Agile Cache Replacement in Edge Computing via Offline-Online Deep Reinforcement Learning

Zhe Wang, Jia Hu, Geyong Min, Zhiwei Zhao, Zi Wang

***Abstract*—One fundamental problem of content caching in edge computing is how to replace contents in edge servers with limited capacities to meet the dynamic requirements of users without knowing their preferences in advance. Recently, online deep reinforcement learning (DRL)-based caching methods have been developed to address this problem by learning an edge cache replacement policy using samples collected from continuous interactions (trial and error) with the environment. However, in practice, the online data collection phase is often expensive and time-consuming, thus hindering the practical deployment of online DRL-based methods. To bridge this gap, we propose a novel Agile edge Cache replacement method based on Offline-online deep Reinforcement learNing (ACORN), which can efficiently learn an edge cache replacement policy offline from a training dataset collected by a behavior policy (e.g., Least Recently Used) and then improve it with fast online fine-tuning. We also design a specific convolutional neural network structure with multiple branches to effectively extract content popularity knowledge from the dataset. Experimental results show that the offline policy generated by ACORN outperforms the behavior policy by up to 38%. Through online fine-tuning, ACORN also achieves the number of cache hits as good as that of several advanced DRL-based methods while significantly reducing the number of training epochs by up to 40%.**

***Index Terms*—Deep reinforcement learning, cache replacement, offline training, convolutional neural network, edge computing**

I. Introduction

With the ever-increasing number of mobile users and rapid advance of smart devices, mobile data traffic has been growing dramatically. It is estimated that mobile data traffic will generate approximately a 4.2x increase from 2021 to 2027 [1]. The explosive growth of mobile data traffic poses a huge burden on the backhaul networks, causing severe issues such as network congestion and server overload, which inevitably introduce high service latency and further impact users' quality of experience (QoE). These issues also hinder the deployment of delay-sensitive and data-intensive applications including augmented reality, cloud gaming, etc.

Content caching in edge computing (i.e., edge caching) is a promising approach to solve the above issues. It stores contents required by users to a cache-equipped edge server close to them, so that the content access latency for users and the backhaul traffic is greatly reduced [2]. Due to the limited cache capacity of the edge servers, they need to store the most popular contents to maximize the number of cache hits and therefore minimize the cache misses that lead to frequent fetching of contents from the remote cloud. However, the popularity of contents is usually unknown in advance and varies with time, thus edge servers should continuously decide how to replace the cached contents to best meet the dynamic requirements of mobile users at the network edge, which is usually a NP-hard problem [3].

To tackle this problem, many studies were devoted to designing an effective edge cache replacement policy that determines which content to be stored and which one to be replaced over time. Heuristic-based methods [4], [5] can capture and use some features of the environment to guide the cache replacement decisions, but these features are often obtained by domain experts. Moreover, these heuristic-based methods usually work well in specific cases but cannot well adapt to dynamic environments of the network edge. To overcome the above weaknesses, deep reinforcement learning (DRL) has emerged as an attractive approach. DRL is a deft combination of reinforcement learning (RL) and deep neural network (DNN), where RL can effectively solve sequential decision-making problems and continuously adapt to the changes in the environment without expert knowledge, while DNN can effectively handle large state/action spaces of complex edge caching systems. The existing works [6]–[8] are mainly developed based on online DRL which needs to continuously interact with the edge caching system to train an optimal cache replacement policy.

However, applying online DRL directly to an edge caching environment carries risks due to the exploration nature of online DRL. To find an optimal cache replacement policy, the online DRL agent needs to constantly try various content replacement decisions during training, even if some of these decisions may be poor. Such trial and error can lead to severe cache performance degradation during training. In addition, learning an optimal policy often requires a large number of interactions with the environment, which is both expensive and time-consuming. These drawbacks of online interactions would harm the revenue of network operators and the QoE of users. Furthermore, online DRL can only self-adapt to small shifts in the environmental dynamics. When the environmental dynamics change drastically, the performance of an optimal policy may drop dramatically, thus requiring to retrain a new optimal policy by online DRL. This implies the presence of

This work was supported in part by UKRI Grant No. EP/X038866/1 and Horizon EU Grant No. 101086159. *(Corresponding authors: Jia Hu; Geyong Min.)*

Zhe Wang, Jia Hu, Geyong Min and Zi Wang are with the Department of Computer Science, University of Exeter, United Kingdom. E-mail: {zw329, j.hu, g.min, z.wang5}@exeter.ac.uk

Zhiwei Zhao is with the School of Computer Science and Engeeneering, University of Electronic Science and Technology of China, China. E-mail: zzw@uestc.edu.cn

frequent retraining processes of the online DRL-based cache replacement methods, which further amplifies the issue of trial and error in online DRL for dynamic edge caching.

To address the above key challenges, this paper proposes a novel Agile edge Cache replacement method based on Offline-online deep Reinforcement learNing (ACORN). The main idea is that some static dataset (e.g., logs of caching records) on edge servers contains key information about the environmental dynamics, thus we can directly extract the information from the dataset to learn a useful policy offline. Afterwards, we can supplement more information about the environmental dynamics to the offline policy by fine-tuning, which includes only a few online interactions with the environment, thus achieving an optimal policy in a cost-efficient way.

The main contributions of this paper can be summarized as follows:

- To the best of our knowledge, we are the first to investigate how to develop an agile, deployable learning-based edge caching scheme using advanced offline-online DRL techniques. Our method ACORN has a two-stage workflow including offline training and online fine-tuning, which can learn an optimal policy while greatly reducing the expensive interactions with the environment.
- We formulate the problem of real-time edge cache replacement without knowing user preferences as an optimization framework to maximize the number of cache hits. To alleviate the performance deterioration caused by frequent content replacement, the delayed hits mechanism is explicitly considered in the problem formulation.
- We design a special Convolutional Neural Network (CNN) with multiple branches to efficiently extract the temporary correlations from historical data with the aim of speeding up the training process and improving cache performance. A mask layer is introduced in the tailored CNN to filter out inefficient content replacement decisions, thus improving the performance of the learned policy.
- Extensive experiments were conducted by using a real-world dataset to simulate the environmental dynamics. The results show that the offline policy learned from the static dataset outperforms the default behavior policy, and is improved by fine tuning to achieve competitive performance using fewer online interactions, compared to the policies learned by advanced DRL-based methods.

The rest of this paper is organized as follows: The related work on the edge cache replacement problem is reviewed in Section II. Section III presents the problem formulation of mobile edge caching with delayed hits. The details of ACORN are shown in Section IV. Section V describes the experimental setting and gives a detailed analysis of the experimental results. Finally, the conclusion of this paper is presented in Section VI.

II. Related Work

The edge cache replacement problem has attracted many researchers to find the optimal cache replacement policy. Heuristic-based methods are widely used to search for an optimal or near-optimal cache replacement policy. A simple heuristic caching method is proposed in [9] to minimize the average download time, but this work assumed that the content popularity was known in advance. Xia $et\ al.$ [10] formulated the edge cache replacement problem under limited storage resources as a constrained optimization problem with the aim of reducing the data retrieval delay of users. They subsequently employed integer programming techniques to determine the optimal cache replacement policy. A proactive retention-aware caching and request routing problem was studied in [5] and formulated as a nonlinear, nonconvex, mixed-integer program. To solve this problem, a low complexity heuristic approach was developed to find a feasible cache replacement policy. [11] formulated the caching problem from the app vendor's perspective without requiring future information about requests, and proposed an online approach to solve it over time. Although heuristic-based methods can find optimal or near-optimal policies, they usually require expert knowledge to formulate the cache replacement problem as an optimization problem and then spend much time and human resources to design the corresponding heuristic algorithm. Additionally, if the edge caching environment changes dramatically, the problem needs to be reformulated, and the corresponding algorithm needs to be redesigned.

By interacting with the edge caching environment, DRL-based methods can automatically find the optimal cache replacement policy without expert knowledge. [12] solved the cache replacement problem in a basic wireless caching network with the objective of improving the cache hit ratio. This work improved the decision-making ability of an agent that is trained with deep Q-learning (DQN) algorithm by using a long short-term neural network and an external memory. A latency-optimized problem for fog radio access networks was proposed in [13], and Q-learning algorithm was then utilized to solve the problem by jointly caching proper contents and allocating a substantial quantity of power resources according to users' demands. To jointly minimize content access delay and bandwidth resource costs in a multi-edge caching environment, [14] leveraged a multi-agent DRL to adaptively learn the best policy for each edge to achieve intelligent caching. However, all the DRL-based methods described above train agents by constantly interacting with the environment, which introduces expensive and risky data collection that inevitably compromises the performance of edge caching and the QoE for users during the training process. The work most similar to this paper is [15], which first used deep learning (DL) with a static dataset to initialize the DNNs. After that, it leveraged Deep Deterministic Policy Gradient algorithm to update the DNNs with online interactions to obtain the optimal cache replacement policy. Nevertheless, this work does not test the performance and convergence speed of the policy learned by DL.

III. Problem Formulation: Edge Caching with Delayed Hits

A basic cache-aided mobile edge computing (MEC) network is shown in Fig. 1, which consists of a remote cloud,

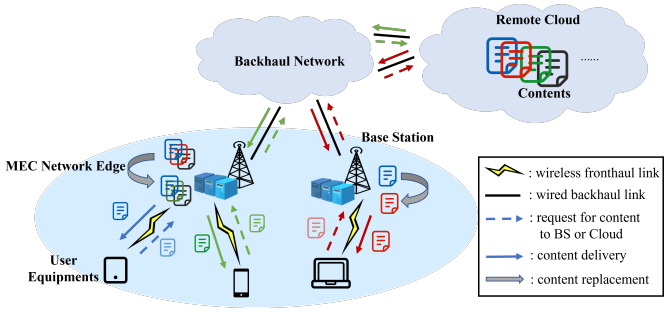


Fig. 1. Architecture of the edge caching environment.

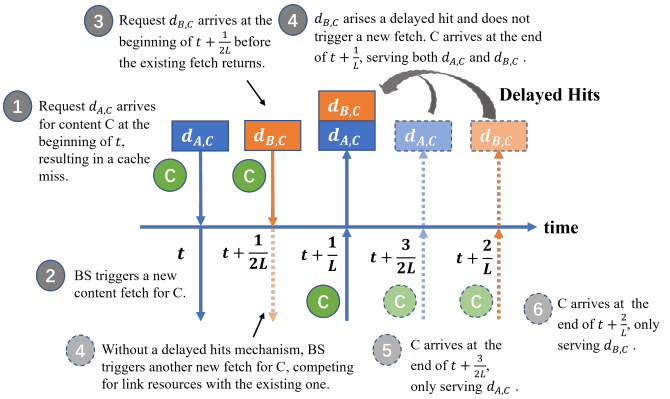


Fig. 2. Example of edge caching with/without delayed hits. Steps 1-3 are the same for both. Opaque step 4 with a solid outline is the process using the delayed hits mechanism, while transparent steps 4-6 with a dashed outline do not use this mechanism.

several base stations (BSs), and multiple mobile users associated with the BSs, where the users communicate with their associated BSs via wireless fronthaul links, and the BSs connects to the remote cloud through wired backhaul links. To achieve agile edge caching, the cache replacement policy on each BS utilizes the request information of the users associated that particular BS. Similar to the approach taken in [16], we avoid exchanging information between BSs in order to maintain the efficiency of decision-making for cache replacement.

It is assumed that the remote cloud has a content library $\mathcal{F}$ containing all the contents requested by users with different indexes, where $\mathcal{F} = \{1, 2, ...j, ..., K\}$. The size of content j is denoted as l_j. Similar to some previous studies [17], [18], we assume that all contents have the same size, where $l_j = 1$ unit. We can readily generalize this assumption to contents of varying sizes by dividing them into units. However, for the sake of illustration and without loss of generality, we consider the case where all contents have the same size in this paper. The cache-equipped BSs can store contents to serve their associated users. Without loss of generality, we consider that the cache size of a particular BS is denoted by M, where $M << K$ means that just a tiny fraction of the contents can be stored on the BSs. In our studied scenario, the service process of the cache-aided MEC network during a finite long-term is divided equally into multiple timeslots, denoted as $\mathcal{T} = \{1, 2, ..., t, ..., T\}$. The division method can be extended to an infinite-time service process with a terminal state, where T represents the timeslot at which the service process reaches its terminal state. For the selected BS, it receives some requests $\mathcal{D}^t = \{d^t_{1,1}, d^t_{1,2}, ..., d^t_{i,j}, ..., d^t_{N,K}\}$ from N users for various contents at the beginning of each timeslot t without knowing the content popularity, where $d^t_{i,j} \in \{0, 1\}$. $d^t_{i,j} = 1$ denotes that the user i demand for the content j in the timeslot t; Otherwise, $d^t_{i,j} = 0$. In this paper, we assume that the request size is much smaller than the content size, so the time consumption of sending requests is negligible.

Let C^t denote the cache strategy that specifies which contents are stored on the selected BS during timeslot t. Specifically, if content j is stored on the BS in timeslot t, then $j \in C^t$; otherwise, if it is not stored on the BS, then $j \notin C^t$ and it is missed. Note that the total size of the contents stored on the BS never exceeds its cache capacity, i.e., $\sum_{j \in C^t} l_j \leq M$. When new requests arrive at the BS, if their required contents are stored on the BS (regarded as cache hits), the BS can deliver contents to users immediately. If the contents are missed on the BS (regarded as cache misses), the BS needs to spend more time fetching the missed contents from the remote cloud and subsequently delivering them to the users in future timeslots. Meanwhile, at the end of the timeslot t, the BS updates the cache strategy C^t with the newly fetched contents $\mathcal{A}^t$ by following a cache replacement policy π. We simplify the cache replacement problem by directly selecting the contents to be stored on the BS from the union of C^t and $\mathcal{A}^t$, rather than evicting contents from C^t and selecting the same number of contents from $\mathcal{A}^t$ to cache. Thus, the updated cache strategy in timeslot $t+1$ can be expressed as:

$$C^{t+1} = \pi(C^t \cup \mathcal{A}^t), \tag{1}$$

where π denotes the cache replacement policy. When $t = 1$, C^t does not contain any contents and is an empty set. To be more specific, since the newly fetched contents $\mathcal{A}^t$ are the missed contents of C^t, we have $C^t \cap \mathcal{A}^t = \emptyset$. If content j is in both C^{t+1} and $\mathcal{A}^t$, then one cached content in C^t is replaced by the newly fetched content j. On the other hand, if content j' is in C^t but not in C^{t+1}, it means that j' is evicted from the BS. As evident from the above, our simplified method can accurately represent the cache replacement process.

When the timeslot is small enough, the cache replacement execution can be considered real-time. However, this also poses more computational resource overhead on the BS because of the increased frequency of replacement decision-making. When no new content arrives at the end of a timeslot, the BS has no optional content to update the cache strategy. In this way, the BS only needs to replace the cached contents when new contents arrive instead of doing so at the end of each timeslot, thus reducing the computational overhead of the BS.

In traditional edge caching scenarios, the fetches triggered by different users are usually independent. It means that even if two users require the same content simultaneously, two independent fetches are triggered when the content is cache-missed, resulting in a waste of backhaul link resources and frequent but meaningless cache replacement processes. Here is a simple example illustrated in Fig. 2. Suppose two requests,

$d_{A,C}$ and $d_{B,C}$ from different users A and B, require content C of size 1 unit. They arrive at the BS at the beginning of the timeslot t and $t+1/2L$, respectively. Since content C is not stored on the BS at the beginning of the timeslot t, when some users require it, C needs to be fetched from the cloud via a backhaul link with a bandwidth of L units per timeslot. In the traditional scenario, $d_{A,C}$ and $d_{B,C}$ trigger two independent fetches for the same content C at the beginning of the timeslot t and $t+1/2L$. Due to competition between the two fetches for the backhaul link, they are eventually finished at the end of the timeslot $t+3/2L$ and $t+2/L$, with a service delay of $3/2L$. Then, two cache replacements with the same content are executed at the end of the timeslot $t+3/2L$ and $t+2/L$, respectively. When numerous requests arrive at the BS in a short period, it may trigger too many content fetches even if most of the requests ask for the same content. In this case, the BS may frequently replace old cached contents with newly fetched contents before old contents work, thus significantly degrading the performance of the edge caching.

The delayed hits mechanism can reduce such competition for backhaul links by using a fetch to serve all requests for the same missed content before this content missing is resolved. When a request triggers a new fetch, all subsequent requests to the same content wait for the fetch to complete instead of triggering new fetches. When the fetch is finished, the BS simultaneously delivers the fetched content to all users in the environment who need that content. These subsequent requests are regarded as "delayed hits" [19]. Reducing repeated fetches for the same content avoids frequent evictions and caching of that content in a short period, thus mitigating the deterioration of the cache performance. When the scenario in Fig. 2 uses the delayed hits mechanism, request $d_{B,C}$ waits for an existing fetch triggered by $d_{A.C}$ to complete. When the BS receives content C at the end of the timeslot $t+1/L$, it simultaneously delivers content C to users A and B. In this way, the service delays of $d_{A,C}$ and $d_{B,C}$ are reduced to $1/L$ and $1/2L$. The frequency of the cache replacement is also reduced to one.

Here, we explicitly consider the impact of delayed hits on our studied scenario by only recording single fetch for each content at each timeslot. Let $f_j^{t'}$ denote the remaining size of content j that still needs to be fetched at the beginning of the timeslot t, while f_j^t is the remaining size at the end of the timeslot t. $f_j^{t'}$ can be determined according to two different cases: In timeslot t, (1) the content j is cache-missed and triggers a new fetch; (2) No new fetch for the content j is triggered. For cases (1), when a fetch for the content j does not exsit and the content j is not stored on the BS, new requests for the missed content j will trigger a new fetch, and $f_j^{t'}$ can be determined:

$$f_j^{t'}(new) = l_j \times \mathbb{I}(f_j^{t-1} == 0) \times \mathbb{I}(\sum_i d_{i,j}^t > 0) \times \mathbb{I}(j \notin C^t), \tag{2}$$

where $\mathbb{I}(\cdot) = 1$ if and only if the condition $\cdot$ within parentheses is true; Otherwise, it is 0.

If case (1) is not satisfied, case (2) needs to be discussed and it can be further divided into three cases: (i) There is no new request for the content j; (ii) There are some new requests for the content j, and j is already stored on the BS; (iii) There are some new requests for the content j, and a fetch for j already exists. In the three cases, $f_j^{t'}$ can be directly determined using the remaining size of content j at the end of the timeslot $t-1$, f_j^{t-1}:

$$f_j^{t'}(nonew) = f_j^{t-1}. \tag{3}$$

We combine the cases above to conclude:

$$f_j^{t'} = \max\{f_j^{t'}(nonew), f_j^{t'}(new)\}. \tag{4}$$

f_j^t can be calculated by subtracting the transmission size of content j during the timeslot t from $f_j^{t'}$:

$$f_j^t = \max\{f_j^{t'} - \frac{L \times \mathbb{I}(f_j^{t'} > 0)}{\max\{\sum_{n=1}^K \mathbb{I}(f_n^{t'} > 0), 1\}}, 0\}, \tag{5}$$

where L is the bandwidth of the backhaul link equally shared by all existing fetches in the environment in each timeslot. $\sum_{n=1}^K \mathbb{I}(f_n^{t'} > 0)$ is the number of fetches at the beginning of the timeslot t. Because there could be no fetches in the environment, we use a $\max$ operation in the denominator to avoid the case where the denominator is zero. While the $\max$ operation for the whole right part of Eq. 5 ensures that the remaining size is not a negative number. When a entire content j is fetched to the BS at the end of the timeslot t, we have $f_j^{t'} > 0$ and $f_j^t == 0$. Then, the set of newly fetched contents during the timeslot t, $\mathcal{A}^t$, can be determined:

$$\mathcal{A}^t = \bigcup_{f_j^{t'}>0 \ and \ f_j^t==0}^{j \in \mathcal{F}} \{j\}, \tag{6}$$

where $j \in \mathcal{F}$ means that the types of fetched contents should belong to the existing types in the content library. Combining Formula 1, we can deduce that the types of cached contents on the BS should also belong to the existing types in the content library. Therefore, users will not require content types that do not exist in the content library during the service.

The goal of this paper is to develop a cache replacement policy that enables each BS to dynamically store the most

TABLE I
SUMMARY OF MAIN NOTATIONS

Notation	Description
$\mathbb{I}(\cdot)$	The value is 1 if the condition $\cdot$ within parentheses is true; Otherwise the value is 0.
K	The total number of content types.
M	The cache storage size of a particular BS.
C^t	The set of cached contents in timeslot t.
D^t	The set of requests arriving in timeslot t.
A^t	The set of contents fetched from the remote cloud at the end of the timeslot t.
f_j^t	The remaining size of content j to be fetched at the end of the timeslot t.
$f_j^{t'}$	The remaining size of content j to be fetched at the beginning of the timeslot t.
π	The decision-making policy for replacing cached contents.

popular contents to maximize cache performance. As the number of cache hits directly reflects the popularity of the content, we choose the total number of cache hits as our optimization objective. In this context, we can safely ignore the delivery phase from the BSs to the mobile users since it does not affect the number of cache hits. The cache replacement problem, which is generic to all BSs, is formulated as an optimization problem as follows:

$$\begin{aligned} &\min_{\pi} \sum_{t}\sum_{i}\sum_{j} d_{i,j}^{t}\mathbb{I}(j \in \pi(C^{t-1} \cup \mathcal{A}^{t-1})), \\ &s.t. \quad C1: C^{t} \cap \mathcal{A}^{t} = \emptyset, \forall t, \\ &\qquad C2: |C^{t}| \leq M, \forall t, \\ &\qquad C3: \textstyle\sum_{j} d_{i,j}^{t} \leq 1, \forall i, t, \end{aligned} \tag{7}$$

where constraint C1 guarantees that the newly fetched contents are not currently stored on the BS. Constraint C2 enforces that the amount of stored contents never exceeds the cache size of the BS. Constraint C3 ensures that there is a maximum of one request per user at any given timeslot. The main notations of this model are presented in Table I.

IV. ACORN: An Agile Edge Cache Replacement Method Based On Offline-Online DRL

This section details the proposed novel agile edge cache replacement method based on offline-online DRL named ACORN. First, we present the background of DRL. Next, we outline the ACORN system framework and explain its workflow. Then, the MDP modeling of the edge cache replacement problem is described. After that, we will show the structure of the policy neural network. Finally, the offline training and online fine-tuning algorithms are presented.

A. Background: Deep Reinforcement Learning

RL has attained great success across a wide range of fields, such as video games [20], robotics [21], task offloading [22], service placement [23], and so on. The problem of RL is usually formulated as a Markov Decision Process (MDP) with a 5-tuple definition $< S, A, T, R, \gamma >$. S is the state space, A is the action space, $T : S \times A \to S$ is the state transition probability matrix, $R : S \times A \to \mathbb{R}$ is the reward function, and $\gamma \in (0, 1)$ is a discount factor. At each discrete time step t, the RL agent receives an immediate reward $r_t(s_t, a_t)$ for executing an action a_t for a given state s_t, then arriving at the next state s_{t+1} in accordance with the state transition probability matrix $T(s_{t+1}|s_t, a_t)$. The agent aims to maximize γ-discounted cumulative rewards in a long term $R_t = \sum_{k=0} \gamma^k r_{t+k}$.

Suppose the agent selects actions following a policy $\pi : S \to A$ that indicates the probability of selecting an action a for a given state s. The policy's Q-function $Q^{\pi}(s_t, a_t) = \mathbb{E}_{\pi}[R_t|s_t, a_t]$ is the expected return when all subsequent actions are selected under the guidance of the policy π after executing the action a_t for the state s_t. To calculate the Q-function of the policy $\pi(s_t, a_t)$, Bellman Operator B_{π} is introduced to update a Q-function from any initial point iteratively:

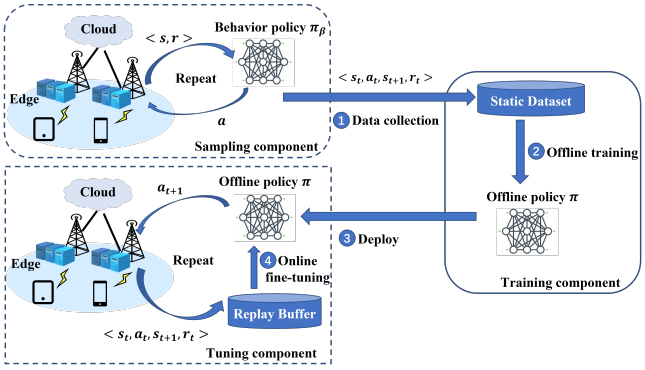


Fig. 3. ACORN System Framework. The workflow consists of four steps: 1. Collect data with the existing behavior policy on a particular BS; 2. Learn a policy offline based on the static dataset; 3. Deploy the well-trained offline policy into the selected BS; 4. Fine-tune the offline policy through online interactions with the environment

$$B_{\pi}Q^{\pi}(s_t, a_t) = \mathbb{E}_{s_{t+1}}[r_t + \gamma Q^{\pi}(s_{t+1}, \pi(a_{t+1}|s_{t+1}))]. \tag{8}$$

This operator converges to a unique fixed point $Q^{\pi}(s_t, a_t)$ for $\gamma \in (0, 1)$. An optimal Q-function can be obtained by $Q^*(s_t, a_t) = \max_{\pi} Q^{\pi}(s_t, a_t)$, then the optimal policy π^* is determined by selecting greedy actions $\pi^*(a_t|s_t) = \arg\max_a Q^*(s_t, a)$. DNN is used to approximate the Q-function due to its powerful representation capability. By adjusting the connection weights across all layers, DNN can implement a diverse range of functions, allowing for effectively representing the complex mapping relationship between the states and estimated Q-values. The optimal Q-function, being the one with the highest value estimations, assigns an estimated value to the current state-action pair that equals the sum of the reward and the highest estimated value of the next state-action pair. Consequently, we can iteratively update the parameters θ using Eq. 9 to directly approximate the optimal Q-function.

$$\theta - \frac{1}{2}\bigtriangledown_{\theta}|Q_{\theta}(s_t, a_t) - \mathbb{E}_{s_{t+1}}[r_t + \gamma \max Q_{\theta}(s_{t+1}, a_{t+1})]|^2, \tag{9}$$

where $\bigtriangledown_{\theta}$ is the gradient calculation for the parameters θ. Once the optimal Q-function is determined, the corresponding optimal policy can be obtained by selecting the greedy actions:

$$\pi(a_{t+1}|s_{t+1}) = \arg\max_{a} Q_{\theta}(s_{t+1}, a). \tag{10}$$

B. ACORN System Framework

The ACORN system consists of three components, a sampling component, a training component, and a tuning component, as shown in Fig. 3. The sampling component utilizes behavior policies on BSs to separately collect a static dataset for each BS by constantly interacting with the environment. If a large amount of log data exists in the environment, the sampling component can also be omitted. The training component only uses the static dataset collected from the sampling component to learn an offline policy for each BS until the policy is well-trained. In this way, the training of

the offline policy does not influence the operations of the behavior policy, which means that even if the offline policy explores undesirable actions during the training process, the cache performance is not degraded. It should be noticed that the dataset is collected only once and not altered during the training process [24]. After the offline policy is well-trained, it is applied to the tuning component, which is further improved with a small number of new interactions.

C. The MDP Model for Edge Caching

To resolve the edge cache replacement problem on a particular BS with offline-online DRL, we should first model it as an MDP. The state space, action space, and reward function should be designed carefully, as they significantly impact the training speed and the cache performance. In addition, the behavior policy can be derived from DRL-based, heuristic-based, or even rule-based methods (e.g., Least Recently Used - LRU, Least Frequently Used - LFU, First In First Out - FIFO), which means that the structure of collected log data may not directly indicate the states, actions, or rewards. Therefore, we should carefully devise the state/action spaces and reward function for extracting them from the raw log efficiently. The following are their definitions while considering the above requirements:

- **State Space:** The content replacement decisions mainly depend on content popularity. The past requested content history reflects the content popularity to a certain extent, so we use them as part of the state inputs $(\mathbf{x_1^t}, \mathbf{x_2^t}, ..., \mathbf{x_M^t})$, where $\mathbf{x_M^t}$ is a vector that records the requested history in the past n timeslots of content cached in the Mth location when the current timeslot is t, $\mathbf{x_M^t} = (x_M^t, x_M^{t-1}, ..., x_M^{t-n})$. Besides, the information of newly fetched contents in the current timeslot t, $\mathbf{x_{ADD}^t}$, is also included in the state inputs.
- **Action Space:** In each timeslot t, several missed contents $\mathcal{A}^t$ are fetched intact to the BS. If $\mathcal{A}^t + C^t \leq M$, the missed contents can be directly cached into the BS. If $\mathcal{A}^t + C^t > M$, the BS needs to choose a part of the contents from C^t to evict and choose the same number of contents from $\mathcal{A}^t$ to cache to meet the dynamic requests from users. The eviction and cache phases can be simplified to select M contents from $\mathcal{A}^t \cup C^t$ and then store them on the BS. Under this design, the action space size is a combination number $\mathbf{C}_M^{|\mathcal{A}^t|+|C^t|}$. However, the number of the fetched contents is not fixed, which makes the action space size dynamic. We extend the candidates to all contents to solve this problem, indicating that the BS selects M contents from $\mathcal{F}$. In this way, the action space size is fixed as $\mathbf{C}_M^K$, where $|\mathcal{A}^t| + |C^t| \leq K$. Unfortunately, this method may introduce some illegal actions because usually, not all the contents in $\mathcal{F}$ will arrive at the BS from the cloud in the same timeslot. Therefore, we use a mask layer introduced later to filter the illegal actions of selecting the contents not in $\mathcal{A}^t \cup C^t$ to cache.
- **Reward Function:** The optimization problem to be solved aims to maximize the total number of cache hits. Therefore, we can define the number of cache hits per timeslot as the reward function:

$$r_t(s_t, a_t) = \sum_i \sum_j d_{i,j}^{t+1} \mathbb{I}(j \in C^{t+1}); \tag{11}$$

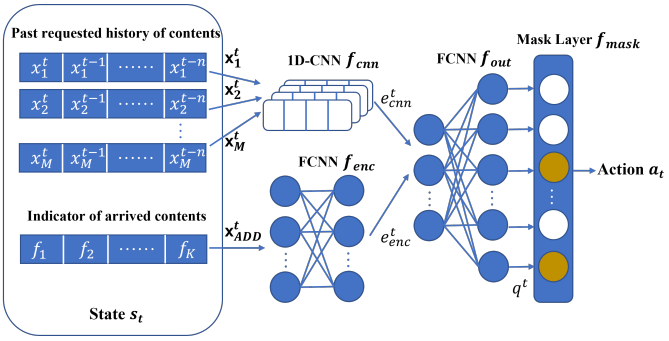


Fig. 4. Structure of the Q-function neural network.

D. ACORN Neural Network Structure

According to the MDP model, we find two problems to be solved in ACORN: (1) To make good content replacement decisions, the learned policy needs a solid understanding of content popularity and its changing trends. How to efficiently learn the relevant knowledge from given states is urgently needed. (2) Part of the content selections from $\mathcal{F}$ may be illegal and cannot be executed since some contents are not included in $\mathcal{A}^t \cup C^t$, which raises the consideration of action legality guarantee.

To address the above problems, we use CNN, which has been proven to extract the temporary correlation knowledge from time series data efficiently [25], [26], to learn the knowledge about content popularity from history. A mask layer is also developed to filter legal actions. Specifically, we will use 1-D CNN, defined as f_{cnn}, to extract the temporary correlation from the history of contents cached in the BS. The history of one cached content corresponds to the input of a f_{cnn}'s channel. The output of CNN, e_{cnn}^t, can be obtained by:

$$e_{cnn}^t = f_{cnn}(\mathbf{x_1^t}, \mathbf{x_2^t}, ..., \mathbf{x_M^t}). \tag{12}$$

In addition to the content popularity of the cached contents, the information about newly fetched contents also affects the content replacement decisions. To make better decisions, we should also extract features of these contents and combine them with e_{cnn}^t to provide sufficient knowledge. Due to the differences in the data structure between the information about the fetched contents $\mathbf{x_{ADD}^t}$ and the history of the cached contents $(\mathbf{x_1^t}, \mathbf{x_2^t}, ..., \mathbf{x_M^t})$, $\mathbf{x_{ADD}^t}$ cannot be used as the input of a f_{cnn}'s channel. To effectively extract the knowledge about the newly fetched contents, we introduce an additional embedding fully connected neural network (FCNN), defined as f_{enc}, to embed $\mathbf{x_{ADD}^t}$ into a high-dimensional feature vector e_{enc}^t:

$$e_{enc}^t = f_{enc}(\mathbf{x_{ADD}^t}). \tag{13}$$

Then, we concatenate the outputs of the two branches, 1-D CNN and the embedding FCNN, into a single vector as the input of a two-layer FCNN, which is defined as f_{out}, to produce the Q-value estimation of all actions, q^t:

$$q^t = f_{out}([e^t_{cnn}, e^t_{enc}]). \tag{14}$$

Given the Q-value estimation q^t, the action can be obtained according to Eq. 10. However, illegal actions are sometimes estimated with high Q-values, causing the policy to select actions that cannot be executed as outputs. To avoid such a case, we develop a mask layer, defined as f_{mask}, to reduce the Q-value estimation of illegal actions to $-inf$ while leaving the Q-value estimation of legal actions unchanged. For any action in the action space $a^t \in \mathbb{U}_{\mathbf{C}^K_M}$, its Q-value can be updated:

$$q^{t'}_{a^t} = f_{mask}(q^t_{a^t}) = \begin{cases} q^t_{a^t}, & if\ a^t \in \mathbb{U}_{\mathbf{C}^{|\mathcal{A}^t|+|C^t|}_M}; \\ -inf, & otherwise. \end{cases} \tag{15}$$

The probability of each action to be selected can be computed by using a softmax layer:

$$prob(a^t) = \frac{exp(q^{t'}_{a^t})}{\sum_{\hat{a}^t \in \mathbb{U}_{\mathbf{C}^K_M}} exp(q^{t'}_{\hat{a}^t})}. \tag{16}$$

The probability of illegal actions is $exp(-inf) = 0$, thus preventing the selection of such actions.

E. Offline-Online DRL Algorithm

In general, by using Eq. 8, the Q-function will correct its Q-value estimates for state-action pairs $< s_t, a_t >$ with the help of the ground truth rewards from new interactions. However, it is impractical in offline RL because only a static dataset, denoted as D, can be used for training. The dataset is pre-collected using a behavior policy, which may not be optimal, and may not contain the best action to be taken at every state. As the offline policy is updated to search for better actions, it will deviate from the dataset and explore unseen state-action pairs outside the dataset. Due to the lack of the ground truth rewards of these unseen state-action pairs, estimating and correcting their Q-values becomes challenging. As a result, the offline policy may make suboptimal decisions and degrade the cache performance. This shift between the state access distribution of an offline policy and that of a static dataset, caused by the updates to the offline policy, is commonly known as distribution shift. It is considered one of the central challenges of offline RL. This paper utilizes an advanced offline training algorithm improved from Conservative Q-Learning (CQL) [27] to address the distribution shift. To be specific, the working concept of Double DQN (DDQN) [28] is incorporated into CQL to enhance the effect of CQL further. The loss function of the improved CQL is shown as follows:

$$\begin{aligned} L_{CQL} = \alpha\dot{(}&\mathbb{E}_{s_t \sim T^D, a_t \sim \pi(a_t|s_t)}[Q_\theta(s_t, a_t)] \\ &- \mathbb{E}_{s_t, a_t \sim T^D}[Q_\theta(s_t, a_t)]) + L_{DDQN}(\theta, \theta^-), \end{aligned} \tag{17}$$

Algorithm 1 Offline training

1: Collect a dataset D with a behavior policy π_β.
2: Create a Q-function neural network Q with randomly generated initial values θ.
3: Create a target Q-function neural network $\hat{Q}$ with values $\theta^- = \theta$.
4: **for** step t = $1, 2, 3, ..., N$ **do**
5: Randomly sample a minibatch of interactions from D to update Q_θ using gradient steps with Eq. 17: $\theta_t \leftarrow \theta_{t-1} - \eta \bigtriangledown_\theta L_{CQL}$.
6: Every P steps reset $\hat{Q}_{\theta^-} = Q_\theta$.
7: **end for**

where T^D is the access distribution of the static dataset, and $L_{DDQN}(\theta, \theta^-)$ is the loss function of DDQN, which consists of a Q-function Q_θ and a target Q-function $\hat{Q}_{\theta^-}$:

$$\begin{aligned} L_{DDQN}(\theta, \theta^-) = \frac{1}{2}&\mathbb{E}_{s_t, a_t, r_t, s_{t+1} \sim T^D}[(Q_\theta(s_t, a_t) \\ &- (r_t + \gamma \hat{Q}_{\theta^-}(s_{t+1}, \arg\max_a Q_\theta(s_{t+1}, a))))^2]. \end{aligned} \tag{18}$$

Compared with DQN, DDQN uses the maximum Q-values estimated by the Q-function network to select the state-action pair, then leverages its Q-value estimated by the target Q-function network to calculate the loss. Because the Q-function is usually updated several times before the target Q-function is updated, its estimated Q-values are smaller or closer to the Q-values estimated by the target Q-function. Compared with the target Q-function, the Q-values estimated by Q-function are closer to the true Q-values for the state-action pairs within the dataset and more conservative for the state-action pairs outside the dataset. In this way, the conservative estimation of CQL is further strengthened. Then, the optimal Q-function can be found by minimizing the loss function of CQL:

$$Q^*_\theta \leftarrow \arg\min_{Q_\theta} L_{CQL}. \tag{19}$$

The first part of L_{CQL} ensures that all state-action pairs estimated by the learned policy have low Q-values. The second part encourages the policy to assign high Q-values to these state-action pairs in the dataset. In combination with the first part, actions in the dataset are estimated with higher Q-values, while actions outside the dataset are conservatively estimated with lower Q-values. The third part corrects the estimated Q-values using the ground truth rewards in the dataset according to the Bellman equation. In this way, the selection of actions available in the dataset is encouraged, and unseen actions are avoided whenever possible. α is a tradeoff factor used to adjust the weight of the conservative estimate part.

The details of the offline training process is shown in Alogithm 1. It should be noticed that θ are the parameters of the Q-function neural network. A static dataset is collected using an existing behavior policy generated by a rule-based, heuristic-based, or DRL-based method at first. Then, we calculate the update gradients using only the static dataset according to Eq. 17 to update the Q-function neural network. This process will

Algorithm 2 Online fine-tuning

Initialize replay memory buffer B.
Load the offline trained Q-function neural network θ.
Load the offline trained target Q-function neural network θ^{-}.
for episode = $1, 2, 3, ..., P$ **do**
  Initialize environment.
  for step t = $1, 2, 3, ..., N$ **do**
    Obeserve a state s_t.
    Select $a_t = \max_a Q_\theta(s_t, a)$ with ϵ-greedy method.
    Execute deicision a_t, and obtain a reward r_t and the next state s_{t+1} from the environment.
    Store the new online interaction (s_t, a_t, r_t, s_{t+1}) in B.
    Randomly sample a minibatch of interactions from B to update Q_θ according to : $\theta_t \leftarrow \theta_{t-1} - \eta \bigtriangledown_\theta L_{CQL}$.
    Every P steps reset $\hat{Q}_{\theta^-} = Q_\theta$.
  end for
end for

repeat multiple times until the Q-function converges. During this process, the target Q-function will be updated every P steps according to the parameters of the Q-function.

Unfortunately, the offline training algorithm can only learn a sub-optimal policy since the static dataset usually contains partial information about the state transition probability, even if the sub-optimal outperforms the behavior policy. To further improve it, an online fine-tuning algorithm is developed to supplement the missing knowledge about the transition probability by collecting new interactions from the environment and correcting the estimated Q-values. The details of the online fine-tuning process are presented in Algorithm 2, similar to the training process of DDQN. Unlike DDQN, the parameters of Q-function neural networks are loaded with the offline trained neural network parameters instead of random values, and the gradients are calculated using the loss function of CQL instead of that of DDQN.

V. EXPERIMENTS

We present the experimental setting of ACORN and give a detailed analysis of its experimental results in this section. The parameter settings of ACORN and the simulation environment are introduced at first. Then, we test the performance of ACORN by comparing it with some rule-based and DRL-based methods. We also evaluate the offline training convergence of ACORN with a DL-based method and an online off-policy DRL-based method, and compare the online interaction reduction of ACORN with several DRL-based methods.

A. Simulation Environment

The simulation environment is a MEC network including a remote cloud and two BSs, where the BSs communicate with the cloud via a wired backhaul network. Referring to [29], [30], we choose similar parameters. The number of types of all contents in the network K is set to 10, the content size is set to 1 GB, and the cache size of all BSs is set to 3 GB. To simulate the long latency of the content fetching process, we set the number of the backhaul links to 10 and their bandwidth to 512 MB/s.

TABLE II
THE PARAMETERS OF NEURAL NETWORK, TRAINING AND TUNING

	Parameter	Value
Offline Training	Learning rate	3×10^{-6}
	Target Q-function update Interval	2000
	Batch Size	32
	Optimization Method	Adam
Online Tuning	Learning rate	5×10^{-6}
	Q-function update Interval	1
	Target Q-function update Interval	2000
	Batch Size	128
	Optimization Method	Adam
Neural Network	CNN Kernel Size	2
	CNN Input Channels	3
	CNN Output Channels	64
	MaxPool Padding Size	2
	Encoding FCNN layers	1
	Encoding FCNN neurons	64
	Output FCNN layers	2
	Output FCNN neurons	64
	Activation Function	ReLU

To simulate the request arrivals and the content popularity, we use a real-world dataset, Movielens [31], by assuming that each movie corresponds to one content and each movie rating item corresponds to a request (which is similar to [32]). Thus, a user's rating of a movie at a specific time corresponds to a user's request for content with the same ID arriving at the same time. To simulate the explosion of user requests in the MEC network, We divide the time into multiple timeslots of one hour in duration, and use the requests of one timeslot to simulate the requests of one second in the MEC network.

To validate that ACORN can effectively cope with various user preferences without human interventions, we evaluate the cache performance of ACORN on the two BSs with different distributions of content popularity. We selected requests for 10 movies from two different periods, then calculated the proportion of the request number for each movie to the total request number for all movies to obtain their content popularity distribution, as shown in Fig. 5. Compared with the distribution of content popularity 2, the content popularity of all contents in distribution 1 is relatively close. On the contrary, distribution 2 shows a very different user preference, with users preferring content 7 and 10 over other contents.

B. Parameters

ACORN is based on a PyTorch implementation. The 1-D CNN of the Q-function neural network is set to have 3 input channels (equal to the cache size), 2 kernel sizes, and 64 output channels, followed by a 1-D MaxPool layer with padding size 2. The encoding FCNN for processing the information of the newly fetched contents is set as a one-layer network with 64 neurons, and the output FCNN has two layers with 64 neurons in each layer. ReLU is selected as the activation function considering its fast updating speed.

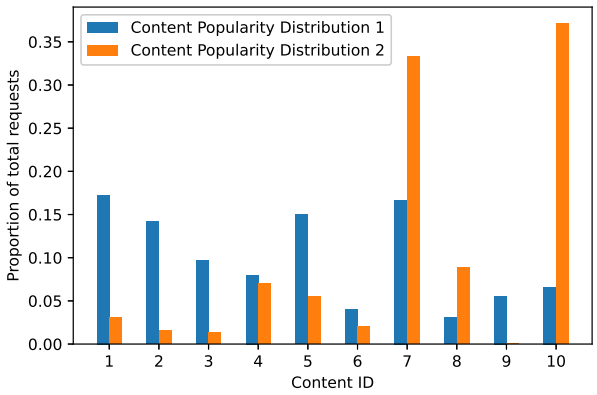


Fig. 5. The comparision between two distributions of content popularity.

For the offline training parameter setting of ACORN, the learning rate is set to 3×10^{-6}. The update interval of the target Q-function is set to 2000, and the sampled batch size is 32. The parameters of the Q-function are optimized via Adam [33].

For the online fine-tuning parameter setting of ACORN, the learning rate is set to 5×10^{-6}. The update interval of Q-function and target Q-function is set to 1 and 2000, respectively. The sampled batch size is 128. The parameters of the Q-function are also optimized via Adam. Overall, the parameter setting of ACORN is summarized in Table II. It is worth noting that the neural network parameters for both BSs are identical.

C. Performance Evaluation

We test the cache performance of the policies learned by ACORN on two BSs associated with users with different user preferences. We also compare the performance of ACORN based on different training methods with the performance of several rule-based and learning-based methods. The learning-based methods are shown as follows:

- *DL:* The mapping relations between states and actions are learned using only the states as inputs and the actions as outputs. The loss function is the MSE loss between the given and output actions. Different from the DRL-based methods, DL does not leverage the rewards in the dataset.
- *DDQN (online):* DDQN stores the collected online interactions in a replay buffer and then randomly samples a minibatch of interactions from the buffer to update the neural network parameters with Eq. 18. This process repeats multiple times until the Q-function converges or the number of training epochs exceeds a preset value.
- *DLDDQN:* DLDDQN is modified from [15]. It uses an offline policy learned by DL as the initial training point for the online DDQN. The training phase of DDQN is then used to improve the offline policy further.

Different training methods of ACORN are introduced as follows:

- *ACORN (offline):* Only use the static dataset sampled by a behavior policy to learn a policy. The policy is trained by the offline training algorithm of ACORN.
- *ACORN (online):* Learn a policy from scratch using the online fine-tuning algorithm of ACORN. The parameters of Q-function neural networks are loaded with random values at first.
- *ACORN:* The offline policy learned by the offline training algorithm of ACORN is used to collect new interactions. After that, the parameters of the offline policy are fine-tuned with these new interactions via the online fine-tuning algorithm.

The rule-based methods are decribed as follows:

- *LRU:* The most recent used time of each cached content is recorded. When new contents are fetched from the cloud to the BS, it will replace the least recent used contents with new ones if the BS is full.
- *LFU:* The number of requests for each cached content is recorded. When newly fetched contents arrive at the BS, it will replace the least requested contents with new ones if the BS is full.
- *FIFO:* The cached time of each cached content is required to be tracked. When newly fetched contents arrive at the BS, it will evict the earliest cached contents and store new ones if the BS is full.
- *Belady:* Belady has a view of future requests. It records the most recent future requests of each cached content. When newly fetched contents arrive at the BS, it will replace the least recent contents in the future with new ones if the BS is full. Although Belady cannot be realized in the real-world as it is not possible to obtain the perfectly accurate information of future requests in reality, Belady can be used as the upper bound of the rule-based methods to evaluate the performance of ACORN.

We use LRU as the behavior policy to sample a static dataset. The length of the future view for Belady is set as 48 seconds.

The results are shown in Fig. 6. We can find that offline ACORN works with about 38% higher performance than LRU on the BS with the content popularity distribution 1, and about 23% higher than LRU on the BS with the content popularity distribution 2. It indicates that offline ACORN can learn a competitive or even better policy from a static dataset sampled by the behavior policy. The results also show that all DRL-based methods, including ACORN, have better performance and are capable of handling different user preferences, compared to the rule-based methods. This proves that DRL is a promising way to solve the cache replacement problem. Moreover, we see that the performance of offline ACORN is not much worse than online DDQN, with a degradation factor of 6% on the BS with the distribution 1 and only 2% on the BS with the distribution 2, respectively. Compared with the performance gap between offline ACORN and rule-based methods, the gap between offline ACORN and the DRL-based methods trained online is much smaller. It means that the policy learned by offline ACORN can be directly applied to the edge caching environment without any tuning in the case of a

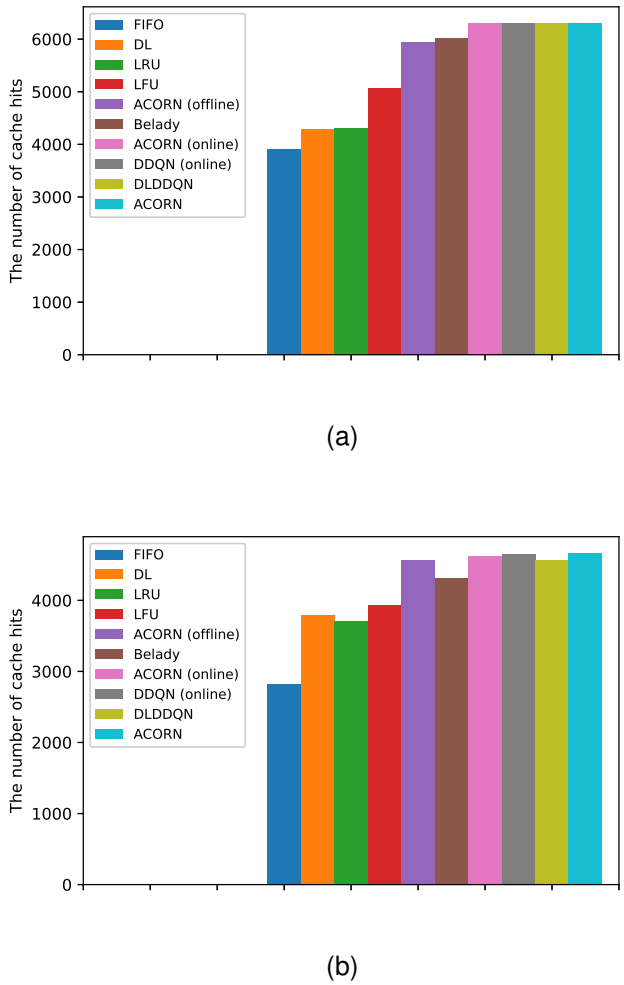


Fig. 6. Comparison of cache performance of policies generated from ACORN and other rule-based and learning-based methods. Offline ACORN learns a better policy from the dataset in two distributions of content popularity. The performance gap between offline ACORN and online DDQN is tiny. (a) Content Popularity Distribution 1. (b) Content Popularity Distribution 2.

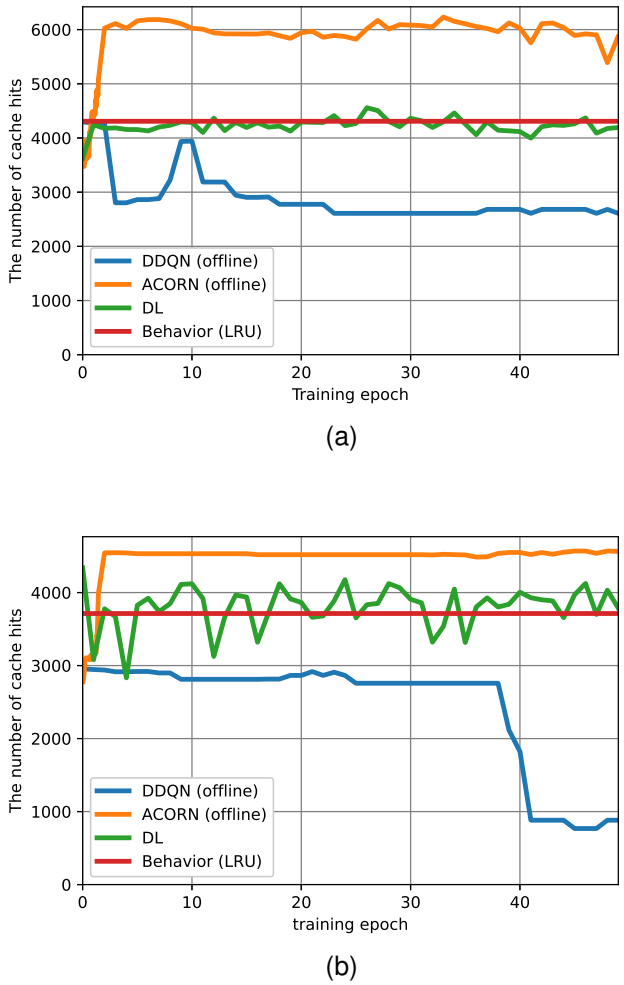


Fig. 7. Comparing the learning curves of the policy generated from ACORN with those from learning-based methods and the behavior policy. ACORN converges successfully, and its learned policy performs better than others. (a) Content Popularity Distribution 1. (b) Content Popularity Distribution 2.

computationally overloaded edge server. In addition, although Belady owns a view of future requests, it still follows a simple rule to replace contents, which makes it perform less well than the online DRL-based methods.

D. Convergence of Offline Training

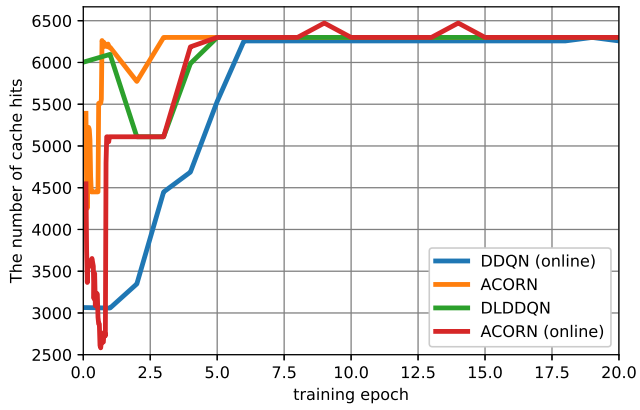
We test the offline training convergence of ACORN and compare it to an offline DRL-based method modified from DDQN:

- *DDQN (offline):* We can easily change online DDQN to offline DDQN by eliminating the data collection phase and replacing the replay buffer with a static dataset.

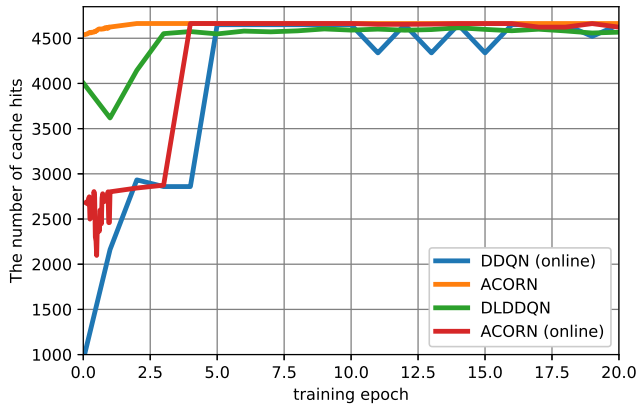
We plot the learning curves for the different methods in Fig. 7. It can be found that ACORN successfully converges at epoch 3 on the BS with distribution 1 and epoch 2 on the BS with distribution 2. The learned policy performs better than the behavior policy LRU in both distributions. Since the loss function of DL is the MSE loss between given actions in the dataset and the output actions, Fig. 7 only shows its performance curve but not its learning curve. The curve of DL fluctuates around the curve of the behavior policy, which implies that DL successfully learns the state-action mapping relations of the dataset. Due to the lack of new samples to correct the estimated Q-values and the unconstrained exploration, offline DDQN cannot learn a better policy and even cannot converge.

E. Reduction of Online Interactions

The reason why the offline policy learned by ACORN does not perform as well as the policies learned by the DRL-based methods trained online is that the static dataset only contains partial information about environmental dynamics. It is challenging for ACORN to learn an optimal policy using only this limited information. Fortunately, the missing information about environmental dynamics can be supplemented with a small number of online interactions. In this part, we want to test how many interactions can be reduced by fine-tuning the offline policy using ACORN to achieve a similar number



(a)



(b)

Fig. 8. Learning curves comparing the policies generated from ACORN based on different training methods with those from DDQN and DLDDQN. Even though all learned policies achieve similar performance, ACORN requires fewer online interactions than DLDDQN, online ACORN, and online DDQN. (a) Content Popularity Distribution 1. (b) Content Popularity Distribution 2.

of cache hits compared to training from scratch using the advanced DRL-based methods.

We plot the tuning curves for these methods in Fig. 8, where each training epoch contains 2000 online interactions. We find that all methods learn an excellent policy by executing the online training/tuning phase, demonstrating that ACORN can compete with DDQN even in the online setting. Although all learned policies have similar cache performance finally, they require different numbers of interactions to reach convergence. ACORN requires fewer interactions than other methods because offline ACORN provides a better initial point for fine-tuning. On the BS with the distribution 1, ACORN converges at epoch 3. It reduces the number of online interactions by 40% compared to DLDDQN and online ACORN that converge at epoch 5, and by 50% compared to online DDQN that converges at epoch 6. For the BS with the distribution 2, ACORN that converges at epoch 2 reduces the number of online interactions by 33%, 50%, and 60% compared to DLDDQN that converges at epoch 3, online ACORN that converges at epoch 4, and online DDQN that converges at epoch 5, respectively. We can find that under different user preferences, ACORN can effectively reduce the number of online interactions while achieving a similar number of cache hits compared to the advanced DRL-based methods. It means that ACORN is quite applicable to edge networks with time-varying user preferences where frequent retraining processes occur. Although DLDDQN also improves the policy from the initial point obtained by DL, it still requires more interactions than ACORN to achieve similar cache performance, which proves the superiority of ACORN again.

VI. Conclusion

A novel dynamic edge cache replacement method, named ACORN, is proposed to meet the dynamic requirements of mobile users in a MEC network. To the best of our knowledge, it is the first work to apply the offline-online DRL to solve the edge cache replacement problem. Distinguishing from the existing work, ACORN can learn a competitive or even better policy from a static dataset sampled by a behavior policy. It can further improve the learned policy within a small number of online interactions via fine-tuning. A CNN is used to efficiently extract the knowledge about content popularity from the request history, and an FCNN is used to further improve the performance by encoding the additional information about newly fetched contents and combing it with the extracted content popularity knowledge. We conduct extensive simulations in our designed edge caching environment with two different user preferences from three aspects, cache performance, offline training convergence, and online interaction reduction. The experimental results demonstrate that ACORN can successfully learn an offline policy by only interacting with a static dataset, and the offline policy outperforms the behavior policy. The results also show that ACORN can fine-tune the offline policy to achieve a number of cache hits as good as that learned by the advanced DRL-based methods while using fewer online interactions.

References

[1] Ericsson mobility report. [Online]. Available: https://www.ericsson.com/en/reports-and-papers/mobility-report/reports/june-2022

[2] J. Shuja, K. Bilal, W. Alasmary, H. Sinky, and E. Alanazi, "Applying machine learning techniques for caching in next-generation edge networks: A comprehensive survey," *J. Netw. Comput. Appl.*, vol. 181, no. 1, p. 103005, 2021.

[3] H. Zhu, Y. Cao, W. Wang, T. Jiang, and S. Jin, "Deep reinforcement learning for mobile edge caching: Review, new features, and open issues," *IEEE Netw.*, vol. 32, no. 6, pp. 50–57, 2018.

[4] A. Mehrabi, M. Siekkinen, and A. Yl-Jaaski, "Qoe-traffic optimization through collaborative edge caching in adaptive mobile video streaming," *IEEE Access*, vol. 6, pp. 52 261–52 276, 2018.

[5] S. Shukla, O. Bhardwaj, A. A. Abouzeid, T. Salonidis, and T. He, "Proactive retention-aware caching with multi-path routing for wireless edge networks," *IEEE J. Sel. Areas Commun.*, vol. 36, no. 6, pp. 1286–1299, 2018.

[6] G. Qiao, S. Leng, S. Maharjan, Y. Zhang, and N. Ansari, "Deep reinforcement learning for cooperative content caching in vehicular edge computing and networks," *IEEE Internet Things J.*, vol. 7, no. 1, pp. 247–257, 2020.

[7] C. Zhong, M. C. Gursoy, and S. Velipasalar, "Deep multi-agent reinforcement learning based cooperative edge caching in wireless networks," in *IEEE ICC*. Shanghai, China: IEEE, 2019, pp. 1–6.

[8] H. Zhou, T. Wu, H. Zhang, and J. Wu, "Incentive-driven deep reinforcement learning for content caching and d2d offloading," *IEEE J. Sel. Areas Commun.*, vol. 39, no. 8, pp. 2445–2460, 2021.

[9] Q. Li, Y. Zhang, Y. Li, Y. Xiao, and X. Ge, "Capacity-aware edge caching in fog computing networks," *IEEE Trans. Veh. Technol.*, vol. 69, no. 8, pp. 9244–9248, 2020.

[10] X. Xia, F. Chen, J. Grundy, M. Abdelrazek, H. Jin, and Q. He, "Constrained app data caching over edge server graphs in edge computing environment," *IEEE Trans. Serv. Comput.*, 2021, early access.

[11] X. Xia, F. Chen, Q. He, G. Cui, J. Grundy, M. Abdelrazek, A. Bouguettaya, and H. Jin, "Ol-medc: An online approach for cost-effective data caching in mobile edge computing systems," *IEEE Trans. Mobile Comput.*, 2021, early access.

[12] P. Wu, J. Li, L. Shi, M. Ding, K. Cai, and F. Yang, "Dynamic content update for wireless edge caching via deep reinforcement learning," *IEEE Commun. Lett.*, vol. 23, no. 10, pp. 1773–1777, 2019.

[13] G. M. S. Rahman, M. Peng, S. Yan, and T. Dang, "Learning based joint cache and power allocation in fog radio access networks," *IEEE Trans. Veh. Technol.*, vol. 69, no. 4, pp. 4401–4411, 2020.

[14] F. Wang, F. Wang, J. Liu, R. Shea, and L. Sun, "Intelligent video caching at network edge: A multi-agent deep reinforcement learning approach," in *IEEE INFOCOM*. Toronto, ON, Canada: IEEE, 2020, pp. 2499–2508.

[15] C. Zhong, M. C. Gursoy, and S. Velipasalar, "A deep reinforcement learning-based framework for content caching," in *52nd Annual CISS*. Princeton, NJ, USA: IEEE, 2018, pp. 1–6.

[16] X. Zhang, Y. Zhou, D. Wu, M. Hu, X. Zheng, M. Chen, and S. Guo, "Optimizing video caching at the edge: A hybrid multi-point process approach," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 10, pp. 2597–2611, 2022.

[17] W. Jing, X. Wen, Z. Lu, and H. Zhang, "User-centric delay-aware joint caching and user association optimization in cache-enabled wireless networks," *IEEE Access*, vol. 7, pp. 74 961–74 972, 2019.

[18] X. Zhou, Z. Liu, M. Guo, J. Zhao, and J. Wang, "Sacc: A size adaptive content caching algorithm in fog/edge computing using deep reinforcement learning," *IEEE Trans. Emerg. Topics Comput.*, 2021.

[19] N. Atre, J. Sherry, W. Wang, and D. S. Berger, "Caching with delayed hits," in *ACM SIGCOMM*. Virtual Event, NY, USA: ACM, 2020, pp. 495–513.

[20] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[21] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," in *ICLR*, 2016.

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

[23] T. Liu, S. Ni, X. Li, Y. Zhu, L. Kong, and Y. Yang, "Deep reinforcement learning based approach for online service placement and computation resource allocation in edge computing," *IEEE Trans. Mobile Comput.*, 2022, early access.

[24] S. Levine, A. Kumar, G. Tucker, and J. Fu, "Offline reinforcement learning: Tutorial, review, and perspectives on open problems," *arXiv preprint arXiv:2005.01643*, 2020.

[25] H. I. Fawaz, B. Lucas, G. Forestier, C. Pelletier, D. F. Schmidt, J. Weber, G. I. Webb, L. Idoumghar, P.-A. Muller, and F. Petitjean, "Inceptiontime: Finding alexnet for time series classification," *Data Min. Knowl. Discov.*, vol. 34, pp. 1936–1962, 2020.

[26] K. Kashiparekh, J. Narwariya, P. Malhotra, L. Vig, and G. Shroff, "Convtimenet: A pre-trained deep convolutional neural network for time series classification," in *IEEE IJCNN*. Budapest, Hungary: IEEE, 2019, pp. 1–8.

[27] A. Kumar, A. Zhou, G. Tucker, and S. Levine, "Conservative q-learning for offline reinforcement learning," in *NeurIPS 33*, 2020, pp. 1179–1191.

[28] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *AAAI*, vol. 30, no. 1, Phoenix, Arizona, USA, 2016.

[29] Z. Yang, Y. Liu, Y. Chen, and L. Jiao, "Learning automata based q-learning for content placement in cooperative caching," *IEEE Trans. Commun.*, vol. 68, no. 6, pp. 3667–3680, 2020.

[30] A. Sadeghi, F. Sheikholeslami, and G. B. Giannakis, "Optimal and scalable caching for 5g using reinforcement learning of space-time popularities," *IEEE Trans. Signal Process.*, vol. 12, no. 1, pp. 180–190, 2018.

[31] F. M. Harper and J. A. Konstan, "The movielens datasets: History and context," *ACM Trans. Interact. Intell. Syst.*, vol. 5, no. 4, pp. 1–19, 2016.

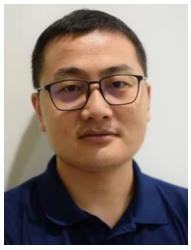
[32] S. Mller, O. Atan, M. van der Schaar, and A. Klein, "Context-aware proactive content caching with service differentiation in wireless networks," *IEEE Trans. Wirel. Commun.*, vol. 16, no. 2, pp. 1024–1036, 2017.

[33] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *ICLR*, 2015, pp. 1–15.

Zhe Wang is a Ph.D. student in Computer Science at the University of Exeter. He received both B.Eng. and M.Eng. degree in Computer Science from the University of Electronic Science and Technology of China (UESTC), Chengdu, China in 2016 and 2019, respectively. His research interests include deep reinforcement learning, cloud and edge computing, and computer system optimization.

Jia Hu received the BEng and MEng degrees in electronic engineering from the Huazhong University of Science and Technology, China, in 2006 and 2004, respectively, and the PhD degree in computer science from the University of Bradford, UK, in 2010. He is an associate professor of computer science at the University of Exeter. His research interests include edge-cloud computing, resource optimization, applied machine learning, and network security.

Geyong Min received the BSc degree in computer science from the Huazhong University of Science and Technology, China, in 1995, and the PhD degree in computing science from the University of Glasgow, United Kingdom, in 2003. He is a professor of high performance computing and networking with the Department of Computer Science within the College of Engineering, Mathematics and Physical Sciences at the University of Exeter, United Kingdom. His research interests include computer networks, wireless communications, parallel and distributed computing, ubiquitous computing, multimedia systems, modeling and performance engineering.



Zhiwei Zhao is currently a full professor at the School of Computer Science and Engineering in University of Electronic Science and Technology of China. He received his PhD degree at the College of Computer Science, Zhejiang University in 2015. His research interests include edge computing, IoT systems, low-power computing, etc. He is a member of IEEE, ACM and CCF.

Zi Wang is currently a Postdoctoral Research Fellow at the University of Exeter, UK. He received his Ph.D. from the University of Exeter, UK, in 2023, and the M.Eng. and B.Eng. in Computer Science from the University of Electronic Science and Technology of China in 2018 and 2015, respectively. His research interests include wireless networks, federated learning, edge-cloud computing, and applied artificial intelligence.