meet the requirements of tasks with strict time limits in most cases. Consequently, exploring new computing architectures has become a shared focus of academic and industrial circles.

Against this backdrop, MEC, as an innovative technological paradigm, has emerged. By pushing computing resources, storage capabilities, and application functionalities to the network edge, MEC significantly reduces data transmission distances, effectively decreases processing latencies, improves the efficiency of the mobile network, and enhances user experiences [3]. However, the high dynamism of the MEC environment, encompassing user mobility, fluctuations in network conditions, and variations in edge server loads, introduces unprecedented complexity into task offloading decisions. Furthermore, due to the multiple factors involved in task offloading in MEC environments, finding an optimal task offloading policy has proven to be a NP-hard problem [4]. Task offloading policy is about deciding whether a task should be executed on the UE or on the MEC edge server. If consecutive tasks are dealt with on the same device, it may reduce the time for uploading and downloading intermediate results, but it could increase the overall computation time. Conversely, if different devices are used to deal with consecutive tasks, it may reduce calculation time but increase the frequency of uploading and downloading intermediate results. Adaptiveness refers to the system's ability to quickly tweak task offloading strategies depending on the existing environment and task requirements, achieving optimal performance. Fig. 1 shows the task offloading scenario in the MEC system architecture, and the specific components of this system will be described in detail in the following chapters. Traditional task offloading methods typically rely on predefined rules or static strategies, which are often inadequate for adapting to complex and variable real-world scenarios. Therefore, how to achieve efficient adaptive task offloading in an uncertain environment to maximize system performance and optimize resource utilization has become a core problem to be solved urgently in the MEC field.

MRL combining Reinforcement Learning (RL) and meta-learning, as an advanced machine learning (ML) method, aims to address the generalization issues faced by traditional RL methods when encountering new and unknown environments. Traditional RL methods typically require substantial amounts of training data and directly learn the mapping from states to actions, neglecting the structure and objectives of the tasks themselves. In contrast, the objective of meta-learning is to enable machines to acquire the ability to learn. By learning a general learning policy across multiple tasks, a system is capable of rapidly adjusting to new tasks. Instead of directly mapping states to actions, MRL algorithms learn how to adjust and optimize their RL strategies based on task characteristics. The MRL framework includes two learning loops: the outer loop gradually adjusts parameters of the meta-policy through experience accumulated across many tasks, while the inner loop, founded on the meta-policy, quickly adapts to new tasks with a few gradient updates. Therefore, in the context of the task offloading problem, utilizing MRL can remarkably enhance the performance of learning new tasks, enabling the system to rapidly adjust task offloading strategies in a dynamic MEC environment.

To grapple with the complexity and dynamism of the MEC environment, we propose an adaptive task offloading mechanism

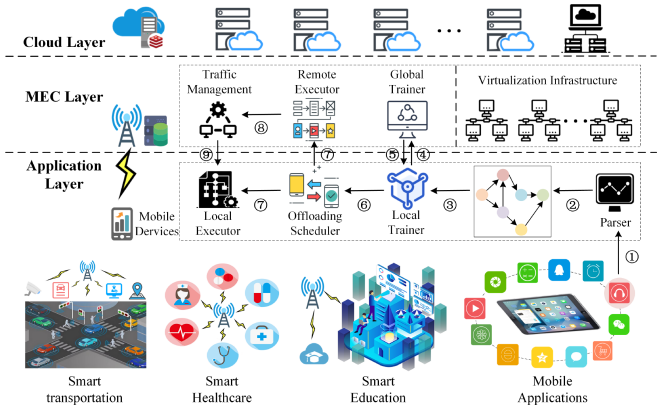

Fig. 1. A scenario example of task offloading in the MEC environment. The succinct description of the process: (1) Input applications; (2) Parse into DAG; (3) Input to local trainer; (4) Upload local training results; (5) Return remote training results; (6) Transfer unloading policy; (7) Schedule subtasks to remote and local executors respectively according to the unloading policy; (8) Transfer remote execution results; (9) Return the execution results to local.

based on MRL, which is called MRLATO. Initially, a general policy is derived for all users, followed by generating effective strategies for each UE based on this general policy and local data. Taking the current popular AR games as an example, players need to process a large amount of image data in real time during the game process to complete complex tasks such as scene recognition and virtual object rendering. By adopting an adaptive offloading algorithm, the system can dynamically decide to offload some computing tasks to edge servers based on current network conditions, task urgency, and other factors, thereby significantly reducing task execution latency, ensuring smooth game graphics, and providing players with a better and more immersive gaming experience. The principal contributions of this study are briefly outlined as follows:

- Introduce MRL to the task offloading problem under the MEC framework. can effectively solve the limitations of traditional RL methods in multi-task environments due to low training efficiency and poor task adaptability, and provides a new optimisation idea for the task offloading problem in MEC.
- Implement a simpler directed acyclic graph (DAG) generator based on the existing DAG synthesis logic, and at the same time optimise the task priority ranking algorithm in complex tasks. Specifically, in the priority calculation, the inheritance relationship of subtasks and the computational cost of tasks are considered comprehensively, and the task weights are updated through dynamic recursion. The improved sorting policy significantly improves the parsing efficiency and sorting accuracy of the task graph.
- Design a neural network model based on Seq2Seq structure. Each task is first modelled as a DAG, and its task characteristics and dependencies are input into the network by converting them into embedding vectors, enabling the model to better capture complex dependencies. The introduction of the multi-head attention mechanism enables the model to process the dependencies between tasks in parallel and capture the task features from multiple perspectives,

which improves the ability to characterise the tasks and the computational efficiency, and helps the decoder to make more accurate offloading decisions.

- With reasonable experimental arrangements, we strongly validate the superiority of the proposed method in edge computing environments. Specifically, in complex task offloading scenarios, MRLATO significantly reduces the overall task execution latency compared to other methods. In addition, the MRL and Seq2Seq model-based scheme demonstrates significant improvements in training efficiency and performance convergence speed.

The subsequent part of the paper is structured in the following manner. Section II reviews related work on task offloading mechanisms. Section III introduces relevant background knowledge and defines the problem. Section IV details the system architecture and algorithm implementation. Section V exhibits the experimental setup and results, trailed by a discussion. Eventually, Section VI wraps up the paper and sketches out the prospective research directions.

II. Related Work on Task Offloading Mechanisms

With the development of edge computing, task offloading has become an important research field. The existing task offloading strategies can be roughly classified into heuristic algorithms, dynamic programming and optimization, and ML algorithms. These methods have their own advantages and disadvantages in handling tasks, and specific applications need to be customized and designed according to actual scenarios.

Given that the MEC offloading problem is inherently NP-hard, early research work mainly focused on heuristic algorithms. Mei et al. [5] adopt a two-stage optimization policy to minimize the energy consumption of mobile devices while satisfying task delay constraints. Li [6] also propose a heuristic algorithm based on a two-stage approach, which comprehensively considers the characteristics of the communication channel, the power consumption model of computation and communication, and the characteristics of the current task, using greedy methods to reduce the energy consumption of mobile devices. Bi et al. [7] consider various factors such as task execution time, transmission time, latency, CPU speed and transmission power, and solve a nonlinear constrained optimisation problem by a particle swarm optimisation algorithm based on genetic simulated annealing to optimise cloud-edge task offloading and resource allocation. In [8], [9], [10], [11], [12], Goli et al.'s research covers a wide range of fields such as organ transplantation, Industry 4.0 project scheduling, and supply chain, etc. They apply the possibility planning framework, ML, meta-heuristic algorithms, and the construction of mathematical models to solve multi-objective optimisation problems in complex environments, which provide references from different perspectives for the improvement and refinement of task offloading strategies.

Although heuristic algorithms provide lightweight optimization solutions, they may have limitations when dealing with more complex offloading scenarios. As a result, researchers have begun to adopt more structured approaches such as dynamic programming to cope with complex optimization problems. Xu et al. [13] propose a task offloading method named COM to address the conflict between the resource limitations of mobile devices and the user demands in the Internet of Things. Fang et al. [14] model the problem as a multiuser computation task offloading game and propose a distributed multiuser computation task offloading algorithm based on better replies, which improves the total user offloading benefits. Goudarzi et al. [15] propose a cost model with weights to optimize the processing time and energy consumption for IoT devices within a heterogeneous computing environment, and also put forward a technique for the placement of batch applications using the Memetic Algorithm. Li et al. [16] propose a cooperative resource allocation model for IoT application offloading in a multi-user MEC environment, and achieved performance improvements by decomposing the target problem into three subproblems to obtain the solution of the objective function. Xia et al. [17] propose an online distributed optimization algorithm based on game theory and perturbation Lyapunov optimization theory, which is used to determine heterogeneous task offloading, demand-oriented computational resource allocation, and battery energy regulation strategies, in order to optimize system performance.

The wide application of MEC technologies and the complexity of wireless network architectures have contributed to the increasing challenges of task offloading, and further promoted the application and development of intelligent algorithms. Wang et al. [18] put forward a task offloading framework for multi-access edge computing based on DRL, which acquires a reduced latency compared to heuristic baselines in various transmission rates and task number scenarios, and can obtain near-optimal results within the polynomi al time complexity. Yang et al. [19] combine the perceptive functions of Deep Learning with the decision-making capabilities of RL, with the aim of minimizing the latency of offloading decisions by taking into account diverse channel conditions, fluctuating latency limitations among users, and restricted computational resources. Alfakih et al. [20] propose an MEC system model that simultaneously considers both computational delay and power consumption, and adopted a RL-based algorithm to solve the resource management problem in edge servers, making optimal offloading decisions to minimize system costs. Hao et al. [21] study the task offloading problem in a multi-drone cooperative assisted MEC system, taking into account task priorities and binary offloading modes. They cast the problem as a mixed integer programming issue and devised a new DRL algorithm predicated on potential space to enhance the long-term average system gain. Zarandi et al. [22] propose a federated DRL framework for solving the problem of IoT devices that involves minimizing the task completion delay and the energy consumption of multi-objective optimization problems in IoT devices and demonstrated the effectiveness of the framework in terms of learning speed. Zheng et al. [23] decompose the problem of minimizing the total computational delay into a primary problem and a subproblem, and solve them separately by constructing a deep neural network (DNN)-based DRL model and designing a worst-WD-adjusting algorithm. Shakarami et al. [24] conduct a detailed and comprehensive inquiry into ML-based task offloading mechanisms in MEC, including the classification and

comparison of existing methods, their respective advantages and disadvantages, as well as the current issues that remain to be addressed.

In summary, heuristic algorithms are simple but limited in complex scenarios, dynamic planning is structurally complex but capable of handling complex problems, and intelligent algorithms are adaptable but computationally expensive. Therefore, although these algorithms perform well in their respective applicable scenarios, in the complex and changing network environment, the dynamic change of resources becomes more and more complex, and the diversity and real-time requirements of the tasks continue to improve, the existing algorithms still suffer from insufficient adaptability to the dynamic change of resources, and it is difficult to ensure the timeliness of the tasks while realising the efficient use of resources.

III. Background Knowledge and Problem Definition

A. Reinforcement Learning

RL is a method for learning from the environment to maximise cumulative reward. It models the learning task as a MDP, which consists of a state space S, an action space A, a reward function R, a state transfer probability matrix P, an initial state distribution P_0 and a discount factor γ. The discount factor serves to measure the significance of future rewards against current rewards and usually takes a value between 0 and 1.

In RL, a policy is a rule by which an intelligent body chooses an action based on its current state. A policy can be represented as a mapping that maps each state to an action or a probability distribution of actions. We denote the policy by $\pi(a|s)$, where $a \in A$ and $s \in S$. When an intelligent body interacts with the environment, it chooses an action based on a policy and receives a corresponding reward. We denote the trajectory of an intelligent body sampled from the environment as $\tau = (s_0, a_0, r_0, s_1, a_1, r_1, \ldots)$, where s_0 is the initial state, a_0 is the action taken in the state, r_0 is the reward obtained after taking the action, and s_1 is the next state to which s_0 is transferred after taking the action. The probability distribution of the trajectory depends on the policy π and the dynamic properties of the environment.

The state-value function $V(s)$ represents the expected cumulative reward that an intelligent body can obtain by following the policy starting from state s. Specifically, the state value function can be computed by the following equation:

$$V_s = \mathbb{E}_{\tau \sim P(\tau|s,\pi)} \left[\sum_{t=k}^{T} \gamma^{t-k} r_t | s_k = s, \pi \right], \tag{1}$$

where $\mathbb{E}_{\tau \sim P(\tau|s,\pi)}$ denotes the expected value of the probability distribution $P(\tau|s,\pi)$ taken over the trajectory τ given the state s and the policy π, T is a finite number of time steps, γ is the discount factor and r_t is the reward acquired at time step t.

The goal of RL is to find an optimal policy $\pi(a|s;\theta^*)$ that allows the intelligence to maximise the expected cumulative reward in any initial state, which is formulated as follows:

$$R_{total} = \sum_{s_0} \mathbb{E}_{s_0 \sim P_0} \left[V\left(s_0\right) \right]. \tag{2}$$

B. Meta Reinforcement Learning

MRL was proposed to solve the problem of insufficient adaptation of RL in the face of a new environment or task. Its core idea is to use previous learning experiences to hasten learning of new tasks. MRL usually consists of two cycles: the outer loop and the inner loop. In the outer loop, the intelligent body gradually adjusts the parameters of the meta-policy by learning on several different tasks in order to adapt the meta-policy to different tasks. In the inner loop, the intelligent body learns quickly for a specific new task based on the meta-policy, in order to obtain a specific policy adapted to that task.

We can think of tasks as different MDPs, and the objective of MRL is to acquire a generic meta-policy that enables the intelligent body to quickly adapt and learn an effective specific policy with a small number of gradient updates when faced with a new MDP. To achieve this goal, MRL usually uses some special algorithms and techniques. For example, a common approach is gradient-based MRL. Specifically, let the parameter of the meta-policy be θ. The objective is to find the optimal parameter θ^* such that for task T_i sampled from the task distribution, the updated policy achieves the maximum expected reward. The objective function can be expressed as:

$$J\left(\theta\right) = \mathbb{E}_{T_i \sim \rho(T)} \left[J_{T_i} \left(U\left(\theta, T_i\right) \right) \right], \tag{3}$$

where $U(\theta, Ti)$ is an update function that updates the parameters to get new parameter values based on the current meta-policy parameter θ and task T_i.

When applying the Vanilla policy Gradient (VPG), the objective function $J_{T_i}(\theta)$ for every task T_i can be formulated as:

$$J_{T_i}\left(\theta\right) = \mathbb{E}_{t \sim P_{T_i}(\tau|\theta)} \left[\sum_{t=0} \gamma^t r_t - b\left(s_t\right) \right]. \tag{4}$$

Here, $\gamma^t r_t$ denotes the discounted value of the reward obtained at the time step, and $b(s_t)$ is an arbitrary baseline that does not vary with action and is used to reduce variance to improve learning stability.

The update function can be defined in the following form:

$$U\left(\theta, T_i\right) = \theta + \alpha \sum_{t=0}^{k} g_t. \tag{5}$$

where α denotes the learning rate for the inner loop and g_t denotes a t-step gradient ascent for T_i. The updating process for the outer loop usually involves an outer loop learning rate β as well, so the updating rule for the parameters can be formulated as:

$$\theta \leftarrow \theta + \beta \mathbb{E}_{T_i \sim \rho(T)} \left[\bigtriangledown_\theta J_{T_i} \left(U\left(\theta, T_i\right) \right) \right]. \tag{6}$$

Here, $\bigtriangledown_\theta J_{T_i}(U(\theta, T_i))$ denotes the gradient of the objective function with respect to parameter θ, indicating in which direction the update should be made. If it is negative, it means that decreasing θ may make the objective function increase.

Since (3) follows Model-Agnostic Meta-Learning (MAML) as defined in [25], and the challenge of MAML mainly lies in the fact that the computational cost of second-order derivatives is

too high. To address this problem, in this paper we use first-order approximation, which improves the computational efficiency in cases involving complex neural networks (e.g., Seq2Seq).

C. Problem Definition

In this paper, we model the task offloading problem as an optimisation problem in which the scheduling and allocation of multiple dependent tasks must be performed between the UE and the MEC host The goal is to minimise the total execution delay of all tasks. We use a DAG to describe the structure of tasks in an application, where vertices represent tasks and edges represent dependencies between tasks. Fig. 2 displays a basic illustration of modeling a realistic application as a DAG.

1) Task Model: We represent the dependent tasks in mobile applications as a DAG, denoted as $D = (N, L)$, where $N = (n_1, n_2, n_3, \ldots n_n)$ is the set of tasks and L is the set of directed edges of task dependencies. For tasks n_i and n_j, when there is a dependency relation $n_i \rightarrow n_j$, it means that task n_i depends on n_j, and n_i can start only when n_j has finished executing.

In case the task is transferred to the MEC host for remote execution, the execution time is limited by the transfer rate and the computational resources of the MEC host. The remote execution process of the task consists of three stages: data upload, MEC host processing, and data download. The specific delay is calculated as follows.

Uploading delay: $T_{ul}(n_i) = \frac{data_i^{send}}{U_r}$, where $data_i^{send}$ is the amount of uploaded data, U_r is the uplink transmission rate.

Processing delay: $T_m(n_i) = \frac{C_{n_i}}{f_{vm}}$, where C_{n_i} is the CPU cycles required for task n_i, and f_{vm} is the computing power of each VM (Assuming that all VMs enjoy the resources of the MEC host equally).

Download delay: $T_{dl}(n_i) = \frac{data_i^{recv}}{D_r}$, where $data_i^{recv}$ is the amount of downloaded data, D_r is the downlink transmission rate.

If the task is executed locally, then the execution time is contingent on the computational power of the UE f_{ue}. Thus the execution time can be expressed as $T_{ue}(n_i) = \frac{C_{n_i}}{f_{ue}}$

2) Available Time Calculation: In the task scheduling process, the start execution time of each task is not only affected by the completion time of its dependent tasks, but also limited by the available time of the resources. We define the available times of the upload channel, MEC host, download channel, and UE as $A_{ul}(n_i)$, $A_m(n_i)$, $A_{dl}(n_i)$ and $A_{ue}(n_i)$.

For each task n_i, its time to start uploading data depends on the completion time of its parent task and the available time of the upload channel, and $A_{ul}(n_i) = \max(A_{ul}(n_{i-1}), ET_{ul}(n_{i-1}))$, so the upload end time $ET_{ul}(n_i)$ can be defined as:

$$ET_{ul}(n_i) = \max(A_{ul}(n_i), \max(ET_{ue}(n_j), ET_{dl}(n_j))) + T_{ul}(n_i). \quad (7)$$

Here, $n_j \in parent(n_i)$.

Similarly, let $A_m(n_i) = \max(A_m(n_{i-1}), ET_m(n_{i-1}))$, and $A_{dl}(n_i) = \max(A_{dl}(n_{i-1}), ET_{dl}(n_{i-1}))$, then the processing

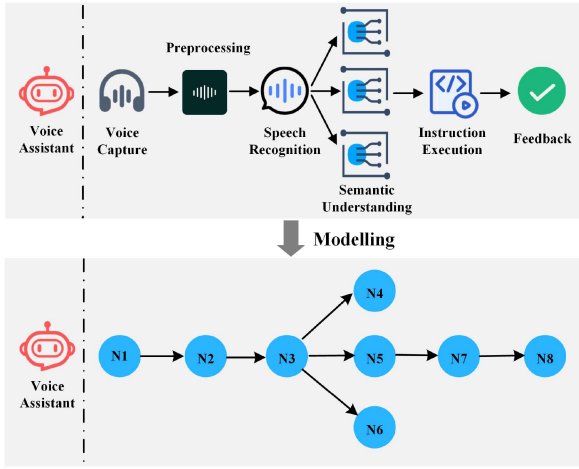

Fig. 2. A basic illustration of modeling a realistic application as a DAG.

end time at the MEC host $ET_m(n_i)$ and the download completion time $ET_{dl}(n_i)$ of task n_i are:

$$ET_m(n_i) = \max(A_m(n_i), \max(ET_{ul}(n_i), ET_m(n_j))) + T_m(n_i). \quad (8)$$

$$ET_{dl}(n_i) = \max(A_{dl}(n_i), ET_m(n_i)) + T_{dl}(n_i). \quad (9)$$

If task n_i is executed on the UE, the completion time $ET_{ue}(n_i)$ is given by:

$$ET_{ue}(n_i) = \max(A_{ue}(n_i), \max(ET_{ue}(n_j), ET_{dl}(n_j))) + T_{ue}(n_i). \quad (10)$$

3) Optimisation Objective: The objective of the whole DAG is to minimise the total execution time of all exit tasks, let the scheduling plan be $A_{1:n} = \{a_1, a_2, \ldots, a_n\}$, where a_i denotes the offloading decision of task n_i. Our optimisation objective is to minimise the total delay of the DAG:

$$T_{total} = \max_{n_e \in \mathcal{K}} (ET_{ue}(n_e), ET_{dl}(n_e)), \quad (11)$$

where $\mathcal{K}$ denotes the set of exit tasks which without any subtasks.

IV. System Architecture and Algorithm Implementation

In order to achieve highly adaptive and low-latency task offloading in MEC environments, this study constructs a task offloading framework that incorporates MRL techniques. This chapter first introduces the way the MEC system architecture is combined with MRL, followed by a detailed description of the modeling of the task offloading process, and gives the implementation of the relevant algorithms at the end.

A. MRL-Driven MEC Architecture Design

This subsection describes how the UE layer and the edge computing layer of the system can achieve adaptive task offloading decision optimisation through bidirectional communication and collaboration. The system is designed with an inner and outer double-loop training mechanism: the inner loop training

is responsible for learning the policy tailored to the task, while the outer loop training manages the optimisation and updating of the overall policy. By combining MRL techniques, the system has the ability to adjust the policy based on historical task data and network conditions, thus achieving efficient task offloading.

1) UE Layer: The UE layer is the initiator of task offloading and mainly consists of heterogeneous UE. Each UE makes the decision of task offloading by communicating with the edge server and combining the current task and network state. The other modules of this layer are specified as follows.

Parser Module: It is responsible for transforming a wide variety of mobile applications into a DAG structure, through which the dependencies of the tasks are clarified, providing the basis for subsequent task offloading decisions.

Local Trainer: It is responsible for running the inner loop training, which locally trains task-specific policies based on meta-policies downloaded from the MEC host. The UE communicates with the MEC host via a local transport unit to upload or download the parameters of the policy network. Since inner loop training only requires passing a small amount of task data and training steps, it allows UEs to learn and make task decisions quickly locally.

Offload Scheduler: It is responsible for using the trained policy network for offloading decisions. When the offloading decisions for all tasks are determined, the local tasks will be executed on the UE, while others will be transmitted to the MEC host.

Local Executor: It is responsible for receiving and executing the local tasks assigned by the offload scheduler, ensuring that these tasks run efficiently on the UE according to the predefined computational resource requirements.

2) Edge Computing Layer (MEC): The edge computing layer mainly consists of the MEC Host and the platform and virtualisation infrastructure on it. As an intermediary layer between the UE and the core network, it is responsible for providing computing and storage resources to help the UE quickly process tasks that require high computing power, reduce the communication delay to and from the core network, and improve the system response speed.

Traffic Management: A part of the MEC platform, responsible for controlling the routing of data flow, rule management and distribution of edge services to ensure the high efficiency of task processing and reasonable use of network resources.

Global Trainer: It is mainly responsible for conducting outer loop training. It needs to collect the execution data of task offloading from multiple UE and train on the MEC host based on these policy parameters to complete the update of the meta-policy. This module ensures that the meta-policy can be continuously improved in long-term operation and continuously adapted to new task environments.

Remote Executor: It is in charge of dealing with tasks offloaded from the UE, allocating them to the corresponding virtual machines for processing, and returning the execution results to the UE.

Virtualised infrastructure: It provides each user with an exclusive and isolated computing and storage environment to execute the tasks offloaded from the UE according to the commands of the remote executor.

3) Summary of the Training Process: The training process for integrating the MRL method into the MEC architecture is shown in Fig. 3. This architecture achieves lower computational complexity and faster response speed by layered design and combining inner and outer loop policy training.

B. MDP-Based Modeling of Task Offloading

MDP is a mathematical framework for describing decision making in dynamic environments. Modelling the offloading process as MDP helps to achieve efficient task scheduling and resource allocation in MEC environments. Because edge computing environments are complex and contain multiple devices, wireless channels, MEC hosts and task dependencies, MDP modelling allows for learning generic meta-policies before optimising the policies for a specific context, thus decomposing the learning process and accelerating the efficiency of system learning through shared experience.

We define finding its optimal offloading scheme for each MDP as a learning task. The distribution of all learning tasks is denoted by $\rho(T)$, where a task can be defined as $T = (S, A, P, P_0, R, \gamma)$, $T \in \rho(T)$.

State: During task scheduling, task execution is affected by various factors such as the amount of data, dependencies, transmission rate, and available resources. Based on the related equations (7)–(10), it can be seen that the availability of computational resources in the MEC is determined by the scheduling policy of the previous tasks, so the state can be further defined by the processed DAG and the previous offloading policy:

$$S := \{si | si = (D(N, L), A_{1:i})\}, \tag{12}$$

where $i \in [1, |T|]$, $D(N, L)$ denotes the processed DAG. Specifically, in order to enable the neural network to better capture the interrelationships among tasks, we process each DAG as an embedding, and each embedding is made up of three parts: a vector containing the current task index and normalized task attributes, as well as vectors containing the parent task index and child task index, respectively. In order to sort each subtask in the graph, we propose a recursion-based priority sorting algorithm, specifically, we compute a rank value for each task based on its own execution time and the priority of the succeeding task as follows:

$$\text{rank}(n_i) = \begin{cases} t_i & \text{if } n_i \in \mathcal{K}, \\ t_i + \max_{n_j \in \text{child}(n_i)} (\text{rank}(n_j)) & \text{if } n_i \notin \mathcal{K}, \end{cases} \tag{13}$$

where t_i is the total time required for n_i to complete the entire execution process. When $n_i \in \mathcal{K}$, the time complexity is $O(1)$. Otherwise, if all its subtasks need to be traversed, the computational complexity of a single task is $O(j)$, where j is the number of subtasks. Under extreme task dependencies, the time complexity of the algorithm reaches a worst-case scenario close to $O(n^2)$, but in reality, the number of subtasks under task dependencies is limited, and the average time complexity is closer to $O(n)$, which may be better in practical applications.

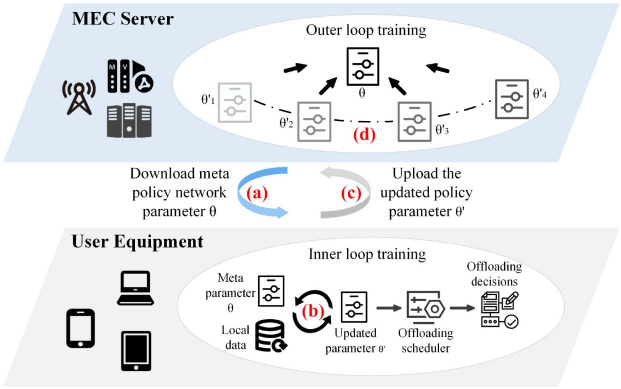

Fig. 3. The training process for integrating the MRL method into the MEC architecture: (a) UE downloads updated metapolicy parameters from the MEC host; (b) UE performs training founded on the meta-policy and current task data, generates offloading decisions and executes them; (c) UE uploads the updated task policy to the MEC host; (d) The MEC host updates the global meta-policy based on the policy parameters from multiple devices and returns it to the UE for subsequent use.

The main space consumption of this algorithm lies in storing the rank value of each task and the possible stack space during recursion, so the total space complexity is $O(n+d)$. Due to the small size of d, the spatial complexity is mainly determined by $O(n)$.

Action: Since there are only two options for scheduling each task, either executing it locally or offloading it to the MEC server, the action can be concisely denoted as $A := \{0, 1\}$.

Reward: In Section III-C we have summarised the optimisation objective of the learning task, i.e., to minimise the total latency of each MDP, T_{total}. Here we define the reward as:

$$\triangle T^{i}_{total} = T_{total}\left(A_{1:i}\right) - T_{total}\left(A_{1:i-1}\right), \tag{14}$$

which denotes the predicted negative increase of the delay obtained when an action is taken on n_i.

After further definition of the above components, the policy when scheduling n_i can be expressed as $\pi(a_i|D(N,L), A_{1:i-1})$. If a task consists of t subtasks, then signify the probability of executing the scheduling scheme $A_{1:n}$ for this task as:

$$\pi\left(A_{1:n}|D(N,L)\right) = \prod_{i=1}^{n} \pi\left(a_i|D(N,L), A_{1:i-1}\right), \tag{15}$$

In order to efficiently represent the above task scheduling strategies, we designed a neural network structure based on the Seq2Seq model. This network is capable of handling complex task scheduling scenarios by encoding task embeddings and decoding offloading decisions, especially for task offloading decisions for DAGs. The network consists of two main components: an encoder and a decoder.

Encoder: The main task of the encoder is to encode the input task embedding sequence and convert it into a high-dimensional representation for decision making. The input task embedding sequence can be represented as $[n_1, n_2, n_3, \ldots, n_n]$, where each n is an embedding vector of tasks. In a concrete implementation, the encoder uses a recurrent neural network (RNN) such as a long short-term memory (LSTM) network to process the input sequence. For each task n_i, the output of the encoder can be obtained by the following recursive formula:

$$e_i = f_e\left(n_i, e_{i-1}\right), \tag{16}$$

where f_e denotes the recursive function of the encoder that depends on the embedding of the current task and the state e_{i-1} of the previous task. The final output of the encoder $[e_1, e_2, e_3, \ldots, e_n]$ is a high-dimensional representation of the entire sequence of tasks, which the attributes of the tasks as well as their topological relationships with other tasks.

Decoder: The task of the decoder is to generate step-by-step offloading decisions for each task based on the encoder's output sequence. The decoder takes the same structure as the encoder. For task n_i, the output of the decoder at step j can be expressed by the following equation:

$$d_j = f_d\left(d_{j-1}, a_{j-1}, c_j\right), \tag{17}$$

where d_{j-1} denotes the output of the decoder at the previous step, a_{j-1} is the decision of the previous task, and c_j is the context vector. Specifically, we take the output of the decoder as the queries of the multi-head attention mechanism, the output of the encoder as the keys and values, and their dimensions as d_{model}. Set the number of heads of the multi-head attention mechanism as h, and the dimension of each head as d_k, where $d_k = d_{model}/h$. Map the inputs to multiple subspaces by the following linear transformation:

$$Q' = W_Q Q,\ K' = W_K K,\ V' = W_V V, \tag{18}$$

where W_Q, W_K and W_V are the projection matrices of queries, keys, and values, respectively. These projections are then split into multiple heads:

$$Q_i, K_i, V_i = split\_heads(Q', K', V', h). \tag{19}$$

For each head, first calculate the dot product of the query and key to get the attention score, and scale it:

$$a\left(Q_i, K_i, V_i\right) = softmax\left(\frac{Q_i K_i^T}{\sqrt{d_k}}\right) V_i. \tag{20}$$

The output of each head is obtained by softmax to get the attention weights and multiplied with the value V_i to get the output of each head. Then, the outputs of all the heads are combined:

$$output = concat\left(a\left(Q_1, K_1, V_1\right), \ldots, a\left(Q_h, K_h, V_h\right)\right). \tag{21}$$

Finally, the merged output is subjected to a linear transformation to obtain the final context vector:

$$c = W_O \cdot output(Q, K, V), \tag{22}$$

where W_O is the projection matrix of the output.

The traditional Seq2Seq model compresses the entire input sequence into a fixed-length vector, and information may be lost when the input sequence is long, especially when dealing with complex DAG structures. By introducing the multi-head attention mechanism, the decoder is able to pay attention to multiple different parts of the encoder's output sequence simultaneously,

which helps the decoder make more accurate task offloading decisions at each step.

The Seq2Seq network is used to approximate not only the scheduling policy $\pi(a_j|s_j)$, but also the value function $v_\pi(s_j)$. In the network structure, the decoder's output $[d_1, d_2, d_3, \ldots, d_n]$ is passed to two independent fully-connected layers for generating the output of the policy and the value function, respectively. Most of the parameters of the encoder and decoder are used jointly by these two components, thus improving computational efficiency when extracting common features of the task graph. During training, the action a_j is obtained by sampling from the policy $\pi(a_j|s_j)$, while during inference, the action a_j is determined by $a_j = arg\max_{a_j} \pi(a_j, s_j)$.

The time complexity of the Seq2Seq depends mainly on the inference process of the encoder and decoder. In the usual case, the time complexity of the network is $O(n^2)$, where n is the number of tasks. Considering that the number of tasks in most mobile applications is usually less than 100 [26], [27], [28], this complexity is feasible in practical applications.

C. Algorithm Implementation Under Two-Tier Training Mechanism

The flow of the algorithm is shown in Algorithm 1. The implementation of the algorithm consists of two training sessions: an inner loop and an outer loop. The two work in tandem to ensure both efficient policy optimisation on specific tasks and fast adaptation under multi-tasking. Unlike the traditional policy gradient approach, we use a proximal policy optimisation (PPO)-based objective function for training in the inner loop.

1) Inner Loop: For each task T_i, we implement the policy optimisation through the following steps:

Generate Trajectories: For a given task T_i, PPO generates multiple trajectories $\tau \sim P_{T_i}(\tau, \theta_i^{old})$ using the sample policy $\pi_{\theta_i}^{old}$, recording the state, action and reward at each step in the task scheduling. The target policy π_{θ_i} is then updated with initial parameters equal to the sample policy.

Compute the PPO loss function: The goal of PPO is to avoid drastic policy updates by clipping alternative goals. The loss function can be expressed as follows:

$$L^P(\theta_i) = \mathbb{E}_\tau \left[\sum_{k=1}^{n} \min \left(Pr_k A(s_k), clip_{1-\epsilon}^{1+\epsilon} (Pr_k) A(s_k) \right) \right], \tag{23}$$

where ϵ is a hyperparameter used in the clips to prevent over updating, thus avoiding instability during training, Pr_k is the ratio of the probability of the sample policy to the target policy:

$$Pr_k = \frac{\pi_{\theta_i}(a_k|D(N,L), A_{1:k})}{\pi_{\theta_i}^{old}(a_k|D(N,L), A_{1:k})}. \tag{24}$$

Calculate the dominance function: The dominance function $A(s_k)$ measures the advantage of taking action a_k in the current state s_k compared to the baseline policy. It is defined as:

$$A(s_k) = \sum_{l=0}^{n-k+1} (\gamma\lambda)^l \left(r_{k+l} + \gamma V(s_{k+l+1}) - V(s_{k+l}) \right), \tag{25}$$

Algorithm 1: MRL-based algorithm for double-loop mechanism.

Input: Task distribution $\rho(T)$
Output: Optimized meta-policy parameters θ

```
Initialize meta-policy parameters θ in a random manner
for each meta-iteration k = 1, ..., K do
    Sample a batch of n tasks {T_1, T_2, ..., T_n} from ρ(T)
    for each task T_i ∈ {T_1, ..., T_n} do
        Initialize specific parameters θ_i^old ← θ and θ_i ← θ
        Collect trajectories T = (τ_1, τ_2, ...) from task T_i using the current task policy π_{θ_i^old}
        Optimize task-specific parameters θ_i using Adam via m steps: θ'_i ← θ_i + α∇_{θ_i} J_in(θ_i)
    end for
    Update meta-policy parameters θ with task-adapted gradients: θ ← θ + β g_meta
end for
return θ
```

where $\lambda \in [0, 1]$ controls the balance between bias and variance.

Value function update: In addition to optimising the policy, we need to update the value function. The loss of the value function is defined as:

$$L^V(\theta_i) = \mathbb{E}_\tau \left[\sum_{k=1}^{n} \left(V(s_k) - \hat{V}(s_k) \right)^2 \right], \tag{26}$$

where $\hat{V}(s_k)$ is the actual return value estimate: $\hat{V}(s_k) = \sum_l^{n-k+1} \gamma^l r_{k+l}$.

Updating the policy: Optimise the policy parameter θ_i by updating the gradient over a number of steps to generate an optimal policy π'_{θ_i} for each task T_i. The objective function of the inner loop can be expressed as:

$$J_{in}(\theta_i) = L^P(\theta_i) - c_1 L^V(\theta_i). \tag{27}$$

2) Outer Loop: The goal of outer loop training is to learn the meta-policy through the inner loop optimisation results of multiple tasks, so that it can be quickly adapted when facing new tasks. The specific steps are as follows.

Task Sampling: Sample multiple tasks $T_1, T_2, \ldots, T_n$ from the task distribution $\rho(T)$.

Meta-policy optimisation objective: Based on the inner-loop update results of tasks, the optimization objective of the meta-policy can be expressed as:

$$J_{meta}(\theta) = \mathbb{E}_{T_i \sim \rho(T)} \left[J_{in}(\theta'_i) \right]. \tag{28}$$

Meta-policy gradient update: To optimize the above objective, it is necessary to update the gradient of the meta-policy to clarify the update direction. For the sake of simplifying the calculation, we use a first-order approximation for the update, avoiding the significant extra cost and difficulty associated with

the need to compute second-order derivatives.

$$g_{meta} = \frac{1}{n}\sum_{i=1}^{n}\frac{\theta' - \theta}{\alpha m}, \tag{29}$$

where the inner loop learning rate α determines the magnitude of the parameter update at each step, and the gradient step m determines the depth of optimisation of the model for each task.

We update the meta-policy parameters by gradient ascent to maximise the meta-policy objective: $\theta = \theta + \beta g_{meta}$.

V. Experimental Design and Analysis

A. Experimental Environment Simulation

The server used in this experiment is equipped with NVIDIA A100-SXM4-40 GB GPU, and the experimental operation is carried out in a terminal environment based on Linux system. This experiment focuses on the cellular network environment. In this context, the CPU clock speed of the UE is assumed to be 1 GHz. For the MEC layer, it is assumed that each task is served by its corresponding VM. Each VM has 4 cores, and the CPU clock speed of each core is 2.5 GHz. Since tasks are able to operate parallelly on all cores, the CPU clock speed of a VM can reach up to 2.5 GHz $\times$ 4.

For the dataset required for the experiments, we learnt the logic of the DAG generator in [29] and wrote a python version of the generator to simulate a real-world application, so that the generation of the dataset and the training of the algorithm can be carried out in a unified environment without the need to configure new compilation environments. The structure of the DAG is primarily determined by four factors: n, which represents the number of subtasks that the task has number, determining the size of the graph scale; fat determines the width of the graph. The smaller the fat, the fewer the number of nodes in each layer; the larger the fat, the more tasks need to be processed simultaneously; density affects the denseness of the edges in the graph. When density is small, the number of task dependency paths is small and the relationships are relatively simple. When it increases, it is likely to form a complex dependency network; ccr, which reflects the ratio between the communication overheads and the computational overheads in the execution of the task. In all the experimental scenarios, given [27], we allocated data sizes between 5 MB and 50 MB for each task, while the CPU cycles required to perform these tasks ranged from 10^7 to 10^8 cycles. Also, considering that real-world applications are generally computationally intensive, the value of ccr was randomly set to 0.3, 0.4, or 0.5 during the dag generation process.

B. Algorithm Hyperarameter Setting

In this paper, MRLATO is implemented under the framework of TensorFlow, the core of which employs a two-layer dynamic LSTM network as the encoder and decoder of the Seq2seq neural network, with 128 implicit units in each layer, and incorporates a layer normalisation technique in order to enhance the performance. Regarding the training hyperparameter configuration of MRLATO, it is shown in Table I.

TABLE I
The Training Hyperparameters

Hyperparameter	Value
Learning Rate α and β	5×10^{-4}
Bias-variance adjustment factor λ	0.95
Reward discount rate γ	0.99
Clipping coefficient ϵ	0.2
Value function balancing factor c_1	0.5
Gradient update step	3
The number of attention heads	8
Training optimizer	Adam
Activation function	Tanh
Number of iterations	2000

C. Experimental Design

In order to verify the performance of MRLATO comprehensively and from multiple perspectives, we carefully designed and conducted three comparison experiments.

Experiment 1: In the first experiment, we focus on the adaptability of MRLATO to dags with different topologies. Taking into account the relevant experimental design ideas in the literature [30] and [31], we determined that each DAG consists of 25 vertices. For the fat and density of the graph, we used a two-by-two combination of values between 0.4 and 0.8 to determine this. In this way, we were able to generate 25 topologically diverse DAG sets, each containing 100 DAGs. 22 of these sets were randomly selected for training to generate a meta-policy. The meta-policy is then used to evaluate the remaining three sets to test the performance of the algorithm in the face of new topologies. The training process is performed for 2000 iterations and the meta-batch size is set to 10.

Experiment 2: In the second experiment, we explored the performance of the algorithm in the face of mobile applications with different numbers of subtasks, specifically by varying the number of vertices in the DAG graph. In order to cover application scenarios with different number of tasks, we set the number of DAG vertices in the training set to be 10, 15, 25, 35, 45, and 50, while the test set uses a collection of DAGs with vertices of 20, 30, and 40. For the fat and density of the DAGs, we randomly chose values between 0.4 and 0.8 during the generation process. Each collection still contains 100 DAG graphs. Unlike Experiment 1, the meta-batch size for this experiment was set to 5.

Experiment 3: The core purpose of the third experiment is to examine the performance of the algorithm in the face of different network transmission rates, here specifically between the UE and the MEC layer. Similar to Experiment 1, this experiment also generates a DAG ensemble of 25 vertices, but instead of limiting the choice of fat and density to a two-by-two combination, these parameters are chosen randomly. For training, we use 24 ensembles for training, during which the transmission rate is randomly chosen between 4 and 22 Mbps with a step size of 3 Mbps.In the testing phase, three fixed values of transmission

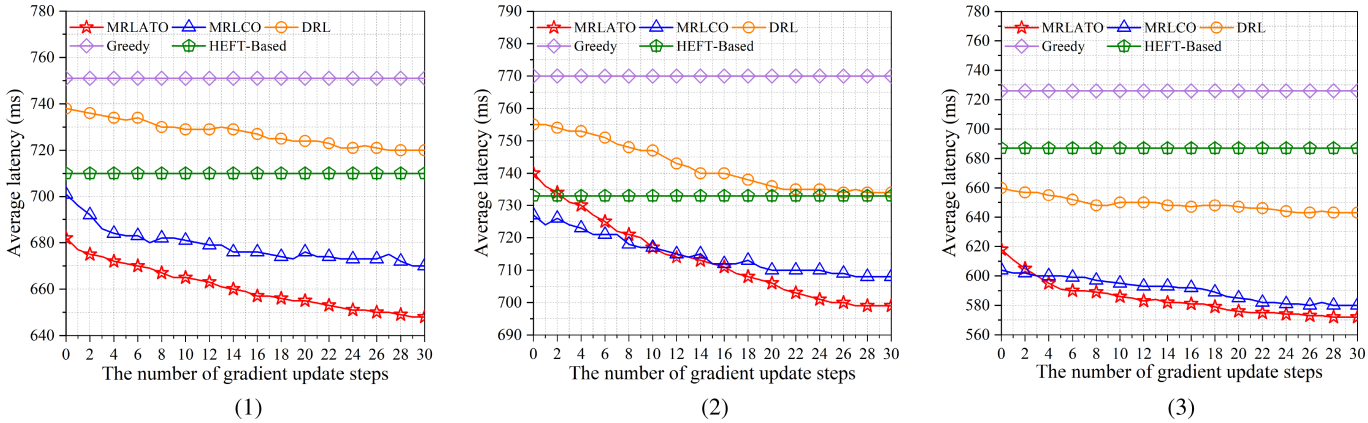

Fig. 4. Performance evaluation results under different topologies. (1) Topology1: $fat = 0.7, density = 0.4$. (2) Topology2: $fat = 0.7, density = 0.7$. (3) Topology3: $fat = 0.8, density = 0.6$.

TABLE II
THE COMPARISON ALGORITHMS

Algorithms	Main Idea
Greedy	Tasks are greedily allocated to the most appropriate computational resources with a view to obtaining the fastest completion time at the current moment.
HEFT-Based [26]	Tasks are first weighted and prioritised, and task scheduling is based on the estimation of the earliest completion time.
DRL [18]	Based on the DRL offloading policy in the literature, pre-train on the training set and further update and optimize on the test set.
MRLCO [30]	Generates meta-policies from the training set and uses them to guide new polic on the test set.

rates of 5.5 Mbps, 8.5 Mbps, and 13.5 Mbps are selected for evaluation.

In order to better evaluate the performance of the proposed algorithm, four comparison algorithms were selected for experimental validation, as shown in Table II.

D. Analysis of Experimental Results

Fig. 4 illustrates the results of Experiment 1, from which it can be clearly seen that the MRLATO algorithm proposed in this paper exhibits significant advantages in terms of delay optimisation and adaptation. In Topology 1, the initial delay of MRLATO is 682 ms, which is lower than the 738 ms of DRL and 701 ms of MRLCO. After 30 gradient updates, the delay of MRLATO drops to 648 ms, a decrease of 34 ms compared to the initial delay, demonstrating the strong optimization capability; During the same period, DRL decreased by 18 ms and MRLCO decreased by 31 ms. In Topologies2 and 3, although the initial latency of MRLATO is not the lowest, it outperforms the other algorithms and reaches the leading level after only a few iterative updates. MRLCO, due to the fact that it also incorporates the idea of meta-learning, demonstrates a better performance than DRL when facing new tasks. The traditional heuristic algorithms Greedy and HEFT-Based, on the other hand, are difficult to flexibly cope with dynamic changes because they are based on predefined strategies. Moreover, no matter how the structure of the topology changes, MRLATO can always quickly optimise the average delay and approach the optimal value within fewer gradient update steps. This performance not only shows the high training efficiency and clear optimisation direction of MRLATO, but also demonstrates its strong adaptive adjustment ability in dynamic environments.

Fig. 5 shows the performance of each algorithm under different numbers of tasks. With a small number of tasks, MRLATO has the lowest latency both at the initial stage and after some updates. And as the number of tasks increases, although it slightly underperforms the HEFT-Based algorithm initially, it again takes the lead after only 6 gradient updates, and eventually reaches the lowest value by a significant margin. Meanwhile, MRLCO's latency steadily decreases during the update process, but is not as optimised or as good as MRLATO. DRL may not perform as well as HEFT-Based when faced with a new task, and the latency convergence appears to be slower in comparison, while Greedy still exhibits high initial latency and no optimisation capability.

Fig. 6 shows the performance of the algorithms when faced with an environment of changing transmission rates. In the link rate changing environment, MRLATO still performs well with the lowest initial latency and quickly optimises to a better performance after 30 updates, demonstrating its good dynamic adaptation capability. In contrast, MRLCO also shows good adaptation, but its final delay is slightly higher than that of MRLATO. DRL's delay optimisation is relatively weak in the face of link rate variations, while Greedy and HEFT-Based continue to have high levels of delay.

To further evaluate the convergence speed and adaptability of the different algorithms more comprehensively, we record the delay performance of DRL, MRLCO, and MRLATO at the 1st, 30th, and 100th gradient updates under nine different experimental settings, and the results are shown in Table III. To quantify the convergence efficiency of the algorithms, we

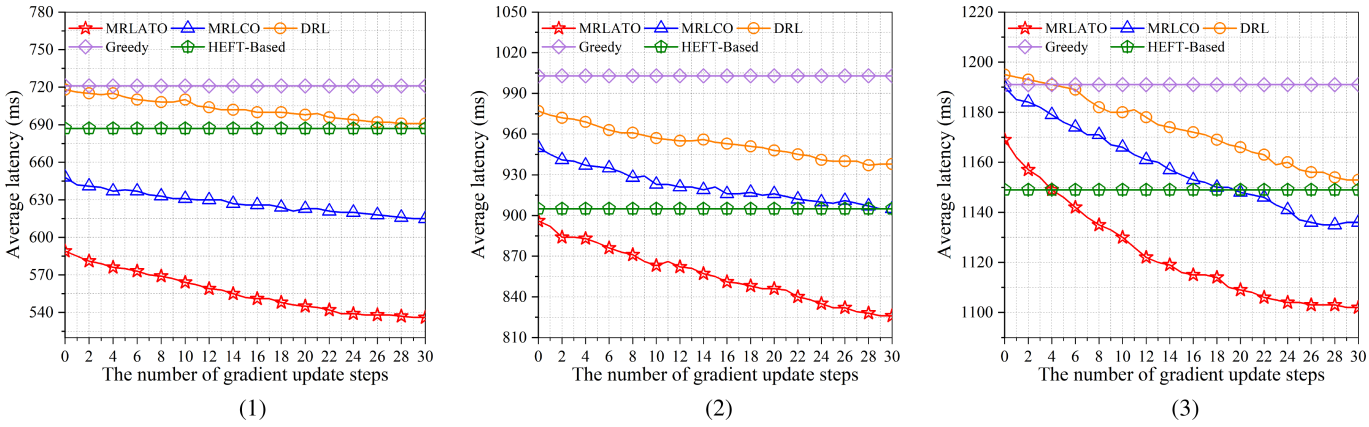

Fig. 5. Performance evaluation results with varying task numbers. (1) $n = 20$. (2) $n = 30$. (3) $n = 40$.

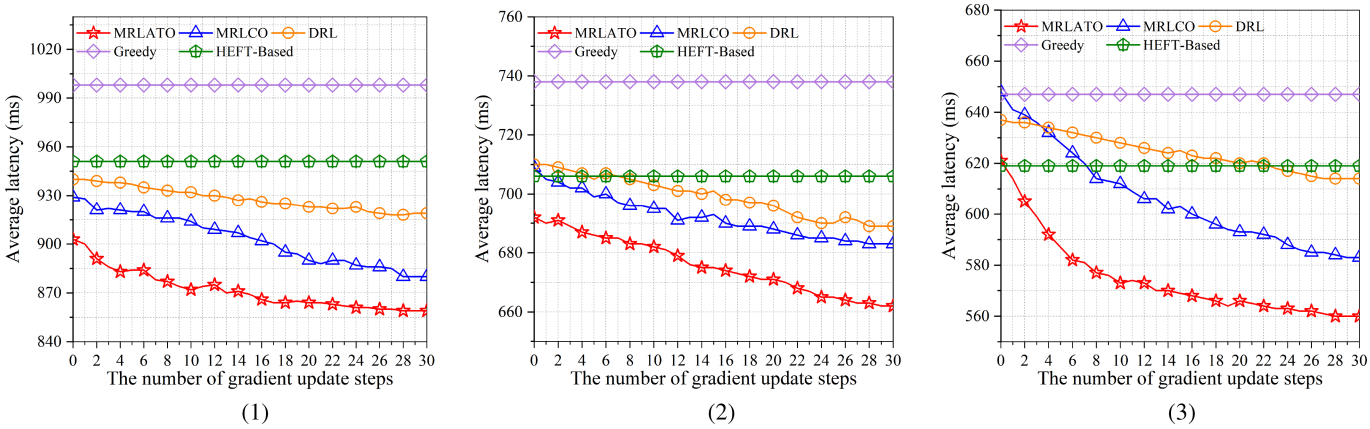

Fig. 6. Performance evaluation results under different transmission rates. (1) $U_r = D_r = 5.5Mbps$. (2) $U_r = D_r = 8.5Mbps$. (3) $U_r = D_r = 11.5Mbps$.

TABLE III
LATENCY OPTIMIZATION COMPARISON UNDER DIFFERENT EXPERIMENTAL SETUPS

Experiment Setup	DRL			MRLCO			MRLATO		
	1 steps	30 steps	100 steps	1 steps	30 steps	100 steps	1 steps	30 steps	100 steps
Topology1	738	720	712	701	670	655	682	648	**638**
Topology2	755	734	719	727	708	700	740	699	**686**
Topology3	660	643	630	604	580	567	618	572	**558**
$n = 20$	718	691	674	648	615	604	589	536	**520**
$n = 30$	977	938	921	950	905	886	896	826	**790**
$n = 40$	1195	1153	1130	1190	1136	1099	1169	1102	**1079**
$U_r = D_r = 5.5$ Mbps	940	919	906	929	880	856	903	859	**832**
$U_r = D_r = 8.5$ Mbps	710	689	680	709	683	667	692	662	**650**
$U_r = D_r = 11.5$ Mbps	637	614	603	648	583	559	621	560	**544**

compute the ratio of the optimisation magnitude of each algorithm for the first 30 gradient updates to the overall optimisation magnitude (1st to 100th update), called the optimisation efficiency ratio. The ratio is calculated to be 0.73 for MRLATO (Taking Topology1 as an example, the calculation process is $(682 - 648)/(682 - 638) \approx 0.77$. Similarly, the results under other experimental settings are 0.76, 0.77, 0.77, 0.66, 0.74, 0.62, 0.71, 0.79, and the average value is taken to 0.73), which is higher than MRLCO's 0.68 and DRL's 0.64. This result clearly demonstrates that MRLATO is able to optimise the task latency much faster in the initial updates, and its convergence speed is significantly better than the other algorithms. In contrast, MRLCO's initial performance improvement is still weaker than that of MRLATO despite its improved adaptability through meta-learning. DRL has the lowest optimisation efficiency, which suggests that it optimises slowly in the initial iterations and is not sufficiently adaptable to new task environments.

VI. Conclusion

This paper presents an innovative task offloading mechanism, MRLATO, to address the challenges of task allocation in dynamic MEC environments. By integrating MRL techniques and designing the Seq2Seq model, the proposed framework effectively captures inter-task dependencies and quickly adapts to environmental changes. Its main innovations include introducing a multi-head attention mechanism to improve the representation accuracy of task dependencies, designing a recursive task priority ranking algorithm to optimise task scheduling, and adopting a two-tier training mechanism to efficiently generate offloading decision strategies. Experimental results validate the superiority of MRLATO, showing that it can reduce task latency, accelerate convergence speed, and enhance adaptability in different scenarios.

The current research findings have laid the foundation for future research. The successful application of multi-head attention mechanism provides experience for further understanding and processing complex task relationships, which is conducive to introducing more comprehensive performance indicators in the future, thereby further enhancing the multi-dimensional optimization ability of the model. For example, precise monitoring and dynamic allocation of network resources to improve resource utilization; Develop energy-saving strategies to reduce system energy consumption; Strengthen access control by utilizing technologies such as anonymization to ensure data security. We will also actively promote the extension of the framework to heterogeneous MEC systems to support more diverse and extensive application scenarios. These improvements will further enhance the robustness and practicality of MRLATO in real-world applications, and drive its application potential in the field of edge computing.

References

[1] Ericsson, "Ericsson mobility report (June 2024)," 2024. [Online]. Available: https://www.ericsson.com/zh-cn/about-us/company-facts/ericsson-worldwide/china/mobility-report

[2] Q. V. Pham et al., "A survey of multi-access edge computing in 5G and beyond: Fundamentals, technology integration, and state-of-the-art," *IEEE Access*, vol. 8, pp. 116974–117017, 2020.

[3] H. Hu, W. Song, Q. Wang, R. Q. Hu, and H. Zhu, "Energy efficiency and delay tradeoff in an MEC-enabled mobile IoT network," *IEEE Internet of Things J.*, vol. 9, no. 17, pp. 15942–15956, Sep. 2022.

[4] W. Li, F. Wang, Y. Pan, L. Zhang, and J. Liu, "Computing cost optimization for multi-BS in MEC by offloading," *Mobile Netw. Appl.*, vol. 27, pp. 236–248, 2022.

[5] J. Mei, Z. Tong, K. Li, L. Zhang, and K. Li, "Energy-efficient heuristic computation offloading with delay constraints in mobile edge computing," *IEEE Trans. Serv. Comput.*, vol. 16, no. 6, pp. 4404–4417, Nov./Dec. 2023.

[6] K. Li, "Heuristic computation offloading algorithms for mobile users in fog computing," *ACM Trans. Embedded Comput. Syst.*, vol. 20, no. 2, pp. 1–28, 2021.

[7] J. Bi, H. Yuan, S. Duanmu, M. Zhou, and A. Abusorrah, "Energy-optimized partial computation offloading in mobile-edge computing with genetic simulated-annealing-based particle swarm optimization," *IEEE Internet of Things J.*, vol. 8, no. 5, pp. 3774–3785, Mar. 2021.

[8] A. Goli, A. Ala, and S. Mirjalili, "A robust possibilistic programming framework for designing an organ transplant supply chain under uncertainty," *Ann. Operations Res.*, vol. 328, no. 1, pp. 493–530, 2023.

[9] A. Goli, "Efficient optimization of robust project scheduling for industry 4.0: A hybrid approach based on machine learning and meta-heuristic algorithms," *Int. J. Prod. Econ.*, vol. 278, 2024, Art. no. 109427.

[10] A. Goli, A. Ala, and M. Hajiaghaei-Keshteli, "Efficient multi-objective meta-heuristic algorithms for energy-aware non-permutation flow-shop scheduling problem," *Expert Syst. Appl.*, vol. 213, 2023, Art. no. 119077.

[11] A. Goli and E. B. Tirkolaee, "Designing a portfolio-based closed-loop supply chain network for dairy products with a financial approach: Accelerated benders decomposition algorithm," *Comput. Operations Res.*, vol. 155, 2023, Art. no. 106244.

[12] A. Goli, "Integration of blockchain-enabled closed-loop supply chain and robust product portfolio design," *Comput. Ind. Eng.*, vol. 179, 2023, Art. no. 109211.

[13] X. Xu et al., "A computation offloading method over big data for IoT-enabled cloud-edge computing," *Future Gener. Comput. Syst.*, vol. 95, pp. 522–533, 2019.

[14] T. Fang, F. Yuan, L. Ao, and J. Chen, "Joint task offloading, D2D pairing, and resource allocation in device-enhanced MEC: A potential game approach," *IEEE Internet of Things J.*, vol. 9, no. 5, pp. 3226–3237, Mar. 2022.

[15] M. Goudarzi, H. Wu, M. Palaniswami, and R. Buyya, "An application placement technique for concurrent IoT applications in edge and fog computing environments," *IEEE Trans. Mobile Comput.*, vol. 20, no. 4, pp. 1298–1311, Apr. 2021.

[16] X. Li, L. Zhao, K. Yu, M. Aloqaily, and Y. Jararweh, "A cooperative resource allocation model for IoT applications in mobile edge computing," *Comput. Commun.*, vol. 173, pp. 183–191, 2021.

[17] S. Xia, Z. Yao, Y. Li, and S. Mao, "Online distributed offloading and computing resource management with energy harvesting for heterogeneous MEC-enabled IoT," *IEEE Trans. Wireless Commun.*, vol. 20, no. 10, pp. 6743–6757, Oct. 2021.

[18] J. Wang, J. Hu, G. Min, W. Zhan, Q. Ni, and N. Georgalas, "Computation offloading in multi-access edge computing using a deep sequential model based on reinforcement learning," *IEEE Commun. Mag.*, vol. 57, no. 5, pp. 64–69, May 2019.

[19] W. Yang, Z. Liu, X. Liu, and Y. Ma, "Deep reinforcement learning-based low-latency task offloading for mobile-edge computing networks," *Appl. Soft Comput.*, vol. 166, 2024, Art. no. 112164.

[20] 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. 54074–54084, 2020.

[21] H. Hao, C. Xu, W. Zhang, S. Yang, and G. M. Muntean, "Joint task offloading, resource allocation, and trajectory design for multi-UAV cooperative edge computing with task priority," *IEEE Trans. Mobile Comput.*, vol. 23, no. 9, pp. 8649–8663, Sep. 2024.

[22] S. Zarandi and H. Tabassum, "Federated double deep Q-learning for joint delay and energy minimization in IoT networks," in *Proc. IEEE Int. Conf. Commun. Workshops (ICC Workshops)*, 2021, pp. 1–6.

[23] K. Zheng, G. Jiang, X. Liu, K. Chi, X. Yao, and J. Liu, "DRL-based offloading for computation delay minimization in wireless-powered multi-access edge computing," *IEEE Trans. Commun.*, vol. 71, no. 3, pp. 1755–1770, Mar. 2023.

[24] A. Shakarami, M. Ghobaei-Arani, and A. Shahidinejad, "A survey on the computation offloading approaches in mobile edge computing: A machine learning-based perspective," *Comput. Netw.*, vol. 182, 2020, Art. no. 107496.

[25] C. Finn, P. Abbeel, and S. Levine, "Model-agnostic meta-learning for fast adaptation of deep networks," in *Proc. 34th Int. Conf. Mach. Learn.,* - 2017, pp. 1126–1135.

[26] X. Lin, Y. Wang, Q. Xie, and M. Pedram, "Task scheduling with dynamic voltage and frequency scaling for energy minimization in the mobile cloud computing environment," *IEEE Trans. Serv. Comput.*, vol. 8, no. 2, pp. 175–186, Mar./Apr. 2015.

[27] T. Q. Dinh, J. Tang, Q. D. La, and T. Q. S. Quek, "Offloading in mobile edge computing: Task allocation and computational frequency scaling," *IEEE Trans. Commun.*, vol. 65, no. 8, pp. 3571–3584, Aug. 2017.

[28] S. E. Mahmoodi, R. N. Uma, and K. P. Subbalakshmi, "Optimal joint scheduling and cloud offloading for mobile applications," *IEEE Trans. Cloud Comput.*, vol. 7, no. 2, pp. 301–313, Apr.–Jun. 2019.

[29] F. Suter and S. Hunold, *Daggen: A Synthetic Task Graph Generator*. 2013. [Online]. Available: https://github.com/frs69wq/daggen

[30] J. Wang, J. Hu, G. Min, A. Y. Zomaya, and N. Georgalas, "Fast adaptive task offloading in edge computing based on meta reinforcement learning," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 1, pp. 242–253, Jan. 2021.

[31] Y. Li, J. Li, Z. Lv, H. Li, Y. Wang, and Z. Xu, "GASTO: A fast adaptive graph learning framework for edge computing empowered task offloading," *IEEE Trans. Netw. Service Manage.*, vol. 20, no. 2, pp. 932–944, Jun. 2023.