Task Offloading in IOT Edge Computing Using Deep Reinforcement Learning

Ermias Melku Tadesse * and Nuru Endris

Posted Date: 23 October 2025

doi: 10.20944/preprints202510.1811.v1

Keywords: task offloading; edge computing; reinforcement learning; internet of things

 doi:10.20944/preprints202510.1811.v1

Article

Task Offloading in IOT Edge Computing Using Deep Reinforcement Learning

Ermias Melku Tadesse * and Nuru Endris

Information Technology Department, Kombolcha Institute of Technology, Wollo University, Ethiopia

* Correspondence: ermiasmelku3400@gmail.com

Abstract

As the IoT continues to grow, the need for efficient and effective task processing at the net- work's edge becomes crucial. This thesis delves into leveraging DRL to enhance real- time task offloading in IoT edge computing aims to optimize resource utilization and minimize latency. A novel approach is presented, combining BLSTM networks for predicting load and the A2C algorithm for making dynamic offloading decisions. This frame- work anticipates the load on MEC servers and strategically offloads tasks to balance computational demands. The research highlights significant contributions, including the implementation of BLSTM for precise load prediction by understanding temporal task request patterns and the use of the A2C algorithm to dynamically optimize offloading decisions based on these predictions and the current system state. Comprehensive experiments show that the proposed model surpasses traditional strategies, such as Deep Q- Networks (DQN), in maximizing rewards and ensuring system stability. These findings underscore the potential of DR-based methods to significantly enhance IoT edge computing efficiency by achieving balanced and responsive task offloading, thus promoting the development of more intelligent and resilient IoT systems.

Keywords: task offloading; edge computing; reinforcement learning; internet of things

1. Introduction

The Internet of Things (IoT) has recently emerged as a significant development in IT, be- coming an essential part of our daily lives. IoT device's function both as data producers and consumers, integrating seamlessly into current IT industries[1]. The number of interconnected IoT devices is expected to rise to 75.44 billion. However, these devices are limited by their battery, storage, and processing capabilities, leading to tasks being transferred to the cloud to enhance the quality of service (QoS)[2].

Fog/edge computing addresses these challenges by bringing storage and computing resources closer to end users, enabling resource-intensive tasks to be offloaded to nearby servers and enhancing IoT application efficiency. Edge computing leverages servers situated close to users[3]. With advancements in 5G, smart devices, and cloud computing, high-performance computing has progressed, and the data generated by IoT devices has exploded. emerging IoT applications are pushing the boundaries of data communication and processing requirements. These include Virtual Reality VR, AR, intelligent transportation systems, smart homes, and smart health systems. These applications demand extremely low latency in both data transmission and computational requirement that traditional cloud computing models struggle to meet. The inherent delays in cloud-based systems make them inadequate for these time-sensitive IoT applications [4].

Mobile Edge Computing (MEC) brings user and edge servers closer, as illustrated in Figure 1. By transferring computation-intensive tasks to computation servers deployed at the user's end, MEC reduces the backhaul traffic to distant cloud data centers, thus reducing the time required for time-sensitive applications and conserving energy. The offloading process involves making crucial decisions on which tasks to offload and determining the optimal location for the offloading process,

 doi:10.20944/preprints202510.1811.v1

directly influencing system QoS and performance. Scholars have suggested various innovative methods to im- prove offloading, including MDP-based, RL-based, and prediction-based approaches.

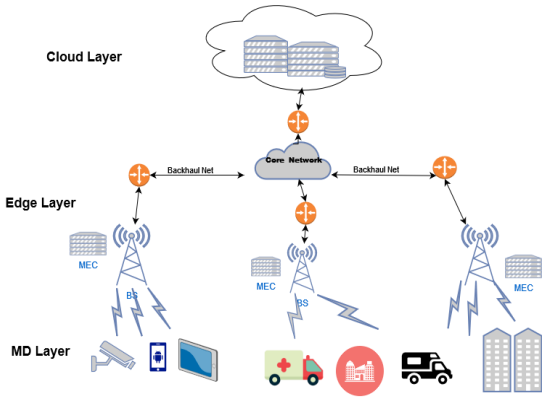


Figure 1. MEC architecture[5].

MEC offers computing services at the network edge, enabling real-time data analytics while preserving privacy, enhancing network capabilities, and alleviating congestion in backbone networks[6]. When IoT-generated data is sent to the cloud/fog/edge for processing, known as task offloading, conventional cloud computing faces challenges due to the distance between the Cloud Server (CS) and users, leading to significant load, congestion, and elevated transmission latency[2]. The deployment of many edge servers allows for collaboration and optimization of computing resources, enhancing task processing efficiency.

Edge computing, which provides computing capabilities at the network's periphery, delivers improved response times and reduced latencies. Fog computing, on the other hand, furnishes computing, networking, and storage services to IoT devices using net- work devices within the Local Area Network (LAN) of the IoT devices. In contrast, edge computing uses small data centers near the IoT devices, facilitated by Wi-Fi access points, to minimize task completion times[7].

Task completion time, a crucial QoS metric, is measured from request submission to result reception and is influenced by waiting, transmission, and computation delays. For mobile devices with limited battery capacity, energy consumption is another critical factor, primarily affected by data processing and transmission. These elements are key considerations in assessing performance and efficiency, especially given the power constraints of portable devices. Effective computation offloading remains a challenge due to its NP-hard nature[2].

Reinforcement learning (RL) involves acquiring the optimal policy through exploratory and exploitative actions to maximize cumulative rewards. This sequential decision- making process is influenced by state changes and comprises key components like environment, agent, state, action, and reward[8]. However, RL algorithms often over- look the dynamic and foreseeable nature of task offloading, potentially leading to sub optimal decisions. Conventional approaches focus on static optimization, while deep reinforcement learning methods address dynamic IoT task computing but may neglect task inference testing, leading to notable response delays[9].

Decision-making in task offloading, determining the task whether offload to the remote server or compute locally, is crucial and merits thorough examination[2]. The decision typically involves addressing a multi-objective optimization problem. Common solutions include transforming multiple objectives into one objective or using one primary objective with others as constraints[2].

BLSTM, an advanced recurrent neural network (RNN), is capable of handling the vanishing gradient problem faced by RNNs and has been effective in forecasting user mobility[10]. LSTM shows promise for trajectory prediction[11]. In our approach, we integrate BLSTM into the A2C algorithm, creating the BLSTM-A2C algorithm, which enhances monitoring of user mobility and system utility. Policy-based RL directly models the policy guiding the agent's behavior, with better learning convergence than value- based RL. However, policy-based RL faces slow learning convergence due

 doi:10.20944/preprints202510.1811.v1

to difficulty in determining optimal actions. To address this The A2C method proposes to addresses the strengths and weaknesses of both value-based and policy-based RL[12].

In this research, the main concern is to balance low latency and low energy consumption while decrease task drop rate and maximizing reward in real-time task offloading with mobility. We will consider an intelligent hybrid model combining BLSTM and A2C for dynamic real-time task offloading in edge computing systems serving IoT devices. This hybrid technique will consider computational constraints of IoT endpoints, applying quantization techniques to reduce computational requirements during training and inference, and adapting to changing network and computational states in real- time to optimize latency, energy consumption, and computational cost, while handling device mobility.

2. Methodology

This research utilizes Google cluster datasets to evaluate a BLSTM architecture for IoT task offloading and resource management datasets proposed in[13]. The dataset, released in 2011, contains 29 days of comprehensive cloud environment data, including information from 10,388 machines, over 670,000 jobs, and 20 million tasks, with details on task arrival times, data sizes, processing times, and deadlines. The data is organized across six separate tables, and to improve data quality and usability, the researchers performed preprocessing steps, notably converting time measurements from microseconds to seconds, while addressing common challenges like noise, inconsistency, and missing data in the raw dataset.

$$\text{Time(s)} = \text{Time(s)} \times 10^{6}$$

A. System Model

The proposed framework employs a Deep Reinforcement Learning (DRL) agent to optimize task offloading in a Mobile Edge Computing (MEC) environment, as shown in Figure. 2. The system consists of N mobile devices and M MEC servers connected to a macro–Base Station (BS) via fiber cables, with the DRL agent centrally managing task distribution based on historical performance patterns. The agent aims to minimize system cost and task drop rates while maximizing rewards by analyzing server load patterns and predicting capable nodes based on their CPU and memory resource history. The MEC servers, co-located with the macro-BS and Small Base Stations (SBSs), are responsible for collecting, processing, and delivering results for computation tasks submitted by mobile devices within specified time slots, with the goal of achieving optimal cost efficiency and reduced latency for time-sensitive data.

1) Task Model

IoT devices generate tasks across different time slots, represented as $T = (1, 2, 3, \ldots, T_n)$, with arrival times and data sizes based on real-world IoT data. Each task is initially sent to the Helper MEC with relevant task and device information. The decision model then determines whether to offload the task to the MEC server or process it locally. In a given time slot $t \in T$, a new task $\lambda_m(n)$ generated by IoT device $n \in N$ is characterized by the tuple D_{nm}, R_{nm}, $t_{nm,max}$, B, where D_{nm} is the task data size, R_{nm} is the required computational resources per bit (in CPU cycles/bit), (B) is the task's communication bandwidth, and $t_{nm,max}$ is the maximum tolerated delay to meet QoS demands. If the task isn't completed by the end of time slot $t + t_{nm,max} - 1$, it's discarded.

2) Decision model

When a new taskλ_{nm} arrives for MD $n \in N$ at the start of time slot $t \in T$, an optimal offloading decision is required. We use a binary variable $X_{nm} \in 0, 1$ to represent this decision: $X_{nm} = 0$ indicates local processing, while$X_{nm} = 1$ signifies offloading to an edge node. The decision model's role is to determine this offloading scheme optimally.

The decision model will provide the offloading scheme for this task.

 doi:10.20944/preprints202510.1811.v1

$$X_{nm} = \begin{cases} 0 = \text{ the task executed Locally} \\ 1 = \text{ the task executed in MEC} \end{cases} \qquad Eq.(1)$$

In our model, IoT devices generate tasks in real-time. Each task is considered atomic, it cannot be divided or split into smaller parts. This indivisibility is a key characteristic that influences the offloading decision process.

3) Channel /communication Model

Our system, depicted in Figure 2, features MEC nodes distributed across various regions. This infrastructure includes multiple APs, edge servers, and N MDs, indexed as n= (1,2,3, . . ., N). MDs connect to the MEC through either APs or mobile networks. The macro-BS must determine the optimal task offloading location within the network, considering factors such as edge server workload, response time, latency, and energy consumption.

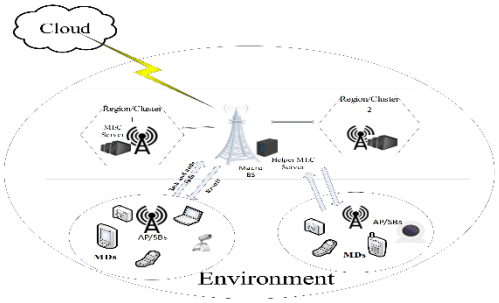


Figure 2. Propose System Model.

The utilization of resources and the introduction of transmission delays occur exclusively when the task is offloaded to the edge server for execution. The offloading of the taskλ_{nm} is denoted as X_{nm} $\{X_n1, X_n2, X_n3, \ldots\ldots, X_nM,\}$ where B_{NM} represents the corresponding bandwidth allocation, consisting of $\{B_n1, B_n2, B_n3, \ldots\ldots, B_nM,\}$ This task offloading is governed by the bandwidth constraint, signifying that X_m must be less than or equal to B_m, which is further constrained by B_{max} Utilizing Shannon's formula enables the determination of the transmission rate[14].

$$r_{nm} = B_m(t)\log 2\left(1 + \frac{P_{nm}^{tran} g(d_t)}{\partial^2}\right)(t), \forall n \in N \qquad Eq.(2)$$

In the context provided, the channel gain of between MD and the AP is represented by g and d_t representing a distance from the MD and the nearest AP., and the power of additive Gaussian white noise is denoted by σ^2. We define $d_t = |l_n - l_m|$ as the distance between MD n and MEC server m, where l_n denotes the current location of m and l_m is the location of m[15]. Consequently, the equations allow for the derivation of the task's transmission delay and transmission energy consumption.

$$T_{m,n^{tran}} = \frac{D_m(t)}{r_{n,m}} \qquad (3)$$

$$E_{nm}^{tran} = P_{nm}^{tran}.T_{nm}^{tran} \qquad (4)$$

we set, $P_{nm}^{tran} = P_{max}$where P_{max} is the maximum transmit power when the battery.

4) Computation Model

Computation time, measured time duration from request to submit result, it is a key QoS metric in offloading design. It's affected by waiting, transmission, and processing delays. Energy consumption, crucial due to limited device batteries, includes data processing and transmission costs[8].

 doi:10.20944/preprints202510.1811.v1

a) Local Execution

Processing a task locally on a device incurs both a time delay and power consumption. If we denote the processing capacity of the IoT device as $f_m^{device}(b/s)$, then the processing delay T_{nm}^L for task m on MD n can be calculated using Equation 5. These factors contribute to the overall cost of local task execution and play a crucial role in decision-making for task offloading.

$$T_{nm^{compu}}^{l} = \frac{R_{nm}}{f_m^{device}} \qquad (5)$$

Similarly, the energy consumption for task m of user n during local computation is denoted by E_{nm}^L and is defined by Equation 6. This equation captures the power consumed during the local execution of the task.

$$E_{nm}^L = T_{nm}^L P_n^L \qquad (6)$$

where E^L, P_n denotes the energy consumption and the power coefficient, which depends on the chip architecture[16] respectively when the task is processed in the MD. Therefore, the computational cost is characterized as the weighted aggregate of both the energy consumed and the time taken for the completion of a task λ_{nm} assigned to node n. The overall cost is then computed as the weighted sum of local costs in the subsequent manner as shown in Equation 7.

$$C_n^L = \sum_{n==1}^{N} (\partial T_n^L + \beta E_n^L) \qquad (7)$$

Where α and β are constant weighting parameters corresponding to the time and power consumption of the task λ_{nm} and the sum of weights is always equal to 1, $\alpha + \beta = 1$ and must fulfill $0 <= \alpha <= 1$ and $0 <= \beta <= 1$. In this scenario, diverse user requirements can be accommodated through the adjustment of weights. For instance, prioritizing ∂T_n^L over βE_n^L indicates a higher emphasis on minimizing delay rather than energy consumption, making it suitable for latency- sensitive applications.

b) Algorithm for calculating local cost

Algorithm 1, which calculates the local cost, begins by setting the initial action for each device to local computation $(X_{total} = 0)$ and the system's total cost to zero $(C_{total} = 0)$. Following this initialization phase, the algorithm takes several parameters as input. These include the size of computation (D_n), the number of CPU cycles required (R_n) the local computation power of the device (f_n^{device}) and the decision weights (α and β). These parameters are used to determine the cost of local computation for each device.

Algorithm 1: calculate the cost

Require: N, R_n, D_n, f_n^{device}, P_n^L, α, β, $t_{n,max}$

Ensure: C_{total}

1: $C_{total} \leftarrow 0$ ▷ Set total system cost

2: for n ∈ N do

3: $T_n^L \leftarrow \frac{D_n}{f_n^{device}}$ ▷ time to execute

4: $E_n^L \leftarrow T_n^L P_n^L$ ▷ energy consumption

5: $C_n^L \leftarrow (\partial T_n^L + \beta E_n^L$ ▷ Calculate cost

6: $C_{total} \leftarrow C_{total} + C_n^L$ ▷ Update the cost

 doi:10.20944/preprints202510.1811.v1

7: end for

8: return C_{total}

c) MEC Execution

When the decision unit opts to offload a task to an edge server (represented by $X_{nm} = 1$), it does so because the edge server's computational resources significantly surpass those of the MD. The processing time per task on the edge server is denoted as T_n^e, which encompasses both transmission and computing delays. The computing delay is influenced by factors such as the edge server's CPU frequency and other available resources. It's important to note that in this study, we make the assumption that the size of the computed results is minimal. Consequently, the time required to download these results from the MEC server is considered negligible. Given these conditions, the processing time can be expressed using equation 8.

$$T_{nm^{compu}}^e = \frac{R_{nm}}{f_m^E} \quad (8)$$

$$T_{nm}^e = T_{nm^{compu}}^e + T_{m,n^{tran}} \quad (9)$$

Similarly, the corresponding power cost for task λ_m of user n is denoted by $E_{nm}^e$and is defined as equation 10.

$$E_{nm}^e = T_n^e P_n^e \quad (10)$$

WhereP_n^e is the power coefficient, The total system cost in edge computing is derived from two primary factors: the time required for computation and the associated power consumption. This can be expressed as:

$$C_n^e = \sum_{n==1}^{N} (\partial T_n^e + \beta E_n^e) \quad (11)$$

The overall expense, denoted as C_{total}, to calculate a task λ_n send from device n can be formulated as...

$$C_{total} = \min\left(C_n^L, \min_{m\in\{1,\ldots,M\}} C_{nm}^e\right) \quad (12)$$

where C_n^L is considered when the $t_{nm,max}$ is not guaranteed when doing offloading.

Table 1 provides a comprehensive summary of the key notations used consistently throughout this paper.

Table 1. Summary Of Notation.

Notation	Definition
c	Represents the cost of task consumed energy and delay.
α, β	The trade-off weight between energy and delay.
	Represents the number of MD servers.

 doi:10.20944/preprints202510.1811.v1

n	
m	Represents the number of tasks.
f^e	Represents the CPU (computational resource) of the edge server.
B_n^e	Represents the communication bandwidth.
R_{nm}	Represents the required size per bit for this type of task (in CPU cycles/bit).
r_{nm}	Represents the transmission rate of the input data to the MEC m.
D_n	Represents the data size.
f_m^{device}	Represents the CPU frequency of the MD (processing capacity (bits/s)).

5) Prediction Model

Traditional RL-based task offloading systems make decisions reactively, leading to potential delays and failures. By analyzing historical patterns of task generation, server usage, and mobile device movements, our system can predict future resource needs and pre-allocate computing resources to optimal MEC nodes before tasks arrive, significantly reducing decision-making delays and task failures.

6) Task Load Prediction Model

Using a trained LSTM model, our system predicts future task data $\widetilde{D}_t$ by analyzing historical task sequences D1, D2, D3, D_{t-1} from mobile devices. The model aims to minimize prediction error $|D_t - \widetilde{D}_t| \approx 0$, enabling proactive resource allocation and service package loading before tasks arrive. and prediction of future MEC server loads using an LSTM model trained on historical CPU utilization data sequences H1, H2, H_{t-1}.

The model outputs predicted server loads $H_t^{\sim}$, identifying potentially idle servers $h_t^{\sim}$ that can be prioritized for task offloading, thus preventing workload imbalances and enabling more efficient resource allocation decisions.

B. Offloading Algorithm Design

To dynamically adjust to changing conditions and make optimal choices, we've developed an innovative decision-making solution that combines two powerful techniques: BLSTM from deep learning and A2C from reinforcement learning. This integration results in a DRL based approach for task offloading, designed to anticipate and respond effectively to environmental shifts.

1) Prediction phase (BLSTM)

During the Prediction Phase, as illustrated in Figure 3, the algorithm forecasts future conditions such as CPU availability, upcoming task loads, and the nearest edge node. Upon arrival of an actual task, the system compares the predicted values with the real data for CPU, task, and node. If the

 doi:10.20944/preprints202510.1811.v1

discrepancy falls within an acceptable range, the task is offloaded to the location determined by the DRL Agent in the Decision Phase. However, if the error exceeds the set threshold, the Decision Phase reevaluates the offloading decision using the actual values. This new information is then incorporated into the historical dataset for future training. The training process of the BLSTM Net- works involves adjusting weights and biases to improve prediction accuracy.

2) Decision Phase (A2C)

Each IoT device generates various tasks at different times, and when a new task is created, there is a delay in the system's response to the decision request for that task. To optimize performance, the system uses information from a prediction phase before it receives real task information. A DRL model called A2C is used to make offloading decisions for newly arrived tasks. The A2C network consists of two sub-networks: an actor and a critic, with shared layers to simplify the model and speed up network convergence. The network structure of the model is shown in Figure 3.

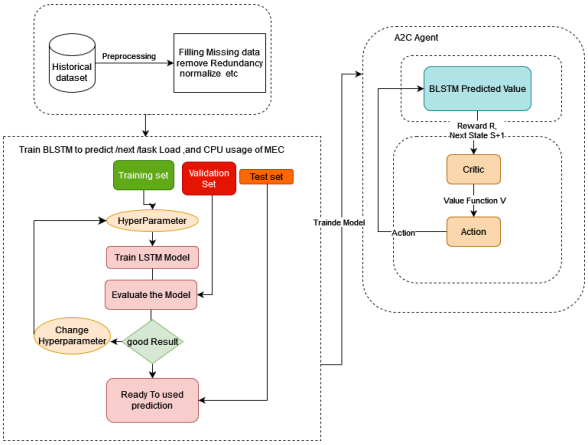
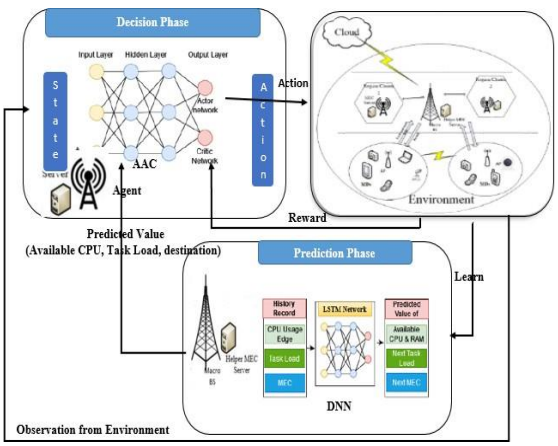


Figure 3. work flow of BLSTM Prediction.

The edge server within A2C-BLSTM-based algorithm functions as a smart brain, where a RL agent interacts with the environment to learn optimal policies. The agent consists of two components: an actor, which defines a policy and performs actions based on observed states, and a critic, which evaluates the current policy and updates its parameters based on rewards from the environment[17].

The A2C algorithm initializes an actor to interact with the environment, using BLSTM networks to predict system parameters and select actions based on policies. After executing actions, the algorithm captures states and rewards in a replay buffer for iterative learning. A critic network estimates the advantage function to evaluate the actor's decisions, and both the policy and critic networks are refined through gradient-based optimization to improve decision-making and maximize cumulative rewards within the A2C ecosystem.



 doi:10.20944/preprints202510.1811.v1

Figure 4. Decision Phase of A2C Network.

C. DRL Decision model Elements

1) Environment State S

The state describes the environment and changes when the agent generates an action. In this model, each state consists of five pieces of information: task size (in bytes), deadline (in milliseconds), processing queue of the MD, transmission queue, and MEC processing queue information (all in bytes), and communication type (5G or Wi-Fi). The state is characterized as follows:

$$S = \{D_{nm}, t_{max}, \partial^{proc}, \partial^{tran}, \partial^{MEC}, r_{nm}, H(t)\} \qquad (13)$$

where D_{nm} is the task size, $H(t)$ denotes the historical load level on each edge node, t_{max} is the task deadline, and $\partial^{proc}, \partial^{tran}, \partial^{MEC}$ represent the processing state of devices, the transmission state of devices, and the MEC node state information for time slot t - 1, respectively. Additionally, r_{nm} represents the transmission rate of the MD.

2) Action a

The action space of this model comprised as a binary

$$A = \begin{cases} \mathrm{X} = 1 \\ \lambda_m^t = N \end{cases}, X = (0,1)^{1+N} \qquad (14)$$

Where N represents the computation server, and X_n determines whether a task λ_m^t is processed at the MEC or locally. Specifically, λ_m^t is local when $X_n = 0$, and N is one of the MEC servers {1, 2, ..., M} when $X_n = 1$. The agent will make decisions based on the task offloading strategy at each time step and receive rewards from the environment in the subsequent time step. When local and MEC processing are both possible, the choice should optimize either latency or energy efficiency. If neither option is suitable, the task should be discarded.

3) Reward

After the agent executes an action, the reward serves as the environment's feed- back to the agent. This reward, a numerical value, reflects the agent's performance; at each time step (t + 1), the agent receives the reward and information about the updated state from the environment. Based on this feedback, the agent learns a policy (π), which is considered optimal (π^*) when consistently yielding the highest rewards. In this section, the costs and rewards are quantified based on local processing, remote processing, and task dropping to execute task λ(n). For every action, the system must choose between executing the task locally or offloading it to external resources.

$$C_n^L = \partial T_n^L + (1-\beta)E_n^L + \Delta T^L \qquad (15)$$

α and β are weighting coefficients that balance the importance of processing time against energy consumption. ΔT^L represents the penalty cost for exceeding the allocated time.

Equation 15 is applied to choose a minor action with lower cost value for all multi- devices capable of processing data locally and remotely. if the allocated server is wrong or busy, that task will be discarded. The cost associated with discarding the task is denoted as C_{dr}, with a fixed value of 1. If the processing time surpasses the deadline, the algorithm imposes ΔT^L . In time-sensitive systems, tasks are evaluated against predetermined deadlines. Those exceeding these limits may be discarded to maintain overall system efficiency. While minor delays are of- ten tolerable, the acceptable threshold varies based on task type and criticality. A penalty system quantifies the cost of delays, helping to prioritize timely task completion and optimize resource allocation. This approach balances the need for thorough processing with the demands of real-time responsiveness across diverse application scenarios. The penalty for breaching the deadline is defined as:

$$\Delta T_t^L = \begin{cases} 0 & \text{if } T_t^L - t \le t_{max} \\ (T_t^L - t_{max})^2 & if \ \ T_t^L - t > t_{max} \end{cases} \qquad (16)$$

 doi:10.20944/preprints202510.1811.v1

When a task meets its deadline, ΔT^L becomes 0, having no impact on the cost. However, for tasks exceeding the deadline, a penalty is applied using a squared function, which amplifies the effect of larger delays. To address the negligible impact of delays less than 1 second due to the squaring effect, time calculations in this range are processed in milliseconds. For instance, a 0.1second delay translates to a ΔT_t^L value of 10,000ms rather than 0.01. This approach ensures that even small delays contribute meaningfully to the penalty calculation. For offloading tasks λ_{nm}, the costs C^e and ΔT_t^L are calculated using Equations 17 and 18, respectively. These equations account for the specific characteristics of edge processing and its associated penalties.

$$C_n^e = \partial T_n^e + (1-\beta)E_n^e + \Delta T^e \qquad (17)$$

$$\Delta T_t^e = \begin{cases} 0 & \text{if } T_t^e - t \le t_{max} \\ (T_t^e - t_{max})^2 & if \ \ T_t^e - t > t_{max} \end{cases} \qquad (18)$$

At each time slot t, the agent is given an immediate reward r_t for choosing action A. Typically, this reward function is inversely related to the cost function. The primary objective of the optimization problem is to minimize overall costs.

$$reward_t = \begin{cases} -C_L & \text{if at} = 0 \\ -C_{ofL} & \text{if at} = 1 \end{cases} \qquad (19)$$

The objective of this research is to simultaneously reduce costs and increase re- wards, optimizing the system's overall performance and efficiency.

D. Prediction Based Decision Task offloading framework

Our algorithm combines A2C (Actor-Critic) with BLSTM (Bidirectional Long Short-Term Memory) to optimize task offloading in MEC environments. The system makes offloading decisions based on both predicted and actual server loads, choosing between edge processing and local execution. With probability ϵ, the agent selects minimum- cost actions; otherwise, it follows predicted or mathematical solutions with probability 1 − ϵ. The optimal policy π^*minimizes the expected long-term cost: $\pi^* = \arg\min\pi \in [\sum_t r^{t-1} C(t)|\pi]$, where the total reward is calculated as $R_t = \sum_{t=1}^{T} \gamma^t r_t(s_t, a_t)$. The model continuously updates through policy gradients and advantage estimates, improving its decision-making capabilities over time.

Algorithm 2: Proposed A2C-BLSTM-Based Task Offloading Decision Algorithm

1: Input: Task T_i in each time slot

2: Output: Optimal offloading decision and total cost

3: Initialize:

4: Actor-Critic model and related parameters (γ)

5: LSTM model for task prediction

6: Error threshold ϵ

7: Maximum tolerance time ψ

8: for episode e = 1 to M do

9: Initialize sequence s sequence

10: for time slot t = 1 to T do

11: Predict Load BLSTM ES_m

12: Predict Task $(Load_n)$

 doi:10.20944/preprints202510.1811.v1

13: Wait for Real task (n)

14: With probability $1-\epsilon$, select mathematical A or predicted A

15: δ =real task load λ_n ▷ real task load

16: if $\delta > \epsilon$ then

17: for each m = 1 to M do

18: $load$[m]= $T_{nm}^{comp} + T_{nm}^{tran}$

19: end for

20: $\text{Min}_m = \min(Load)$

21: if $\text{Min}_m \leq \psi$ then

22: $a_t = m$ ▷ Select the edge server with minimum load

23: else

24: $a_t = MD_n$ ▷ Select to process the task locally

25: end if

26: else

27: Select $ES_m \rightarrow$ Predic Action ($Task_n$, ES_m)

28: if Predic Action $\leq \psi$ then

29: a_t = selected ES_m ▷ Select the edge server to processed the task

30: else

31: $a_t = MD_n$ ▷ Select to process the task locally

32: end if

33: end if

34: Execute action a_t (local or MEC)

35: Calculate the cost of the action taken:

36: if $a_t = MD_n$ then

37: Calculate local cost: C_t^L ▷ Eq. 15

38: else

39: Calculate MEC cost: C_t^e ▷ Eq. 17

40: Receive reward $r_t = -C_t$ ▷ Reward is the negative cost

41: Observe next state S_{t+1}

 doi:10.20944/preprints202510.1811.v1

42: Calculate advantages:

43: $\delta_t = r_t + \gamma V(S_{t+1}) - V(S_t)$

44: Update Critic: Minimize δ_t^2

45: Update Actor: Use policy gradient $\nabla \log \pi(a_t|s_t)\delta_t$

46: end for

47: end for

2. Experimental Result and Discussion

We use PYTHON simulation to evaluate the performance of our proposed algorithm. The A2C-BLSTM algorithm is conducted using Keras plus TensorFlow[18], where Keras is adopted to build the DNN training model and TensorFlow supplies the back- ground support. We utilize a dataset from Google Cluster[13], which contains details about task arrival times, data sizes, processing times, and deadlines of the task. Tasks vary from complex big data analysis to simpler image processing, each with unique processing requirements and data volumes. We preprocess and normalize the data based on task characteristics to ensure compatibility with our model, accounting for the interrelation between processing density, time, and data size across different task types. Based on similar research in the field[13,19,20], the authors have recommended and used specific simulation parameters to evaluate task offloading solutions. Accordingly, we have adopted and simulated their recommended parameters, as shown in Table 2

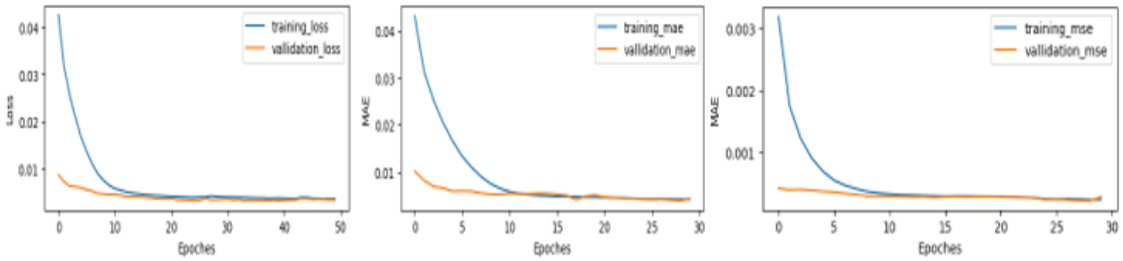
Table 2. Simulation Parameter.

Parameter	Value	Description
B_m^t	10MHz	Bandwidth for offloading devices (MHz)
f^{device}	2.5GHz	Local CPU capacity (GHz/sec)
f^{MEC}	41.8GHz	MEC CPU capacity (GHz/sec)
P_{nm}^{tran}	$10^{-27}(f$	Energy consumption per CPU cycle
δ	1	Gaussian channel noise
P_{nm}^{exe}	5w	Energy consumption per CPU cycle

 doi:10.20944/preprints202510.1811.v1

A. Training process of BLSTM & DRL

Figure 5 shows that the model that used to predict the task load and CPU usage is well fitted, the LOSS decreases for larger epoch values hence it shows how the model starts to optimize properly heavily for a larger epoch. The model learns any pattern or context and adapts the workload fluctuation. This will lead to that out validation LOSS will decrease for longer epochs.



(A) Loss (B) Mean Absolute Error (C) Mean Square Error

Figure 5. Model Loss MAE and MSE.

Prediction on Historical Data from the Google Cluster Dataset

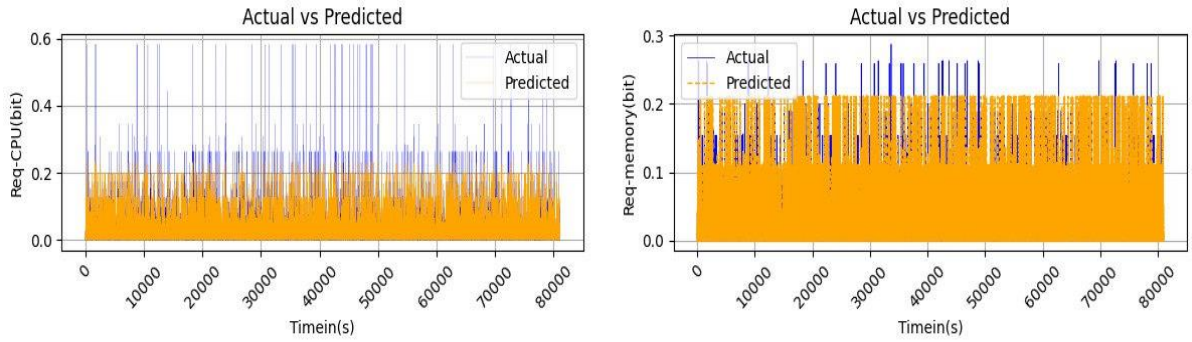
The proposed model demonstrates strong predictive accuracy when compared against actual values using a historical data dataset and is applied to authentic cloud data sets, significantly impacting accuracy and performance. Figures 6a and 6b illustrate the prediction outcomes of the BLSTM model, contrasting actual data (blue line) with predicted data (orange line). The close alignment between the two lines (actual and predicted) in these figures indicates the robust fit of the proposed model.

B. Discussion on Performance Comparison

To evaluate the performance of our A2C-BLSTM-based scheme in real-time for IoT, we compared our model with three benchmark algorithms: MADDPG[21], MAAC-based algorithm[17], and DQN-based algorithm[22]. We trained our model and the other three algorithms simultaneously in the same environment, including A2C-BLSTM with prediction and A2C without prediction.

These benchmark algorithms are commonly employed in MDP scenarios. The DQN algorithm operates as a single agent for dependent computation offloading, while MAD- DPG represents a state-of-the-art Multi-Agent Deep Reinforcement Learning (MADRL) framework. The MAAC algorithm, like our approach, utilizes a multi-agent reinforcement learning strategy based on the actor-critic method. Our comparison focused on key performance indicators: task drop rate, energy consumption, and task execution delay. To assess prediction accuracy, we employed Mean Squared Error (MSE), Mean Absolute Error (MAE), and Mean Absolute Percentage Error (MAPE). For optimization evaluation, we examined total system cost, total reward, and task throughput volume. The experimental setup involved eight edge servers and 80 terminal devices. As illustrated in Figure 6, our model consistently outperformed the benchmark methods across all metrics. This superior performance can be attributed to the model's effective integration of predictive and decision-making techniques, which optimally utilize task characteristics and edge server load information.

 doi:10.20944/preprints202510.1811.v1



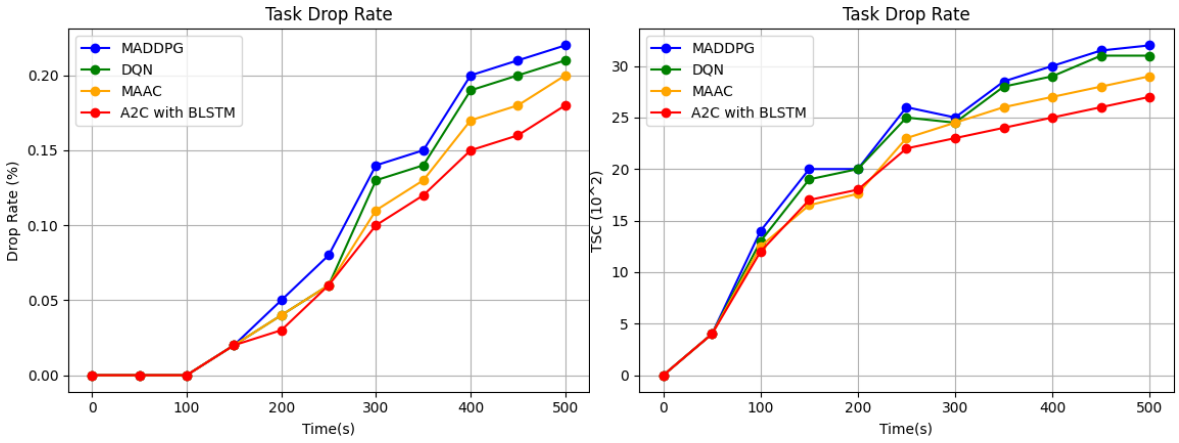
(A) CPU (B) Memory

Figure 6. prediction of Requested of memory and CPU against time in(s).

1) Impact of Task Number

Previous studies have demonstrated that the task arrival rate significantly affects the system state due to the random generation of tasks. In this paper, we utilize data from real scenarios for data streams. Unlike traditional approaches that rely on task arrival rates to evaluate the system state, our study uses different time slots to assess the impact of the number of tasks on system cost, average task delay, and task discard rate.

Specifically, we set the time slots in the dataset to T = 0, T = 100, 200, and 500 and compare the performance of MADDPG, MAAC, DQN, and our A2C-BLSTM scheme under these different time slots. The experimental results are presented in Figure 7.



(A) Task throw Rate comparison (B) Total System Cost

Figure 7. Performance evaluation under different no of Task in time slot.

Figure 7a presents a comprehensive comparison of the task rejection ratios across four different schemes. As the number of tasks increases, the rejection ratio for all schemes also rises. However, our scheme demonstrates a significantly lower increase compared to the other schemes, indicating better performance in terms of task acceptance ratio and a lower rejection rate. The proposed model's ability to accept more tasks stems from its intelligent assignment of tasks to servers that are optimally matched in terms of resource requirements and availability. This approach not only preserves resources for future use but also allows for more tasks to be accepted, reducing the overall time to complete tasks, which is a critical factor for real-time communication scenarios. A high acceptance rate is beneficial as it leads to higher resource utilization and reduces system idle time. Consequently,

 doi:10.20944/preprints202510.1811.v1

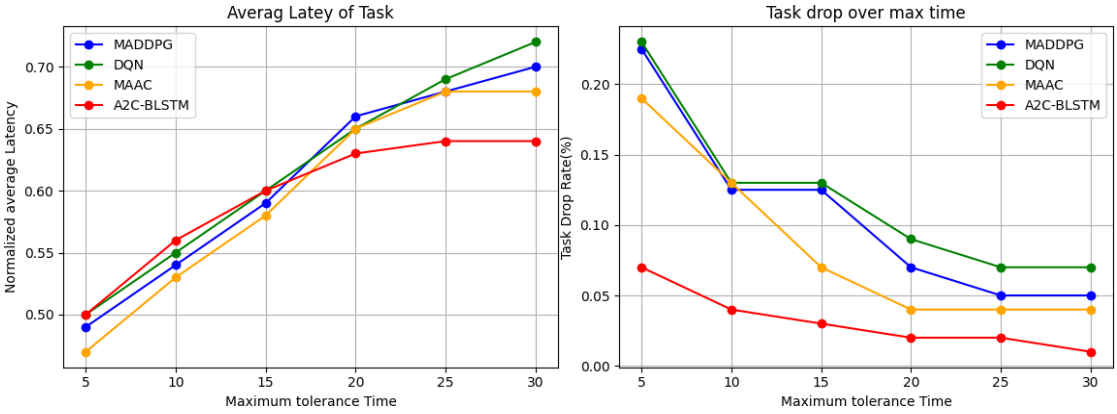
our proposed work outperforms other methods in terms of cost (latency and power consumption), demonstrating its effectiveness in improving the proposed MEC system.

The task throw rate is a crucial metric directly impacting QoS. A low task rejection ratio indicates high QoS. The proposed A2C-BLSTM-based scheme employs a robust mechanism for selecting the best servers by assessing CPU usage and task requirements, thereby enhancing the efficiency of the MEC system. Additionally, it leverages intelligent resource allocation strategies, resulting in an increased task acceptance rate while maintaining QoS. A higher acceptance rate typically leads to higher average resource utilization relative to cost. The results demonstrate that our model achieves a higher utilization rate than other algorithms, underscoring the effectiveness of the proposed approach in improving MEC system performance.

Figure 7b also presents a comprehensive comparison of the cost ratios of three different schemes as the number of tasks increases. It can be observed that the cost ratio for all schemes increases with the number of tasks. However, the proposed scheme exhibits a significantly lower increase compared to the other schemes, indicating better performance in terms of cost ratio. Additionally, it has a significantly lower task rejection rate compared to the MADDPG, MAAC and DQN algorithms, implying that it accepts more tasks for offloading and enhances the QoS of the MEC system. The key factor enabling the proposed one to achieve this is its ability to intelligently assign tasks to servers that are optimally matched in terms of resource requirements and availability, thus minimizing overall cost and maximizing resource utilization of the MEC system.

2) Impacts of Max tolerable time

Figure 8a illustrates our system's performance across varying maximum tolerance latencies and algorithm configurations. The data reveals an inverse relationship between maximum tolerance latency and task drop rate for all four methods tested. As the tolerance increases from 5 to 30 seconds, task drop rates decrease markedly. Our proposed A2C-BLSTM algorithm consistently maintains a lower task drop rate compared to its counterparts. This advantage is particularly pronounced at lower maximum tolerance latencies, highlighting our algorithm's efficacy for time-critical tasks. Conversely, Figure 8b demonstrates that average system latency increases for all methods as maximum tolerance latency extends from 5 to 30 seconds. This trend is attributed to task completion dynamics: at shorter tolerance levels, many tasks fail to complete, whereas longer tolerances allow more tasks to be processed and transmitted, resulting in higher aver- age system latencies.



(A) Average dropped Task (B) Average Latency of the task

Figure 8. Performance evaluation under different max tolerance latency (in seconds)

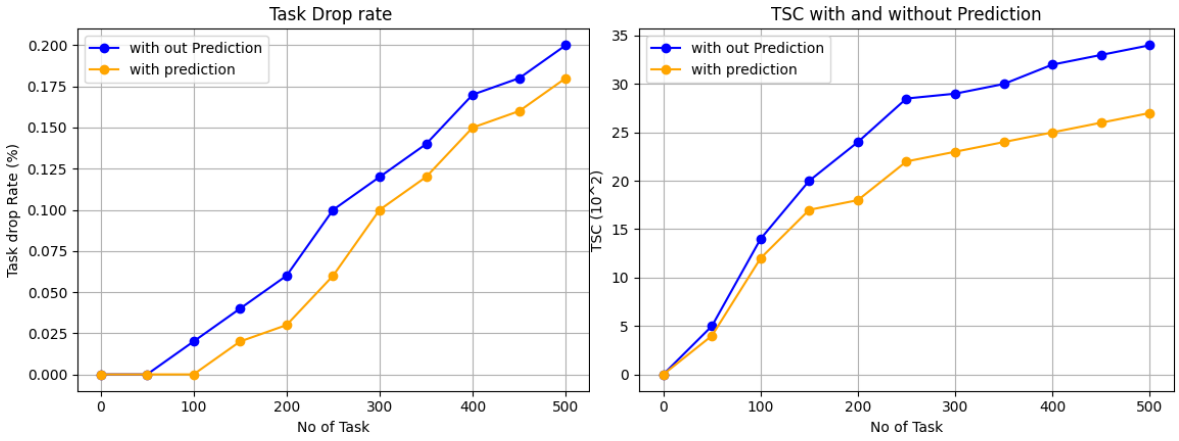
3) Impacts of Prediction

The newly generated tasks demand optimal allocation of computational resources, which is achieved through a decision model. From the moment of task generation until the al- location

 doi:10.20944/preprints202510.1811.v1

decision is made, these tasks experience queuing and decision response delays within the cache queue. Our system employs a BLSTM prediction model to forecast incoming tasks based on historical data. This prediction then feeds into the decision model, which proposes an offloading scheme for task assignment. For the purpose of our experiment, we analyze the performance using the first 500 tasks generated. This approach combines predictive modeling with decision-making to optimize resource allocation and task management in a dynamic computational environment.

The results depicted in Figure 9 considerable advantages of utilizing prediction in task execution, comparing total task drop rate and system cost when doing prediction with the pro- posed algorithm and with just only implementing A2C without BLSTM prediction for decision making. Both graphs show a linear increase in task drop and cost as the number of tasks grows, but these increases are significantly steeper without prediction. The blue lines, representing execution without prediction, indicate higher task drop and costs than the orange lines, which represent a BLSTM prediction based A2C decision making to offload and a predicted resource allocation. Using prediction reduces both tasks drop rate and system cost, resulting in more efficient and cost-effective task management. These benefits become increasingly pronounced with a higher number of tasks, underscoring the scalability and strategic importance of predictive mechanisms in task management.



(A) Total dropped Task

(B) Total system cost over No of task

Figure 9. Diagram of real-time simulated decision results.

When comparing the results of prediction-based approaches versus those without pre- diction, it is evident that our method performs significantly better overall. However, our scheme requires additional energy initially to train the model until it reaches an error-tolerant threshold during prediction. This increased energy consumption occurs only during the initial deployment and training stage. Additionally, the training process is conducted on the MEC server and macro base stations, ensuring that this initial energy expenditure does not significantly impact the overall performance of the system.

Once the model is trained and deployed, the energy requirements stabilize, and the benefits of the prediction-based approach become more pronounced. These benefits include improved task offloading efficiency, reduced latency, and higher system reliability. By leveraging the computational resources of the MEC server and macro base stations, we ensure that the end-user experience remains unaffected during the training phase. This strategic deployment of training processes helps maintain optimal system performance while reaping the long-term advantages of our enhanced predictive model.

3. Conclusion

This study addresses the critical challenge of task offloading and resource allocation in mobile edge computing (MEC) environments, focusing on minimizing energy consumption and response

 doi:10.20944/preprints202510.1811.v1

times while maintaining high-quality real-time performance. By framing the problem as a Markov Decision Process (MDP) and employing an innovative Actor-Critic (A2C) algorithm integrated with BLSTM prediction, we developed an intelligent offloading strategy for mobile IoT devices. Simulation results confirm that the proposed approach outperforms traditional methods, achieving lower system costs, higher task acceptance rates, and improved resource utilization, ultimately enhancing the quality of service (QoS) for end-users.

Future research should focus on extending the A2C-based model to handle the dynamic mobility of devices in MEC environments. Developing algorithms that adapt to rapid changes in device locations and network conditions will be critical for seamless task offloading. Additionally, incorporating advanced AI techniques like federated learning, meta-learning, and enhanced predictive models can further optimize resource allocation and task scheduling, paving the way for more robust and efficient MEC systems.

References

1. Y. Bin Zikria, R. Ali, M. K. Afzal, and S. W. Kim, "Next-Generation Internet of Things (IoT): Opportunities, Challenges, and Solutions," pp. 1–7, 2021.
2. S. Hossain, C. I. Nwakanma, J. M. Lee, and D. Kim, "Edge computational task offloading scheme using reinforcement learning for IIoT scenario," *ICT Express*, vol. 6, no. 4, pp. 291–299, 2020, doi: 10.1016/j.icte.2020.06.002.
3. H. Lin, S. Zeadally, Z. Chen, H. Labiod, and L. Wang, "Jo ur l Pr e r f," *J. Netw. Comput. Appl.*, p. 102781, 2020, doi: 10.1016/j.jnca.2020.102781.
4. T. Kim and S. Yoo, "Edge / Fog Computing Technologies for IoT Infrastructure II," pp. 2–4, 2023.
5. M. Pendo, J. Mahenge, C. Li, and C. A. Sanga, "Energy-ef fi cient task of fl oading strategy in mobile edge computing for resource-intensive mobile applications," *Digit. Commun. Networks*, vol. 8, no. 6, pp. 1048–1058, 2022, doi: 10.1016/j.dcan.2022.04.001.
6. W. Sun, J. Liu, and Y. Yue, "AI-Enhanced Offloading in Edge Computing : When Machine Learning Meets Industrial IoT," no. October, pp. 68–74, 2019.
7. J. P. Jue, "All One Needs to Know about Fog Computing and Related Edge Computing Paradigms All One Needs to Know about Fog Computing and Related Edge Computing Paradigms : A Complete Survey," no. February, 2019, doi: 10.1016/j.sysarc.2019.02.009.
8. L. Yan, "A Task Offloading Algorithm With Cloud Edge Jointly Load Balance Optimization Based on Deep Reinforcement Learning for Unmanned Surface Vehicles," *IEEE Access*, vol. 10, pp. 16566–16576, 2022, doi: 10.1109/ACCESS.2022.3150406.
9. Y. Tu, H. Chen, and L. Yan, "Task Offloading Based on LSTM Prediction and Deep Reinforcement Learning for Efficient Edge Computing in IoT," pp. 1–19, 2022.
10. R. Li, C. Wang, Z. Zhao, R. Guo, and H. Zhang, "The LSTM-based Advantage Actor-Critic Learning for Resource Management in Network Slicing with User Mobility," vol. 7798, no. 1, pp. 1–5, 2020, doi: 10.1109/LCOMM.2020.3001227.
11. C. Wang, L. I. N. Ma, R. Li, and H. Zhang, "Exploring Trajectory Prediction Through Machine Learning Methods," pp. 101441–101452, 2019.
12. T. Haarnoja, H. Zhu, G. Tucker, and P. Abbeel, "Soft Actor-Critic Algorithms and Applications," 2015.
13. H. Yuan *et al.*, "Online Dispatching and Fair Scheduling of Edge Computing Tasks : a Learning-based Approach," vol. 4662, no. c, 2021, doi: 10.1109/JIOT.2021.3073034.
14. J. Wang, K. Liu, M. Ni, and J. Pan, "Learning Based Mobility Management Under Uncertainties for Mobile Edge Computing," no. 1, pp. 1–6.
15. Y. Miao, G. Wu, M. Li, A. Ghoneim, and M. Al-rakhami, "Intelligent task prediction and computation offloading based on mobile-edge cloud computing," *Futur. Gener. Comput. Syst.*, vol. 102, pp. 925–931, 2020, doi: 10.1016/j.future.2019.09.035.
16. V. D. Tuong, T. P. Truong, T. Nguyen, W. Noh, and S. Cho, "Partial Computation Offloading in NOMA-Assisted Mobile Edge Computing Systems Using Deep," vol. XX, no. XX, 2021, doi: 10.1109/JIOT.2021.3064995.

17. Z. Li and C. Guo, "Multi-Agent Deep Reinforcement Learning based Spectrum Allocation for D2D Underlay," *IEEE Trans. Veh. Technol.*, vol. PP, no. c, p. 1, 2019, doi: 10.1109/TVT.2019.2961405.
18. M. Abadi *et al.*, "TensorFlow : A System for Large-Scale Machine Learning This paper is included in the Proceedings of the TensorFlow : A system for large-scale machine learning," 2016.
19. M. Tang and V. W. S. Wong, "Deep Reinforcement Learning for Task Offloading in Mobile Edge Computing Systems," vol. 1233, no. 99, 2020, doi: 10.1109/TMC.2020.3036871.
20. M. Chen, T. Wang, S. Zhang, and A. Liu, "Deep reinforcement learning for computation offloading in mobile edge computing environment," *Comput. Commun.*, vol. 175, no. April, pp. 1–12, 2021, doi: 10.1016/j.comcom.2021.04.028.
21. W. Hou, G. S. Member, H. Wen, and S. Member, "Multiagent Deep Reinforcement Learning for Task Offloading and Resource Allocation in Cybertwin-Based Networks," no. April, 2022, doi: 10.1109/JIOT.2021.3095677.
22. A. Garmendia-orbegozo, J. D. Nunez-gonzalez, and M. A. Anton, "Task Offloading in Edge Computing Using GNNs and DQN," 2024, doi: 10.32604/cmes.2024.045912.