The Journal of Supercomputing (2024) 80:22885–22912
https://doi.org/10.1007/s11227-024-06310-7

A collaborative cache allocation strategy for performance and link cost in mobile edge computing

Hui Xiao[1] · Xinyu Zhang[1] · Zhigang Hu[1] · Meiguang Zheng[1] · Yang Liang[1,2]

Accepted: 17 June 2024 / Published online: 29 June 2024

Abstract

Mobile Edge Computing (MEC) represents a novel paradigm dedicated to addressing the challenge of facilitating rapid access to an immense volume of content over mobile networks. However, improper cache placement and usage, coupled with fluctuating requests for cached data at diverse timeframes, exhibits considerable variability. Despite the abundance of optimization techniques, a majority of them lack the adaptive capacities needed to navigate dynamic caching environments efficiently. Furthermore, many studies employ online deep learning methodologies, but a slow convergence speed during the training process can potentially compromise caching performance and hinder dynamic goal adjustment in alignment with realistic provider requirements. We propose an integrative utility function encapsulating the worth of cached content and the cost associated with transmission links. By dynamically modifying weight values, this function can concurrently meet the performance and link cost demands of edge computing caching systems. To enhance the real-time response of the caching policy and the efficiency of deep learning, we introduce a Collaborative two-stage Deep Reinforcement Learning (CDRL) framework for devising the caching policy model. CDRL utilizes Double Deep Reinforcement Learning (DDQN) for pre-training in the caching environment to make pre-caching decisions and employs a Deep State-Action-Reward-State-Action (SARSA) algorithm for online training and caching decision-making. Experimental results convincingly demonstrate the proposed method's efficacy in improving the cache hit rate, service latency, and link cost.

Keywords Mobile edge computing (MEC) · Cache strategy · Collaborative two-stage deep reinforcement learning (CDRL) · Double deep reinforcement learning (DDQN) · State-action-reward-state-action (SARSA)

Extended author information available on the last page of the article

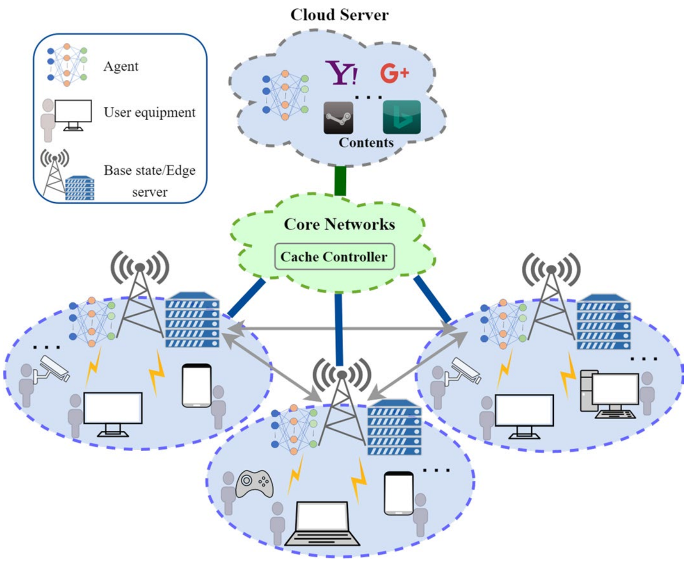

Fig. 1 MEC architecture that supports caching

1 Introduction

In recent years, the widespread popularity of 5 G networks and the massive data generated by mobile devices (such as short videos, images, and mobile games) have given rise to many new issues [1]. In conventional cloud computing, while storage and computational services are extended to mobile devices, orchestrating data collection and processing within the cloud can induce significant and erratic network latencies. Recognizing the constraints of standard cloud computing within the IoT domain, scholars have conceptualized a refreshed computing framework denoted as MEC. As depicted in Fig. 1, a typical MEC architecture is delineated across three levels: the end-user level, the proximal edge server level, and the overarching cloud data center level. Given the inherent constraints in computing and storage capabilities of terminal apparatuses to satisfy a multitude of application demands, the task of storage and computation is frequently relegated to adjacent edge servers. These edge servers, essentially diminutive storage and computational nodes situated in proximity to terminal devices, are tethered through established wired connections [2]. They offer cloud-analogous environments at the network's frontier, facilitating the caching and computational exigencies of the terminal devices.

MEC holds significant importance across various domains, encompassing smart urban developments, telehealth services, expansive manufacturing units, and cohesive business sectors. Within the confines of classic cloud computing, cloud data centers are often burdened with supplying extensive cache data. Yet, constraints associated with network throughput and bandwidth can detrimentally impact the Quality of Service (QoS) extended to users. Given that edge servers are strategically positioned nearer to users compared to cloud servers, it's feasible to handle data processing at the edge, sidestepping the need for cloud-centric operations. This approach curtails back-and-forth data transmissions, minimizes latency, and enhances the QoS for end-users. The extensive storage and computation demands ushered in by mobile devices introduce nuanced challenges to the prevailing MEC landscape. The escalating adoption of mobile devices coupled with the influx of data traffic exerts pronounced pressure on edge servers, especially those grappling with constrained caching capabilities [3]. Concurrently, age-old caching methodologies struggle to keep pace with today's fluid tech milieu. Consequently, suboptimal caching resource distribution on edge servers emerges as a prevalent issue, necessitating a shift towards cooperative caching [4].

The domain of academic research has delved deeply into the realm of collaborative caching. Within the context of cloud computing, the role of collaborative cache is multifaceted: it seeks to curtail energy expenditure within data centers [5], bolster user access velocities [6], and diminish the overarching computational burden [7]. A predominant portion of extant literature on edge collaborative caching portrays the content caching conundrum as a linear programming quandary, which anchors its strategies on content popularity. Such methodologies further seek to augment energy efficacy by minimizing data redundancies via cooperative caching measures [8–10]. Yet, this popularity-centric methodology is riddled with challenges. For instance, it often neglects considerations such as the constraints of edge cache storage while hoarding highly sought-after content, the costs associated with supplanting cached data, or the infrastructural costs linked to user requests. To rectify these oversights, our approach holistically contemplates the merits of caching in tandem with the infrastructural costs, aiming to dynamically harmonize cache utility.

Furthermore, traditional methods, whether based on common protocols or advanced strategies, struggle with adapting to the environment, mainly because of the changing and unpredictable nature inherent to users. Online methods supported by learning techniques offer the flexibility to adjust caching strategies in line with changing environmental needs. As a result, researchers have promoted deep learning-based predictive caching methods [11, 12], cooperative deep reinforcement learning for edge caching [13], and caching systems based on Q-learning [14]. These learning-focused approaches mainly use models that are trained and then sent to the cloud/edge, meeting immediate operational needs. However, challenges remain, especially in achieving effective real-time methods in a short time, mostly due to the extended learning times during training. To address these problems, we propose a new cache utility model that can be adjusted according to the needs of the service provider. To match the dynamic caching environment in real-time, we propose a two-stage Collaborative Deep Reinforcement Learning (CDRL) caching mechanism. The model is deployed on the cache controller to update the caching strategy in real-time. Our contributions are as follows:

- In order to balance the efficiency of edge caching and communication costs in MEC environments and enable cache policies to meet actual performance/link cost requirements in specific environments, the edge collaborative caching problem is modeled, and a new dynamic cache utility function is proposed. The utility function can be changed according to the needs of service providers/actual environments.

- To address the real-time and efficient requirements of edge caching, a two-stage CDRL mechanism is proposed to dynamically optimize the edge caching problem. Due to the fact that the CDRL algorithm shares a neural network in both pre-training and online training, and uses a Double Deep Reinforcement Learning (DDQN) in pre-training to achieve the current best caching policy and a State-Actor-Reward-State-Actor (SARSA) algorithm in online training to improve learning efficiency, better caching results can be achieved in delay-sensitive edge computing environments.
- The performance of the strategy based on dynamic cache utility and CDRL is evaluated through simulation experiments. The simulation results show that, compared with traditional cache replacement algorithms and other deep reinforcement learning (DRL) algorithms, this optimization method can effectively increase cache hit rate, reduce cache cost, and latency.

This paper is organized as follows: Sect. 2 summarizes the current work progress, and Sect. 3 describes the system model and NP-complete proof. In Sect. 4, CDRL and cache algorithms are introduced. Section 5 analyzes the experimental results and gives conclusions. Finally, we summarized the article and looked forward to it.

2 Related work

Contemporary research in this domain is normally divided into three primary streams, including the conventional methods (e.g., linear programming, heuristic strategies), the deep/reinforcement learning methods and the DRL methods.

2.1 Conventional methods

Challenges like transmission link overloads and limited buffer capacity have emerged as pressing concerns in the caching optimization problem. Vo et al. [6], for instance, proposed a collaborative caching scheme tailored for eNBs to channel video content requisitions. This effort frames the optimization issue as a broad integer linear programming problem, proposing a distributed approach that matches the simpler version of the problem and includes nearby request routing with efficient collaborative eviction strategies. However, this scheme lacks real-time performance. Echoing this sentiment, Xie et al. [7] worked on improving network router content caching against the background of traffic management, separating collaborative caching from traffic patterns and creating a set of collaborative caching measures to design effective caching plans. Going further, Ostovari et al. [8] presented a guided real-time collaborative caching method to handle cache cost challenges. Looking at timing, Ferragut et al. [15] explored a time-based (TTL) cache approach. Broadening the scope, Ioannidis et al. [16] introduced a distributed adaptive content placement method, using gradient rise on a predictable cache output while arranging content chances aimed at the expected best result. In subsequent studies, many research projects [17–21] strengthened these basic concepts. Somesula et al. [22] proposed a greedy submodular optimization method which incorporates user preference prediction and clustered mechanism for cooperative content caching. In addition, they [23] jointly optimized the service placement and the request routing to maximize the time utility and presented a greedy rounding-based service caching method and a randomized rounding-based request routing method. Hu et al. [24] employed a many-objective evolutionary cooperative caching method which jointly optimizes delay, load balance, offloaded traffic and prediction accuracy for cloud-edge-end collaborative IoT networks.

While many of these traditional methods can provide real-time cache instructions, making accurate decisions in large-scale networks becomes difficult for these conventional caching approaches given the rapid expansion and constant evolution of multimedia service demands within MEC.

2.2 Deep/reinforcement learning methods

Recently, the academic community has delved deeply into cache optimization strategies, leveraging both machine learning and deep learning for MEC cache enhancement. For instance, Li et al. [10] employed a neural network for anticipating subsequent user requests, preloading these onto the optimal edge node to mitigate issues of latency and operational costs in video caching. Another study by Li et al. [11] introduced the Edge Collaborative Cache (CVC) framework, integrating a request prediction tool grounded in federated learning along with a cooperative cache decision strategy to boost hit rates and trim user latency. Anselme et al. [12] devised a deep learning-based cache decision method, focusing on reducing content download delays by capturing passenger attributes. Somesula et al. [25] focused on the cache placement problem considering the mobility and velocity of user devices and random contact duration, and proposed a reinforcement learning-enabled caching approach to tackle the problem.

While these methodologies, which preload content based on historical data or specific characteristics, enhance cache efficiency, they often lean on intricate rule-based caching directives within real-time frameworks and lack efficiency in solving complex problems.

2.3 DRL methods

Transitioning towards DRL-driven real-time cache strategies can further optimize the problem-solving efficiency by combining the perception ability of deep learning and decision ability of reinforcement learning. Wang et al. [13], for instance, highlighted a real-time caching approach rooted in federated deep reinforcement learning (FADE) for IoT settings, aiming to slash network delays. Chien et al. [14] harnessed Q-learning to craft a caching system, formulating action-selection approaches for caching challenges and pinpointing optimal cache states. In similar veins, Sun et al. [26] deployed Deep Reinforcement Learning (DRL) for holistic edge resource management, and Wan et al. [27] actualized collaborative caching via layered aggregated federated learning. The potency of these online, learning-centric caching strategies is markedly bolstered by their profound environmental interaction capabilities. Somesula et al. [28] adopted multi-agent DRL for solving cooperative cache replacement problem to minimize delay with resource and deadline constraints. Beside, they developed a DRL-based mechanism [29] for cooperative caching which adopts deep deterministic policy gradient (DDPG) to enhance the long term time-saving performance and accelerated the learning process.

While numerous collaborative caching mechanisms exist, only a handful of optimization strategies account for balancing cache value against link costs based on real-world demands. Furthermore, learning-driven caching methods often struggle to meet immediate performance benchmarks since the learning process consumes a certain amount of time and resources. In this study, we introduce a novel two-stage learning-centric approach which first utilize rich cloud resources to obtain an effective and stable initial DDQN model via pre-training in the cloud and then enable each UE to rapidly adjust the model to environmental fluctuations via SARSA-based local online training.

3 System model

This section introduces the network structure of MEC systems bolstered by collaborative edge caching and elaborates on the cache utility model that factors in both cache value and link costs. We then delineate the cache placement optimization challenge within MEC. Conclusively, we demonstrate that the cache optimization issue in MEC is NP-Complete. Table 1 lists some key parameters.

3.1 Network model

As illustrated in Fig. 2, we propose the overall architecture of our system. The left side of the figure includes an introduction to the network architecture, where all cached contents are sourced from the cloud, which is the fundamental support of the entire system and also the part we aim to optimize. The right side of the figure contains a description of the CDRL approach.

Figure 2 showcases the CDRL framework's interaction with a standard edge system. CDRL's primary goal is to offer a control framework attuned to both performance and cost. This framework dynamically manages cache operations and utility setups within the edge system, striving for an optimal balance between performance and connectivity costs. Hence, CDRL employs a two-tier control framework, leveraging reinforcement learning to make informed resource management decisions in real-time. The CDRL encompasses four primary segments: a bi-level learning model, caching controller, caching operations, and utility adjustments.

- MEC: MEC consists of edge, and User Equipments (UEs), where the edge servers as the carriers of content caching. The cloud and edge are connected

Table 1 Summary of notations

Notation	Description
$N = \{1, \dots, n, \dots, N\}$	Set of BSs
$U = \{1, \dots, u, \dots, U\}$	Set of UEs
$F = \{1, \dots, f, \dots, F\}$	Set of contents
$T = \{1, \dots, t, \dots, T\}$	Set of time slots
$W_n(t)$	Utility of the BS_n
$value_{n,t}$	Cache value of BS_n when caching content f
$cost_{u,t}$	Transmission cost of BS_n when caching content f
$ReP_n(t)$	The cache replacement rate of BS_n
$\rho(t)$	Popularity of content f
$V_n(t)$	Available cache size of BS_n
C_f	Size of content f
C_n	Cache capacity of BS_n
$d_{c,n}(t)$	Cloud-edge link cost
$d_{e,n}(t)$	Edge-edge link cost
$d_{u,n}(t)$	Edge-User link cost
B_n	Network bandwidth of BS_n

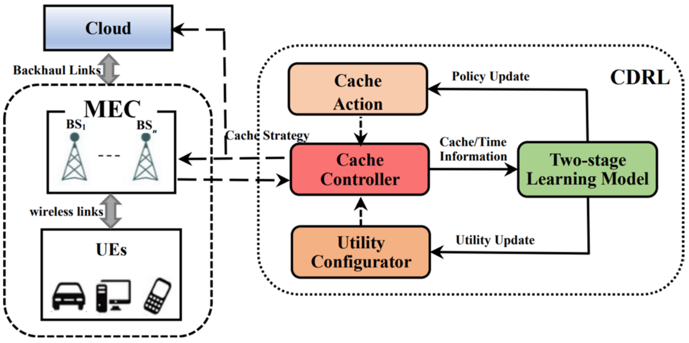

Fig. 2 The overall network architecture of the proposed system

via a backhaul link. The edge comprises a set of base stations (BSs) connected by optical fibers. For simplicity, we use the term BSs to refer to edge servers, base stations, and nodes. The BSs provide direct caching services for UEs through wireless links. It should be noted that the UEs are dynamic and can change over time.

- BS: let $N = \{1, ..., n, ..., N\}$ represent the set of BSs. Each BS has the storage capacity C_n to cache contents and the bandwidth B_n to communicate with the UE. Due to the limited capacity of BS, it is necessary to cache the most valuable content within its service scope. Cache replacement can be performed when the capacity of BS is insufficient.

- UEs: let $U = \{1, ..., u, ..., U\}$ represent the set of UEs, randomly distributed within the service range of each BS. The set of requested contents is denoted as $F = \{1, ..., f, ..., F\}$, for clarity, each piece of content, represented by f, has a size denoted as C_f. We consider this size as an integer, using megabits (Mbit) as the unit for simplicity.
- Cache strategy: given a BS set N and a content set F, the caching strategy operates as a many-to-many relationship. This means items from set F must be positioned within portions of the base station set N. Depending on the specifics of an edge network, there are multiple potential caching strategies. The BS must retrieve the content, sourcing data from its local storage, neighboring BSs, or directly from the cloud.
- Cache controller: this pivotal element of the edge caching setup gathers data about the system's varied resources and directly governs the caching operations. It conveys cache and timing details to the learning model, directly affecting the caching system by adjusting cache operations and utility settings. Facilitating to capture and serve the dynamic content requests, we consider a discrete time-slotted system where the timeline is discretized into time slots $T = \{1, \ldots, t, \ldots, T\}$ and enable time-slotted caching operations in each $t \in T$.
- Two-stage learning model: this part integrates two reinforcement learning architectures. The initial phase is an offline learning model utilizing DDQN, which processes past cache data to craft an introductory caching strategy. Subsequently, an online training system, grounded in the principles of SARSA and building upon the insights from the DDQN model, consistently acquires data to dynamically update the caching strategy.
- Utility configurator: this module fine-tunes the computation methodology of cache utility in alignment with learning outcomes, relaying the evolved utility computation back to the cache controller.
- Cache action: the caching activity is segmented into three operations: retrieving from the cloud, saving to the local BS, and transferring to other BSs. Actions related to caching are derived from the learning outcomes, and the resultant cache activity matrix is forwarded to the caching controller.

3.2 Cache value and link cost

In much of the existing literature, cache value is predominantly linked to content popularity. Yet, considering the limited capacity of edge servers and the regular cache turnover at BSs, the cache value can be influenced. As such, this paper offers the subsequent characterization for cache value:

$$\begin{aligned} &Value_{n,f}(t) = \frac{\rho_f(t) \times V_n(t)}{Rep_n(t)} \\ &\text{s.t.} \;\; \rho_f(t) \in (0, 1) \end{aligned} \tag{1}$$

where $\rho_f(t)$, $V_n(t)$ and $Rep_n(t)$ represent the popularity of content f for BSn, the available cache size, and the cache replacement rate before time slot t of the BSn, respectively.

The content's popularity is defined as the probability distribution of content requests across all UEs within the network. The $f-th$ element can be derived by determining the proportion of requests for content f relative to the total content requests in the MEC. The $\rho_f(t)$ is generated by Mandelbrot–Zipf (MZipf) distribution as [30]:

$$\begin{aligned} \rho_f(t) &= \frac{(r_f(t)+q)^a}{\sum\limits_{f=1}^{F}(r_f(t)+q)^a}. \\ \text{s.t. } & f \in F \end{aligned} \tag{2}$$

The MZipf distribution is more in line with the distribution of users' actual resource requests in the cache system, and it is widely used in the simulation of cache algorithms [31]. In the above formula, at time slot t, the resource popularity ranking of content f is $r_f(t)$, and the parameters $q \le 0$ is the plateau factor, and $a < 0$ is the skewness factor. Moreover, we assume that the content popularity changes slowly.

When the residual cache $V_n(t)$ of a BS falls below a designated threshold, prompting proactive caching, it initiates a cache replacement. Given the diverse characteristics of BSs, many may experience regular content substitutions in their cache. This recurrent cache turnover can induce increased overheads. Thus, content should ideally be proactively cached on BSs exhibiting reduced cache replacement frequencies. Denote $Rep_n(t)$ as the cache replacement rate of BSn. A high value of $Rep_n(t)$ signifies considerable costs associated with substituting the cached content on BSn, while a lower value suggests that caching content on that particular server is more advantageous. The cache replacement rate is defined as:

$$Rep_n(t) = \frac{1}{C_n - V_n}\sum_{i=1}^{k_n(t)} C_{n,i}, \tag{3}$$

where $C_{n,i}$ represents the content size of caches to be replaced in $i-th$ cache replacement on BSn. The total number of cache replacements on BSn that occurred before time slot t is denoted as $k_n(t)$, whereas $C_n - V_n$ represents the amount of used cache space on BSn. $Rep_n(t)$ serves as an indicator of the level of cache space contention on BSn, signifying that content cached on edge servers with high cache replacement rates is more prone to experience frequent replacements.

Considering that minimizing link cost is crucial in deciding the source of content acquisition, it's essential to rank content acquisition sources in the cache decision-making process. Based on the model's assumptions, link costs are ordered in ascending sequence, with the priority being local BSs, followed by other BSs, and then cloud servers. In determining the link cost for obtaining a specific resource, the content placement strategy can guide the caching decision according to this hierarchy, allowing for the respective link cost to be identified.

As depicted in Fig. 3, within the collaborative caching framework of MEC, users can potentially retrieve requested content from local BS, other BSs, or directly from the cloud. It's vital to consider the link cost associated with each retrieval source. In our model, the link cost symbolizes the network's transmission cost, derived not from a direct physical measurement but from a combination of delay and bandwidth factors. We can gauge link quality by assessing these delay and bandwidth values, and our model operates under the assumption that this link quality is known. Notably, a rise in link cost corresponds to increased delay, reduced bandwidth, and decreased network stability. Thus, we can obtain the link cost when a user requests content *f*,

$$cost_{u,f}(t) = \alpha((1 - x_f^{(e)})d_{c,n}(t) + x_f^{(e)}d_{e,n}(t) + d_{u,n,f}(t)),$$
$$\text{s.t.} \begin{cases} x_f^{(e)} \in \{0, 1\}, \\ d_{e,n}(t) \ll d_{c,n}(t), \\ f \in F, \\ n \in N. \end{cases} \tag{4}$$

where α is a positive constant coefficient, introduced for the purpose of standardizing units and magnitudes. $d_{c,n}(t)$ represents the transmission delay from the cloud to BSn for content *f*, $d_{e,n}(t)$ denotes the transmission delay from other BS to BS n for content *f*, and $d_{u,n,f}(t)$ represents the average transmission delay between all UEs and the BS n up to time slot t for content *f* [13, 29]. When a user request arrives at the local BS and results in a cache miss, the system needs to determine whether to perform cache replacement and where to retrieve the content *f* from. Due to the lower delay of retrieving content from other BSs compared to the cloud, we propose a decision variable $x_f^{(e)}$, where $x_f^{(e)} = 1$ indicates that the content *f* is stored in other

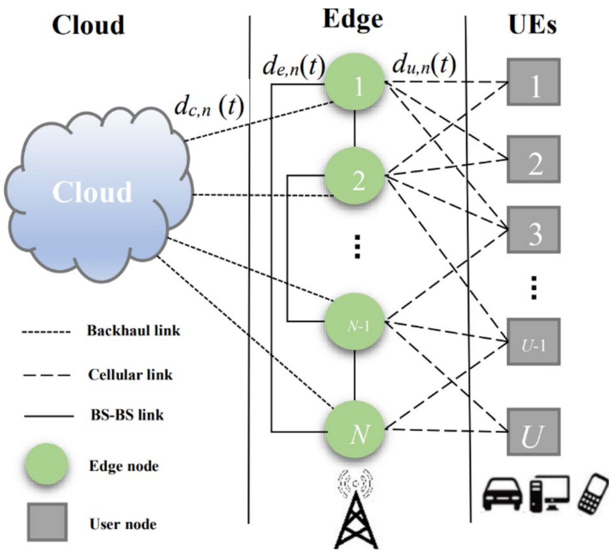

Fig. 3 MEC system topology supporting cache

BSs, otherwise $x_f^{(e)} = 0$ and the content needs to be retrieved from the cloud. We can obtain the average latency of UEs requests for content *f* prior to time slot *t*:

$$d_{u,n,f}(t) = \frac{1}{\left|U_f\right|} \sum_{t' \leq t} \sum_{u \in U_f} d_{u,n,f}(t'), \\ \text{s.t. } U_f \in U. \tag{5}$$

where U_f denotes the set of UEs that have requested content *f*, and the wireless transmission delay between BS and UE is computed using Shannon's formula [13].

3.3 Problem definition

When a user seeks particular content via BSn, potential sources encompass the local BS, neighboring BSs, and cloud repositories. Assuming efficient content searching, request dispatch, and reception processes, user preferences can typically be discerned by examining past content requests or using real-time online techniques. Typically, resource discovery and transmission leverage buffer exchange protocols and GPRS tunneling methods.

BSs are usually deployed in commercial or industrial sites; we consider an MEC scenario in a city's commercial area containing a set of *N* BSs. These BSs are scattered in the city's commercial area, and each BS has a cache utility value $W_n(t)$. At a specific time, all BSs need to cache a total of F_m contents, which are distributed within the range of *N* BSs, and there are U_m users in the area. Where F_m is a subset of F and U_m is a subset of *U*.

This paper centers around harmonizing cache value and link cost, leading to the introduction of an integrated utility function. It merges content popularity, cache capacity, and BS cache replacement frequency to define cache value, which is then offset against the link cost. This utility function offers adaptability based on specific needs. The overarching aim of this caching framework is to optimize the aggregate cache utility, ensuring compliance with standard caching prerequisites and attaining peak caching efficiency within MEC. When a BS is requested to cache content, set the cache utility of the server as $W_n(t)$, the cache value as Valuen,*f*(*t*), and the link cost as $cost_n$,*f*(*t*). The cache utility model of a single BS is obtained as follows,

$$W_n(t) = x_{n,f}((1-\sigma)Value_{n,f}(t) - \sigma cost_{u,f}(t)), \\ \text{s.t. } \begin{cases} x_{n,f} \in \{0,1\}, \\ \sigma \in [0,1], \\ f \in F_m, \\ F_m \subset F, \\ n \in N \end{cases} \tag{6}$$

where $x_{n,f} = 1$ represents the content *f* needs to be cached in BSn, $x_{n,f} = 0$ otherwise. Parameter σ represents the weight of proactive caching link cost. An elevated weight value typically reduces the total link cost but might compromise the aggregate hit rate, steering the system away from proactive caching. The weight factor offers tangible benefits in real-world scenarios by facilitating the adjustment of caching system performance. For example, in settings characterized by rapid content turnover, such as sporting events or holiday shopping locales, diminishing the weight value could enhance caching efficiency. On the flip side, in scenarios with minimal caching demands, augmenting the weight can minimize proactive caching. Given that the link burden is insubstantial, sourcing content outside of the local base station can maintain satisfactory user experience, simultaneously curbing both link and energy consumption. According to formula (6), the high caching utility of the BSn indicates that the BSn has a higher caching value and a lower link cost when caching content. Therefore, in the MEC environment, the primary objective of caching is to maximize the caching utility of all BSs,

$$max \sum_{n \in N} \sum_{f \in f_m} W_n(t) \tag{7}$$

3.4 NP-complete proof

If the Set Cover (SC) problem, known to be NP-complete, can be mapped to the cache placement challenge, it would establish the decision problem of cache placement as NP-complete. Initially, we identify two sets: Set $G = g_1, g_2, .., g_I$ and its counterpart, Set $H = h_1, h_2, .., h_J$, which contains subsets of G. Each member of H, denoted h_j, is associated with a weight. The SC problem's goal is to choose a subset from H such that its union encompasses all elements in G while either minimizing or maximizing the cumulative weight.

To adapt the SC problem to our cache placement scenario, proceed as follows: first, fix the number of content items at 1. Allow $G = g_1, g_2, ..., g_I$ to symbolize all BSs, while $H = h_1, h_2, ..., h_J$ captures the distinct configurations of content caching across BSs. For instance, if $h_1 = 1, 3, 5$, it implies content is cached at BSs g_1, g_3, and g_5. Similarly, $h_2 = 2, 4, 6$ signifies caching at g_2, g_4, and g_6. Now, equate the weights in the SC problem to our cache value and link cost.

Therefore, our problem can be formulated as a SC problem. Let's use $V = v_1, v_2, ..., v_N$ to represent all edge servers. Each member of the set $C = c_1, c_2, ..., c_K$ denotes potential caching decisions for content across all BSs. Each caching decision c_k, carries an associated cache value and link cost. The net benefit from a caching decision c_k is the difference between its cache value and link cost. The goal is to pinpoint a subset $C' = c_1, c_2, ..., c'_K$ from C such that the union of all members in C' captures V, aiming to optimize the aggregate content caching benefits.

The set cover problem can be mapped to the data caching placement dilemma using the aforementioned transformation. By weaving real-world parameters into the set cover problem's structure, it's clear that these parameters align with the set cover's generic variables. At its core, our challenge mirrors the elements of the set cover problem, making it a variant of the same. As a result, the data caching placement issue is deemed NP-complete.

4 Collaborative two-stage learning strategy for cache algorithms

4.1 Two-stage learning for state, action and reward

The proposed two-stage learning framework integrates the pre-training capabilities of the DDQN algorithm with the online adaptability of the Deep SARSA algorithm to tackle proactive caching challenges in BSs. By merging these two DRL methodologies, the framework capitalizes on the distinctive advantages of each. Specifically, DDQN aids in mitigating overestimations of Q-values and sensitivity to miscalculations, whereas SARSA bolsters model stability by emphasizing the current policy over historical ones. The utilization of DDQN in the pre-training phase not only speeds up the learning process but also lays a solid foundation for model performance. In contrast, deploying SARSA for online training equips the model to adjust to environmental fluctuations, thanks to its emphasis on present policies and adaptability to evolving scenarios. Collectively, the integration of DDQN's pre-training with SARSA's online training enhances the model's efficacy and resilience, ensuring superior adaptability and broader application potential.

We model the content cache placement process in BSs as Markov decision process (MDP) [13]. The state of caches, cache action, utility configuration, and feedback rewards are as follows.

State of caches: During each decision epoch i, the cache status of the content is represented by $s^f_{i,n}$, and $s^f_{i,n} = 1$ means that BSn caches content f, otherwise. Furthermore, we use sfi,u to denote the request state from UE u. Similarly, $s^f_{i,u} = 1$ means UE u sends a request for content f, $s^f_{i,u} = 0$ otherwise. At each decision epoch i, the cache state of BSn and the set of request states within its service range are combined as its state s_i,

$$s_{i,n} = \{s^{F_n}_{i,n}, s^{F_u}_{i,u}\} \tag{8}$$

F_n and F_u denote the sets of cached contents in BSn and content requests from UEs, respectively.

Cache action: in order to adapt to the continuous changes in a dynamic environment, a BS may choose which content to replace and decide where to process the request. The first option is to process the request at the local BS, $a^{local}_i = \{a^{local}_{i,0}, a^{local}_{i,1}, ..., a^{local}_{i,F}\}$ denoted by a local function. If $a^{local}_{i,f} = 1$, it indicates that the content f requested by the UE needs to be replaced in the local BS, and the content request is executed locally at the BS. The second option is to route the UE's request to a neighboring BS if the requested content f is not cached locally at the BS. This is achieved through a neighboring processing function denoted by $a^{neibour}_i = \{a^{neibour}_{i,1}, a^{neibour}_{i,2} ..., a^{neibour}_{i,N}\}$. If $a^{neibour}_{i,n} = 1$, it indicates that the BSn is processing the current user request. The third option is to process the request in the cloud, denoted by aicloud. If the UE is unable to obtain the content f from the local and neighboring BSs, then $a^{cloud}_i = 1$, indicating that the request is processed in the cloud,

$$\phi(s_n^i) = \{a_i^{local}, a_i^{neibour}, a_i^{cloud}\} \tag{9}$$

Feedback reward: when the local BS takes action A, the system receives feedback rewards, which are composed of two components: the overall cache value and the link cost. To this end, we designed two reward functions, aimed at maximizing the system's total reward and ensuring maximum utility in the dynamic process.

$$\begin{cases} R_{n,value}(s_{i,n}, \phi(s_{i,n})) = \sum\limits_{f \in F_n} value_{n,f} \\ R_{n,cost}(s_{i,n}, \phi(s_{i,n})) = -\sum\limits_{f \in F_n} cost_{u,f} \end{cases} \tag{10}$$

Formula (10) proposes two rewards for the system. The first is a positive cache value function defined by Eq. (1), indicating that caching proactively to increase cache value provides positive rewards. The second is a negative link cost function defined by Equation (4), indicating that caching should not be performed blindly and should be combined with link cost considerations.

As illustrated in Fig. 4, the blue rectangles denote the shared lower-level network module, while the squares represent branches in the higher-level network. The shared network module is a Long Short-Term Memory (LSTM) with internal unit size of 512. The high-level network branches consist of two fully-connected layers, each approximating $Q_{n,value}$ and $Q_{n,cost}$, respectively.

4.2 Pre-training and online training process

Pre-training aims to enhance the initial performance and consistency of the DRL model. For this purpose, we employ the DDQN algorithm, a refined version of the Q-learning algorithm. It utilizes dual Q-networks to decrease bias in Q-value estimations: one for determining actions and another for evaluating action values. This

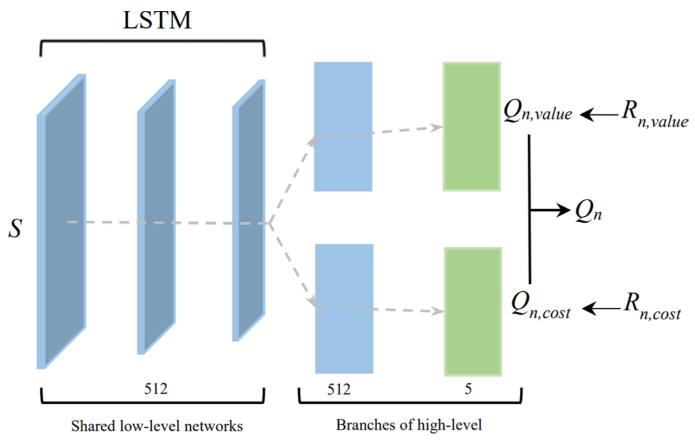

Fig. 4 Proposed dual reward architecture

setup circumvents the instability stemming from overestimating Q-values in the initial phase. Notably, DDQN proves effective when pre-trained on the cloud, given the accessibility to comprehensive cache information [30].

For online training at the edge nodes, we adopt the SARSA algorithm. This method stands out due to its on-the-fly learning of the optimal policy, paired with its minimal computational and memory demands. Additionally, SARSA offers commendable convergence and consistency, adeptly sidestepping issues like unsteady learning from biased value functions. Stationing the model at the edge minimizes communication exchanges between the edge and cloud, thus promoting efficient and unwavering online learning [32].

Figure 5 illustrates the collaborative two-stage learning framework. During the DDQN pre-training phase, global historical data provides the state matrix, action matrix, and reward matrix. By training both the primary and secondary networks of DDQN, and by reducing the loss function L across each iteration, an initial caching strategy is formulated in the cloud. Once this strategy is established, both the devised caching strategy and the primary DDQN network are dispatched to every BS. Each BS, equipped with this neural network, employs the SARSA algorithm for dynamic caching, aiming to achieve an optimal real-time caching decision. In the online training phase powered by the SARSA algorithm, the system executes an action a_{t+1} before recognizing the state s_{t+1} and the associated reward r_{t+1}. This dynamic caching method, driven by the SARSA algorithm, can adaptively modify the edge cache state in real-time. To maintain data training integrity, the experience replay method stores all sample points, eliminating potential biases through randomized sampling.

The SARSA algorithm dynamically shapes the caching strategy. In contrast to the DDQN algorithm, where the agent takes action a_{t+1} based on state s_{t+1} and then receives the reward r_{t+1} before updating the Q-function, the SARSA algorithm's approach to predicting future rewards relies on the real reward from the taken action. The SARSA algorithm consistently updates the policy and adjusts its approach for the Q-function. By using experience replay, the agent can

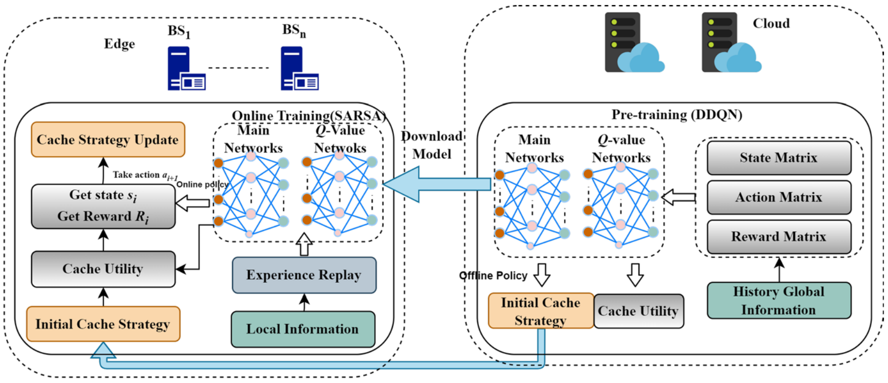

Fig. 5 Collaborative two-stage learning frameworks

effectively navigate through past states, avoiding sudden learning disruptions or deviations. Each learning step mirrors the weight of the online network, strengthening the neural network's stability compared to traditional online Q-learning. The total reward value shifts when the server conducts a caching action. Based on (10), we can obtain the system's reward as follows:

$$R_n(s_{i,n}, \phi(s_{i,n})) = \sigma R_{n,value}(s_{i,n}, \phi(s_{i,n})) + (1 - \sigma)R_{n,cost}(s_{i,n}, \phi(s_{i,n})), \tag{11}$$

where ϕ symbolizes the caching strategy, depicted as a blend of actions. Drawing from the formulas (10) and (11), the aim is to optimize the long-term reward starting from an initial cache state s_1:

$$R(s, \phi) = \max_{\phi} E\left[\lim_{I\to\infty} \frac{1}{I}\sum_{i=1}^{I} R(s_i, \phi(s_i))|s_1 = s\right], \tag{12}$$

where $R(s_i, \phi(s_i)) = \sum_{n\in N} R_n(s_{i,n}, \phi(s_{i,n}))$ represents the over rewards of all BSs. Additionally, a single-agent infinite-horizon MDP using a discounted utility (12) is often used to estimate the expected infinite-horizon unrecognized value, especially when γ is within the range [0, 1). We conduct the pre-training phase in the cloud with DDQN. As a result, the iterative formula for the Q-value can be expressed as:

$$Q_{i+1}(s, \phi) = Q_i(s, \phi) + \alpha_i(R(s, \phi) + \gamma \times \max_{\phi'} Q_i(s', \phi') - Q_i(s, \phi)) \tag{13}$$

where $\alpha_i \in [0, 1)$ is the learning rate, $\gamma \times \max_{\phi'} Q_i(s', \phi')$ signifies the accumulated reward value achieved after executing strategy $\phi(s)$ in the state s, transitioning to state s', and then choosing the optimal strategy ϕ. The discount factor, denoted by γ, can progressively reduce the value of forthcoming rewards. Once the server carries out the operation $\phi(s_i)$, the current state s_i transitions to s_{i+1}, yielding an immediate reward $R(s_i, \phi(s_i))$. This equation depicts the total reward value post operation $\phi(s)$ subtracted from the previous period's total reward value.

The hybrid Q-value structure proposed in this paper is used for caching policy selection, as shown in Fig. 4. The final Q-value is calculated by combining the Q_{value}, Q_{cost}, and network weight ω, as shown in the following equation:

$$Q(s, \phi) := \sigma Q_{value}(s, \phi) + (1 - \sigma)Q_{\cos t}(s, \phi) \tag{14}$$

The approximate optimal Q-value can be obtained according to the following equation:

$$Q(s, \phi) \approx Q((s, \phi);\omega_i) \tag{15}$$

where ω_i denote the neural network weights. Given that the two neural networks have identical low-level weights (albeit for different Q-values), it's more beneficial than starting training from scratch. In deep Q-networks, the Q-learning update procedure can be straightforwardly applied to a multi-layer neural structure. Here, the neural networks act as substitutes for the table of state-action pairings. It should be noted that the pre-training of DDQN is done by taking random minibatches from the experience replay pool, including the current s_i, a_i, R_i, and then using a target network to calculate the maximum Q-value under $s_i + 1$, thus reducing the overestimation problem. The online training of SARSA is done by taking random minibatches from the experience replay pool, including the current $si, ai, Ri, si + 1, ai + 1$, and then using the current network to calculate the Q-value under s_{i+1}, thus achieving on-policy learning. The loss function of DDQN and SARSA are mean squared error (MSE). The traditional TD error is adapted into a revised loss function, denoted as $L(\omega_i)$. Subsequently, the Q-networks are trained by aiming to minimize this loss function.

$$L(\omega_i) = E_{(s,\phi)}\left[Y_i^Q - Q(s,\phi;\omega_i)^2\right]. \tag{16}$$

The original target in DDQN is Y_i^Q:

$$Y_i^Q = R + \gamma\, Q(s, \arg\max_{\phi'} Q(s', \phi'; \omega_i); \omega_i^-), \tag{17}$$

where ω_i^- denotes target network weights. Unlike the conventional DDQN, which maintains two value functions and updates each function using a value from the alternate function for the subsequent state, this study introduces an innovative original Q function by structuring the neural networks in layers. This newly proposed Q function is then updated collaboratively with the target Q function. Moreover, we can obtain the gradient updates of ω_i by $\nabla_{\omega_i} L(\omega_i)$ as follows:

$$\nabla_{\omega_i} L(\omega_i) = E_{(s,A)}[(Y_i^Q - Q(s,A;\omega_i))\nabla_{w_i} Q(s,A;\omega_i)]. \tag{18}$$

To curb the overestimation in every episode during both pre-training and online training and to reduce the loss function $L(\omega_i)$, we introduce the "Two-stage DRL algorithm for caching," delineated in Algorithm 1. This method doesn't merely employ DDQN for pre-training followed by SARSA for dynamic training. Instead, after the DDQN pre-training phase, an action is chosen with a probability ε during dynamic training, where ε is set to 0.1. This means there's a chance of taking a random action, as well as a likelihood of opting for an action that maximizes the Q-value.

Procedure 1 (Pre-training process): Firstly, Begin with the initialization of the pertinent DDQN parameters within the experience replay, facilitating the acquisition of weights for both the primary and target networks. (lines 1–3). Then, using comprehensive historical data, both primary and target networks are trained to determine the foundational caching policy. Capture and store the identified states, actions, rewards, and value functions within the experience replay (lines 4–6). Finally, dispatch the initial neural network parameters and the foundational caching policy to every BS.

Procedure 2 (Online training process): Upon receiving a content request at the BS, determine the caching action through the $\varepsilon - greedy$ method initially. Following this, reset and record both the cache state and the associated policy within the experience replay (lines 9–13). While the online network assesses the greedy action, the target network is employed for its value approximation. Then, SARSA is executed to train the caching process and update all parameters (lines 14–16).

The complexity of Algorithm 1 primarily considers the number of state transitions and backpropagation steps in dual-depth reinforcement learning [31]. The difference from [31] is that we employ the DRL network twice to optimize the learning objectives. Pre-training can be conducted before the deployment of the strategy, as it only requires historical information, and its training process does not interfere with actual cache execution. Therefore, the execution efficiency of the online SARSA strategy executed after deployment is not significantly different from that of other DRL methods.

Algorithm 1 Two-stage deep reinforcement learning algorithm for caching

```
1: Pre-training process (DDQN in cloud):
2: Initialize reply memory $\Omega$.
3: Initialize action-value function $Q$ with random weights $\omega_1$.
4: Initialize target action-value function $Q$ with weights $\omega^- = \omega_1$.
5: Pretrain the main and target network with $\langle s_i, \phi(s_i)\rangle$ and the corresponding $Q(s_i, \phi(s_i); \omega_i)$ by equations (13) and (17).
6: Keep the neural network parameters $\omega$ and obtain the pre-caching strategy.
7: Send $\omega$ and pre-caching strategy to each BS.
8: Online training process ( in each BS):
9: for each episode do
10:     for each $f \in F$ do
11:         if BSn receive a request content $f$ then
12:             With probability $\epsilon$ select a random action $\phi_i$, otherwise select $\phi_i = \arg\max_a Q(s_i, \phi(s_i); w_i)$.
13:             Execute action $\phi_i$ and observe reward $R_i$ by equations (11), (12) and caching state $s_{i+1}$.
14:             Set $S_{i+1} = S_i, \phi_i, R_{i+1}$ and $\phi_{i+1} = \phi(s_{i+1})$.
15:             Store transition $(s_i, \phi_i, R_i, s_{i+1}, \phi_{i+1})$ in $\Omega$.
16:             Sample random minibatch of transitions $(s_i, \phi_i, R_i, s_{i+1}, \phi_{i+1})$ from $\Omega$.
17:             Update $\omega_i$ by minimizing the gradient by equation (17).
18:             Update the caching state $s_i$.
19:         end if
20:     end for
21: end for
```

4.3 Cache placement algorithms

The problem of edge cache placement is an NP-complete problem. Taking into account the value of cached content and link costs, we designed an Online caching placement algorithm starting from user requests. The key to the algorithm lies in the decision-making process of caching, which must ensure the long-term increase in the target value of global cache utility.

Algorithm 2 presents the pseudo-code of the cache placement algorithm based on dual-depth DRL. Firstly, we construct reward functions R_{value} and R_{cost}, initialize an empty hash map to store cache decisions, and initialize the Q network (as shown in lines 1–2). If BS *n* receives a request for content f and has cached it, the content is directly provided (as shown in lines 4–6). In the case where BS *n* has not cached the requested content, if the capacity of BS *n* that needs to cache the content is not full, the content f is directly cached, and then provided to UE, and the status s_i is updated synchronously (as shown in lines 7–9). If BS *n* has not cached the requested content and the capacity is full, an online policy is executed to train the cache process and update the cache policy (as shown in lines 11–12). Finally, the caching strategy is put into the hash map (as shown in lines 15–16).

Algorithm 2 Optimize placement results

```
1:  Input: Content set $F$, BS set $N$.
2:  Output: Optimize placement results HashMap$< F_O, N_O >$.
3:  Construct the reward function $R_{\text{value}}$ and $R_{\text{cost}}$.
4:  Initialize HashMap$< F_O, N_O > \leftarrow \emptyset$
5:  for each $f \in F$ do
6:      UEs request content $f$ to BS $n$.
7:      if the cache state of BS $n$ is $s_{i,n}^{f} = 1$ then
8:          break.
9:      else
10:         if BS $n$ not full then
11:             Cache the content $f$ in BS $n$.
12:             Update the state $s_i$.
13:         else
14:             Replace the content of BS $n$ for content $f$ by online cache policy of
    two-stage deep reinforcement learning algorithm.
15:             Update the state $s_i$.
16:         end if
17:     end if
18: end for
19: HashMap$< F_O, N_O > \leftarrow$ Overall cache decision
20: return HashMap$< F_O, N_O >$
```

5 Experiment analysis

In this section, the proposed algorithm will be evaluated from the perspective of comprehensive utilization of edge servers. Four base stations (BSs) were considered for simulation, with a maximum coverage radius of 250 m. The channel gain was modeled as $g_{u,n} = 30.6 + 36.7lgl_{u,n}$, $l_{u,n}$ is the distance of the UE u and BSn, the channel bandwidth B_n was 20 MHz, and the transmission power was $40W$. The learning model construction was based on Python 3.8 and Pytorch 1.13.0, The model uses a shared bottom layer network of 512-size LSTM, with two additional fully connected layers of size 512 and two decision layers of size 5 for different Q-value outputs. The Adam optimizer is used for optimization. Other experimental parameters are presented in Table 2. The parameters are set according to [13].

Table 2 Experimental parameters

Parameter name	Value
Content numbers F	10,000 100,000
BSs numbers N	4
BS radius	250
Channel bandwidth	20 MHz
UE fluctuations	0.8 1.2
Noise power	−95 dBm
Content size C_f	(0, 10] Mbit
Delay of BS-Cloud $d_{c,n}$	200 ms
Delay of BS-BS $d_{e,n}$	20 ms
Pre-training epoch	200
Capacity of replay memory	1000
Minibatch	32
Reward decay γ	0.9
State transition probability ϵ	0.1
Learning rate α	0.01
The period of replacing target Q network	200

The simulation environment consisted of a primary Dell server acting as the cloud. This server was equipped with a 20-core CPU, 128 GB RAM, and 2TB of storage capacity. In addition, we utilized four Dell edge servers, each boasting a 4-core CPU, 8 GB RAM, and 500 MB storage capacity. Throughout the simulation, pertinent details for every request were diligently logged: this encompassed request category, the entire completion duration, associated link expenses, and the determination of a cache hit or miss. Such recorded metrics facilitate the computation of essential performance indicators like the cache hit ratio, mean latency, count of backhauls, and overall link costs.

Our experiments employed a comprehensive offline dataset sourced from the Xender application, a prominent content-sharing platform extensively adopted in India. The dataset spanned the month of May 2018, specifically from the 1st to the 31st. Within this timeframe, the dataset revealed activities from 472 to 833 distinct user pathways, encompassing the sharing of approximately 149,262 files and generating between 291,255 and 1021 requests [33].

5.1 Experimental value of weight parameter

In this experiment, we manipulated the number of episodes and content to investigate how these parameters impact the overall cache performance of each algorithm. As stated in [14], the delay between the edge server and the user is directly proportional to the Euclidean distance. In our simulation, we assume that each kilometer introduces a one-millisecond delay, given that the user and edge server locations are known.

Our initial analysis focused on the weight σ of link cost. Specifically, we varied the number of contents (10,000 and 100,000), and the value of σ (ranging from 0 to 1). We evaluated the average cache hit rate and overall link cost over 10 periods (1000 online iterations). Figure 6a, b indicates that the hit rate and link cost decrease with increasing. We found that changes to the value have a significant impact on cached results. Consequently, our proposed dual learning framework was used to determine the final experimental outcomes.

5.2 Comparison and analysis of experimental results

To appraise the efficacy of our suggested cache placement methodology, we benchmarked it against several sophisticated caching strategies, detailed as follows:

1. Least Recently Used (LRU): this scheme prioritizes the removal of content that hasn't been accessed for the longest duration.
2. Least Frequently Used (LFU): under this strategy, content that is least accessed or fetched gets earmarked for replacement.
3. Local greed for content popularity (GREEDY): in this approach, the content exhibiting the lowest popularity metrics is the primary candidate for replacement.
4. Deep Q-network (DQN): this caching mechanism hinges on decisions derived from a singular deep Q-network.
5. Federated deep-reinforcement-learning-based cooperative edge caching (FADE) [13]: treat each BS as a client, regard the cloud as a central aggregator, and generate caching strategies to minimize content access delay using the federated learning approach.
6. Fuzzy reinforcement learning based cache placement (FRCP) [25]: a reinforcement learning-based caching placement strategy focusing on saving delay.
7. Deep deterministic policy gradient (DDPG) [29]: an optimized DRL caching method model focusing on long-term time-saving.

Figures 7 and 8 provide illustrative comparisons regarding average content access latency, hit rate, backhaul traffic, and cumulative link cost. A visual examination of the graphs suggests that both GREEDY, DQN, FADE, FRCP and DDPG outpace

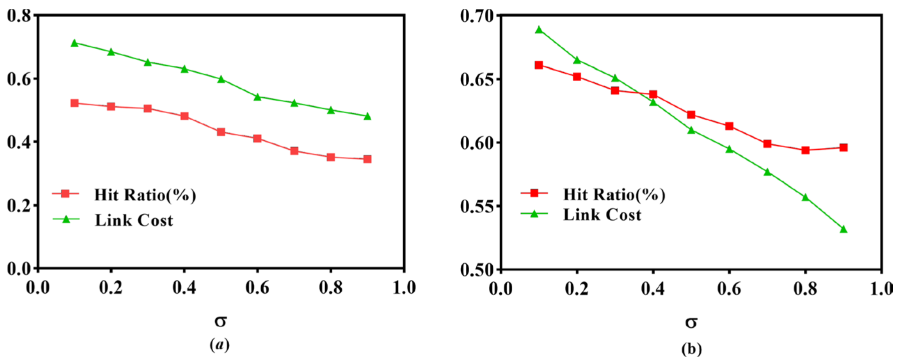

Fig. 6 Performance evaluation when **a** the content number is 10,000, and **b** the content number is 100,000

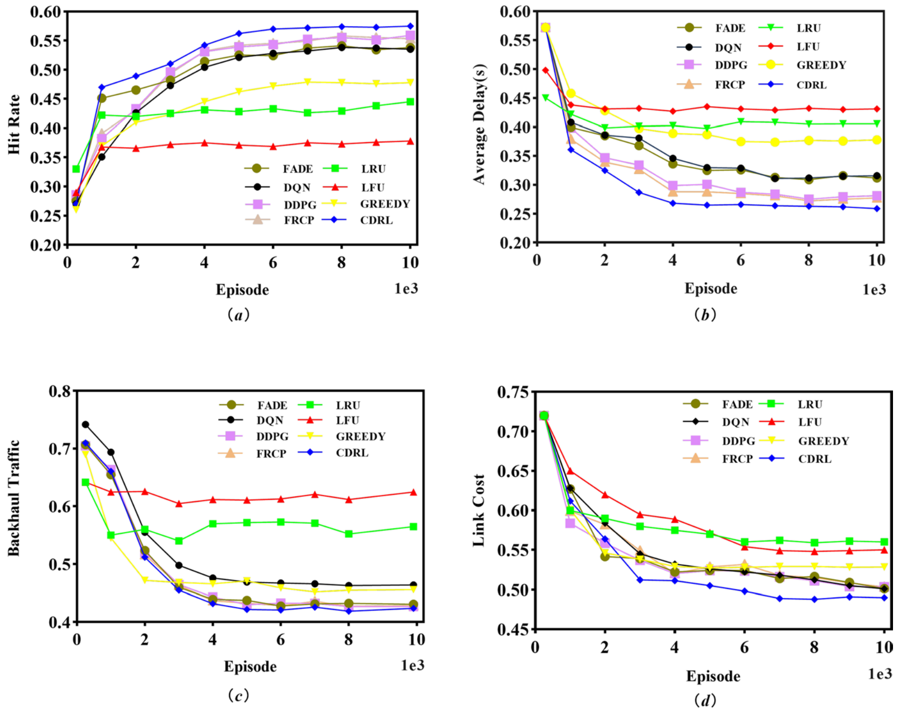

Fig. 7 Performance evaluation of the **a** hit rate, **b** average delay, **c** traffic with respect to time, and **d** link cost respect to time

traditional methods like LRU and LFU in terms of performance. However, when juxtaposed against these, the merits of CDRL become even more discernible and pronounced.

The manuscript delineates the symbol ε as equivalent to $f_j(t)$, where the determination of $f_j(t)$ is contingent on real-world circumstances. Specifically, we characterize $f_1(t)$ to follow a distribution reminiscent of the Poisson distribution:

$$f_1(t) = 1 - \frac{5.5^t}{t!}5.5^{-8} \tag{19}$$

where the value of t is [1, 10].

Figure 7 illustrates the temporal evolution of caching strategy performance. As the duration increases, the overall system tends to become stable. The CDRL strategy is compared with classic caching methods such as LFU, LRU, GREEDY, DQN, DDPG and FRCP. From the Fig. 7, it can be observed that the CDRL strategy outperforms other methods in terms of cache hit rate, delay, backhaul traffic, and overall link cost. This is because CDRL has a better real-time caching strategy when dealing with dynamically changing requests. The GREEDY strategy is a local greedy algorithm based on content popularity. Although it is more

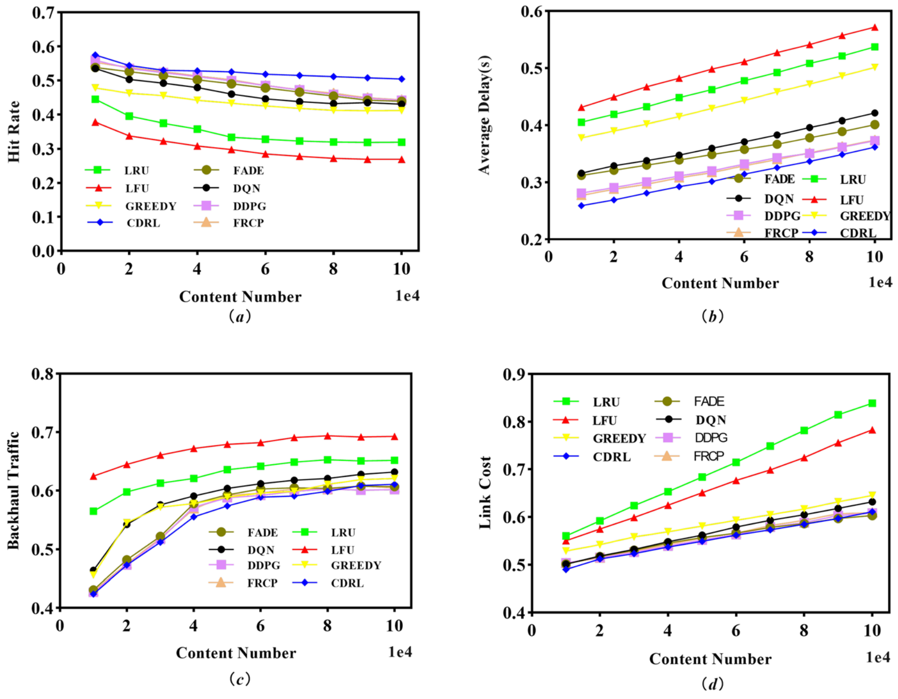

Fig. 8 Performance evaluation of the **a** hit rate, **b** average delay, **c** traffic with respect to time, and **d** link cost respect to content number

advantageous than traditional LFU and LRU, it still has a significant gap compared to the DQN and CDRL strategies based on real-time policies. The CDRL strategy is superior to DQN in two aspects: firstly, the CDRL pre-training process can obtain a better initial policy through DDQN, and secondly, the SARSA algorithm has stronger real-time interactivity during online caching. The reason why CDRL outperforms DDPG and FRCP is that the pre-training in the cloud enables the CDRL model to achieve better performance, and it quickly infers online caching strategies in the BS through the SARSA algorithm. Furthermore, the weight ε Poisson-like distribution matches the dataset better. the CDRL approach, a higher number of requests find a match in the cache node, which leads to a decrease in the average resource access link cost and subsequently enhances the user experience. As evident from Fig. 7a b, the CDRL approach exhibits a notable enhancement in hit rate when juxtaposed with the other methodologies (LRU, LFU, GREEDY, DQN, FADE, DDPG and FRCP) is 22.1, 28.8, 19.6, 7.4, 6.9, 2.9, and 3.9% respectively. The delay reduction is 35.1, 31.3, 29.5, 22.0, 20.5, 8.5 and 6.9%. As illustrated in Fig. 7c, our proposed method is adept at offloading a greater amount of backhaul services, exhibiting improvements of 14.2, 21.1, 10.1, 6.3, 3.1, 1.9, and 1.5% over the other algorithms. The computation of the comprehensive link cost for each cache, as highlighted in Fig. 7d, reveals an average decline in the overall link cost by 9.4, 9.3, 3.3, 1.9, 1.2, 0.9, and 0.8% respectively when compared to the other strategies.

Figure 8 delineates the performance trajectory of the BS when juxtaposed with policies like LFU, LRU, and GREEDY, over a spectrum of content quantities from 10,000 to 100,000. Observations from Fig. 8a–d highlight the pronounced supremacy of the CDRL strategy over its counterparts in latency metrics. Figure 8a reveals that when content count is on the lower end, the CDRL algorithm stands out in performance against LRU, LFU, GREEDY, and DQN. Yet, an uptrend in content numbers-with a static server count-correlates with a dip in the overall hit rate. Under such circumstances, the hit rate of CDRL burgeons by 32.9, 53.9, 25.1, 11.8, 10.4, 7.8 and 8.3% respective to the other methods. Figure 8b underscores that in scenarios of limited content, latency across algorithms remains minimal. But, as content swells, the need for frequent content rotation intensifies under diverse policies, escalating latency. Herein, the CDRL approach trims the average latency by margins of 26.4, 29.2, 21, 18.6, 14.8, 5.3, 4.8% when stacked against other methods. In Fig. 8c, a surge in backhaul traffic emerges as an inevitable consequence of augmenting content numbers. Within this context, the CDRL approach curtails backhaul traffic by 7.9, 14.3, and 6.6, 5.8, 0.2, 0.5, and 0.2% in relation to other strategies. Figure 8d posits that during phases of scant content, disparities in link costs across algorithms are negligible. However, with a rising content count, every algorithm's link cost elevates due to amplified link burdens. Contrarily, the CDRL algorithm's link expense diminishes by 10.1, 9.1, 7.7, 2.9, 2.7, 0.9 and 0.1% when paralleled with its peers. Although other reinforcement learning-based caching strategies also achieve good results, they lack the capability to fully utilize cloud-edge computing resources. The proposed algorithm, CDRL, has already established a better initial strategy in the cloud, and can quickly update its caching policy using the SARSA algorithm when facing more complex environments, giving CDRL a distinct advantage. In essence, these insights affirm the robustness and efficiency of the CDRL strategy across diverse content volume scenarios.

Finally, the values of exploration probability ε and scene function $f(t)$ are discussed. Figure 9 shows the performance comparisons for CDRL in terms of the hit

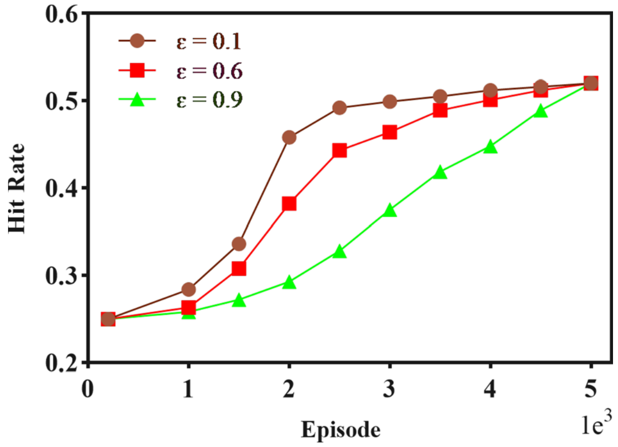

Fig. 9 Performance of the hit rate under different exploration probabilities

rate with different exploration probabilities $\varepsilon = 0.1$, $\varepsilon = 0.6$, and $\varepsilon = 0.9$. Exploration probabilities can greatly affect caching efficiency, so a wide range of experiments are needed to determine the best exploration probabilities before deploying a caching policy. Moreover, we consider three different scenarios, which are shopping malls, factories, and schools. And accordingly, we propose three different scenario functions: $f_1(t)$ denotes cache request rate model for malls scenario. The factory is distinctly different from the commute, so the cache request rate model for the factory scenario is defined as

$$f_2(t) = \begin{cases} 0.1\sin\left(\frac{\pi}{2}t\right) + 0.4, 3 \le t \le 10, \\ 0.1\sin\left(\frac{\pi}{2}t\right) + 0.12, \text{otherwize.} \end{cases} \tag{20}$$

Assuming a typical commute window from 3 to 10, there's a surge in cache request rates during working hours, while it drops considerably during off-peak times. In the context of schools, this paper defines the cache request rate to oscillate randomly within a specified time frame, as:

$$f_3(t) = \frac{1}{3}\left(\sin\left(\frac{\pi}{2}t\right) + 0.5\right). \tag{21}$$

We conducted experiments to evaluate the cache hit rate and link cost across three $f_j(t)$ functions under identical scenarios, as depicted in Fig. 10.

As shown in Fig. 10, changing the weights ε results in increased fluctuations in cache hit rate and link consumption, which do not comply with the special scenarios proposed in Eqs. (19) and (20). Therefore, it is necessary to examine the local characteristics of different scenarios to develop better weights that ensure cache performance.

From the aforementioned experimental outcomes, it's evident that the system's overall performance and link cost metrics can be dynamically tuned based on real-world conditions. For demanding performance scenarios, the parameter ε can be decreased, while it can be increased when there's a preference for reduced link

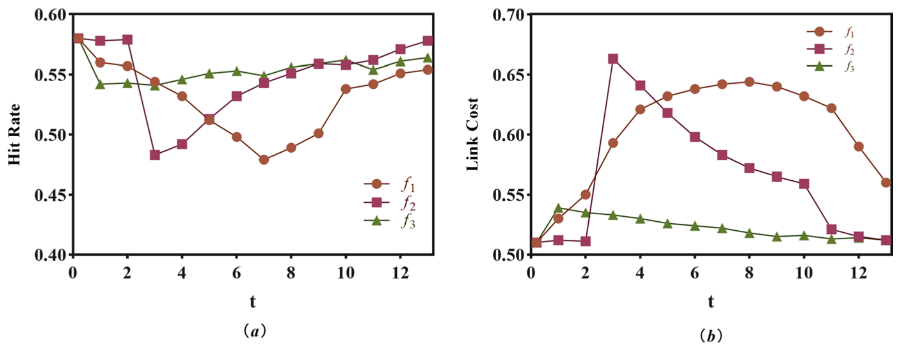

Fig. 10 Hit rate (**a**) and link cost (**b**) for different scenarios and scene functions

costs. Given the holistic accounting of caching costs in performance evaluation, our method yields superior results compared to other prominent cache replacement algorithms.

6 Conclusion and outlook

In this study, we present a novel cache placement methodology tailored for MEC. Our approach hinges on an integrated system framework that leverages global variable utility and a two-tier DRL mechanism. We delve deeply into the intricacies of the general utility function and the nuances of the two-stage DRL paradigm. The utility metric amalgamates cache value, link costs, and variable parameters, while our reinforcement learning model interweaves elements of DDQN and SARSA. Empirical evaluations underscore the potency of our approach in the edge computing milieu, particularly when juxtaposed against other established caching techniques.

Yet, as promising as our findings are, there remains a swath of areas ripe for refinement. Key among these is the enhancement of the utility model. Currently, content popularity leans on the Zipf distribution. However, the mercurial nature of content popularity, influenced by time and evolving events, demands more nuanced treatment. Enriching our model by harnessing machine learning and deep learning methodologies to predict content popularity with heightened precision could usher in marked improvements in overall cache utility. Further, the link cost model requires recalibration. The paper's link cost estimation rests partly on a streamlined predefined value, which doesn't fully encapsulate the intricacies of real-world networks, particularly those entwined with bandwidth allocation and fluid network traffic. Thus, devising a more robust framework to ascertain link costs remains a pressing priority.

Author Contributions Hui Xiao wrote the main manuscript text. Xinyu Zhang and Yang Liang helped conduct the experiments. Zhigang Hu and Meiguang Zheng put forward research ideas. All authors have reviewed the manuscript.

Declarations

Conflict of interest The authors declare no conflict of interest.

References

1. Cisco annual internet report (2018–2023) white paper, [Online]. Available: https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html
2. Guo Y, Zou B, Ren J, Liu Q, Zhang D, Zhang Y (2019) Distributed and efficient object detection via interactions among devices, edge, and cloud. IEEE Trans Multimedia 21(11):2903–2915
3. Ren J, Zhang D, He S, Zhang Y, Li T (2019) A survey on end-edge-cloud orchestrated network computing paradigms: transparent computing, mobile edge computing, fog computing, and cloudlet. ACM Comput Surv (CSUR) 52(6):1–36

4. Zhu Z, Peng J, Gu X, Li H, Liu K, Zhou Z, Liu W (2018) Fair resource allocation for system throughput maximization in mobile edge computing. IEEE Access 6:5332–5340
5. Hou B, Chen F (2017) Gds-lc: a latency-and cost-aware client caching scheme for cloud storage. ACM Trans Storage (TOS) 13(4):1–33
6. Vo PL, Tran NH (2019) Cooperative caching for http-based adaptive streaming contents in cache-enabled radio access networks. Computing 101(5):435–453
7. Xie H, Shi G, Wang P (2012) Tecc: towards collaborative in-network caching guided by traffic engineering. In: 2012 Proceedings IEEE INFOCOM. IEEE, pp 2546–2550
8. Ostovari P, Wu J, Khreishah A (2016) Efficient online collaborative caching in cellular networks with multiple base stations. In: 2016 IEEE 13th International Conference on Mobile Ad Hoc and Sensor Systems (MASS). IEEE, pp 136–144
9. Zhi J, Li J, Wu H et al (2017) Edge-first-based cooperative caching strategy in information centric networking. J Commun 38(3):53–64
10. Li C, Zhang Y, Sun Q, Luo Y (2021) Collaborative caching strategy based on optimization of latency and energy consumption in MEC. Knowl-Based Syst 233:107523
11. Li Y, Hu S, Li G (2021) Cvc: a collaborative video caching framework based on federated learning at the edge. IEEE Trans Netw Serv Manage 19(2):1399–1412
12. Ndikumana A, Tran NH, Kim KT, Hong CS et al (2020) Deep learning based caching for self-driving cars in multi-access edge computing. IEEE Trans Intell Transp Syst 22(5):2862–2877
13. Wang X, Wang C, Li X, Leung VC, Taleb T (2020) Federated deep reinforcement learning for internet of things with decentralized cooperative edge caching. IEEE Internet Things J 7(10):9441–9455
14. Chien W-C, Weng H-Y, Lai C-F (2020) Q-learning based collaborative cache allocation in mobile edge computing. Futur Gener Comput Syst 102:603–610
15. Ferragut A, Rodríguez I, Paganini F (2016) Optimizing ttl caches under heavy-tailed demands. ACM SIGMETRICS Perform Eval Rev 44(1):101–112
16. Ioannidis S, Yeh E (2018) Adaptive caching networks with optimality guarantees. IEEE/ACM Trans Netw 26(2):737–750
17. Wang Y, Wang W, Cui Y, Shin KG, Zhang Z (2018) Distributed packet forwarding and caching based on stochastic network utility maximization. IEEE/ACM Trans Netw 26(3):1264–1277
18. Yan B, Xu Y, Chao HJ (2018) Adaptive wildcard rule cache management for software-defined networks. IEEE/ACM Trans Netw 26(2):962–975
19. Wu D, Zhou L, Cai Y, Qian Y (2018) Collaborative caching and matching for d2d content sharing. IEEE Wirel Commun 25(3):43–49
20. Breslau L, Cao P, Fan L, Phillips G, Shenker S (1999) Web caching and zipf-like distributions: evidence and implications. In: IEEE INFOCOM'99. Conference on Computer Communications. Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. The Future Is Now (Cat. No. 99CH36320), vol 1. IEEE, pp 126–134
21. Wang L, Jiao L, He T, Li J, Mühlhäuser M (2018) Service entity placement for social virtual reality applications in edge computing. In: IEEE INFOCOM 2018-IEEE Conference on Computer Communications. IEEE, pp 468–476
22. Somesula MK, Rout RR, Somayajulu DVLN (2023) Greedy cooperative cache placement for mobile edge networks with user preferences prediction and adaptive clustering. Ad Hoc Netw 140:103051
23. Somesula MK, Mothku SK, Annadanam SC (2023) Cooperative service placement and request routing in mobile edge networks for latency-sensitive applications. IEEE Syst J 17(3):4050–4061
24. Hu Z, Fang C, Wang Z, Tseng S-M, Dong M (2024) Many-objective optimization-based content popularity prediction for cache-assisted cloud-edge-end collaborative iot networks. IEEE Internet Things J 11(1):1190–1200
25. Somesula MK, Kotte A, Annadanam SC, Mothku SK (2022) Deadline-aware cache placement scheme using fuzzy reinforcement learning in device-to-device mobile edge networks. Mob Netw Appl 27(5):2100–2117
26. Sun Y, Peng M, Mao S (2018) Deep reinforcement learning-based mode selection and resource management for green fog radio access networks. IEEE Internet Things J 6(2):1960–1971
27. Wan Z, Li Y (2020) Deep reinforcement learning-based collaborative video caching and transcoding in clustered and intelligent edge b5g networks. Wirel Commun Mob Comput 2020:1–16
28. Somesula MK, Rout RR, Somayajulu DVLN (2022) Cooperative cache update using multi-agent recurrent deep reinforcement learning for mobile edge networks. Comput Netw 209:108876
29. Somesula MK, Mothku SK, Kotte A (2023) Deep reinforcement learning mechanism for deadline-aware cache placement in device-to-device mobile edge networks. Wirel Netw 29(2):569–588

30. Van Hasselt H, Guez A, Silver D (2016) Deep reinforcement learning with double q-learning. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol 30
31. Chen B, Liu L, Sun M, Ma H (2019) Iotcache: toward data-driven network caching for internet of things. IEEE Internet Things J 6(6):10064–10076
32. Sutton RS, Barto AG (2018) Reinforcement learning: an introduction. A Bradford Book, Cambridge
33. Li X, Wang X, Wan P-J, Han Z, Leung VC (2018) Hierarchical edge caching in device-to-device aided mobile networks: modeling, optimization, and design. IEEE J Sel Areas Commun 36(8):1768–1785

Authors and Affiliations

Hui Xiao[1] · Xinyu Zhang[1] · Zhigang Hu[1] · Meiguang Zheng[1] · Yang Liang[1,2]

✉ Zhigang Hu
zhiganghu666@163.com

Hui Xiao
huixiaoyt@163.com

Xinyu Zhang
zhangxinyu11014@163.com

Meiguang Zheng
zhengmeiguang@csu.edu.cn

Yang Liang
liangyang1987@tom.com

1 School of Computer Science and Engineering, Central South University, 932 South Lu Shan Road, Changsha 410083, Hunan, China

2 School of Informatics, Hunan University of Chinese Medicine, 300 Scholars Road, Changsha 410083, Hunan, China