IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received June 8, 2019, accepted June 19, 2019, date of publication June 26, 2019, date of current version July 11, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2925019

A Deep Reinforcement Learning Approach to Proactive Content Pushing and Recommendation for Mobile Users

DONG LIU, (Student Member, IEEE), AND CHENYANG YANG, (Senior Member, IEEE)
School of Electronics and Information Engineering, Beihang University (BUAA), Beijing 100191, China

Corresponding author: Dong Liu (dliu@buaa.edu.cn)

This work was supported in part by the MOE-CMCC Science Foundation of China under Project 1-4 MCM2017, and in part by the National Natural Science Foundation of China (NSFC) under Grant 61731002.

ABSTRACT The gain from proactive caching at mobile devices highly relies on the accurate prediction of user demands and mobility, which, however, is hard to achieve due to the random user behavior. In this paper, we leverage personalized content recommendation to reduce the uncertainty of user demands in sending requests. We formulate a joint content pushing and recommendation problem that maximizes the net profit of a mobile network operator. To cope with the challenges in modeling and learning user behavior, we establish a reinforcement learning (RL) framework to resolve the problem. To circumvent the curse of dimensionality of reinforcement learning for the joint problem, that is, with very large action and state spaces, we decompose the original problem into two RL problems, where two agents with different goals operate together, and we limit the number of possible actions in each state of the pushing agent by harnessing the well-learned recommendation policy. To enable the generalization of action values from experienced states to the unexperienced states with function approximation, we find a proper way to represent the state and action of the pushing agent. Then, we resort to double deep-Q network with dueling architecture to solve the two problems. The simulation results show that the learned recommendation and pushing policies are able to converge and can increase the net profit significantly compared with baseline policies.

INDEX TERMS Wireless edge caching, content recommendation, pushing, deep reinforcement learning.

I. INTRODUCTION

Content caching at the wireless edge has been acknowledged as a promising way to support the explosively increasing traffic demands and improve user experience [1]–[4]. By caching at base stations (BSs), the traffic load of backhaul and service latency of users can be reduced, and network throughput and energy efficiency can be improved dramatically [5]–[7]. By further proactively pushing the contents into the cache of mobile devices at favorable channel conditions [3], [8], [9], users can enjoy zero latency with low cost of the network if the requested contents are pushed.

Due to the huge number of available contents at content providers but limited cache size at each mobile device, the benefit of proactive pushing highly relies on the prediction of user behavior in requesting contents. Although user preference, i.e., the probability distribution of a user requesting every content in a library [10], [11] or the rating of the user for each content [12], can be learned via machine learning techniques such as collaborative filtering [11]–[13], the uncertainty of user behaviors still makes it hard to precisely predict when and where a user will request which content. Consequently, the efficiency of proactive caching, e.g., the cache-hit ratio, at mobile devices may be unsatisfactory [3].

On the other hand, user demands are increasingly driven by recommendation systems, whose goal is to relieve users from information overload by recommending the contents best matching the preference of individual user. This, in turn, improves user stickiness and boost the number of content requests [14]. For example, driven by a powerful recommendation algorithm, Douyin (Tik Tok in the US), a short video app in China, was the most downloaded iPhone app in the world for the first quarter of 2018 [15]. Considering that

The associate editor coordinating the review of this manuscript and approving it for publication was Zhenhui Yuan.

people often do not know what they want until you show it to them, recommendation systems can be leveraged as a powerful tool to reduce the uncertainty in user demands, which has the potential to unleash the caching gain.

Considering that recommendation and wireless edge caching are operated so far by different entities, i.e., content providers and mobile network operators (MNOs), respectively, recommendation policy and caching policy have been designed independently in different communities [1]–[14], [16]–[18]. Yet content caching and recommendation are coupled with each other, since recommendation has large influence on user demands, which further affects caching policy. Recently, there is a trend towards integrating content recommendation with wireless edge caching [19]–[23]. In [19], user demands are rendered less uncertain by modifying the rating of contents shown to each user so that the cost of content provider for proactive pushing can be reduced. In [20]–[22], by recommending contents that are both cached at BSs and appealing to each user, the cache-hit ratio at BSs can be increased. In [20], the BS first optimizes caching policy based on recommending top-N contents according to the preference of each user, and then adjusts the recommendation lists based on the cached contents. In [21], a caching policy was optimized to maximize a "soft" cache-hit ratio by recommending related contents in the cache if the originally requested content is not cached in nearby BSs.

However, how user demands are affected by recommendation is assumed known *a priori* in [19]–[21]. In particular, the probability distribution of a user requesting for each content after recommendation is known in [19], [20], and the probability of a user accepting a recommended content is known in [21]. Such assumption is unrealistic, because the reaction of users to recommendation is hard to observe without a controlled experiment to measure the requests of a user for contents before and after recommendation. In [22], the impact of recommendation on user demands was modeled by a user-specific psychological threshold. Then, an ε-greedy algorithm was proposed to find the policy by learning the threshold via interactions with users. In [23], reinforcement learning (RL) was adopted to learn the caching policy for BSs with recommendation, where the recommended contents are the cached contents at the BS and are identical for every user due to not differentiating the preferences of users. Such recommendation improves the caching gain by making the users aware about the files locally cached at nearby BS. Nonetheless, user preferences are heterogeneous in practice [11], hence the recommended contents may not match the taste of each user, which leads to the performance loss from personalized recommendation as evaluated in [22].

In this paper, we jointly consider proactive content pushing and recommendation for mobile users to maximize the net profit of MNO, which is the revenue minus the transmission cost. To deal with the difficulty in modeling individual user behavior, we formulate the problem under RL framework. To deal with the large action and state spaces, we decompose the problem into two RL problems by decoupling the role of recommendation and pushing, and leverage the ability of recommendation in reducing user demands uncertainty. The first problem aims to help a user find a preferred content to increase user stickiness, which also boosts user requests and hence the revenue. The second problem aims at reducing the transmission delay (hence improving user experience), which also reduces transmission cost. It is worthy to note that these two problems are not independent, where the pushing policy relies on the recommendation policy. Then, we resort to deep reinforcement learning (DRL) algorithm, specifically double deep Q-network (DDQN) with dueling architecture [24], to solve the two RL problems. Finally, we compare such a value-based DRL algorithm with several state-of-the-art policy-based DRL algorithms, i.e., deep deterministic policy-gradient (DDPG) [25], advantage actor-critic (A2C) [26], and proximal policy optimization (PPO) [27].

The major contributions are summarized as follows:

- Different from [19]–[21] that assume a known model of how user requests are affected by content recommendation, the proposed framework is model-free, which does not require any priori knowledge of user behaviors in requesting contents as well as mobility pattern. Different from [22] where a psychological threshold is learned, we directly learn the recommendation and pushing policies. Different from [23] where the recommendation is common for all users and the caching policy is designed for BS, we consider personalized recommendation and pushing to user devices. To the best of our knowledge, this is the first attempt to jointly optimize content recommendation and pushing under the RL framework.
- The complexity of learning is reduced by three means: 1) We decompose the problem so that two agents with different sub-goals are trained sequentially but work together after training to achieve the final goal. The first agent learns the recommendation policy to boost revenue and reduce the uncertainty of user requests, while the second agent learns the pushing policy to reduce transmission cost. 2) We limit the number of possible actions of the second agent by only pushing the contents that the user is likely to request in near-future time steps, which can be predicted after the recommendation policy is well-learned by the first agent. 3) We represent the states and actions of the pushing agent in a proper way to enable better generalization of action values from experienced states to unexperienced states.
- Simulation results show that the learned recommendation and pushing policies can converge and increase the net profit significantly compared with baseline policies, and the adopted value-based DRL algorithm outperforms the policy-based DRL algorithms.

The rest of the paper is organized as follows. In Section II, we introduce the system model. Section III formulates the joint recommendation and pushing problem, decomposes the problem into two problems, and solves them by DRL.

Simulations results are given in Section IV. Conclusions and future works are discussed in Section V.

II. SYSTEM MODEL

In this section, we first introduce the notions to be used throughout the paper and the network architecture to support reinforcement learning. Then, we describe the model of the system with content recommendation and pushing as well as the basic idea of joint recommendation and pushing policy.

A. BASIC NOTIONS IN REINFORCEMENT LEARNING

A standard RL problem can be described as a Markov decision process, where an *agent* learns from interactions with an *environment* to achieve a goal [28]. For an episodic Markov decision process, the agent and environment interact in a sequence of discrete time steps $t = 1, 2, \cdots, T$ constituting an *episode*. At each time step t, the agent observes the *state* of the environment $\mathbf{s}(t)$ and executes an *action* $\mathbf{a}(t)$. Then, the agent receives a reward $r(t)$ from the environment and transits into a new state $\mathbf{s}(t+1)$. The interaction of the agent with the environment is then captured by an *experience* vector $\mathbf{e}(t) \triangleq [\mathbf{s}(t), \mathbf{a}(t), r(t), \mathbf{s}(t+1)]$.

The goal of the agent is to learn a *policy* from its experiences to maximize an expected *return*, which reflects the cumulative reward received by the agent during the T-time-step episode. The policy (denoted by π) determines which action should be executed in which state. At each time step t, π is learned from the past experiences $\mathcal{D} = \{\mathbf{e}(1), \cdots, \mathbf{e}(t-1)\}$. The expected return is defined as $\mathbb{E}[\sum_{i=1}^{T} \gamma^{i-1} r(i)]$, where γ denotes the discount factor.

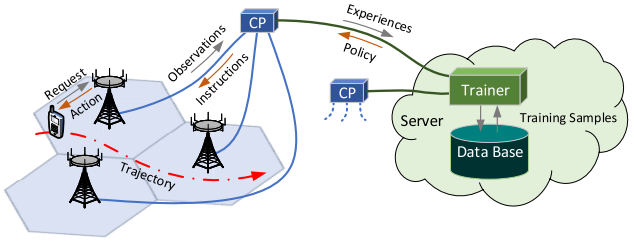


FIGURE 1. Learning-enabled network architecture. The CPs and the trainer jointly serve as the agent but reside in different network entities.

B. LEARNING-ENABLED NETWORK ARCHITECTURE

Consider a learning-enabled cellular network architecture introduced in [29], where a server is connected with multiple central processors (CPs) and each CP is connected with multiple BSs in an area, as shown in Fig. 1. The server consists of a trainer and a data base. Each CP can monitor the status of mobile users (say monitor content requests and gather channel conditions) via BSs and process the collected observations into experience vectors, which are then sent to the server and stored in the data base. A centralized trainer at the server learns the policy based on the experiences stored in the data base, and issues the learned policy to the CPs. Each CP stores the learned policy, based on which the CP controls the BSs to execute actions by sending instructions.

Each CP and the trainer jointly serve as the so-called agent in RL parlance, but reside in different network entities owing to the following reasons. The CP is located closer to BSs, hence the delay and overhead of control signaling (e.g., instructions updated in the timescale of seconds) can be reduced. The policy stored at the CP can be updated less frequently (say hours or days). Hence, the trainer is centralized at the top of the network to enable sufficient use of computing resources (e.g, GPUs) and data resources (i.e., experiences) gathered throughout the whole network.

C. RECOMMENDATION, PUSHING, AND USER REQUESTS

Consider mobile users traveling across multiple cells covered by a CP, which handles user association and controls the associated BS to execute the recommendation and pushing actions. As an illustration, assume that the mobility pattern of each user is characterized by a Markov process, where the probability distribution of the next location of a user depends on its current location and last location.[1]

Each user is equipped with a cache that can pre-store C contents and may request contents from a library of contents (stored at the server or BSs).

A mobile user may continuously request contents during a period of time, e.g., watch short videos one by one on a smart phone app. We call such period of continuously sending requests by a user as a session (i.e., an episode in RL parlance), which can be divided into discrete time steps, as shown in Fig. 2. After a user starts a session, the user requests and consumes a content in each time step.

At the beginning of each time step, the CP controls the associated BS of the user to recommend a content to the user from a pre-determined recommendation candidate set containing F contents[2] according to the stored policy. The recommendation is issued to the user by only presenting the title or thumbnail of the content through the app, which incurs negligible transmission cost. If the user accepts the recommendation, the user will request the recommended content, and close the app (i.e., terminate the session) with probability q_1 after consuming the requested content. If the user rejects the recommendation, the user will instead request another content according to a specific distribution, and terminate the session with probability q_2 after consuming the requested content. Because users are more likely to continue requesting contents if their preferred contents are recommended to them, $q_1 < q_2$. This reflects the impact of recommendation on user stickiness.

[1] The last location is considered because the difference between the current location and the last location of a user can reflect the moving direction and speed. Our RL framework can be extended into the case with more complicated mobility pattern by including the transmission costs during multiple past time steps into the state vector as discussed later.

[2] We assume that the recommendation candidates is winnowed down to hundreds contents by machine learning methods, say by deep neural networks [30].

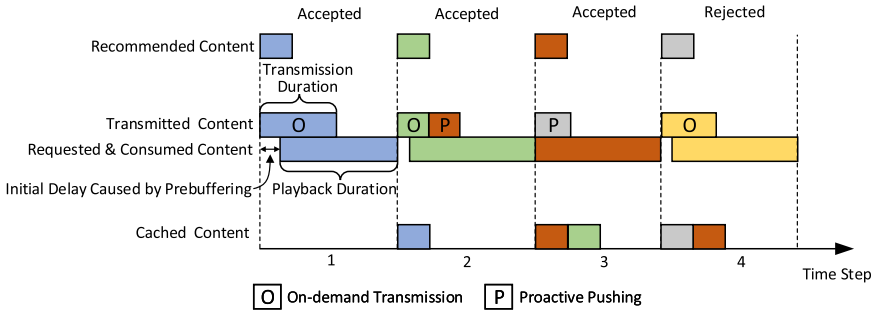


FIGURE 2. Illustration of a session with four requests (i.e., $T = 4$ time steps), each color represents a content. The BS recommends one content in each time step and pushes one content in time steps 2 and 3.

Meanwhile, at each time step, the CP can also control the associated BS to push (or not to push, e.g., when the channel is not good) one content proactively into the cache of the user device. By proactive pushing, if a user requests a content prestored in its own cache, the content can be directly retrieved from the cache without the need of prebuffering, i.e., the user can enjoy zero latency. If the requested content is not cached in its own device, the associated BS will transmit the content to the user. Such an on-demand transmission introduces cost to the network (e.g., energy consumption at the BS) and initial delay to the user. The value of the initial delay depends on where the content is stored: the server or the associated BS.

Content recommendation affects which content the user will request, while pushing introduces transmission cost. When a user undergoes bad channel condition so that the pushing cost is high, or the pushed content is finally not requested by the user before the end of the session, the total transmission cost will increase. Therefore, in each time step, the CP should intelligently decide which content to recommend, whether or not to push, and which content to push into the user device, according to a policy π learned from past experiences $\mathcal{D}$.

In Fig. 2, we provide an example to help understand the recommendation and pushing procedure in a session with duration of $T = 4$ time steps. In time step 1, the user starts a session (i.e., opens the app). During the session, the BS recommends a content at the beginning of each time step. In time step 1, the user accepts the recommendation. Since the recommended content has not been cached in the mobile device, the user is served by on-demand transmission, hence prebuffering is required. According to the instruction from the CP, the BS does not push any content in time step 1, because the transmission cost is higher than the predicted transmission costs in future time steps. In time steps 2 and 3, the BS pushes the contents to be recommended in time steps 3 and 4, respectively. Because the recommended content in time step 3 is already pushed to the user in time step 2, when the user accepts the recommendation in time step 3, prebuffering is unnecessary. In time step 4, the user rejects the recommendation, and terminates the session after it consumes the 4th requested content.

The request history of a user reflects the preference of the user, and a user will accept a recommendation with high probability if the user prefers the recommended content. This suggests that the probability of a user accepting a recommendation (called as *acceptance probability*) depends on its recently consumed contents [31]. Let $\mathcal{Q}$ denote the set of recently consumed contents of a user and $p_{\mathcal{Q},j}$ denote the probability that the user will accept the recommendation of content j. For notational simplicity, we consider the case where the acceptance probability depends on the most recently consumed content of the user as an illustration. In this case, $p_{\mathcal{Q},j}$ can be simplified into p_{ij}, which denotes the probability that the users who have just consumed content i will accept the recommendation of content j. Then, the reaction of a user to recommendation can be characterized by a matrix $\mathbf{P} = [p_{ij}]_{F\times F}$.

Without recommendation, it may not be easy for a user to find a preferred content due to the vast amount of available contents. Hence, the request probability for each content is low. By contrast, the probability of a user requesting a recommended content is high if the content is sufficiently attractive. In other words, a good recommendation can reduce the uncertainty in user requests.

III. RECOMMENDATION AND PUSHING BY DRL

In this section, we first formulate the joint recommendation and pushing problem into RL framework. Since the original problem is with large state and action spaces, we decompose the joint optimization into two RL problems and resort to DRL to solve them.

A. PROBLEM FORMULATION

Considering that the energy consumed at a BS for transmitting a content to a user increases with the duration of transmission, the transmission cost, denoted as $m(t)$, is modeled as proportional to the transmission duration. Since mobile users experience fast fading channels, the duration for transmitting a content is much longer than the channel coherence time (in millisecond timescale), and hence is inversely proportional to the average rate. Therefore, $m(t)$ can be modeled as $m(t) = \beta/R(t)$, where β is a scaling factor representing

the cost for transmitting a content per unit time and $R(t)$ is the average rate of a user.

Assume that each user is associated to the BS with the strongest average received power, and each BS serves the associated users over orthogonal time-frequency resources and allocates a fixed amount of bandwidth to each associated user. Then, $R(t)$ can be expressed as

$$R(t)=\mathbb{E}_{h_i}\left[W\log_2\left(1+\frac{Ph_0d_0(t)^{-\alpha}}{\sum_{i\in\Phi\setminus 0}Ph_id_i(t)^{-\alpha}+\sigma^2}\right)\right] \quad (1)$$

where W is the transmission bandwidth for the user, h_0 and $d_0(t)$ are the small-scale fading channel and the distance between the associated BS to the user, respectively, α is the path-loss exponent, P and σ^2 are the transmit power of each BS and the noise power, respectively, Φ is the set of the BSs in the network, $\sum_{i\in\Phi\setminus 0}Ph_id_i(t)^{-\alpha}$ is the total interference power from other BSs, and $\mathbb{E}_{h_i}$ denotes the expectation taken over the small-scale fading.

Then, the total transmission cost in time step t caused by both on-demand transmission and proactive pushing (if any) can be expressed as

$$m(t)[I_{\mathrm{o}}(t)+I_{\mathrm{p}}(t)] \quad (2)$$

where $I_{\mathrm{o}}(t)$ and $I_{\mathrm{p}}(t)$ are indicator functions. $I_{\mathrm{o}}(t)=1$ if the requested content is not in a user's cache and requires on-demand transmission in time step t, and $I_{\mathrm{o}}(t)=0$ otherwise. $I_{\mathrm{p}}(t)=1$ if pushing occurs in time step t, and $I_{\mathrm{p}}(t)=0$ otherwise.

The goal of the recommendation and pushing policy π is to maximize the average accumulated net profit of MNO during each session (i.e., an episode).[3] Hence, the optimal policy can be found from the following problem

$$\max_{\pi}\ \mathbb{E}\left[\sum_{t=1}^{T}\eta-\sum_{t=1}^{T}m(t)[I_{\mathrm{o}}(t)+I_{\mathrm{p}}(t)]\right] \quad (3a)$$

$$s.t.\ \sum_{f=1}^{F}c_f(t)\le C \quad (3b)$$

where η is another scaling factor representing the revenue of each request whose value is determined by how much the MNO charges a user. The expectation is taken over all random variables in each time step, including user locations, fading channels, which content a user requests, and whether a session terminates after a content is consumed (which determines the session length T). The first and second terms of (3a), respectively, reflect the accumulated revenue and transmission cost for the session. (3b) is the cache size constraint, which limits the number of cached contents at each user, where $c_f(t)=1$ if content f is cached in time step t and $c_f(t)=0$ otherwise.

The user behaviors, including the mobility patterns, acceptance probability matrix $\mathbf{P}$, the user requests distribution when rejects a recommendation, and the session end probabilities q_1 and q_2, as well as channel distributions are all unknown by the agent in advance. More importantly, such user behaviors are in general hard to model. This calls for a model-free approach. We resort to RL to solve the optimization problem.

B. RL FRAMEWORK FOR JOINT OPTIMIZATION

The joint recommendation and pushing problem in (3) can be formulated as the following RL problem.

State: The recommendation and pushing action in time step t should depend on a user's last consumed content with index denoted as $f_{\mathrm{q}}(t-1)$,[4] the cache status of the user denoted as $\mathbf{c}(t)=[c_1(t),\cdots,c_F(t)]$, as well as the costs to transmit a content in current and future time steps. The transmission cost in a time step depends on the channel condition of the user, which further depends on the user location. The agent is unaware of the future transmission costs. According to the Markov property of user mobility, we include the transmission costs during the last two time steps $m(t-1)$, $m(t-2)$ into the state.[5] It is noteworthy that the difference between the costs of successive time steps can reflect the change of channel conditions. For example, if $m(t-1)<m(t-2)$, then it implies that the user was moving towards the cell center in time step $(t-1)$. The whole state can be denoted by vector

$$\mathbf{s}(t)=[f_{\mathrm{q}}(t-1),\mathbf{c}(t),m(t-1),m(t-2)] \quad (4)$$

which has $F\times\sum_{i=0}^{C}\binom{F}{i}$ possible discrete combinations due to $f_{\mathrm{q}}(t-1)$ and $\mathbf{c}(t)$, and involves continuous variable due to $m(t-1)$, $m(t-2)$.

Action: Let $f_{\mathrm{r}}(t)$ and $f_{\mathrm{p}}(t)$ denote the indexes of the content to be recommended and the content to be pushed in time step t, respectively. Considering that pushing introduces transmission cost as shown in (2), it is better for a BS not to push anything to a user in some time steps, e.g., when the channel conditions are not good enough. We denote $f_{\mathrm{p}}(t)=0$ if no content is pushed in time step t. When the cache of the user is full and a content is needed to push to the user, the least-recently consumed content in the cache will be removed to ensure the cache size constraint (3b) because a consumed content is not likely to be requested by the user again.[6] Then, the action can be denoted by vector

$$\mathbf{a}(t)=[f_{\mathrm{r}}(t),f_{\mathrm{p}}(t)] \quad (5)$$

which has $F\times(F+1)$ possible combinations.

[3] We can also consider other goals, e.g, minimizing the transmission delay, by adjusting the reward.

[4] If the acceptance probability depends on multiple recently consumed contents, then $f_{\mathrm{q}}(t-2), f_{\mathrm{q}}(t-3),\ldots,$ should also be included in the state.

[5] To capture more complex user mobility pattern, $m(t-3), m(t-4),\cdots$ can also be included into the state.

[6] If all the cached contents have not been consumed yet, the least-recently cached content will be removed. Such a content is cached with the longest duration but still has not been requested, indicating that it is the content least likely to be requested. We can also include which content to be removed from the cache into the action so that the agent can learn it, which however will increase the already large action space.

Reward: Since our goal is to maximize the average accumulated net profit of MNO during a session, the reward function can be naturally designed as the net profit in each time step, i.e.,

$$r(t) = \eta - m(t)[I_{\mathrm{o}}(t) + I_{\mathrm{p}}(t)] \tag{6}$$

Define the action-value function (also called as Q-function)

$$Q^{\pi}(\mathbf{s}, \mathbf{a}) \triangleq \mathbb{E}\left[\sum_{i=t}^{T} \gamma^{i-t} r(i) \,\middle|\, \mathbf{s}(t) = \mathbf{s}, \mathbf{a}(t) = \mathbf{a}, \pi\right] \tag{7}$$

as the expected return (i.e., the average accumulated net profit for $\gamma = 1$) achieved by policy π when taking action $\mathbf{a}$ from state $\mathbf{s}$.

By defining the optimal action-value function as $Q^*(\mathbf{s}, \mathbf{a}) \triangleq \max_{\pi} Q^{\pi}(\mathbf{s}, \mathbf{a})$, the optimal policy can be easily obtained from $Q^*(\mathbf{s}, \mathbf{a})$ as

$$\pi^*(\mathbf{s}) \triangleq \arg\max_{\mathbf{a}} Q^*(\mathbf{s}, \mathbf{a}) \tag{8}$$

and hence the goal of the agent is to learn the optimal action value $Q^*(\mathbf{s}, \mathbf{a})$ in each state.

Q-learning is a commonly used algorithm to learn $Q^*(\mathbf{s}, \mathbf{a})$ from experience $\mathbf{e}(t) = [\mathbf{s}(t), \mathbf{a}(t), r(t), \mathbf{s}(t+1)]$ by the following iteration

$$\begin{aligned} Q(\mathbf{s}(t), \mathbf{a}(t)) &\leftarrow Q(\mathbf{s}(t), \mathbf{a}(t)) \\ &+\delta\left[r(t) + \gamma \max_{\mathbf{a}} Q(\mathbf{s}(t+1), \mathbf{a}) - Q(\mathbf{s}(t), \mathbf{a}(t))\right] \end{aligned} \tag{9}$$

which is with guaranteed convergence to $Q^*(\mathbf{s}, \mathbf{a})$ with sufficient experiences and properly chosen learning rate δ [32].

The state and action spaces of such a formulation of RL problem (called "*Non-decomposed*" framework in the sequel) however are too large to learn the action values in every single state. Even if we resort to DRL, e.g., deep Q-network $Q(\mathbf{s}, \mathbf{a}; \boldsymbol{\theta})$ with parameter $\boldsymbol{\theta}$ to approximate $Q^*(\mathbf{s}, \mathbf{a})$ [33], the problem is still very hard to solve due to the combinatorial actions and states involved.

C. DECOMPOSED RL FRAMEWORK

The large action and state spaces come from the fact that recommendation policy and pushing policy are coupled with each other. On one hand, recommendation affects which content will be requested by a user and hence affects which content should be pushed. On the other hand, pushing policy determines the cached content, based on which recommendation policy can be leveraged to stimulate the user to request a cached content. For example, the BS can recommend a cached content instead of the most attractive but non-cached content to a user [20].

Nonetheless, the gain from exploiting the impact of pushing on recommendation is much lower than the gain from exploiting the impact of recommendation on pushing. This can be explained as follows. Given the recommendation policy that can recommend a preferred content to a user, the probability that the requested content is cached is high if the BS can push the content to be recommended into the user's cache in advance. On the contrary, if the pushing policy is not designed based on the recommendation policy that recommends a preferred content to a user, the probability that a preferred content is cached is low. Since a user is not likely to accept the recommendation of a non-preferred content, recommendation does not help much in stimulating the user to request a cached content.

Moreover, recommendation can increase the revenue by boosting user requests [14], while pushing can help reduce system cost and improve user experience if the pushed content matches the taste of the user, which can only increase the net profit remarkably if the revenue has already been boosted.

This implies that the pushing policy should be designed based on the recommendation policy, instead of the other way around. Furthermore, thanks to the ability of recommendation in reducing the uncertainty in user requests as previously explained, it is possible to precisely predict the future requested content based on a well-learned recommendation policy. Recall that unlike pushing, recommendation itself causes negligible transmission cost during the interactions with users. This suggests that training the recommendation policy first and the pushing policy afterwards can help the learned pushing policy to achieve low cost without compromising user experience.

Therefore, we can decompose the problem into two RL problems by distinguishing the roles of recommendation and pushing policies explicitly, in order to reduce the state and action spaces for viable learning. We consider two agents, where the first agent learns the recommendation policy, and the second agent learns the pushing policy based on the learned recommendation policy.

1) RECOMMENDATION PROBLEM

The goal of personalized recommendation is to increase user stickiness by helping a user find the preferred contents. If the recommended contents match the taste of a user sufficiently well, the user will accept the recommendation and will request more contents, say during a session. Hence, the recommendation policy can be designed to maximize the average accumulated revenue $\mathbb{E}[\sum_{t=1}^{T} \eta]$.

The action and reward of the recommendation agent are denoted by $a_1(t) = f_{\mathrm{r}}(t)$ and $r_1(t) = \eta$, respectively. Since which content to recommend should depend on the user's last consumed content, the state in recommendation problem is represented by $s_1(t) = f_{\mathrm{q}}(t-1)$. Then, the index of the content recommended in time step t is

$$f_{\mathrm{r}}(t) = \pi_{\mathrm{r}}(f_{\mathrm{q}}(t-1)) \tag{10}$$

where $\pi_{\mathrm{r}}(\cdot)$ denotes the recommendation policy.

Both the state and action respectively have F possible values, which are much smaller than the direct formulation because the cache status $\mathbf{c}(t)$ and transmission cost $m(t-1)$, $m(t-2)$ are excluded from the state and the pushing action $f_{\mathrm{p}}(t)$ is excluded from the action of the recommendation agent.

2) PUSHING PROBLEM

The goal of the pushing policy is to reduce the cost of the network and improve the experience of a user by pre-downloading contents that the user is very likely to request in future time steps. If a pushed content is requested, then the user can enjoy zero initial delay by retrieving the content directly from its own cache. If the content was pushed under better channel condition than the channel when user initiates the request, then the transmission cost of the network can be reduced. Hence, the pushing policy can be designed to minimize the average accumulated cost $\sum_{t=1}^{T} m(t)[I_o(t) + I_p(t)]$ for a well-learned recommendation policy.

Albeit the intention of pushing policy is to push the contents that the user will request with high probability in future time steps, when establishing the RL framework for joint recommendation and pushing, the agent does not explicitly predict the contents to be requested. Instead, it directly learns the policy by maximizing the expected return (i.e., the accumulated net profit). Due to the large state and action spaces, huge number of experience vectors (i.e., training samples) are required for trial-and-error. When we decompose the original RL problem into two RL problems and first learns the recommendation policy, the pushing agent can explicitly predict the contents to be requested in multiple time steps ahead with high precision thanks to the ability of recommendation in reducing the uncertainty in user requests. Then, the state and action spaces of the pushing agent can be further reduced dramatically, as detailed below.

When the recommendation policy is learned good enough that it can recommend preferred contents to a user, the user will accept the recommendation with high probability. Therefore, it is of high precision to predict that the content to be requested/consumed in time step t (with index denoted as $\hat{f}_q(t)$) is the recommended content, i.e., $\hat{f}_q(t) = f_r(t)$. With a well-learned recommendation policy, the content to be recommended in the next time step can also be predicted by considering (10) as

$$\hat{f}_r(t+1) = \pi_r(\hat{f}_q(t)) = \pi_r(f_r(t)) = \pi_r(\pi_r(f_q(t-1))) \quad (11)$$

In a similar fashion, we can predict that the content to be recommended (and hence to be requested by the user) in the next n-step of time step t as

$$\hat{f}_r(t+n) = \pi_r(\cdots \pi_r(f_q(t-1))) \triangleq \pi_r^{n+1}(f_q(t-1)) \quad (12)$$

Since a precise prediction of the next n-step recommendation relies on the fact that the user accepts the recommended contents in the 1st $\sim (n-1)$th steps when the session does not terminate within next n steps, the prediction becomes less precise with the increase of n. Consequently, pushing the content to be recommended in too many time steps ahead may bring no benefit except transmission cost. Further considering the limited cache size of the user device, the pushing agent should only push the contents to be recommended (which are thereby the contents likely to be requested by the user) within next N ($N \leq C$) steps of time step t (called *pushing candidate set*), i.e.,

$$f_p(t) \in \{f | f = \hat{f}_r(t+n) \text{ for } n = 1, \cdots, N\} \quad (13)$$

This reduces the action space of the pushing agent. In particular, the number of possible actions can be reduced from $F+1$ to $N+1$ (including no pushing) for each state, by introducing the pushing candidate set based on the well-learned recommendation policy π_r.

However, if we simply use the index of the content to be pushed (i.e., $f_p(t)$) to represent the action, the possible actions are distinct for each possible value of $f_q(t-1)$ (which is a part of the state vector of the pushing agent as will be introduced later). For example, the last consumed content is the file with index 1, i.e., $f_q(t-1) = 1$, and the contents to be recommended in one and two time steps ahead are $\hat{f}_r(t+1) = 2$ and $\hat{f}_r(t+2) = 3$, respectively. Then, the possible action set for $f_q(t-1) = 1$ with $N = 2$ is $f_p(t) \in \{0, 2, 3\}$, where $f_p(t) = 0$ stands for no pushing. However, when $f_q(t-1) = 2$, we may have $\hat{f}_r(t+1) = 3$ and $\hat{f}_r(t+2) = 4$, and then the possible action set for $f_q(t-1) = 2$ with $N = 2$ is $f_p(t) \in \{0, 3, 4\}$. This makes it difficult for the agent to generalize the action values from experienced states to the unexperienced states with function approximation, say by deep neural networks.

Fortunately, we can find a mapping from any given value of $f_q(t-1)$ to the index of the content to be pushed in time step t by substituting (12) to (13) as,

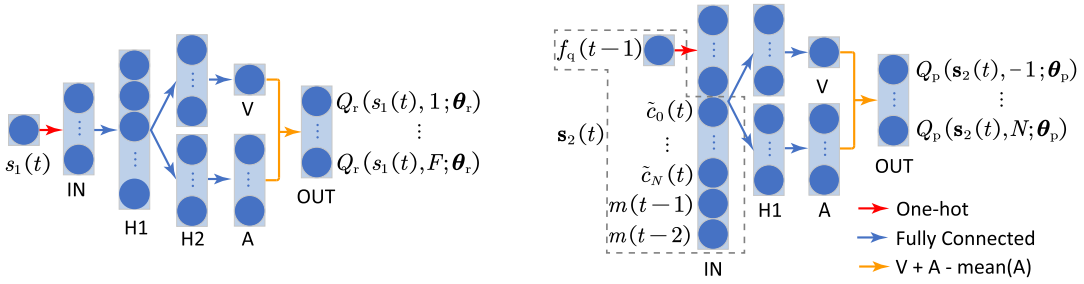
$$f_p(t) = \pi_r^{n+1}(f_q(t-1)) \quad (14)$$

With this mapping, the agent only needs to decide how many time steps ahead a content to be recommended should be pushed in the current time step t (i.e., decide n), instead of directly deciding the specific index of the content to be pushed (i.e., $f_p(t)$). Hence, the action can be represented by $a_2(t) = n$. When $n = 1, \cdots, N$, the content to be recommended in the next n-step (i.e., $\hat{f}_r(t+n)$) is pushed in time step t. When $n = -1$, no content is pushed. In this way, the possible action set, i.e., $\{-1, 1, 2, \cdots, N\}$, is the same for each possible value of $f_q(t-1)$, which makes the agent able to learn the approximated optimal Q-function with less experiences. Then, the index of the content to be pushed at time step t can be expressed as

$$f_p(t) = \pi_r^{a_2(t)+1}(f_q(t-1)) \quad (15)$$

We define $\pi_r^0(\cdot) \triangleq 0$, i.e., no content is pushed.

Since the agent only considers pushing from the pushing candidate set that is determined by $\hat{f}_r(t+n)$, it is the cache status of the contents to be recommended (rather than the cache status of all the recommendation candidates) that affects the reward and hence the pushing policy. We let $\tilde{c}_n(t) = 1$ denote that content $\hat{f}_r(t+n)$ is cached and $\tilde{c}_n(t) = 0$ otherwise. Then, the cache status can be represented as $\tilde{\mathbf{c}}(t) \triangleq [\tilde{c}_0(t), \tilde{c}_1(t), \cdots, \tilde{c}_N(t)]$ (instead of $\mathbf{c}(t)$ defined in section III-B) for the pushing agent. Now, the number of possible cache status is reduced from $\sum_{i=0}^{C} \binom{F}{i}$ to 2^N. In practice, 2^N is not a large number due to the limited cache size at user device and $N \leq C \ll F$.



(a) Recommendation agent: $Q_r(s_1(t), a_1(t); \boldsymbol{\theta}_r)$

(b) Pushing agent: $Q_p(\mathbf{s}_2(t), a_2(t); \boldsymbol{\theta}_p)$

FIGURE 3. Q-networks with dueling architecture.

Further including $m(t-1)$ and $m(t-2)$, we can represent the state vector of the pushing agent as

$$\mathbf{s}_2(t) = [f_q(t-1), \tilde{\mathbf{c}}(t), m(t-1), m(t-2)] \tag{16}$$

The reward of the pushing agent is

$$r_2(t) = -m(t)[I_o(t) + I_p(t)] \tag{17}$$

By decoupling recommendation from pushing, limiting the number of possible actions, and representing the state and action of the pushing agent in a proper manner, the two problems can be efficiently solved by DRL. This RL framework can be implemented with value-based RL algorithms such as deep Q-network (DQN), or with policy-based (or more specifically, actor-critic) RL algorithms. However, among state-of-the-art policy-based RL algorithms, DDPG [25] is more appropriate in continuous action space, while A2C [26] and PPO [27] are on-policy algorithms that are less sample-efficient. Moreover, A2C and PPO may converge to stochastic policies, while a deterministic recommendation policy is more appealing for the problem at hand. This is because a deterministic recommendation policy allows the agent to accurately predict the contents to be recommended in future time steps, based on which the pushing candidates set can be determined. In this work, we consider dueling DDQN. Specifically, we use two Q-networks to approximate the optimal action-value functions for recommendation and pushing agents, respectively.

D. DRL WITH DUELING DDQN

We build the Q-networks based on an improved version of DQN (namely dueling DDQN [24]), which modifies the original DQN [33] with dueling network architecture and introduces double Q-learning update [34] for more efficient and stable learning.

In particular, the Q-networks of the two agents are shown in Fig. 3, which are stored at CPs. The parameters $\boldsymbol{\theta}_r$, $\boldsymbol{\theta}_p$ are learned by the trainer at the server as shown in Fig. 1 and then sent to CPs to update the Q-networks, based on which the CP selects actions for the BS to execute under each state.

For the recommendation agent, the state $s_1(t) = f_q(t-1)$ is the index of last consumed content. The state is first converted into a F-dimensional one-hot vector for the input layer and then goes through two fully-connected hidden layers. The second hidden layer H2 is split into two streams, V-stream $V_r(s_1(t); \boldsymbol{\theta}_r)$ and A-stream $A_r(s_1(t), a_1(t); \boldsymbol{\theta}_r)$, forming the dueling architecture [24], which learns the action value more efficiently by separately estimating the state value and advantages for each action. The output layer implements the following mapping

$$Q_r(s_1(t), a_1(t); \boldsymbol{\theta}_r) = V_r(s_1(t); \boldsymbol{\theta}_r) + A_r(s_1(t), a_1(t); \boldsymbol{\theta}_r) - \frac{1}{F}\sum_{a=1}^{F} A_r(s_1(t), a; \boldsymbol{\theta}_r) \tag{18}$$

For the pushing agent, a part of state $\mathbf{s}_2(t)$, $f_q(t-1)$, is first converted into a F-dimensional one-hot vector, and then concatenated with other parts of $\mathbf{s}_2(t)$ as the input layer. After going through a fully connected hidden layer and the dueling architecture, the output layer (similar to (18)) yields the action-value function $Q_p(\mathbf{s}_2(t), a_2(t); \boldsymbol{\theta}_p)$.

Since the training procedures for the two networks are the same, we omit all the subscripts in the following for notational simplicity. The parameter $\boldsymbol{\theta}$ of the Q-network is learned by minimizing the loss function

$$L(\boldsymbol{\theta}) = \mathbb{E}\left[\left(y(t) - Q(\mathbf{s}(t), \mathbf{a}(t); \boldsymbol{\theta})\right)^2\right] \tag{19}$$

with

$$y(t) = r(t) + \hat{Q}(\mathbf{s}(t+1), \arg\max_{\mathbf{a}} Q(\mathbf{s}(t+1), \mathbf{a}; \boldsymbol{\theta}); \hat{\boldsymbol{\theta}}) \tag{20}$$

where $\hat{Q}(\mathbf{s}, \mathbf{a}; \hat{\boldsymbol{\theta}})$ is the target network. It is the same as the Q-network and updated by $\hat{\boldsymbol{\theta}} \leftarrow \tau\boldsymbol{\theta} + (1-\tau)\hat{\boldsymbol{\theta}}$ with very small value of τ to reduce the correlations between the action-values $Q(\mathbf{s}, \mathbf{a}; \boldsymbol{\theta})$ and the target values $y(t)$, which improves the convergence of learning [33].

In (20), we employ the double Q-learning update where the optimal action is chosen by the Q-network, i.e.,

$$\mathbf{a}_{\max} \triangleq \arg\max_{\mathbf{a}} Q(\mathbf{s}(t+1), \mathbf{a}; \boldsymbol{\theta}) \tag{21}$$

The optimal action value is evaluated by the target network, i.e., $\hat{Q}(\mathbf{s}(t+1), \mathbf{a}_{\max}; \hat{\boldsymbol{\theta}})$, which can reduce the overestimation of action value compared with traditional Q-learning [34].

During the interaction with the environment, every CP collects the experience $\mathbf{e}(t) = [\mathbf{s}(t), \mathbf{a}(t), r(t), \mathbf{s}(t+1)]$ from

every user in its coverage into the data base at the server as $\mathcal{D} = \{\mathbf{e}(1), \cdots, \mathbf{e}(t-1)\}$. During the training of Q-network, the trainer iteratively samples a mini-batch of the experiences from $\mathcal{D}$ uniformly, and updates parameter $\boldsymbol{\theta}$ with gradient descent as

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \frac{\delta}{|\mathcal{B}|}\nabla_{\boldsymbol{\theta}} \sum_{j\in\mathcal{B}} (y(j) - Q(\mathbf{s}(j), \mathbf{a}(j); \boldsymbol{\theta}))^2 \qquad (22)$$

in each iteration, where $\mathcal{B}$ denotes the set of indexes of sampled experiences. By such an *experience replay*, data efficiency can be improved through re-sampling the experiences stored in the data base as training samples, and the correlation among training samples can be reduced, which also improves the convergence of learning [33].

As previously explained, effective pushing relies on the precise prediction of the contents to be requested in future time steps, which further depends on the acceptance probability of recommendation. To avoid high transmission cost during interactions with the environment, recommendation policy should be first trained, while the pushing agent does not push any content until enough experiences are gathered (say after L_1 episodes) for the recommendation agent to learn to recommend attractive contents to a user. In addition, we let the pushing agent learn the pushing policy in an emulated environment while not really pushing any content, until the pushing policy is learned well enough (say after L_2 episodes). Otherwise, the pushing agent may not be able to push the right content at the right time step, which may make pushing even inferior to not pushing.

To be specific, during the period between the (L_1+1)th episode and the L_2th episode, the recommendation agent continues to take actions in real environment, while the pushing agent observes the user requests and transmission costs from the real environment but only takes virtual actions. In the emulated environment, the recommendation agent operates the same as in the real environment, and the user's reaction to the recommendation is also the same as in the real environment. Hence, the reward obtained by the recommendation agent is the reward obtained from the real environment, i.e., $r_1(t)$. Besides, in the emulated environment the observations related to user mobility is the same as those observed in the real environment (because pushing does not affect user mobility), and hence the cost for transmitting a content at time step t is also the same as the cost observed in the real environment, i.e., $m(t)$. Yet the pushing agent operates (i.e., pushes contents) in the emulated environment to learn the pushing policy. Then, based on (17), the reward obtained by the pushing agent in the emulated environment can be computed by

$$r_2'(t) = -m(t)[I_{\mathrm{o}}'(t) + I_{\mathrm{p}}'(t)] \qquad (23)$$

where $I_{\mathrm{o}}'(t) = 1$ if the requested content is not cached by the user in the emulated environment, and $I_{\mathrm{o}}'(t) = 0$ otherwise, $I_{\mathrm{p}}'(t) = 1$ if pushing occurs in the emulated environment, and $I_{\mathrm{p}}'(t) = 0$ otherwise. Based on what content is pushed in the emulated environment, the cache status in the next time step (denoted by $\mathbf{c}'(t+1)$) can be obtained. Because pushing only affects the cache status in the state vector defined in (4), the next state of the emulated environment can be obtained as $\mathbf{s}'(t+1) = [f_{\mathrm{q}}(t), \mathbf{c}'(t+1), m(t), m(t-1)]$, where the only difference with the next state in the real environment is the cache status. The experiences obtained in the emulated environment are also put into the data base as training samples.

In each time step of the first L_1 episodes, the recommendation agent either randomly selects a content to recommend with probability ε_{r}, or selects the optimal action $a_1(t) = \arg\max_a Q_{\mathrm{r}}(s_1(t), a; \boldsymbol{\theta}_{\mathrm{r}})$ based on the stored Q-network at the CP with probability $1-\varepsilon_{\mathrm{r}}$. Such ε-greedy method is able to balance the tradeoff between finding a better action (i.e., exploration) and maximizing the return based on currently estimated Q-network (i.e., exploitation).

In each time step of the (L_1+1)th to the L_2th episodes, the recommendation agent continues to act ε-greedily and the pushing agent starts to act ε-greedily in the emulated environment to learn the pushing policy.

After the L_2th episode, both agents act ε-greedily in real environment to learn the recommendation policy and pushing policy simultaneously. The overall algorithm for the learning procedure is given in Algorithm 1.

Different from existing algorithms, we consider two agents working together with different goals, where the policy of the pushing agent depends on the policy of the recommendation agent. We also consider an emulated environment for the pushing agent to learn its policy without any interaction with real environment, which improves the on-line performance.

IV. SIMULATION RESULTS

In this section, we compare the performance of the learned policy by DRL with baseline policies via simulation. Since the way we representing the action and state of the pushing agent is not easy to follow, we illustrate how the learned pushing policy behaves, again with simulation. We also compare the adopted DRL algorithm with several state-of-the-art policy-based DRL algorithms.

A. SIMULATION ENVIRONMENT SETUP

We consider a scenario where users are moving along a road across multiple cells, as shown in Fig. 4. The distance between adjacent BSs is 500 m and the transmit power of each BS is 46 dBm. The pathloss is modeled as $35.3 + 37.6\log_{10}(d_i)$ in dB, and the small-scale channel is Rayleigh fading. The duration of each time step is 10 s, which is for short videos (e.g., Douyin app [15]). To simulate a general case without specific mobility pattern, in each time step each user is assumed to move forward with speed uniformly selected between (5 ± 0.5) m/s with probability 0.8 or to stop (due to traffic light or traffic jam) with probability 0.2.

Since the net profit actually depends on the ratio between η and β, we can normalize the revenue $\eta = 1$ without loss of optimality. Considering that the transmission cost may exceed the revenue for a user located in cell-edge area, the scaling factor for transmission cost is set as $\beta = 3$. By this

Algorithm 1 Content Recommendation and Pushing Learning

Initialize replay memory, i.e., the data base $\mathcal{D}$.
Initialize Q-network $Q_{\rm r}, Q_{\rm p}$ with random weights $\boldsymbol{\theta}_{\rm r}, \boldsymbol{\theta}_{\rm p}$.
Initialize target network $\hat{Q}_{\rm r}, \hat{Q}_{\rm p}$ with weights $\hat{\boldsymbol{\theta}}_{\rm r} = \boldsymbol{\theta}_{\rm r}$, $\hat{\boldsymbol{\theta}}_{\rm p} = \boldsymbol{\theta}_{\rm p}$.
for episode $= 1, 2, \cdots$ **do**
  for time step $t = 1, \cdots, T$ **do**
    With probability $\varepsilon_{\rm r}$ select a random recommendation $a_1(t)$, otherwise $a_1(t) = \pi_{\rm r}(s_1(t)) \triangleq \arg\max_a Q_{\rm r}(s_1(t), a; \boldsymbol{\theta}_{\rm r})$.
    if episode $\leq L_1$ **then**
      Do not push any content, $a_2(t) = -1$.
    else
      Update the predicted next 1- to N-step recommendations by
      $\hat{f}_{\rm r}(t+1) = \pi_{\rm r}^2(f_{\rm q}(t-1)), \cdots$
      $\hat{f}_{\rm r}(t+N) = \pi_{\rm r}^{N+1}(f_{\rm q}(t-1))$.
      With probability $\varepsilon_{\rm p}$ randomly select $a_2(t) \in \{-1, 1, \cdots, N\}$, otherwise select $a_2(t) = \arg\max_n Q_{\rm p}(\mathbf{s}_2(t), n; \boldsymbol{\theta}_{\rm r})$.
    end if
    if $L_1 <$ episode $\leq L_2$ **then**
      if $t = 1$ **then**
        Initialize the emulated environment, $\mathbf{s}'(t) = \mathbf{s}(t)$.
      end if
      Execute action $\mathbf{a}(t) = [a_1(t), 0]$, observe reward $r_1(t)$, and record transmission cost $m(t)$.
      Execute action $\mathbf{a}'(t) = [a_1(t), \pi_{\rm r}^{a_2(t)+1}(f_{\rm q}(t-1))]$ in the emulated environment, obtain the reward for pushing agent by substituting the recorded $m(t)$ into (23), and observe next state $\mathbf{s}'(t+1)$.
      Store experience $[\mathbf{s}'(t), \mathbf{a}'(t), r_1(t), r_2'(t), \mathbf{s}'(t+1)]$ in $\mathcal{D}$.
    else
      Execute action $\mathbf{a}(t) = [a_1(t), \pi_{\rm r}^{a_2(t)+1}(f_{\rm q}(t-1))]$, observe reward $r_1(t)$, $r_2(t)$ and next state $\mathbf{s}(t+1)$
      Store experience $[\mathbf{s}(t), \mathbf{a}(t), r_1(t), r_2(t), \mathbf{s}(t+1)]$ in $\mathcal{D}$.
    end if
    Randomly sample a mini-batch of experiences from $\mathcal{D}$ as
    $\mathcal{B} = \{[\mathbf{s}(j), \mathbf{a}(j), r_1(j), r_2(j), \mathbf{s}(j+1)]\}$
    Get $s_1(t)$ from $\mathbf{s}(t)$
    Get $a_{\max} = \arg\max_a Q_{\rm r}(s_1(j+1), a; \boldsymbol{\theta}_{\rm r})$ and set
    $$y_1(j) = \begin{cases} r_1(j), & \text{if episode terminates at step } j+1 \\ r_1(j) + \gamma \hat{Q}_{\rm r}(s_1(j+1), a_{\max}; \hat{\boldsymbol{\theta}}_{\rm r}), & \text{otherwise} \end{cases}$$
    Perform a gradient descent step minimizing $(y_1(j) - Q_{\rm r}(s_1(t), a_1(t); \boldsymbol{\theta}_{\rm r}))^2$ with respect to $\boldsymbol{\theta}_{\rm r}$
    if episode $> L_1$ **then**
      Get state representation $\mathbf{s}_2(j)$ from $\mathbf{s}(j)$ based on $\pi_{\rm r}$
      Get $n_{\max} = \arg\max_n Q_{\rm p}(\mathbf{s}_2(j+1), n; \boldsymbol{\theta}_{\rm p})$ and set
      $$y_2(j) = \begin{cases} r_2(j), & \text{if episode terminates at step } j+1 \\ r_2(j) + \gamma \hat{Q}_{\rm p}(\mathbf{s}_2(j+1), n_{\max}; \hat{\boldsymbol{\theta}}_{\rm p}), & \text{otherwise} \end{cases}$$
      Perform a gradient descent step minimizing $(y_2(j) - Q_{\rm p}(\mathbf{s}_2(t), a_2(t); \boldsymbol{\theta}_{\rm p}))^2$ with respect to $\boldsymbol{\theta}_{\rm p}$
    end if
    Update the target networks:
    $\hat{\boldsymbol{\theta}}_{\rm r} \leftarrow \tau \boldsymbol{\theta}_{\rm r} + (1-\tau)\hat{\boldsymbol{\theta}}_{\rm r}$, $\hat{\boldsymbol{\theta}}_{\rm p} \leftarrow \tau \boldsymbol{\theta}_{\rm p} + (1-\tau)\hat{\boldsymbol{\theta}}_{\rm p}$
  end for
end for

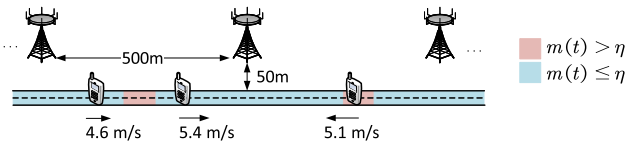


FIGURE 4. Simulation scenario.

setting, the cost will be higher than the revenue if the distance between user and BS is larger than 220 m.

The recommendation candidate set consists of $F = 100$ contents. The cache size of each mobile device is $C = 5$. The size of the pushing candidate set is $N = 5$. The request probability for content f when rejecting the recommendation (i.e., the probability of requests from a user without recommendation) is modeled by Zipf distribution $p_f = f^{-\zeta} / \sum_{j=1}^{N_f} j^{-\zeta}$ with the skewness parameter $\zeta = 0.6$ [35].

To evaluate the performance of a recommendation policy, one needs to know how a user reacts to a content when the content is recommended to the user, i.e., accepts the recommended content or not. Existing datasets, e.g., MovieLens dataset [36] and Netflix dataset [14], only provides the rating or request records, which cannot reflect whether or not a user accepts a recommendation. Therefore, the performance of a recommendation policy is not able to be evaluated with existing datasets. To evaluate the performance of a recommendation policy, one needs to conduct real experiments, e.g., A/B test [37], which however is rather labor intensive and time consuming. In this paper, we provide a simulated environment based on well-acknowledged intuitions and facts revealed from real data sets analysis.

We set the acceptance probability matrix $\mathbf{P} = [p_{ij}]_{F \times F}$ for each user as the following form,

$$\mathbf{P} = \begin{bmatrix} ☹ & ☻ & \cdots & ☻ & ☹ & ☹ & \cdots & ☹ \\ ☹ & ☹ & ☻ & \cdots & ☻ & ☹ & \cdots & ☹ \\ & & & & \cdots & & & \\ ☻ & \cdots & ☻ & ☹ & ☹ & ☹ & \cdots & ☹ \end{bmatrix} \quad (24)$$

where "☹" represents a value uniformly chosen from [0, 0.1], and "☻" represents a value uniformly chosen from [0.9, 1]. For $p_{ij} \in [0, 0.1]$, the users who just have consumed content i are likely to reject the recommendation of content j. For $p_{ij} \in [0.9, 1]$, the users who just have consumed content i are likely to accept the recommendation of content j. Compared with user demands without recommendation (e.g., Zipf distribution with $\zeta = 0.6$, where the content most likely to be requested is requested with probability $1^{-0.6} / \sum_{j=1}^{100} j^{-0.6} = 0.07$), the uncertainty in user request can be reduced significantly if recommending a content with acceptance probability greater than 0.9 (i.e., the content with "☻"). The users share the same structure of $\mathbf{P}$, but the specific values of p_{ij} differ among users.

We set 5% elements in each row of $\mathbf{P}$ as values randomly selected from [0.9, 1] and the rest of 95% elements from [0, 0.1]. This means that the contents likely to be accepted

by the users who have just consumed a content are 5% of the whole recommendation candidate set. These 5% of the contents can be regarded as similar or related contents, e.g., the contents with the same genre. To show a clear structure of $\mathbf{P}$, we have re-ordered the indexes of contents so that similar contents lie together with each other.

We set the diagonal elements of $\mathbf{P}$ as values randomly selected from $[0, 0.1]$, because a user is less likely to accept the recommendation of a content that has just been consumed.

Considering that the requests of a user are also not likely to bounce between two contents, e.g., a user is not likely to request contents in sequence such as $1 \rightarrow 2 \rightarrow 1 \rightarrow 2$ during a session, p_{ij} and p_{ji} should not both lie in $[0.9, 1]$. One possible way to reflect such a fact is to set the ith row as the circulating shift of the $(i-1)$th row. Such a circulating-shift structure can reflect the fact that a content may have multiple genres. Take the first two rows of $\mathbf{P}$ for example, as shown in Fig. 5, contents $2 \sim 6$ have the same genre (say genre 1, *Romance*), and contents $3 \sim 7$ have the same genre (say genre 2, *Comedy*). Then, the contents $3 \sim 6$ belong to both genres 1 and 2.

FIGURE 5. Overlap in genres.

Despite that the structure of $\mathbf{P}$ given by (24) may not capture more complex user demand with recommendation, it agrees with intuition and can reflect some basic facts (e.g., non-repeated requests, no bouncing requests between two contents, and contents with multiple genres) revealed by real datasets [10], [36]. An advantage of using simulated environment is that we can flexibly control users' reaction towards recommendation by adjusting the parameters of matrix $\mathbf{P}$, which allows us to analyze the impact of user behavior on recommendation and pushing. Moreover, with such circulating-shift structure of $\mathbf{P}$, it is not hard to see that the optimal recommendation policy maximizing the average accumulated revenue $\mathbb{E}[\sum_{t=1}^{T} \eta]$ is to recommend a content with "☻" based on the last consumed content. For example, when the last consumed content of a user is content 1, we look up the 1st row of $\mathbf{P}$ given in (24) and find that contents $2 \sim 6$ is with "☻". Then, any one of contents $2 \sim 6$ can be recommended. Such a policy can be used as a baseline to show whether the recommendation policy learned by DRL can converge to the optimal policy. It is worthy to note that although we consider a certain structure of $\mathbf{P}$ in simulation, the proposed RL framework does not use any priori knowledge on such structure.

To capture the impact of recommendation on user stickiness, the session ending probabilities for each user varies within $q_1 = 0.05 \pm 0.005$ if the user accepts the recommendation and $q_2 = 0.5 \pm 0.05$ otherwise.

The simulations are conducted on a work station with Intel Core i7-8700K CPU and single Nvidia Geforce GTX 1080Ti GPU. The proposed RL framework is implement by TensorFlow 1.8.0 [38] with Python 3.6 on Windows 10.

B. FINE-TUNED HYPER-PARAMETERS OF ALGORITHM 1

The hidden layers H1 and H2 consist of 800 and 600 nodes, respectively, and use rectified linear unit (ReLU) as the activation function. Both V and A layers of the two Q-networks have half of the nodes as H2 for Q_{r} and the other half of the nodes as H1 for Q_{p}, respectively, and have no activation function. The output layers of Q_{r} and Q_{f} have F and $N+1$ nodes, respectively, where each node returns an action value of the input state.

The replay memory size is $|\mathcal{D}| = 10^5$. The discount factor is set as $\gamma = 1$ because we aim to maximize the average accumulated net profit during every session.

The whole process of simulation contains two consecutive phases, namely training phase and testing phase.

1) TRAINING PHASE

The training phase of the recommendation agent starts from the first episode. The exploration probability is $\varepsilon_{\mathrm{r}} = 1$ for the first 5×10^3 episodes and then decreases linearly to 0.01 within 3.5×10^4 episodes. The training phase of the pushing agent starts from the $L_1 = 4 \times 10^4$th episode, during which the pushing agent first interacts with the emulated environment within the 6.5×10^3 episodes, and then starts to interact with real environment from the $L_2 = 4.65 \times 10^4$th episode. The exploration probability is $\varepsilon_{\mathrm{p}} = 1$ at the 4×10^4th episode and then decreases linearly to 0.01 within 10^4 episodes. Adam [39] is used to adjust the learning rate during training, and the initial learning rate is $\delta = 10^{-4}$. The update rate for the target network is $\tau = 0.0025$. The mini-batch size for gradient descent is $|\mathcal{B}| = 32$.

2) TESTING PHASE

The testing phase starts from the 5×10^4th episode, during which the exploration probability is set as zero for both recommendation and pushing agents. In the testing phase, the neural network weights $\boldsymbol{\theta}_{\mathrm{r}}$, $\hat{\boldsymbol{\theta}}_{\mathrm{r}}$, $\boldsymbol{\theta}_{\mathrm{p}}$, $\hat{\boldsymbol{\theta}}_{\mathrm{p}}$ are frozen and no longer updated. The performance evaluation are based on the results obtained during testing phase.

C. COMPARING WITH POLICY-BASED DRL ALGORITHMS

To justify why we employ dueling DDQN for the decomposed RL framework, we compare the adopted value-based algorithm with several policy-based DRL algorithms, which are PPO, A2C, and the DDPG with the k-nearest neighbor (KNN) search [40] (called "DDPG + KNN" for short). We take the recommendation problem as an example for illustration. The details of these policy-based DRL algorithms

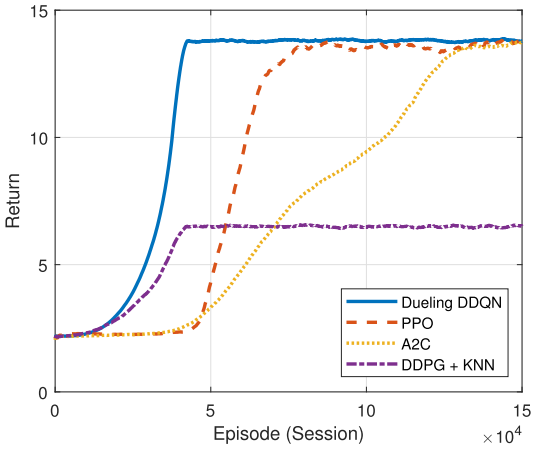


FIGURE 6. Comparison of dueling DDQN with several policy-based DRL algorithms in recommendation problem.

are given in the appendix. The returns achieved by different DRL algorithms are provided in Fig. 6.

We can see from the figure that dueling DDQN converges faster than all the policy-based algorithms. "DDPG + KNN" converges faster than PPO and A2C, but can only achieve 50% of the return as dueling DDQN. Both PPO and A2C can achieve a return close to dueling DDQN, where PPO converges faster than A2C. Since both PPO and A2C are on-policy algorithms, they are less sample-efficient than off-policy algorithm with experience replay such as DQN and hence converge slower. Moreover, our simulation results show that A2C and PPO may converge to stochastic policy if we do not decrease the entropy coefficient (i.e., β_2 defined in the appendix) during the training phase, while a deterministic recommendation policy (as learned from dueling DDQN) is more appealing for the pushing problem as we discussed in Section III-C-2).

In the sequel, we employ dueling DDQN as the DRL algorithm.

D. COMPARING WITH BASELINE POLICIES

There are no existing works jointly considering recommendation and pushing with unknown users behavior. Existing policies in [20]–[23] are not applicable to our considered scenario due to the following reasons: (1) We consider pushing to user devices while [20]–[23] consider caching at the BSs. (2) We consider mobile users and hence exploit mobility pattern, but [20]–[23] do not. We consider the fact that a user will not request a content again if the user has requested the content in a short period (say minutes), but [20]–[23] assumes that a user may request a content repeatedly. (3) The optimization objectives of this work and those in [20]–[23] are very different. Therefore, we do not compare with policies in [20]–[23].

To show the gain respectively from optimizing recommendation and optimizing pushing by decomposing the joint optimization problem, we compare the learned policy by DRL with the following baseline policies:

1) *Optimal Rec. & No pushing*: In each time step, the associated BS of a user recommends a content according to the optimal recommendation policy that maximizes the average accumulated revenue $\mathbb{E}[\sum_{t=1}^{T} \eta]$ and does not push any content. The optimal recommendation policy is obtained with known structure of $\mathbf{P}$, which recommends a content with "☻" based on the last consumed content. This policy can be regarded as a performance upper-bound of the state-of-the-art recommendation policies without pushing.
2) *No Rec. & Optimal pushing*: The BS does not recommend any contents to each user. Then, the optimal pushing policy is to let each user cache the contents that are most likely to be requested by the user. This policy provides a performance upper-bound of the state-of-the-art pushing policies without recommendation.
3) *Non-decomposed*: This policy is learned by directly applying dueling DDQN to the RL framework for joint pushing and recommendation optimization in Section III-B. Since the action and state spaces are huge for the non-decomposed RL framework, we only consider a recommendation candidate set consisting of $F = 10$ contents (specifically, each row of $\mathbf{P}$ has ten elements, among which two elements are with "☻") for "*Non-decomposed*" to reduce the complexity. This simplified scenario makes it easier for the agent to learn which content should be recommended because the action and state spaces are smaller, and the percentage of preferred content increases (i.e., 20% for "*Non-decomposed*" compared with 5% for "decomposed"). It is noteworthy that such a scenario does not affect the optimal performance, because the cache size and the acceptance probability of a preferred content remain unchanged. For the training phase, the exploration probability is first set as one for the first 5×10^3 episodes and then decreases linearly to 0.1 within 4.5×10^4 episodes. The testing phase starts from the 6×10^4th episodes. Other fine-tuned hyper-parameters are the same as those for Algorithm 1 under the decomposed RL framework.

In Fig. 7, we show the learning curve of proposed RL framework. For the first 4×10^4 episodes, the recommendation policy is learned without pushing any contents to the user and finally achieves 500% of the return (i.e., accumulated net profit) over "*No Rec. & Optimal pushing*". From the 4×10^4th to 4.65×10^4th episodes, the pushing agent learns pushing policy in the emulated environment while does not push in real environment. Hence, the return is almost the same as "*Optimal Rec. & No pushing*". After that, the pushing agent interacts with real environment. The return finally achieves 800% of the return over "*No Rec. & Optimal pushing*". It is worthy to note that although the dueling DDQN [24] is not able to guarantee convergence for all scenarios, Algorithm 1 always converges for all the tests we have done. We also show the performance if pushing agent directly interacts with real environment at the

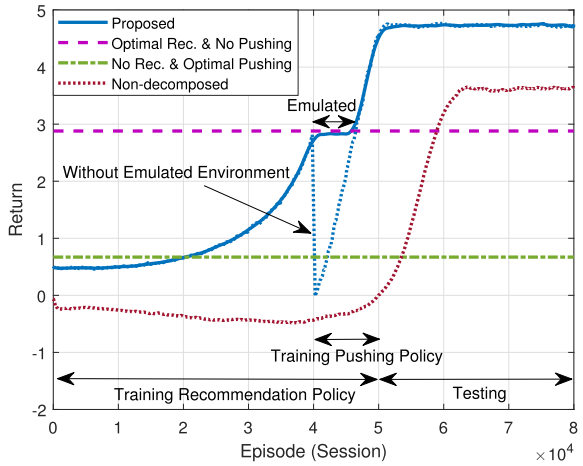


FIGURE 7. Learning curve. The result of each episode are obtained by averaging over 50 Monte Carlo trials and moving average over 50 successive episodes. In each trail, the matrix **P**, user mobility, fading channels, and the content request of a user when it rejects the recommendation are randomly generated.

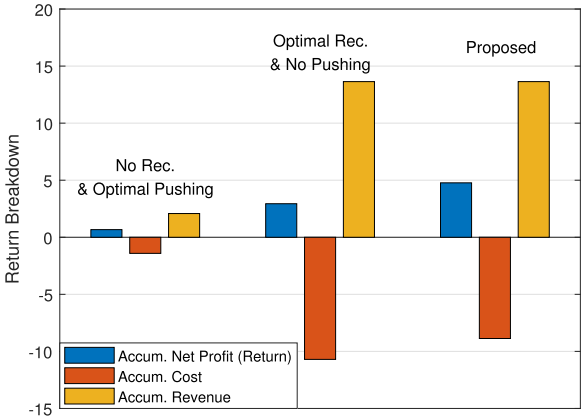


FIGURE 8. Return breakdown after Algorithm 1 converges.

4×10^4th episode in the figure. The return first drops because pushing policy has not been learned well enough, which increases the transmission cost compared with no pushing. We can see that "*Non-decomposed*" is inferior to the proposed decomposed framework even in the simplified scenario, which is due to the "delayed reward" as explained in the sequel. Simulation results show that "*Non-decomposed*" tends to not push anything to the user. Without pushing, "*Non-decomposed*" actually minimizes the immediate cost in the current time step (because pushing incurs transmission cost) and hence maximizes the immediate reward but does not necessarily maximize the accumulated reward in the long run. In other word, the benefit of pushing may only be observed several time steps later (e.g., when the user experiences bad channel condition and requests the pushed content.)

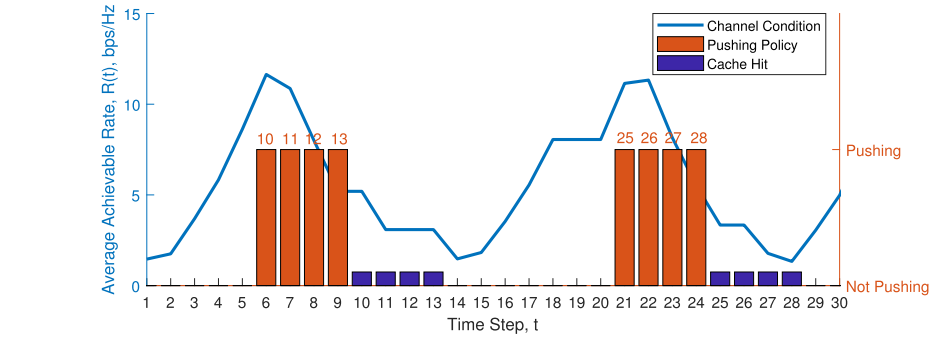
In Fig. 8, we compare the return breakdown to understand where the gain comes from. Compared with "*No Rec. & Optimal pushing*" the accumulated revenue achieved by "*Optimal Rec. & No pushing*" is boosted 600% because users will request more contents during a session. However, the accumulated transmission cost also increases significantly due to increased number of content requests. The proposed framework can achieve the same accumulated revenue as "*Optimal Rec. & No pushing*", which suggests that the learned recommendation policy is optimal. By learning the optimal pushing policy, the accumulated transmission cost of the proposed framework can be reduced and hence the accumulated net profit increases significantly compared with baselines.

E. BEHAVIOR OF THE PUSHING POLICY

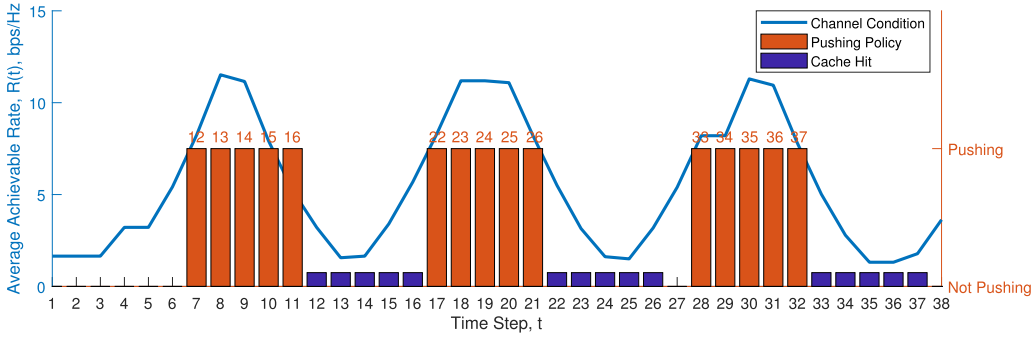
In Fig. 9, we show the average achievable rate (reflecting channel condition), which content is pushed to a user (if any), and whether a requested content is cached in each time step during a simulated session to illustrate how the learned pushing policy behaves. The result is obtained from one snapshot of a session after the pushing policy learned by Algorithm 1 converges.

From Fig. 9(a), we can see that the contents to be recommended in time steps $10 \sim 13$ are proactively pushed to the user in time steps $6 \sim 9$, respectively. The user accepts the recommendation in time steps $10 \sim 13$, leading to cache hits shown by the blue bars. Because the requested contents have already been cached, on-demand transmission is not required in time steps $10 \sim 13$ where the average achievable rate is low (i.e., transmission cost is high). These pushing actions indicate that the learned pushing policy can intelligently adapt to the mobility pattern of the users and the propagation environment of the network, so that the contents to be requested under bad channel condition are proactively pushed under good channel condition. Besides, we can see that only the content to be recommended in four time steps ahead (i.e., $n = 4$) is pushed to the user, although the size of pushing candidate set is $N = C = 5$. This is because with the increase of n, the prediction of the next n-step recommendation (and hence user request) becomes less precise as we explained in Section IV-B-2). From $\mathbf{P}$, we can compute the probability that the user accepts the predicted recommendation (which also indicates the probability of precisely predicting the content to be requested) in the next fourth time step can be as low as $(0.9 \times (1 - q_1))^4 \approx 0.52$.

In Fig. 9(b), we show the impact of prediction precision of the next-n step recommendation by adjusting the acceptance probability. Specifically, the elements with "☻" in $\mathbf{P}$ are set as one, which means that users will definitely accept the recommendation of a sufficiently attractive content. This is the optimistic case where the prediction of next-n step recommendation is perfect as long as the session does not terminate. We can see that, with more precise prediction of the next-n step recommendation, the content pushed to the user is the content to be recommended in five time steps ahead. Compared with Fig. 9(a), more contents are pushed under good channel conditions and hence the cache hit probability under bad channel conditions increases, which reduces the transmission cost. Moreover, since the acceptance probability



(a) ☻ ∈ [0.9, 1]



(b) ☻ = 1

FIGURE 9. Illustration of the learned pushing policy in a simulated session. The number above each orange bar denotes in which time step a content to be recommended is pushed to the user in the current time step. For example, in (a), the content pushed in time step 6 is the content to be recommended in time step 10.

is higher, a user is more likely to continue requesting contents, which results in longer session duration, e.g., $T = 38$ as shown Fig. 9(b). This increases the accumulated revenue during a session.

V. CONCLUSION AND FUTURE WORKS

In this paper, we considered joint content pushing and recommendation for mobile users to increase the net profit of MNO without comprising user experience. To cope with the challenge caused by unknown user behaviors, we resorted to RL framework. Considering that the joint pushing and recommendation problem has too large state and action spaces to be solved even with DRL algorithms, we decoupled the original problem into a recommendation problem and a pushing problem by differentiating the roles of recommendation and pushing. By first letting the recommendation agent learning the recommendation policy, the pushing agent is able to predict when a user will request which content explicitly, which can further reduce the state and action spaces and can also reduce the transmission cost during interactions with the environment. By learning the pushing policy in an emulated environment, the cost can be further cut down. Simulation results showed that the revenue of MNO can be increased by boosting user requests due to the enhanced user stickiness by recommendation. By further integrating proactive pushing based on the content recommendation and mobility pattern, the transmission cost of the network can be reduced. As a final consequence, the net profit of MNO can be increased remarkably.

This work is an early attempt to apply RL for joint content recommendation and pushing. Although the result is still preliminary, it suggests an alternative while promising way to relieve the negative impact of user behavior uncertainty on proactive caching in wireless edge and to learn the user behavior that is hard to model. Due to the lack of experiments and models for users' interaction with content recommendation, the evaluation of the proposed framework in real world scenarios are left for future works.

APPENDIX A
POLICY-BASED DRL ALGORITHMS AND THEIR FINE-TUNED HYPER-PARAMETERS

In this appendix, we provide the details of several policy-based DRL algorithms, by taking the recommendation problem as an example for illustration.

A. DDPG + KNN

DDPG maintains an actor network $\mu(\mathbf{s}; \boldsymbol{\theta}_\mu)$ and a critic network $Q(\mathbf{s}, \mathbf{a}; \boldsymbol{\theta}_Q)$. The actor network specifies the current policy by deterministically mapping states into a specific continuous action, and the critic network is used to approximate the action-value function. To apply DDPG to discrete action space, [40] proposes to embedding discrete action into a continuous space and employ k-nearest neighbor search (KNN)

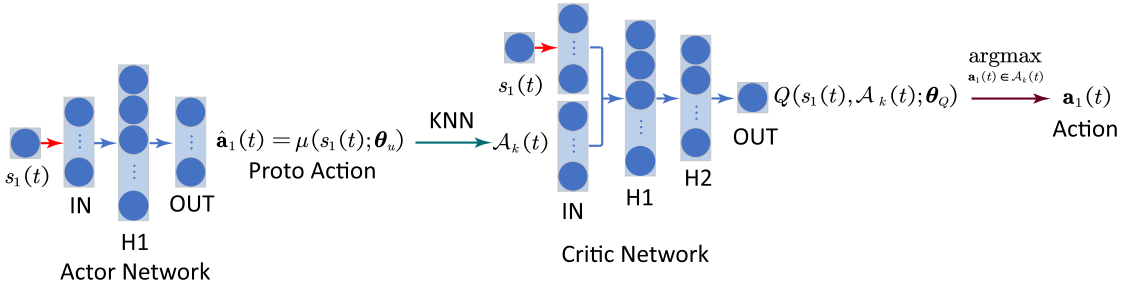


FIGURE 10. The architecture of "DDPG + KNN".

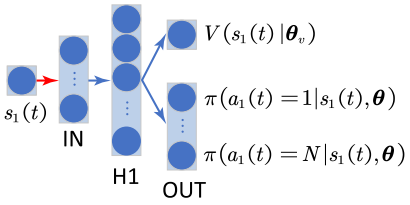


FIGURE 11. A2C with shared layers.

to find the discrete actions that are close to the output of the actor network.

The architecture of actor network and critic network used for recommendation problem are shown in Fig. 10.

The action $\mathbf{a}_1(t)$ (i.e., the content to be recommended in current time step) is denoted by a F-dimensional one-hot vector, e.g., the content with index n is denoted by a vector that only the nth element is "1" and all the other elements are "0"s. The state $s_1(t)$ (i.e., the index of last requested content) is first converted into a F-dimensional one-hot vector for the input layer of actor network and then goes through a fully-connected hidden layer. The actor network outputs a proto-action in $\mathbb{R}^F$ as $\hat{\mathbf{a}}_1(t)$, which may not be a valid action because the valid action set consists of F-dimensional one-hot vector. Then, KNN is used to find k actions in the valid action set that are closest to the proto-action $\hat{\mathbf{a}}_1(t)$ by Euclidean distance, which is denoted by $\mathcal{A}_k(t)$. Finally, the agent will recommend a content in $\mathcal{A}_k(t)$ that has the largest action value according to the output of critic network. The detailed training algorithm for "DDPG + KNN" is given by [40].

B. A2C

A2C is a synchronous variant of Asynchronous Advantage Actor Critic (A3C) [26], which has been shown to achieve the same or better performance than A3C and is more cost-effective when using single-GPU machines [41]. In A2C, there are multiple actors interact with each parallel environment simultaneously. Different from A3C, the actor and critic networks are updated until every actor has collected the experience from the environment. The architecture of A2C implementation used for recommendation problem is shown in Fig. 11.

The actor network has a hidden layer and a softmax output layer for the policy $\pi(a_1(t)|s_1(t);\boldsymbol{\theta})$, which denotes the probability that the agent executes action $a_1(t)$ for state $s_1(t)$. The critic network shares all the non-output layers with the actor network and has linear output layer for the value function $V(s_1(t);\boldsymbol{\theta}_v)$. The loss function for updating the networks is given by

$$L_{A2C} = \mathbb{E}\Big[-\log \pi(a_1(t)|s_1(t);\boldsymbol{\theta})\hat{A}(t) + \beta_1[R(t) - V(s_1(t);\boldsymbol{\theta}_v)]^2 - \beta_2 H(\pi(s_1(t);\boldsymbol{\theta}))\Big] \tag{25}$$

where $\hat{A}(t) = \sum_{i=0}^{k-1}\gamma^i r_1(t+i) + \gamma^k V(s_1(t+k);\boldsymbol{\theta}_v) - V(s_1(t);\boldsymbol{\theta}_v) \triangleq R(t) - V(s_1(t);\boldsymbol{\theta}_v)$ is the estimate of the advantage function, and the first term $-\log \pi(a_1(t)|s_1(t);\boldsymbol{\theta})\hat{A}(t)$ is the policy gradient loss for updating the actor network, $[R(t) - V(s_1(t);\boldsymbol{\theta}_v)]^2$ is the value loss for updating the critic network, and $H(\pi(s_1(t);\boldsymbol{\theta}))$ is the entropy of the policy, and β_1 and β_2 are the value loss coefficient and entropy coefficient, respectively. The entropy coefficient β_2 encourages the policy to be stochastic, which is beneficial for exploration.

C. PPO

PPO aims to improve the stability of policy gradient algorithms by ensuring the deviation from the previous policy is relatively small [27]. As mentioned in an OpenAI blog [42] "PPO has become the default reinforcement learning algorithm at OpenAI because of its ease of use and good performance". PPO shares the same network architecture as A2C but has a different loss function given by

$$L_{PPO} = \mathbb{E}\Big[-L_{CLIP}(\boldsymbol{\theta}) + \beta_1[R(t) - V(s_1(t);\boldsymbol{\theta}_v)]^2 - \beta_2 H(\pi(s_1(t);\boldsymbol{\theta})\Big] \tag{26}$$

where the first term $L_{CLIP}(\boldsymbol{\theta})$ is the clipped surrogate objective expressed by

$$L_{CLIP}(\boldsymbol{\theta}) = \min\left\{\frac{\pi(a_1(t)|s_1(t);\boldsymbol{\theta})}{\pi(a_1(t)|s_1(t);\boldsymbol{\theta}_{old})}\hat{A}(t), \text{clip}\left(\frac{\pi(a_1(t)|s_1(t);\boldsymbol{\theta})}{\pi(a_1(t)|s_1(t);\boldsymbol{\theta}_{old})}, 1-\epsilon, 1+\epsilon\right)\hat{A}(t)\right\} \tag{27}$$

whose second term $\text{clip}\left(\frac{\pi(a_1(t)|s_1(t);\boldsymbol{\theta})}{\pi(a_1(t)|s_1(t);\boldsymbol{\theta}_{old})}, 1-\epsilon, 1+\epsilon\right)\hat{A}(t)$ modifies the surrogate objective by clipping the probability ratio within $[1-\epsilon, 1+\epsilon]$.

D. FINE-TUNED HYPER-PARAMETERS

We have tried our best to tune the hyper-parameters for each algorithm for a fair comparison. For all the algorithms, the number of nodes for hidden layer H_1 is 800 and Adam is used to adjust the learning rate. The fine tuned hyper-parameters for each algorithm are listed as follows:

1) DDPG + KNN: The hidden layer H_2 in the critic network has 600 nodes. The learning rates for the actor network and critic network are 2×10^{-5} and 10^{-4}, respectively. The update rates for the target actor network and target critic network are 0.01 and 0.0025, respectively. The number of neighbors in KNN is set as 25%. Other settings are the same as dueling DDQN.
2) A2C: Four environments run in parallel and the learning rate is set as 10^{-4}. The value loss coefficient is $\beta_1 = 0.05$. To obtain a deterministic recommendation policy, the entropy coefficient is set as $\beta_2 = 0.5$ for the first 7.5×10^3 episodes and then decreases linearly to zero for the next 7.5×10^3 episodes. Other settings are set the same as the implementation by OpenAI [41].
3) PPO: In the clip function, ϵ is set as 0.2. The value loss coefficient is $\beta_1 = 1$. Again, to obtain a deterministic recommendation policy, the entropy coefficient is $\beta_2 = 0.02$ for the first 7.5×10^3 episodes and then decreases linearly to zero for the next 7.5×10^3 episodes. Other settings are the same as that of A2C.

REFERENCES

[1] N. Golrezaei, A. F. Molisch, A. G. Dimakis, and G. Caire, "Femto-caching and device-to-device collaboration: A new architecture for wireless video distribution," *IEEE Commun. Mag.*, vol. 51, no. 4, pp. 142–149, Apr. 2013.

[2] E. Zeydan, E. Bastug, M. Bennis, M. A. Kader, I. A. Karatepe, A. S. Er, and M. Debbah, "Big data caching for networking: Moving from cloud to edge," *IEEE Commun. Mag.*, vol. 54, no. 9, pp. 36–42, Sep. 2016.

[3] D. Liu, B. Chen, C. Yang, and A. F. Molisch, "Caching at the wireless edge: Design aspects, challenges, and future directions," *IEEE Commun. Mag.*, vol. 54, no. 9, pp. 22–28, Sep. 2016.

[4] L. Li, G. Zhao, and R. S. Blum, "A survey of caching techniques in cellular networks: Research issues and challenges in content placement and delivery strategies," *IEEE Commun. Surveys Tuts.*, vol. 20, no. 3, pp. 1710–1732, 3rd Quart., 2018.

[5] X. Li, X. Wang, K. Li, Z. Han, and V. C. M. Leung, "Collaborative multi-tier caching in heterogeneous networks: Modeling, analysis, and design," *IEEE Trans. Wireless Commun.*, vol. 16, no. 10, pp. 6926–6939, Oct. 2017.

[6] D. Liu and C. Yang, "Energy efficiency of downlink networks with caching at base stations," *IEEE J. Sel. Areas Commun.*, vol. 34, no. 4, pp. 907–922, Apr. 2016.

[7] J. Liu, B. Bai, J. Zhang, and K. B. Letaief, "Cache placement in Fog-RANs: From centralized to distributed algorithms," *IEEE Trans. Wireless Commu.*, vol. 16, no. 11, pp. 7039–7051, Nov. 2017.

[8] K. Wang, Z. Chen, and H. Liu, "Push-based wireless converged networks for massive multimedia content delivery," *IEEE Trans. Wireless Commun.*, vol. 13, no. 5, pp. 2894–2905, May 2014.

[9] W. Chen and H. V. Poor, "Content pushing with request delay information," *IEEE Trans. Commun.*, vol. 65, no. 3, pp. 1146–1161, Mar. 2017.

[10] M. C. Lee, A. F. Molisch, N. Sastry, and A. Raman, "Individual preference probability modeling for video content in wireless caching networks," in *Proc. IEEE GLOBECOM*, Dec. 2017, pp. 1–7.

[11] B. Chen and C. Yang, "Caching policy for cache-enabled D2D communications by learning user preference," *IEEE Trans. Commun.*, vol. 66, no. 12, pp. 6586–6601, Dec. 2018.

[12] M. D. Ekstrand, J. T. Riedl, and J. A. Konstan, "Collaborative filtering recommender systems," *Found. Trends Hum.-Comput. Interact.*, vol. 4, no. 2, pp. 81–173, Feb. 2010.

[13] T. Hofmann, "Latent semantic models for collaborative filtering," *ACM Trans. Inf. Syst.*, vol. 22, no. 1, pp. 89–115, Jan. 2004.

[14] C. A. Gomez-Uribe and N. Hunt, "The Netflix recommender system: Algorithms, business value, and innovation," *ACM Trans. Manage. Inf. Syst.*, vol. 6, no. 4, p. 13, 2016.

[15] M. Wittenberg. (May 2018). *Introducing Douyin, China's Incredibly Sticky Short Video App, Posted on Medium*. [Online]. Available: https://mondaynote.com/introducing-douyin-chinas-ridiculously-sticky-short-video-app-ab005727d89e

[16] A. Singhal, P. Sinha, and R. Pant, "Use of deep learning in modern recommendation system: A summary of recent works," *Int. J. Comput. Appl.*, vol. 180, no. 7, pp. 17–22, Dec. 2017.

[17] D. Liu and C. Yang, "Caching policy toward maximal success probability and area spectral efficiency of cache-enabled HetNets," *IEEE Trans. Commun.*, vol. 65, no. 6, pp. 2699–2714, Jun. 2017.

[18] Z. Chen, J. Lee, T. Q. S. Quek, and M. Kountouris, "Cooperative caching and transmission design in cluster-centric small cell networks," *IEEE Trans. Wireless Commun.*, vol. 16, no. 5, pp. 3401–3415, May 2017.

[19] J. Tadrous, A. Eryilmaz, and H. El Gamal, "Proactive content download and user demand shaping for data networks," *IEEE/ACM Trans. Netw.*, vol. 23, no. 6, pp. 1917–1930, Dec. 2015.

[20] L. E. Chatzieleftheriou, M. Karaliopoulos, and I. Koutsopoulos, "Caching-aware recommendations: Nudging user preferences towards better caching performance," in *Proc. IEEE INFOCOM*, May 2017, pp. 1–9.

[21] P. Sermpezis, T. Giannakas, T. Spyropoulos, and L. Vigneri, "Soft cache hits: Improving performance through recommendation and delivery of related content," *IEEE J. Sel. Areas Commun.*, vol. 36, no. 6, pp. 1300–1313, Jun. 2018.

[22] D. Liu and C. Yang, "A learning-based approach to joint content caching and recommendation at base stations," in *Proc. IEEE Globecom*, Dec. 2018, pp. 1–7.

[23] K. Guo, C. Yang, and T. Liu, "Caching in base station with recommendation via Q-learning," in *Proc. IEEE WCNC*, Mar. 2017, pp. 1–6.

[24] Z. Wang, T. Schaul, M. Hessel, H. Van Hasselt, M. Lanctot, and N. De Freitas, "Dueling network architectures for deep reinforcement learning," in *Proc. ICML*, Jun. 2016, pp. 1995–2003.

[25] 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 *Proc. ICLR*, Jun. 2016, pp. 1–3.

[26] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *Proc. ICML*, Jun. 2016, pp. 1928–1937.

[27] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," 2017. *arXiv:1707.06347*. [Online]. Available: https://arxiv.org/abs/1707.06347

[28] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: MIT Press, 1998.

[29] F. Calabrese, L. Wang, E. Ghadimi, P. Gunnar, L. Hanzo, and P. Soldati, "Learning radio resource management in RANs: Framework, opportunities and challenges," *arXiv preprint: 1611.10253*. [Online]. Available: http://arxiv.org/abs/1611.10253

[30] P. Covington, J. Adams, and E. Sargin, "Deep neural networks for youtube recommendations," in *Proc. ACM RecSys*, Sep. 2016, pp. 191–198.

[31] G. Shani, D. Heckerman, and R. I. Brafman, "An MDP-based recommender system," *J. Mach. Learn. Res.*, vol. 6, pp. 1265–1295, Sep. 2005.
[32] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Mach. Learn.*, vol. 8, nos. 3–4, pp. 279–292, 1992.
[33] 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, p. 529, 2015.
[34] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *Proc. AAAI*, 2016, pp. 1–10.
[35] P. Gill, M. Arlitt, Z. Li, and A. Mahanti, "Youtube traffic characterization: A view from the edge," in *Proc. ACM SIGCOMM IMC*, Oct. 2007, pp. 15–28.
[36] F. M. Harper and J. A. Konstan, "The movielens datasets: History and context," *ACM Trans. Interact. Intell. Syst.*, vol. 5, pp. 19:1–19:19, Dec. 2015.
[37] R. Kohavi, R. Longbotham, D. Sommerfield, and R. M. Henne, "Controlled experiments on the web: Survey and practical guide," *Data Mining Knowl. Discovery*, vol. 18, no. 1, pp. 140–181, Feb. 2009.
[38] M. Abadi, A. Agarwal, P. Barham, and E. Brevdo. (2018). *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. [Online]. Available: http://tensorflow.org/
[39] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proc. ICLR*, 2014, pp. 1998–2006.
[40] G. Dulac-Arnold, R. Evans, H. van Hasselt, P. Sunehag, T. Lillicrap, J. Hunt, T. Mann, T. Weber, T. Degris, and B. Coppin, "Deep reinforcement learning in large discrete action spaces," 2015, *arXiv:1512.07679*. [Online]. Available: https://arxiv.org/abs/1512.07679
[41] O. AI. (2014). *OpenAI Baselines: ACKTR & A2C*. [Online]. Available: https://openai.com/blog/baselines-acktr-a2c
[42] (2014). *Proximal Policy Optimization*. [Online]. Available: https://openai.com/blog/openai-baselines-ppo/

DONG LIU (S'13) received the B.S. degree in electronics engineering and the Ph.D. degree in signal and information processing from Beihang University (formerly Beijing University of Aeronautics and Astronautics), Beijing, China, in 2013 and 2019, respectively. His current research interest includes caching and machine learning in wireless networks.

CHENYANG YANG (SM'08) received the Ph.D. degree in electrical engineering from Beihang University (formerly Beijing University of Aeronautics and Astronautics, BUAA), China, in 1997, where she has been a Full Professor with the School of Electronics and Information Engineering, since 1999. She has published over 200 papers in wireless caching, URLLC, energy-efficient transmission, CoMP, interference management, cognitive radio, and relay. Her recent research interests include mobile AI, wireless caching, and URLLC. She was supported by the 1st Teaching and Research Award Program for Outstanding Young Teachers of Higher Education Institutions by the Ministry of Education of China. She was the Chair of the Beijing Chapter of the IEEE Communications Society, from 2008 to 2012. She has ever served as an Associate Editor for the IEEE TRANSACTIONS ON WIRELESS COMMUNICATION, a Guest Editor for the IEEE JOURNAL OF SELECTED TOPICS IN SIGNAL PROCESSING and the IEEE JOURNAL OF SELECTED AREAS IN COMMUNICATIONS, and as a TPC Member, the TPC Co-Chair or the Track Co-Chair for many IEEE conferences.