Learning-based Offloading of Tasks with Diverse Delay Sensitivities for Mobile Edge Computing

Tianyu Zhang*†, Yi-Han Chiang†, Cristian Borcea‡, and Yusheng Ji†*
*Department of Informatics, SOKENDAI (The Graduate University for Advanced Studies), Tokyo, Japan
†Information Systems Architecture Science Research Division, National Institute of Informatics, Tokyo, Japan
‡Department of Computer Science, New Jersey Institute of Technology, NJ, USA
E-mail: {zty,yhchiang}@nii.ac.jp, borcea@njit.edu, kei@nii.ac.jp

***Abstract*—The ever-evolving mobile applications need more and more computing resources to smooth user experience and sometimes meet delay requirements. Therefore, mobile devices (MDs) are gradually having difficulties to complete all tasks in time due to the limitations of computing power and battery life. To cope with this problem, mobile edge computing (MEC) systems were created to help with task processing for MDs at nearby edge servers. Existing works have been devoted to solving MEC task offloading problems, including those with simple delay constraints, but most of them neglect the coexistence of deadline-constrained and delay-sensitive tasks (i.e., the diverse delay sensitivities of tasks). In this paper, we propose an actor-critic based deep reinforcement learning (ADRL) model that takes the diverse delay sensitivities into account and offloads tasks adaptively to minimize the total penalty caused by deadline misses of deadline-constrained tasks and the lateness of delay-sensitive tasks. We train the ADRL model using a real data set that consists of the diverse delay sensitivities of tasks. Our simulation results show that the proposed solution outperforms several heuristic algorithms in terms of total penalty, and it also retains its performance gains under different system settings.**

***Index Terms*—Mobile edge computing, task offloading, diverse delay sensitivities, actor-critic method, deep reinforcement learning.**

I. Introduction

The emergence of newly designed mobile applications requires powerful computing resources to facilitate smooth user experience. In spite of their resource increases, mobile devices (MDs) are still constrained in terms of both computing power and battery life. Therefore, computation-intensive and delay-sensitive tasks are difficult to be performed entirely on MDs. To overcome these challenges, mobile edge computing (MEC) [1]–[3] was proposed to enable MDs to offload tasks to edge servers (ESs) that are deployed in the vicinity. Compared with conventional cloud computing, MEC not only offers lower latency and higher scalability, but also reduces the amount of data traffic in the core network. Thanks to these advantages, MEC becomes a promising platform to help process tasks on behalf of MDs, thereby ensuring these tasks can be completed on time.

Each MD may have applications that generate tasks with different sensitivity levels toward task completion. To differentiate tasks with diverse delay sensitivities, we categorize them into two service classes[1]: *deadline-constrained* and *delay-sensitive*. A *deadline-constrained* task (e.g., real-time object detection in a safety application) has a task deadline before which it should be finished; otherwise, the computation result is useless, which could affect adversely the user experience. On the other hand, a *delay-sensitive* task (e.g., augmented reality gaming, non-real time video analytics) does not have a task deadline to meet, but it is relatively sensitive to the lateness of task completion time. Due to the different sensitivity levels of tasks, it is therefore important to offload tasks adaptively while tasks with the diverse delay sensitivities coexist in MEC systems[2].

Prior works have studied task offloading to satisfy delay constraints. Wang *et al.* [8] aim to minimize the usage of computing resources while ensuring that all computational demands can be answered on time. Yin *et al.* [9] investigate a multi-resource allocation problem where each task is associated with a profit if it can be executed before its deadline. Feng *et al.* [10] propose a framework with the purpose of increasing the computational capabilities of vehicles with deadlines. Tao *et al.* [11] introduce an opportunistic helper to minimize energy consumption under delay constraints. Pu *et al.* [12] propose a framework to minimize energy consumption, while considering application deadlines and vehicle incentives. All these works neglect the coexistence of tasks with diverse delay sensitivities in MEC, which may result in unsatisfactory user experience.

Finding an optimal task offloading strategy in MEC systems is in computationally prohibitive [13]. To address such an intractable problem, several solutions can be found in literature (e.g., queueing theory, combinatorial optimization, or machine learning), among which deep reinforcement learning (DRL) has been lately regarded as a promising one. Yu *et al.* [14] propose a deep supervised learning model to minimize computation and offloading costs. Xu *et al.* [15] develop a post-decision state-based RL algorithm to minimize service delays and operational costs. Li *et al.* [16] aim to minimize delay and energy consumption through Q-learning and deep Q-network (DQN) schemes. Le *et al.* [17] propose a DRL-based offloading algorithm to maximize user utility

[1]Each service class can be further divided into multiple sub-classes to support finer-grained delay sensitivities.

[2]Task offloading can also be designed in accordance with energy harvesting [4], multiple connectivity [5], vehicles [6], or the availability of both computing and radio resources [7]. In this paper, we focus on how to intelligently offload tasks with diverse delay sensitivities.

while minimizing the incurred cost. Chen *et al.* [18] design a double DQN-based RL algorithm to maximize long-term utility. Although these works can guide us to learning-based task offloading, it remains unclear how to leverage state-of-the-art RL to offload tasks with diverse delay sensitivities.

This paper investigates the problem of task offloading in MEC systems while considering the diverse delay sensitivities of tasks. We propose an actor-critic [19] based DRL (ADRL) model to minimize the total penalty that is attributed to the deadline misses of deadline-constrained tasks and the lateness of delay-sensitive tasks. We train the ADRL model using a real data set, from which we can observe the diverse delay sensitivities of tasks. Our simulation results show that the ADRL model outperforms several heuristic algorithms in terms of total penalty, as it can offload deadline-aware and delay-sensitive tasks differently. Even if the system is heavily loaded or it does not have a lot of computing power at the edge, the ADRL model can still reach low total penalty as it can learn well from the environment.

The rest of this paper is organized as follows. Sec. II presents the system model. Sec. III describes our solution, ADRL. Sec. IV shows simulation results and analysis. The paper concludes in Sec. V.

II. System Model

A. Network Environment

We consider a MEC system that consists of a set $\mathcal{U}$ of MDs generating tasks, the set of all tasks $\mathcal{J}$, and a set $\mathcal{N}$ of ESs (as depicted in Fig. 1). We assume ESs are either co-located with the base stations (BSs) or close to the BSs, such that they can interact with MDs with low latency. All these BSs connect to an offloading controller which employs an offloading policy to select an appropriate ES for each task. Both MDs and ESs are capable of processing tasks. Each task generated by an MD is indivisible, and it can be either processed locally or offloaded to an ES in $\mathcal{N}$.

Each time an MD generates a task, it sends this task to the controller through a nearby BS. The controller collects the state of each ES and makes offloading decisions: offload this task to a specific ES or guide the MD to process the task locally. After receiving the task from the controller, an ES stores this task in its dedicated queue and process it according to a scheduling policy. Although various scheduling policies can be employed in a MEC system, here we choose the first-come-first-served policy. We assume that each ES has a limited capacity, and it can store at most l tasks in its queue. The ES executes the tasks according to the scheduling policy the task and sends the results back to MDs. For simplicity, we do not allow ESs to migrate a task among them after offloading.

B. Time Information of Tasks

Let t_j^{rel} denote the release time of task j to indicate that the hosting MD generates this task at time t_j^{rel}. The uploading delay t_{jk}^{up} and the downloading delay t_{jk}^{down} characterize the effects of the wireless network quality between the hosting MD and the nearby BS, as well as the network delay between the controller and ES k.

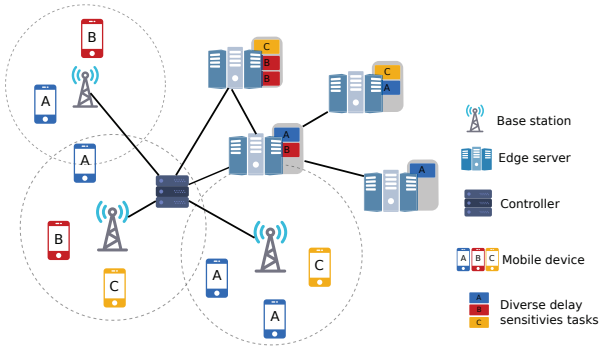


Fig. 1: In the considered MEC system, mobile devices generate tasks with diverse delay sensitivities and the task offloading controller manages the computing resources of ESs, collects task information (through BSs), then makes offloading decisions.

If task j is offloaded to ES k, ES k cannot start processing task j before $t_j^{\text{rel}} + t_{jk}^{\text{up}}$. On completion of the computation by ES k, the hosting MD needs the additional time of t_{jk}^{down} to get the result. If the hosting MD processes the task locally, it can get the result right after the task completion.

The processing time for task j is denoted by t_{jk}^{edge} and t_j^{local} for the edge computation at ES k and the local computation, respectively. Generally, t_{jk}^{edge} is substantially lower than t_j^{local} because ESs have more powerful computing resources than MDs. We define the computability ratio ρ_k between ES k and the MD: $t_j^{\text{local}} = \rho_k \cdot t_{jk}^{\text{edge}}$.

All tasks must be processed either locally or through offloading. If all the queues of ESs are full, a newly generated task must be executed locally. We assume that uploading and downloading a task do not consume any computation resources at the ES. The completion time of a task j, t_j^{com}, is defined as:

$$t_j^{\text{com}} = \begin{cases} t_j^{\text{local}}, & \text{(locally)} \\ t_{jk}^{\text{up}} + t_{jk}^{\text{wait}} + t_{jk}^{\text{edge}} + t_{jk}^{\text{down}}, & \text{(at ES } k\text{)} \end{cases} \tag{1}$$

where t_{jk}^{wait} is the waiting time of task j at ES k.

C. Diverse Delay Sensitivities

The diverse delay sensitivities indicate the coexistence of deadline-constrained and delay-sensitive tasks. For simplicity, we partition $\mathcal{J}$ into two disjoint sets $\mathcal{J}^{\text{dc}}$ and $\mathcal{J}^{\text{ds}}$, which represent the sets of deadline-constrained and delay-sensitive tasks, respectively.

- For each deadline-constrained task, the goal is to complete the task within its predefined deadline, and hence the resulting penalty can be expressed as

$$f_j = \begin{cases} w_{D_j}, & \text{if } t_j^{\text{com}} \geq D_j, \\ 0, & \text{otherwise,} \end{cases} \quad \forall j \in \mathcal{J}^{\text{dc}}, \tag{2}$$

  where D_j is the deadline of task j and w_{D_j} is the penalty for missing this deadline, which is typically set to a large constant value.

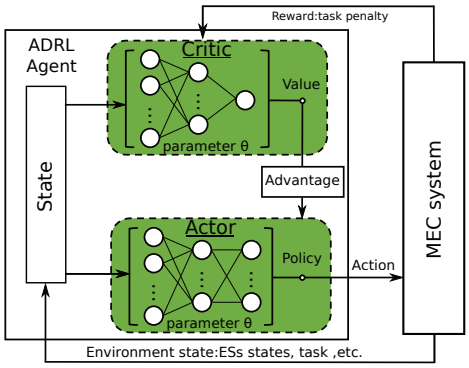


Fig. 2: The basic ADRL model.

- For each delay-sensitive task, there is no predefined deadline to meet, but the service satisfaction decreases with the task completion time. Without loss of generality, we write the penalty as

$$f_j = g_j(t_j^{\mathrm{com}}), \quad \forall j \in \mathcal{J}^{\mathrm{ds}}, \tag{3}$$

where g_j is a monotonically increasing function (which is often modeled as a linear or a convex function).

III. ADRL Model

A. Introduction to DRL

Reinforcement learning (RL) [20] is an approach to learn how to map states to actions in order to maximize a numerical reward. An RL agent interacts with an environment over time. At each time step t, the agent first observes the state s_t of the environment in a state space $\mathcal{S}$, and selects an action a_t in an action space $\mathcal{A}$ based on a policy π. After selecting the action, the state of the environment transitions to the next state s_{t+1}, and the agent receives a reward r_t according to the environment dynamics. The state transition probabilities and rewards are stochastic and are assumed to have the Markov property, namely they depend only on the previous state of the environment and the action taken by the agent. This process continues until the agent reaches a terminal state. The return $R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$ is the accumulated discounted reward with the discount factor $\gamma \in (0, 1]$, which determines the present value of future rewards. The agent aims to maximize the expectation of R.

In most problems of practical interest, the state space is enormous, and there are too many possible state-action pairs. Hence, it is not viable to store the policy π entirely in a tabular form and yield an optimal policy in a short time. Therefore, our alternative goal is to find a good function approximation of π using limited computing resources.

In many forms of function approximations, deep neural networks (DNNs) have recently been used successfully to solve large-scale RL problems [21], also known as deep reinforcement learning (DRL) [22]. In this method, the learning algorithm adjusts the parameters θ of a DNN over the entire state space. Hence, we represent the policy as $\pi(a|s;\theta)$, which means the agent adopts a policy determined by parameters θ.

B. Asynchronous Advantage Actor-Critic (A3C) Algorithm

Fig. 2 shows our ADRL model, which uses A3C, a state-of-the-art actor-critic RL algorithm [19]. The agent will train two neural networks: a critic estimates the value function $V(s_t;\theta_v)$ that measures how good a certain state is to be in; an actor maintains a policy $\pi(a_t|s_t;\theta)$ that controls how RL agent behaves.

Instead of learning the policy that maps states to actions $\pi(a_t|s_t;\theta)$ directly, the RL agent uses the estimate of value function to update the policy more intelligently than traditional policy gradient methods.

The A3C algorithm updates parameters via

$$\Delta\theta = \underbrace{\alpha}_{\text{learning rate}} \cdot \underbrace{\nabla_\theta \log \pi(a_t|s_t;\theta)}_{\text{policy gradient}} \cdot \underbrace{A(s_t, a_t; \theta, \theta_v)}_{\text{estimated advantage function}}, \tag{4}$$

where α is the learning rate that controls how much we adjust the weights of our DNN with respect to the gradient, $\nabla_\theta \log \pi(a_t|s_t;\theta)$ gives the direction on how to change the policy parameters, and $A(s_t, a_t; \theta, \theta_v)$ is an estimate of the advantage function given by $R_t - V(s_t;\theta_v)$ which tells the RL agent how much better or worse the accumulated discounted reward turned out to be than expected. Here, the accumulated discounted reward R_t can be seen as an estimate of expected return for selecting action a_t in state s_t and following policy π.

A3C utilizes multiple incarnations of the above RL agent to learn more efficiently. Each RL agent in A3C has its own network parameters regarding the global network. Each of these RL agents interacts with its own copy of the environment at the same time with the other agents. The experience of each RL agent is different from the others; hence, the overall experience available for training becomes more diverse.

C. ADRL Formulation

The controller runs a trained RL agent to make offloading decisions, and the whole MEC system can be seen as the environment of the ADRL model. The problem of making an offloading decision for MDs is formulated as a Markov Decision Process (MDP), and the main goal of the system is to find the optimal policy π which minimizes the total penalty of all tasks.

1) State space: The state space S of this system consists of the current state of all queues and the information of the task waiting for offloading, and it is represented as

$$\mathcal{S} = \left\{ (x_j, Q(t_j^{\mathrm{rel}})) \right\}, \tag{5}$$

where $Q(t_j^{\mathrm{rel}})$ is the state of all ESs' queues (defined below) at time t_j^{rel}, and x_j is time information of the current task j. x_j is defined as

$$x_j = \left(t_j^{\mathrm{up}}, t_j^{\mathrm{down}}, t_j^{\mathrm{local}}, t_j^{\mathrm{edge}}, class \right), \tag{6}$$

where $class$ indicates which delay sensitivity task j has, $t_j^{\mathrm{up}} = (t_{j1}^{\mathrm{up}}, \ldots, t_{j|\mathcal{N}|}^{\mathrm{up}})$, $t_j^{\mathrm{down}} = (t_{j1}^{\mathrm{down}}, \ldots, t_{j|\mathcal{N}|}^{\mathrm{down}})$, and $t_j^{\mathrm{edge}} = (t_{j1}^{\mathrm{edge}}, \ldots, t_{j|\mathcal{N}|}^{\mathrm{edge}})$ are uploading delays from the

hosting MD to all ESs, downloading delays from ESs to MD, and edge processing time at each ES, respectively.

The state of all queues is defined as

$$Q(t) = \left(q_1(t), q_2(t), \ldots, q_{|\mathcal{N}|}(t)\right), \tag{7}$$

where $q_k(t) = ((\tau_{k1}^{\text{up}}(t), \tau_{k1}^{\text{proc}}(t)), \ldots, (\tau_{kl}^{\text{up}}(t), \tau_{kl}^{\text{proc}}(t)))$ refers to the state of the queue of ES k at time t. To characterize the queueing behaviors of tasks in each ES e_k, we use $\tau_{ki}^{\text{up}}(t)$ and $\tau_{ki}^{\text{proc}}(t)$ to represent the residual uploading and processing time of the i-th scheduled task at time t, respectively, which can be expressed as

$$\tau_{ki}^{\text{up}}(t) = \max\left(t_{\sigma_k^i}^{\text{rel}} + t_{\sigma_k^i k}^{\text{up}} - t, 0\right), \tag{8}$$

$$\tau_{ki}^{\text{proc}}(t) = \max\left(t_{\sigma_k^i}^{\text{rel}} + t_{\sigma_k^i k}^{\text{up}} + t_{\sigma_k^i k}^{\text{wait}} + t_{\sigma_k^i k}^{\text{edge}} - t, 0\right), \tag{9}$$

where σ_k^i corresponds to the actual task that is scheduled as the i-th task in ES e_k.

2) Action space: The action space is defined as:

$$\mathcal{A} = \{a \in \{0, 1, 2, ..., |\mathcal{N}|\}\}, \tag{10}$$

where $a = 0$ is to execute this task locally, and $a = k > 0$ indicates to offload this task to ES k. Our system only chooses valid actions, i.e., it can only offload the task to the ESs where the queues are not full. If all the queues are full, the controller can only choose the action $a = 0$.

3) Reward: The objective of the system is to minimize the total penalty $\sum_{j \in \mathcal{J}} f_j$, which is equivalent to maximizing the total reward of the ADRL model. Therefore, we can simply set the reward at each step to $-f_j$ for each task j in the ADRL model.

4) Training method: We use a variable time step size learning method to train our ADRL model instead of slicing the time into uniform time slices. This means our approach only makes offloading decisions when an MD generates a task rather than at each time slice.

IV. Performance Evaluation

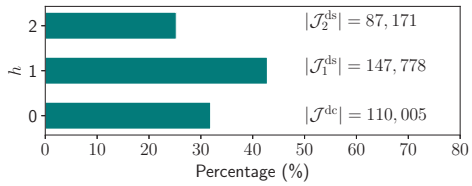
This section compares the performance of our ADRL model with several heuristic algorithms, and then shows how the achieved performance varies with system settings.

A. Simulation Settings

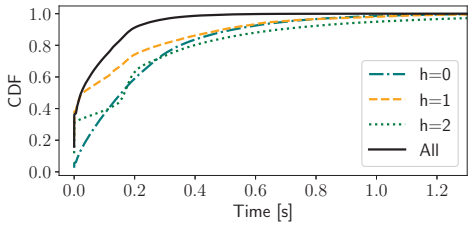
The default parameter settings (unless stated otherwise) for our system and the ADRL model are shown in TABLE I, and the detailed task settings are described below.

TABLE I: Simulation Settings

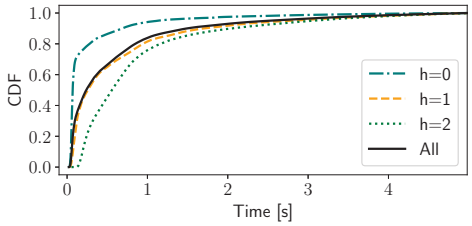
Parameters	Symbols	Values
Number of MDs	$\lvert\mathcal{U}\rvert$	500
Number of ESs	$\lvert\mathcal{N}\rvert$	4
Maximum queue length	l	8
Uploading delay	t^{up}	$\text{Unif}(0.03, 0.2)$ [s]
Downloading delay	t^{down}	$\text{Unif}(0.01, 0.1)$ [s]
Computability ratio	ρ_k	4.0
Deadline	D_j	1.0 [s]
Penalty (h=0)	w_{D_j}	20
Penalty (h=1)	g_j	$1.0 \cdot t_j^{\text{com}}$
Penalty (h=2)	g_j	$0.1 \cdot t_j^{\text{com}}$



(a) The diversity of task delay sensitivities.



(b) The cumulative distribution of task inter-arrival time.



(c) The cumulative distribution of task processing time.

Fig. 3: Task modeling.

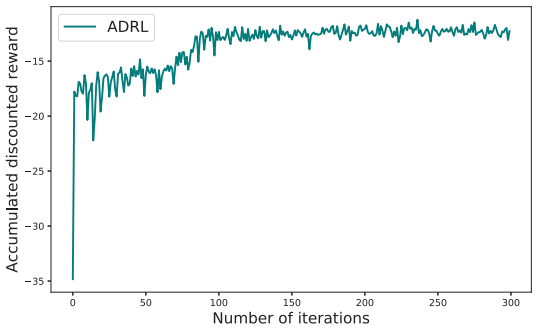
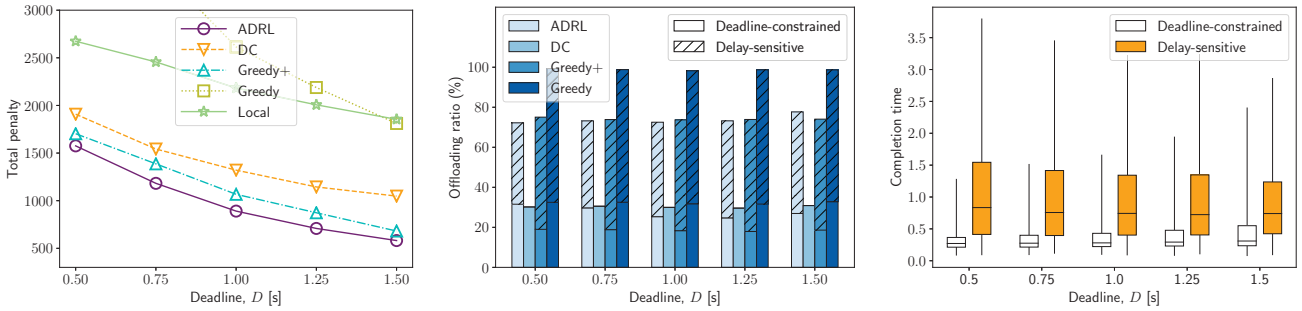


Fig. 4: The accumulated discounted reward over iterations.

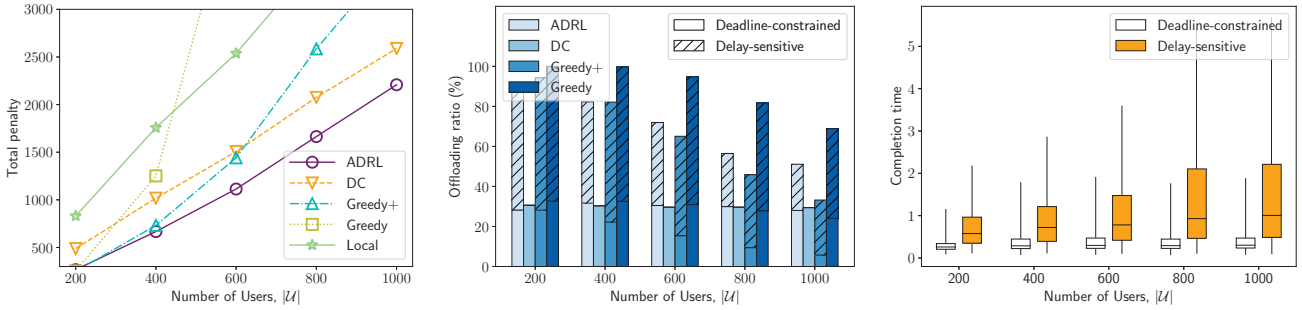
1) Data set: To have more realistic simulation results, we use a real data set, Google Cluster [23], instead of ideal stochastic processes. From this data set, we collect 344,954 tasks and uniformly map them into 1,000 groups, each of which represents the set of tasks generated by an MD. In the following, we will randomly pick up $|\mathcal{U}|$ groups and use the corresponding sets of tasks.

2) Diverse delay sensitivities: Every task in the data set is associated with a specific value $h \in \{0, 1, 2\}$ that indicates the delay sensitivity of the task, where 0 and 2 are the highest and lowest delay sensitivities, respectively. Let $\mathcal{J}^{\text{dc}}$, $\mathcal{J}_1^{\text{ds}}$ and $\mathcal{J}_2^{\text{ds}}$ be the sets of tasks labeled as $h = 0$, 1 and 2, respectively. From Fig. 3a, we see that there are a large number of tasks in every delay sensitivity, revealing that tasks with the diverse delay sensitivities indeed take place in real world.



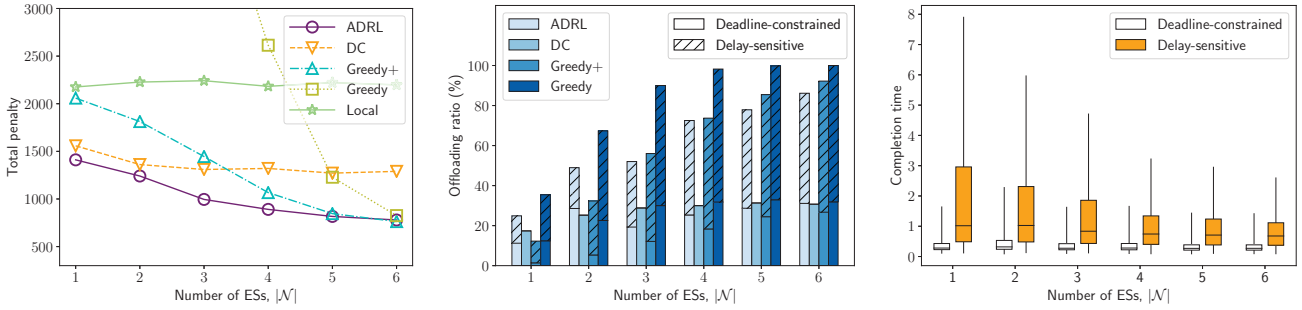
(a) Total penalty comparison (b) Offloading ratio comparison (c) Completion time profile (in ADRL)

Fig. 5: The impact of deadlines.



(a) Total penalty comparison (b) Offloading ratio comparison (c) Completion time profile (in ADRL)

Fig. 6: The impact of the system load.



(a) Total penalty comparison (b) Offloading ratio comparison (c) Completion time profile (in ADRL)

Fig. 7: The impact of the edge computability.

3) Task arrival: The arrival time information of the data set can be used to characterize the release time of tasks. From Fig. 3b, we see that up to 90% of the tasks have inter-arrival time below 0.5 seconds. This observation shows that the tasks are generated frequently and thus motivates MDs to offload.

4) Task processing: Each ES schedules tasks in a first-come-first-served fashion. The processing time of a task on an ES is set to the values shown in Fig. 3c, where we see that the deadline-constrained tasks have shorter processing time than the delay-sensitive tasks. This observation also conforms with the intuition that the processing time of deadline-constrained tasks is typically short, so that all of them can hopefully be completed on time.

5) Training for the ADRL model: We use two DNNs (each of them is a three-layer fully connected NN with a Relu activation function $\mathrm{Relu}(x) = \max(x, 0)$) to set up the ADRL model. The optimization model is implemented with the Adam gradient descent optimization algorithm [24] for training the agent. For convergence, we set the discount factor to 0.99. In each iteration, we use 80% of the task data for training and the rest for testing.

B. Heuristic Algorithms

For performance comparison, we use the following heuristic algorithms.

- Local: all tasks are processed locally.
- Greedy: each task will be processed by an ES that minimizes its completion time.
- Greedy+: each task will be processed by a machine (MD or ES) that minimizes its completion time.
- DC: each deadline-constrained task will be processed by an ES that minimizes its completion time, while all delay-sensitive tasks are processed locally.

C. Simulation Results

1) The convergence of training: It is essential to understand how the training of the proposed solution converges over iterations. We see in Fig. 4 that the accumulated discounted reward increases steadily with the number of iterations. In addition, the performance boosts rapidly in the first 100 iterations and soon becomes stable in later iterations.

2) The impact of deadlines: Fig. 5 depicts how the system performance varies with deadlines. We see in Fig. 5a that ADRL achieves the lowest total penalty, since it properly controls offloading ratios as compared with other algorithms (see Fig. 5b). As the deadline increases, delay-sensitive tasks achieve shorter completion time at the price of slightly longer completion time for deadline-constrained tasks (see Fig. 5c). This is because there is no penalty for deadline-constrained tasks as long as no deadline misses take place.

3) The impact of the system load: Fig. 6 shows how the number of MDs influences the system performance. We see in Fig. 6a that ADRL retains the lowest total penalty, revealing that it is robust to the system load. Even with a large number of MDs, ADRL can still keep the offloading ratio of deadline-constrained tasks stable by decreasing the total offloading ratio (see Fig. 6b), while slightly sacrificing the completion time of delay-sensitive tasks (see Fig. 6c).

4) The impact of the edge computing power: Fig. 7 depicts how the number of ESs affects the system performance. We see in Fig. 7a that ADRL can better utilize the abundant computing resources to keep the total penalty low, thanks to its appropriate offloading ratio for each delay sensitivity class (see Fig. 7b). Furthermore, the higher the number of ESs, the lower the completion time of both deadline-constrained and delay-sensitive tasks (see Fig. 7c).

V. Conclusion

The performance of task offloading, and implicitly of task scheduling, is key to the success of MEC systems. Task offloading is challenging when taking the diverse delay sensitivities of tasks into account. To address this problem, we propose the ADRL model to optimize the scheduling of all tasks by minimizing their total penalty. ADRL is trained using a real data set that consists of deadline-constrained and delay-sensitive tasks. Our simulation results demonstrate that the ADRL model outperforms other heuristic algorithms in that it intelligently controls the offloading rates and balances the offloading ratios between deadline-constrained and delay-sensitive tasks. When the system load increases or the edge computing power decreases, the ADRL model can still keep the total penalty low as it provides better protection to deadline-constrained tasks, at the price of longer completion time for delay-sensitive tasks.

Acknowledgment

This work was supported in part by JSPS KAKENHI under Grant 18H06471, Grant 18KK0279 and Grant 19K21539.

References

[1] T. Taleb, S. Dutta, A. Ksentini, M. Iqbal, and H. Flinck, "Mobile edge computing potential in making cities smarter," *IEEE Commun. Mag.*, vol. 55, no. 3, pp. 38–43, Mar. 2017.

[2] P. Mach and Z. Becvar, "Mobile edge computing: A survey on architecture and computation offloading," *IEEE Commun. Surveys Tuts.*, vol. 19, no. 3, pp. 1628–1656, 3rd Quart. 2017.

[3] (2018, Nov.) Multi-access edge computing (MEC). [Online]. Available: https://www.etsi.org/technologies-clusters/technologies/multi-access-edge-computing

[4] C. You, K. Huang, and H. Chae, "Energy efficient mobile cloud computing powered by wireless energy transfer," *IEEE J. Sel. Areas Commun.*, vol. 34, no. 5, pp. 1757–1771, May 2016.

[5] F. Guo, H. Zhang, H. Ji, X. Li, and V. C. M. Leung, "Energy efficient computation offloading for multi-access MEC enabled small cell networks," in *Proc. IEEE ICC WKSHPS*, May 2018.

[6] J. Feng, Z. Liu, C. Wu, and Y. Ji, "AVE: Autonomous vehicular edge computing framework with ACO-based scheduling," *IEEE Trans. Veh. Technol.*, vol. 66, no. 12, pp. 10 660–10 675, 2017.

[7] S. Sardellitti, G. Scutari, and S. Barbarossa, "Joint optimization of radio and computational resources for multicell mobile-edge computing," *IEEE Trans. Signal Inf. Process. Netw.*, vol. 1, no. 2, pp. 89–103, Jun. 2015.

[8] W. Wang and W. Zhou, "Computational offloading with delay and capacity constraints in mobile edge," in *Proc. IEEE ICC*, May 2017.

[9] B. Yin, Y. Cheng, L. X. Cai, and X. Cao, "Online SLA-aware multi-resource allocation for deadline sensitive jobs in edge-clouds," in *Proc. IEEE GLOBECOM*, Dec. 2017.

[10] J. Feng, Z. Liu, C. Wu, and Y. Ji, "HVC: A hybrid cloud computing framework in vehicular environments," in *Proc. IEEE MobileCloud*, Apr. 2017.

[11] Y. Tao, C. You, P. Zhang, and K. Huang, "Stochastic control of computation offloading to a dynamic helper," in *Proc. IEEE ICC WKSHPS*, May 2018.

[12] L. Pu, X. Chen, G. Mao, Q. Xie, and J. Xu, "Chimera: An energy-efficient and deadline-aware hybrid edge computing framework for vehicular crowdsensing applications," *IEEE Internet Things J.*, vol. 6, no. 1, pp. 84–99, Feb. 2019.

[13] X. Chen, L. Jiao, W. Li, and X. Fu, "Efficient multi-user computation offloading for mobile-edge cloud computing," *IEEE/ACM Trans. Netw.*, vol. 24, no. 5, pp. 2795–2808, Oct. 2016.

[14] S. Yu, X. Wang, and R. Langar, "Computation offloading for mobile edge computing: A deep learning approach," in *Proc. IEEE PIMRC*, Oct. 2017.

[15] J. Xu, L. Chen, and S. Ren, "Online learning for offloading and autoscaling in energy harvesting mobile edge computing," *IEEE Trans. Cogn. Commun. Netw.*, vol. 3, no. 3, pp. 361–373, Sep. 2017.

[16] J. Li, H. Gao, T. Lv, and Y. Lu, "Deep reinforcement learning based computation offloading and resource allocation for MEC," in *Proc. IEEE WCNC*, Apr. 2018.

[17] D. V. Le and C. Tham, "A deep reinforcement learning based offloading scheme in ad-hoc mobile clouds," in *Proc. IEEE INFOCOM WKSHPS*, Apr. 2018.

[18] X. Chen, H. Zhang, C. Wu, S. Mao, Y. Ji, and M. Bennis, "Optimized computation offloading performance in virtual edge computing systems via deep reinforcement learning," *IEEE Internet Things J.*, vol. 6, no. 3, pp. 4005–4018, Jun. 2019.

[19] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *Proc. ICML*, Jun. 2016.

[20] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. Cambridge, MA, USA: MIT Press, 1998.

[21] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015.

[22] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, "A brief survey of deep reinforcement learning," *arXiv preprint arXiv:1708.05866 [cs.LG]*, 2017.

[23] (2011, Nov.) Google cluster-usage traces: format+schema. [Online]. Available: https://github.com/google/cluster-data

[24] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980 [cs.LG]*, 2014.