Adaptive Caching via Deep Reinforcement Learning
Alireza Sadeghi, Gang Wang, Member, IEEE, and Georgios B. Giannakis, Fellow, IEEE

Abstract—Caching is envisioned to play a critical role in next-generation content delivery infrastructure, cellular networks, and Internet architectures. By smartly storing the most popular contents at the storage-enabled network entities during off-peak demand instances, caching can benefit both network infrastructure as well as end users, during on-peak periods. In this context, distributing the limited storage capacity across network entities calls for decentralized caching schemes. Many practical caching systems involve a parent caching node connected to multiple leaf nodes to serve user file requests. To model the two-way interactive influence between caching decisions at the parent and leaf nodes, a reinforcement learning framework is put forth. To handle the large continuous state space, a scalable deep reinforcement learning approach is pursued. The novel approach relies on a deep Q-network to learn the Q-function, and thus the optimal caching policy, in an online fashion. Reinforcing the parent node with ability to learn-and-adapt to unknown policies of leaf nodes as well as spatio-temporal dynamic evolution of file requests, results in remarkable caching performance, as corroborated through numerical tests.

Index Terms—Caching, deep reinforcement learning, deep Q-network, next-generation networks, function approximation.

I. INTRODUCTION

In light of the tremendous growth of data traffic over both wireline and wireless communications, next-generation networks including future Internet architectures, content delivery infrastructure, and cellular networks stand in need of emerging technologies to meet the ever-increasing data demand. Recognized as an appealing solution is caching, which amounts to storing reusable contents in geographically distributed storage-enabled network entities so that future requests for those contents can be served faster. The rationale is that unfavorable shocks of peak traffic periods can be smoothed by proactively storing ‘anticipated’ highly popular contents at those storage devices and during off-peak periods [24], [3]. Caching popular content is envisioned to achieve major savings in terms of energy, bandwidth, and cost, in addition to user satisfaction [24].

To fully unleash its potential, a content-agnostic caching entity has to rely on available observations to learn what and when to cache. Toward this objective, contemporary machine learning and artificial intelligence tools hold the promise to empower the next-generation networks with ‘smart’ caching control units (CCU), that can learn, track, and adapt to unknown dynamic environments, including space-time evolution of content popularities and network topology, as well as entity-specific caching policies.

Deep neural networks have lately boosted the notion of “learning from data” with field-changing performance improvements reported in diverse machine learning and artificial intelligence tasks [14]. Deep neural networks (DNNs) can cope with the ‘curse of dimensionality’ by providing compact low-dimensional representations of high-dimensional input data [4], [39]. Combining deep learning with reinforcement learning [17], deep reinforcement learning has created the first artificial agents to achieve human-level performance across various challenging domains, including AlphaGo beating the world Go champion [21], machine translation [38], and intelligent transportation systems [2]. As another example, a DNN system was built to operate Google’s data centers, and shown able to consistently achieve a 40% reduction in the amount of energy consumed for cooling [13]. The system provides a general-purpose framework to understand complex dynamics, which has been applied to address other challenges in the data center environment and beyond.

A. Prior art on caching

Early approaches to caching include the least recently used (LRU), least frequently used (LFU), first in first out (FIFO), random replacement (RR) policies, and their variants. Albeit simple, these schemes cannot deal with the dynamics of content popularities and network topologies. Recent efforts have gradually shifted toward developing learning and optimization based approaches, that can ‘intelligently’ manage the cache resources. For unknown yet time-invariant content popularities, multi-armed bandit online learning (e.g., [18]) was pursued in [5], while distributed, coded, and convexified generalizations were lately reported in [30], [15].

In realistic networks however, popularities exhibit dynamics, which motivate well the so-called dynamic caching. A Poisson shot noise model was adopted to approximate the evolution of popularities in [34], for which an age-based caching solution was developed in [16]. Reinforcement learning based methods have been pursued in [28], [32], [29]. Specifically, a Q-learning based caching scheme was developed in [28] to model global and local content popularities as Markovian processes. Considering the Poisson shot noise popularity dynamics, a policy gradient reinforcement learning based caching scheme was devised in [32]. Assuming stationary file popularities and service costs, a dual-decomposition based Q-learning approach was pursued in [29]. However, these approaches entail discrete or quantized states, which prevents them from dealing with a practically large continuous state-action space.

On the other hand, the aforementioned contributions focus on developing caching policies for a single caching entity. A more common setting emerging in next-generation networks however, involves a network of interconnected caching nodes.

This work was supported in part by NSF grants 1711471, 1500713, and 1514056. The authors are with the Digital Technology Center and the Department of Electrical and Computer Engineering, University of Minnesota, Minneapolis, MN 55455, USA. E-mails: {sade012, gangwang, georgios} @umn.edu.
It has been shown that considering a network of connected caches jointly can considerably improve performance \cite{20, 7}. For instance, leveraging network topology and the broadcast nature of links, the coded caching strategy in \cite{20} further reduces data traffic over a network. This idea has been generalized in \cite{25} to an online setting, where popularities are modeled Markov processes. Collaborative and distributed online learning approaches (e.g., \cite{10, 7, 15, 36}) have also been pursued.

We identify the following challenges that need to be addressed when designing practical caching methods for next-generation networks.

\textbf{c1) Networked caching.} Caching decisions of a node, in a network of caches, influences decisions of all other nodes. Thus, a desired caching policy must adapt to the network topology and policies of neighboring nodes.

\textbf{c2) Complex dynamics.} Content popularities are random and exhibit unknown space-time, heterogeneous, and often non-stationary dynamics over the entire network.

\textbf{c3) Large continuous state space.} Due to the sheer size of available content, caching nodes, and possible realizations of content requests, the decision space is huge.

\subsection{This work}

Targeting these challenges, this paper considers a two-level network caching where a parent node is connected to multiple leaf nodes to serve user file requests. Such a network indeed acts as the building block in many practical caching settings, including content delivery and 5G networks. We model the interaction between caching decisions of the parent and leaf nodes, along with the space-time evolution of file requests, from a reinforcement learning vantage point. Then, we reinforce the parent node with a policy, that can learn-and-adapt to space-time evolution of file requests as well as local policies used by leaf nodes. This policy just relies on a few observations from leaf nodes to make caching decisions. Nevertheless, the continuous state space and large number of actions challenge classical reinforcement learning algorithms. To address this challenge, a deep Q-network along with experience replay and target network is used to first approximate and then learn the so-called Q-function, hence the optimal policy in an online fashion.

\section{Modeling and Problem Statement}

Consider a two-level network of interconnected caching nodes, where a parent node is connected to \( N \) leaf nodes, indexed by \( n \in \mathcal{N} := \{1, \ldots, N\} \). The parent node is connected to the cloud through a (typically congested) backhaul link; see Fig. 1. One could consider this network as a part of a large hierarchical caching system, where the parent node is connected to a higher level caching node instead of the cloud; see Fig. 2. In a content delivery network for instance, edge servers (a.k.a. points of presence or PoPs) are the leaf nodes, and a fog server acts as the parent node. Likewise, (small) base stations in a 5G cellular network are the leaf nodes, while a serving gate way (S-GW) may be considered as the parent node; see also \cite{9} p. 110.

All nodes in this network store files to serve file requests. Every leaf node serves its locally connected end users, by providing their requested files. If a requested content is locally available at a leaf node, the content will be served immediately at no cost. If it is not locally available due to limited caching capacity, the content will be fetched from its parent node, at a certain cost. Similarly, if the file is available at the parent node, it will be served to the leaf at no cost; otherwise, the file must be fetched from the cloud at a higher cost.

To mitigate the burden with local requests on the network, each leaf node stores ‘anticipated’ locally popular files. In addition, this paper considers that each parent node stores files to serve requests that are not locally served by leaf nodes. Since leaf nodes are closer to end users, they frequently receive file requests that exhibit rapid temporal evolution at a fast timescale. The parent node on the other hand, observes aggregate requests over a large number of users served by the \( N \) leaf nodes, which naturally exhibit smaller fluctuations and thus evolve at a slow timescale.

This motivated us to pursue a two-timescale approach to managing such a network of caching nodes. To that end, let \( \tau = 1, 2, \ldots \) denote the slow time intervals, each of which is further divided into \( T \) fast time slots indexed by \( t = 1, \ldots, T \); see Fig. 3 for an illustration. Each fast time slot may be e.g., 1-2 minutes depending on the dynamics of local requests, while each slow time interval is a period of say 4-5 minutes. We assume that the network state remains unchanged during each fast time slot \( t \), but can change from \( t \) to \( t + 1 \).

Consider a total of \( F \) files in the cloud, which are col-
lected in the set $F = \{1, \ldots, F\}$. At the beginning of each slot $t$, every leaf node $n$ selects a subset of files in $F$ to prefetch and store for possible use in this slot. To determine which files to store, every leaf node relies on a local caching policy function denoted by $\pi_n$, to take (cache or no-cache) action $a_n(t+1, \tau) = \pi_n(s_n(t, \tau))$ at the beginning of slot $t+1$, based on its state vector $s_n$ at the end of slot $t$. We assume this action takes a negligible amount of time relative to the slot duration; and define the state vector $s_n(t, \tau) := [r^0_n(t, \tau) \cdots r^n_n(t, \tau)]^T$ to collect the number of requests received at leaf node $n$ for individual files over the duration of slot $t$ on interval $\tau$. Likewise, to serve file requests that have not been served by leaf nodes, the parent node takes action $a_0(\tau)$ to store files at the beginning of every interval $\tau$, according to a certain policy $\pi_0$. Again, as aggregation smoothes out request fluctuations, the parent node observes slowly varying file requests, and can thus make caching decisions at a relatively slow timescale. In the next section, we present a two-timescale approach to managing such a network of caching nodes.

III. Two-timescale Problem Formulation

File transmission over any network link consumes resources, including e.g., energy, time, and bandwidth. Hence, serving any requested file that is not locally stored at a node, incurs a cost. Among possible choices, the present paper considers the following cost for node $n \in \mathcal{N}$, at slot $t+1$ of interval $\tau$

$$c_n(\pi_n(s_n(t, \tau)), r_n(t+1, \tau), a_0(\tau)) := r_n(t+1, \tau) \odot (1-a_0(\tau))$$

$$\odot (1-a_n(t+1, \tau)) + r_n(t+1, \tau) \odot (1-a_n(t+1, \tau))$$

(1)

where $c_n(\cdot) := [c^0_n(\cdot) \cdots c^n_n(\cdot)]^T$ concatenates the cost for serving individual files per node $n$; symbol $\odot$ denotes entrywise vector multiplication; entries of $a_0$ and $a_n$ are either 1 (cache, hence no need to fetch), or, 0 (no-cache, hence fetch); and 1 stands for the all-one vector. Specifically, the second summand in (1) captures the cost of the parent node fetching files for end users, while the first summand corresponds to that of the parent fetching files from the cloud.

We model user file requests as Markov processes with unknown transition probabilities [28]. Per interval $\tau$, a reasonable caching scheme for leaf node $n \in \mathcal{N}$ could entail minimizing the expected cumulative cost; that is,

$$\pi^*_n,\tau := \arg\min_{\pi_n \in \Pi_n} E_{t=1}^{T} c_n(\pi_n(s_n(t, \tau)), r_n(t+1, \tau), a_0(\tau))$$

(2)

where $\Pi_n$ represents the set of all feasible policies for node $n$. Although solving (2) is in general challenging, efficient near-optimal solutions have been introduced in several recent contributions; see e.g., [28], [32], [5], and references therein. In particular, a reinforcement learning based approach using tabular Q-learning was pursued in our precursor [28], which can be employed here to tackle this fast timescale optimization. The remainder of this paper will thus be on designing the caching policy $\pi_0$ for the parent node, that can learn, track, and adapt to the leaf node policies as well as user file requests.

At the beginning of every interval $\tau + 1$, the parent node takes action to refresh its cache; see Fig. 4. To design a caching policy that can account for both leaf node policies and file popularity dynamics, the parent collects local information from all leaf nodes. At the end of interval $\tau$, each leaf node $n \in \mathcal{N}$ first obtains the time-average state vector $\bar{s}_n(\tau) := (1/T) \sum_{t=1}^{T} s_n(t, \tau)$, and subsequently forms and forwards the per-node vector $s_n(\tau) \odot (1 - \pi_n(\bar{s}_n(\tau)))$ to its parent node. This vector has nonzero entries the average number of file requests received during interval $\tau$, and zero entries if $\pi_n$ stores the corresponding files. Using the latter, the parent node forms its ‘weighted’ state vector as

$$s_0(\tau) := \sum_{n=1}^{N} w_n \bar{s}_n(\tau) \odot (1 - \pi_n(\bar{s}_n(\tau)))$$

(3)

where the weights $\{w_n \geq 0\}$ control the influence of leaf nodes $n \in \mathcal{N}$ on the parent node’s policy. Similarly, having received at the end of interval $\tau + 1$ the slot-averaged costs

$$\bar{c}_n(\pi_0(s_0(\tau)), \{a_n(t, \tau + 1), r_n(t, \tau + 1)\}_{t=1}^{T})$$

$$= \frac{1}{\tau + 1} \sum_{t=1}^{T} c_n(\pi_0(s_0(\tau)), \{a_n(t, \tau + 1), r_n(t, \tau + 1), a_0(\tau + 1)\})$$

(4)

from all leaf nodes, the parent node determines its cost

$$c_0(s_0(\tau), \pi_0(s_0(\tau)))$$

$$= \sum_{n=1}^{N} w_n \bar{c}_n(\pi_0(s_0(\tau)), \{a_n(t, \tau + 1), r_n(t, \tau + 1)\}_{t=1}^{T})$$

(5)

Having observed $\{s_0(t'), a_0(t'), c_0(s_0(t'-1), a_0(t'))\}_{t'=1}^{\tau}$, the objective is to take a well-thought-out action $a_0(\tau + 1)$ for next interval. This is tantamount to finding an optimal policy function $\pi^*_0$ to take a caching action $a_0(\tau + 1) = \pi^*_0(s_0(\tau))$.

Since the requests $\{r_n(t, \tau)\}_{n,t}$ are Markovian and present actions $\{a_n(t, \tau)\}_{n,t}$ also affect future costs, the optimal RL policy for the parent node will minimize the expected cumulative cost over all leaf nodes in the long term, namely

$$\pi^*_0 := \arg\min_{\pi_0 \in \Pi_0} E_{t=1}^{\infty} \gamma^{t-1} c_0(\pi_0(s_0(\tau), \pi_0(s_0(\tau)))$$

(6)

where $\Pi_0$ represents the set of all feasible policies; expectation is over $\{r_n(t, \tau)\}_{n,t}$, as well as (possibly random) parent
\(a_0(\tau)\) and leaf actions \(\{a_n(t, \tau)\}_{n,t}\); and the discount factor \(\gamma \in [0, 1)\) trades off emphasis of current versus future costs.

It is evident from (6) that the decision taken at a given state \(s_0(\tau)\), namely \(a_0(\tau + 1) = \pi_0(s_0(\tau))\), influences the next state \(s_0(\tau + 1)\) through \(\pi_{n,\tau}(\cdot)\) in (2), as well as the cost \(c_0(\cdot)\) in (5). Therefore, problem (6) is a discounted infinite time horizon Markov decision process (MDP). Finding the optimal policy of an MDP is NP-hard [23]. To cope with this complexity of solving (6), an adaptive reinforcement learning approach is pursued next.

IV. ADAPTIVE RL-BASED CACHING

Reinforcement learning deals with action-taking policy function estimation in an environment with dynamically evolving states, so as to minimize a long-term cumulative cost. By interacting with the environment (through successive actions and observed states and costs), RL seeks a policy function (of interacting states) to draw actions from, in order to minimize the average and observed states and costs), RL seeks a policy function (of states, so as to minimize a long-term cumulative cost. By function estimation in an environment with dynamically evolving states, the parent node will be dropped whenever it is clear from the knowledge of the remaining states [33, p. 46]. To formalize this, define first the state transition probability from the current state \(s\) to the next state \(s'\) upon taking action \(a\) as

\[
P_{ss'} = \Pr\{s(\tau + 1) = s' | s(\tau) = s, a = \pi(s)\}. \tag{8}
\]

Leveraging (7) and (8), the Bellman equation for the value function \(V_\pi(\cdot)\) is given by

\[
V_\pi(s) = \mathbb{E}\left[\sum_{\tau=0}^{\infty} \gamma^\tau c_0(s_\tau, \pi_\tau(s_\tau)) \right] \tag{9}
\]

where the average immediate cost can be found as

\[
\mathbb{E}\left[\sum_{s'} P_{ss'} \mathbb{E}\left[\sum_{\tau=0}^{\infty} \gamma^\tau c_0(s_\tau, \pi_\tau(s_\tau))|s', \pi(s)\right]\right]. \tag{10}
\]

With \(A(S)\) denoting the action (state) space, if \(P_{ss'}^a\), were known \(\forall \alpha \in A\) and \(\forall s, s' \in S\), then finding \(V_\pi(\cdot)\) would be equivalent to solving the system of linear equations (9). Indeed, if one could afford the complexity of evaluating \(V_\pi(\cdot)\) for all \(\pi \in \Pi_0\), the optimal policy \(\pi^*\) is the one that minimizes the value function for all states. However, the large (possibly infinite) number of policies in practice discourages such an approach. An alternative is to employ the so-termed policy iteration algorithm [33, p. 64] outlined next. Define first the state-action value function, also called Q-function for policy \(\pi\)

\[
Q_\pi(s, a) := \mathbb{E}\left[\sum_{\tau=0}^{\infty} \gamma^\tau c_0(s_\tau, \pi_\tau(s_\tau)) | s_{(0)} = s, a = \pi(s)\right] + \gamma \sum_{s'} P_{ss'}^a V_\pi(s'). \tag{11}
\]

This function captures the expected immediate cost of starting from state \(s\), taking the first action to be \(a\), and subsequently following policy \(\pi\) to take future actions onwards. The only difference between the value function in (7) and that of Q-function in (11) is that the former takes the first action according to policy \(\pi\), while the latter begins with \(a\) as the first action, which may not necessarily be taken when adhering to \(\pi\). Having defined the Q-function, we are ready to present the policy iteration algorithm, in which each iteration \(i\) entails the following two steps:

\begin{align*}
\text{Policy Evaluation.} & \quad \text{Find } V_{\pi^i}(\cdot) \text{ for the current policy } \pi^i \text{ by solving the system of linear equations in } (9). \\
\text{Policy Improvement.} & \quad \text{Update the current policy greedily as }
\begin{align*}
\pi^{i+1}(s) = \arg\min_{a \in A} Q_{\pi^i}(s, a). \tag{12}
\end{align*}
\end{align*}

To perform policy evaluation, we rely on knowledge of \(P_{ss'}^a(\cdot)\). However, this is impractical in our setup that involves dynamic evolution of file requests, and unknown caching policies for the leaf nodes. This calls for approaches that target directly the optimal policy \(\pi^*\), without knowing \(P_{ss'}^a, \forall a, s, s'\). One such approach is Q-learning [33, p. 107].

In the ensuing section, we first introduce a Q-learning based adaptive caching scheme, which is subsequently generalized in Section V by invoking a deep Q-network.

A. Q-learning based Adaptive Caching

The Q-learning algorithm finds the optimal policy \(\pi^*\) by estimating \(Q_{\pi^*}(\cdot, \cdot)\) ‘on-the-fly.’ It turns out that \(\pi^*\) is the greedy policy over the optimal Q-function [33, p. 64], that is

\[
\pi^*(s) = \arg\min_{a \in A} Q_{\pi^*}(s, a), \quad \forall s \in S \tag{13}
\]

where \(Q_{\pi^*}\) is estimated using Bellman’s equation for the Q-function. This is possible because the \(V\)-function is linked with the Q-function under \(\pi^*\) through (see [33] p. 51) for details

\[
V_{\pi^*}(s) = \min_{a \in A} Q_{\pi^*}(s, a), \quad \forall s. \tag{14}
\]

Substituting (14) into (11), Bellman’s equation for the Q-function under \(\pi^*\) is expressed as

\[
Q_{\pi^*}(s, a) = \mathbb{E}\left[\sum_{s'} P_{ss'}^a V_{\pi^*}(s') + \gamma \sum_{s'} P_{ss'}^a \min_{a'} Q_{\pi^*}(s', a')\right] \tag{15}
\]

which plays a key role in many reinforcement learning algorithms. Examples include Q-learning [37], and SARSA [26], where one relies on (15) to update estimates of the Q-function in a stochastic manner. In particular, the Q-learning algorithm follows an exploration-exploitation procedure to take some action \(a\) in a given state \(s\). Specifically, it chooses the action minimizing the current estimate of \(Q_{\pi^*}(\cdot, \cdot)\) denoted by \(\hat{Q}(\cdot, \cdot)\), with probability (w.p.) \(1 - \epsilon\), or, it takes a random action \(a \in A\) otherwise; that is,

\[
a = \begin{cases} 
\arg\min_{a \in A} \hat{Q}(s, a), & \text{w.p. } 1 - \epsilon \\
\text{random } a \in A, & \text{w.p. } \epsilon
\end{cases}
\]
Algorithm 1: Q-learning based adaptive caching

Initialize: \(s(0), Q_0(s,a), \forall s,a,\) and \(s_n(t,\tau), \forall n\)

\[
\text{for } t = 1, 2, \ldots \text{ do } \\
\text{Take action } a(\tau) \text{ via exploration-exploitation} \\
a(\tau) = \arg\min_{a \in A} \{ \pi_n(s(\tau - 1), a), \text{ w.p. } 1 - \epsilon \tau \} \\
\text{random } a \in A, \text{ w.p. } \epsilon \tau \\
\text{for } t = 1, \ldots, T \text{ do } \\
\text{for } n = 1, \ldots, N \text{ do } \\
\text{Take action } a_n(t,\tau) \text{ via } \pi_n \\
a_n(t,\tau) = \{ \pi_n(s(t-1), \tau), \text{ if } t \neq 1 \}
\{ \pi_n(s(T, t-1)), \text{ if } t = 1 \}
\text{Requests } \{r_n(t,\tau)\} \text{ are revealed} \\
\text{Set } s_n(t,\tau) = r_n(t,\tau) \\
\text{Incur cost } c_n(\cdot), \text{ cf. } [1] \\
\text{end } \\
\text{Leaf nodes} \\
\text{Set } \tilde{s}_n(\tau) := \left( 1/T \right) \sum_{t=1}^{T} s_n(t,\tau) \\
\text{Send } \tilde{s}_n(\tau) \oplus \left( 1 - \pi_n(s_n(\tau)) \right) \text{ to parent node} \\
\text{Send } c_n(\cdot), \text{ cf. } [1], \text{ to parent node} \\
\text{Parent node} \\
\text{Set } s(\tau) := \sum_{n=1}^{N} w_n \tilde{s}_n(\tau) \oplus \left( 1 - \pi_n(s_n(\tau)) \right) \\
\text{Find } c(s(\tau - 1), a(\tau)), \text{ cf. } [5] \\
\text{Update} \\
\hat{Q}_{\tau+1}(s(\tau - 1), a(\tau)) = (1 - \beta) \hat{Q}_{\tau}(s(\tau - 1), a(\tau)) + \\
\beta \left[ 1^T c(s(\tau - 1), a(\tau)) + \gamma \min_{a \in A} \hat{Q}_{\tau}(s(\tau), a) \right] \\
\text{end}
\]

After taking action \(a\), moving to some new state \(s'\), and incurring cost \(\epsilon\), the Q-learning algorithm adopts the following loss function for the state-action pair \((s, a)\)

\[
L(s, a) = \frac{1}{2} \left( 1^T c(s, a) + \min_{a \in A} \hat{Q}_\tau(s', a) - \hat{Q}_\tau(s, a) \right)^2. 
\tag{16}
\]

The estimated Q-function for a single state-action pair is subsequently updated, by following a gradient descent step to minimize the loss in (16), which yields the update

\[
\hat{Q}_{\tau+1}(s,a) = \hat{Q}_\tau(s,a) - \beta \frac{\partial L(s,a)}{\partial \hat{Q}_\tau(s,a)} 
\tag{17}
\]

where \(\beta > 0\) is some step size. Upon evaluating the gradient and merging terms, the update in (17) boils down to

\[
\hat{Q}_{\tau+1}(s,a) = (1 - \beta) \hat{Q}_\tau(s,a) + \beta \left[ 1^T c(s,a) + \gamma \min_{a \in A} \hat{Q}_\tau(s',a) \right].
\]

For easy reference, the main steps of our two-timescale Q-learning based caching scheme are listed as Alg. [1].

Three remarks are worth making at this point.

Remark 1. As far as the fast-timescale caching strategy of leaf nodes is concerned, multiple choices are possible, including, e.g., LRU, LFU, FIFO, [12], the multi-armed bandit scheme [5], and even reinforcement learning ones [28, 32].

Remark 2. The exploration-exploitation step for taking actions guarantees continuously visiting state-action pairs, and ensures convergence to the optimal Q-function [23]. Instead of the \(\epsilon\)-greedy exploration-exploitation step in Alg. [1] one can employ the upper confidence bound scheme [2]. Technically, any exploration-exploitation scheme should be greedy in the limit of infinite exploration (GLIE) [27, p. 840]. An example obeying GLIE is the \(\epsilon\)-greedy algorithm [27, p. 840] with \(\epsilon = 1/\tau\). It converges to an optimal policy, albeit at a very slow rate. On the other hand, using a constant \(\epsilon = \epsilon\) approaches the optimal \(Q^*(\cdot,\cdot)\) faster, but its exact convergence is not guaranteed as it is not GLIE.

Clearly, finding \(Q_\pi(s,a)\) entails estimating a function defined over state and action spaces. In several applications however, at least one of the two vector variables is either continuous or takes values from an alphabet of high cardinality. Revisiting every state-action pair in such settings is impossible, due to the so-called curse of dimensionality – a typical case in practice. To deal with it function approximation techniques offer as a promising solution [33]. These aim at finding the original Q-function over all feasible state-action pairs, by judicious generalization from a few observed pairs.

Early function approximators for reinforcement learning design good hand-crafted features that can properly approximate \(V_\pi(\cdot), Q_\pi(\cdot, \cdot),\) or, \(\pi^*(\cdot)\) [8]. Their applicability has only been limited to domains, where such features can be discovered, or, to state spaces that are low dimensional [11].

Deep learning approaches on the other hand, have recently demonstrated remarkable potential in applications such as object detection, speech recognition, and language translation, to name a few [11]. This is because DNNs are capable of extracting compact low-dimensional features from high-dimensional data. Wedding DNNs with reinforcement learning results in ‘deep reinforcement learning’ that can effectively deal with the curse of dimensionality, by eliminating the need of hand Crafting good features. These considerations have inspired the use of DNNs to estimate either the Q-function, value function, or, the policy [11]. The most remarkable success in playing AI games adopted DNNs to estimate the Q-function, what is termed deep Q-network (DQN) in [21].

Prompted by this success, the next leverages the power of function approximation through DNNs to develop an adaptive caching scheme for the parent node.

V. ADAPTIVE DQN-BASED CACHING

To find the optimal caching policy \(\pi^*\) for the parent node, the success of function approximators for RL (e.g., [8]), motivated us to pursue a parametric approach to estimating \(Q(s,a)\) with a DNN (cf. [13]). This DNN has as input pairs of vectors \((s,a)\), and as scalar output the corresponding estimated \(Q(s,a)\) values. Clearly, the joint state-action space of the sought Q-function has cardinality \(|A| \times |S|\) = \(|A||S|\). To reduce the search space, we shall take a parametric DQN approach [21] that we adapt to our setup, as elaborated next.

Consider a deep feedforward NN with \(L\) fully connected layers, with input the \(F \times 1\) state vector \(s(\tau)\) as in [3], and \(F \times 1\) output cost vector \(a(\tau + 1)\) [21]. Note that input does not include the action vector \(a\). Each hidden layer \(l \in \{2, \ldots, L - 1\}\) comprises \(n_l\) neurons with rectified
linear unit (ReLU) activation functions \( h(z) := \max(0, z) \) for \( z \in \mathbb{R} \); see e.g., [35]. Neurons of the \( L \)-th output layer use a softmax nonlinearity \( \sigma \) to yield for the \( f \)-th entry of \( o \), an estimated long term cost \( o_f \) that would incur, if file \( f \) is not stored at the cache.

If this cache memory can store up to \( M \) file entries at the parent node, the \( M \) largest entries of the DQN output \( o(\tau + 1) \) are chosen by the decision module in Fig. 5 to obtain the action vector \( a(\tau + 1) \). We can think of our DQN as a ‘soft policy’ function estimator, and \( o(\tau + 1) \) as a ‘predicted cost’ or a ‘soft action’ vector for interval \( \tau + 1 \), whose ‘hard thresholding’ yields \( a(\tau + 1) \). It will turn out that excluding \( a \) from the DQN input and picking it up at the output lowers the search space from \(|\mathcal{A}||\mathcal{S}|\) to \(|\mathcal{S}|\).

To train our reduced-complexity DQN amounts to finding a set of weights collected in the vector \( \theta \) that parameterizes the input-output relationship \( o(\tau + 1) = Q(s(\tau); \theta) \). To recursively update \( \theta \) to \( \theta_{\tau+1} \), consider two successive intervals along with corresponding states \( s(\tau) \) and \( s(\tau + 1) \); the action \( a(\tau + 1) \) taken at the beginning of interval \( \tau + 1 \); and, the cost \( c(\tau+1) \) revealed at the end of interval \( \tau + 1 \). The instantaneous approximation of the optimal cost-to-go from interval \( \tau + 1 \) is given by \( Q(s(\tau + 1); \theta_{\tau}) \), where \( Q(s(\tau + 1); \theta_{\tau}) \) is the immediate cost, and \( Q(s(\tau + 1); \theta_{\tau}) \) represents the predicted cost-to-go from interval \( \tau + 2 \) that is provided by our DQN with \( \theta_{\tau} \), and discounted by \( \gamma \). Since our DQN offers \( Q(s(\tau); \theta_{\tau}) \) as the predicted cost for interval \( \tau + 1 \), the prediction error of this cost as a function of \( \theta_{\tau} \) is given by

\[
\delta(\theta_{\tau}) := [c(\tau + 1) + \gamma Q(s(\tau + 1); \theta_{\tau}) - Q(s(\tau); \theta_{\tau})] \\
\quad \quad \quad \quad \implies (1 - a(\tau + 1)) \tag{18}
\]

and has non-zero entries for files not stored at interval \( \tau + 1 \).

Using the so-called experience \( E_{\tau+1} := [s(\tau), a(\tau + 1), c(\tau + 1), s(\tau + 1)] \), and the \( \ell_2 \)-norm of \( \delta(\theta_{\tau}) \) as criterion

\[
L(\theta_{\tau}) = \|\delta(\theta_{\tau})\|_2^2 \tag{19}
\]

the sought parameter update minimizing (19) is given by the stochastic gradient descent (SGD) iteration as

\[
\theta_{\tau+1} = \theta_{\tau} - \beta_{\tau} \nabla L(\theta_{\tau})_{|_{\theta=\theta_{\tau}}} \tag{20}
\]

where \( \beta_{\tau} > 0 \) denotes the learning rate.

Since the dimensionality of \( \theta \) can be much smaller than \(|\mathcal{S}|\), the DQN is efficiently trained with few experiences, and generalizes to unseen state vectors. Unfortunately, DQN model inaccuracy can propagate in the cost prediction error in (18) that can cause instability in (20), which can lead to performance degradation, and even divergence [31, 33]. Moreover, (20) leverages solely a single most recent experience \( E_{\tau+1} \). These limitations will be mitigated as elaborated next.

A. Target Network and Experience Replay

NN function approximation, along with the loss (19) and the update (20), often result in unstable RL algorithms [21]. This is due to: i) correlated experiences used to update the DQN parameters \( \theta \); and, ii) the influence of any change in policy on subsequent experiences and vice versa.

Possible remedies include the so-called experience replay and target network to update the DQN weights. In experience replay, the parent node stores all past experiences \( E_{\tau} \in \mathcal{E} := \{E_{1}, \ldots, E_{\tau}\} \), and utilizes a batch of \( B \) uniformly sampled experiences from this data set, namely \( \{E_{i_{\tau}}\}_{i_{\tau}=1}^{B} \sim U(\mathcal{E}) \). By sampling and replaying previously observed experiences, experience replay can overcome the two challenges. On the other hand, to obtain decorrelated target values in (18), a second NN (called target network) with structure identical to the DQN is invoked with parameter vector \( \theta_{\text{Tar}} \). Interestingly, \( \theta_{\text{Tar}} \) can be periodically replaced with \( \theta_{\tau} \) every \( C \) training iterations of the DQN, which enables the target network to smooth out fluctuations in updating the DQN [21].

With a randomly sampled experience \( E_{i_{\tau}} \in \mathcal{E} \), the prediction error with the target cost-to-go estimated using the target network (instead of the DQN) is

\[
\delta_{\text{Tar}}(\theta; E_{i_{\tau}}) := [c(i_{\tau} + 1) + \gamma Q(s(i_{\tau} + 1); \theta_{\text{Tar}}) - Q(s(i_{\tau}); \theta)] \\
\quad \quad \quad \quad \implies (1 - a(i_{\tau} + 1)). \tag{21}
\]

Different from (18), the target values here are found through the target network with weights \( \theta_{\text{Tar}} \). In addition, the error in
Algorithm 2: Deep RL for adaptive caching.

Initialize: $s(0)$, $s_n(t, \tau)$, $\forall n$, $\theta_r$, and $\theta^\tau$

for $\tau = 1, 2, \ldots$ do
  Take action $a(\tau)$ via exploration-exploitation
  $a(\tau) = \begin{cases} 
  \text{Best files via } Q(s(t-1); \theta_r) \text{ w.p. } 1 - \epsilon_r \\
  \text{random } a \in A \text{ w.p. } \epsilon_r
  \end{cases}$
  for $t = 1, \ldots, T$ do
    for $n = 1, \ldots, N$ do
      Take action $a_n(t, \tau)$ using local policy
      $a_n(t, \tau) = \begin{cases} 
      \pi_n(s(t-1, \tau)) & \text{if } t \neq 1 \\
      \pi_n(s(T, \tau-1)) & \text{if } t = 1
      \end{cases}$
      Requests $r_n(t, \tau)$ are revealed
      Set $s_n(t, \tau) = r_n(t, \tau)$
      Incur $c_n(\cdot)$, cf. [1]
    end
  end
  Leaf nodes
  Set $\bar{s}_n(\tau) := (1/T) \sum_{t=1}^T s_n(t, \tau)$
  Send $\bar{s}_n(\tau) \odot (1 - \pi_n(\bar{s}_n))$ to parent node
  Send $\bar{c}(\cdot)$, cf. [4], to parent node
  Parent node
  Set $s(\tau) := \sum_{n=1}^N w_n \bar{s}_n(\tau) \odot (1 - \pi_n(\bar{s}_n))$\n  Find $c(s(\tau-1), a(\tau))$
  Save $(s(\tau-1), a(\tau), c(s(\tau-1), a(\tau), s(\tau))$ in $E$
  Uniformly sample $B$ experiences from $E$
  Find $\nabla^{\theta^\tau}L^\tau(\theta)$ for these samples, using [22]
  Update $\theta_{\tau + 1} = \theta_{\tau} - \beta_r \nabla^{\theta^\tau}L^\tau(\theta)$
  If $\text{mod}(\tau, C) = 0$, then update $\theta^\tau = \theta_r$
end

(18) is found by using the most recent experience, while the experience here is randomly drawn from past experiences in $\mathcal{E}$. As a result, the loss function becomes

$$ L^\tau(\theta) = \mathbb{E}[\|\delta_{\theta^\tau}(\theta; E)\|^2_2] $$

(22)

where the expectation is taken with respect to the uniformly sampled experience $\mathcal{E}$. In practice however, only a batch of $B$ samples is available and used to update $\theta_r$, so the expectation will be replaced with the sample mean. Finally, following a gradient descent step over the sampled experiences, we have

$$ \theta_{\tau + 1} = \theta_{\tau} - \beta_r \nabla L^\tau(\theta) \big|_{\theta = \theta_r}. $$

Both the experience replay and the target network help stabilize the DQN updates. Incorporating these remedies, Alg. 2 tabulates our deep reinforcement learning based adaptive caching scheme for the parent node.

VI. NUMERICAL TESTS

In this section, we present several numerical tests to assess the performance of our deep reinforcement learning based caching schemes that rely on Algs. 1 and 2.

The first experiment considers a simple setup, consisting of a parent node that directly serves user file requests. A total of $F = 50$ files were assumed in the cloud, each having the same size, while $M_0 = 5$ files can be stored in the cache, amounting to 10% of the total. The file popularities were drawn randomly from $[0, 1]$, invariant across all simulated time instants, but unknown to the caching entity. A fully connected feed-forward NN of 3 layers was implemented for DQN with each layer comprising 50 hidden neurons. For simplicity, we assume that the dataset $\mathcal{E}$ can store a number $R$ of most recent experiences, which is also known as the replay memory. It was set to $R = 10$ in our test, along with mini-batch size $B = 1$ to update the target network every $C = 10$ iterations.

Fig. 6 shows the convergence of the DQN parameter $\theta_r$ to that of the target network $\theta^\tau$. Since $\theta^\tau$ is updated with $\theta_r$ per $C$ iteration, the error $\|\theta_r - \theta^\tau\|^2_2$ vanishes periodically. While the error changes just in a few iterations after the $\theta^\tau$ update, it is constant in several iterations before the next $\theta^\tau$ update, suggesting that $\theta_r$ is fixed across those iterations. This motivates investigating the impact of $C$ on $\theta_r$’s convergence. Fig. 7 shows the convergence of $\theta_r$ to $\theta^\tau$ for $C = 20, 5, 3, 2$. Starting with $C = 20$, vector $\theta_r$ is not updated in most iterations between two consecutive updates of $\theta^\tau$. Clearly, the smaller $C$ is, the faster the convergence for $\theta_r$ is achieved. Indeed, this is a direct result of having static file popularity throughout the simulated intervals. As will be shown later, having small $C$ values is also preferable in dynamic settings.
Fig. 8: Temporal evolution of user file requests.

Fig. 9: Convergence of DQN to the target network in a dynamic popularity setting.

Fig. 10: Instantaneous reduced cost obtained by deep reinforcement learning compared with that of the optimal policy.

Fig. 11: Performance of deep reinforcement learning caching policy compared with the optimal noncausal policy.

Fig. 12: Impact of $C$ on DQN convergence in a dynamic setup.

For the second experiment, a similar setup having solely the parent node, plus $M_0 = 5$ and $F = 50$, was considered. Dynamic file popularities were generated with time evolutions modeled by Markov random processes, and again they are unknown to the caching node. An illustration of temporal evolution of such file popularities is depicted in Fig. 8. Here, each color represents the number of received requests for a particular file. Convergence of $\theta_\tau$ to $\theta_{Tar}$ with $C = 10$, is corroborated in Fig. 9. Compared with Figs. 6 and 7, the error here never vanishes, which implies that the DQN’s and the target network’s parameters are continuously changing. Intuitively, this is due to the dynamic evolution of file requests. As will be shown in the sequel, although $\theta_\tau$ never converges, it is powerful enough to find the proper files to store. In this regard, we define a non-causal optimal policy as benchmark, which unrealistically assumes knowledge of future requests and stores the most frequently requested files. In fact, this is the best policy that one may ideally implement. By drawing 100 samples randomly from all 2,000 time intervals, the actual cost reduction obtained by adopting this optimal policy is plotted with a red line in Fig. 10. Under the same setting, the cost reduction achieved by following our proposed
The deep reinforcement learning based caching approach is plotted in blue. This figure shows remarkable performance of our proposed approach relative to the non-causal optimal one. Furthermore, the empirical CDFs for reduced costs obtained over these 100 random samples, are presented in Fig. 11. This further verifies how our deep reinforcement learning based approach performs relative to the optimal policy. Finally, Fig. 12 shows convergence of the DQN parameters to the target network’s parameters when $C = 20, 10, 5, 2$. Again, the error never goes to zero. Further, similar to the static case (cf. Fig. 7), smaller $C$ values typically yield faster error decay.

To further examine performance of the proposed approach, consider now $N = 20$ leaf nodes are connected to the parent node. They receive user file requests during fast time slots, where there are $T = 2$ fast time slots within every slow timescale interval. File requests exhibit different Markovian evolution locally at different leaf nodes. There are a total of 2,000 time intervals, a total of $F = 50$ files, $M_0 = 5$ for the parent node, and the weights are all set to $w_n = 1$, $\forall n$.

Consider first the leaf nodes have no caching capacity. Fig. 13 compares the deep reinforcement learning based caching policy with baselines in terms of the empirical CDF of reduced cost. Here, the random policy, namely storing files randomly, and the worst case is a policy which stores the least popular files while assumed to have non-causal information about file requests. Corresponding instantaneous reduced costs, at the sampled intervals, are reported in Fig. 14. Both Figs. 13 and 14 clearly show the remarkable performance of the deep reinforcement learning based policy relative to the optimal one. In addition, compared with previous results, it becomes clear that, the more the leaf nodes, the better the performance of our proposed approach.

Finally, consider every leaf node has a caching capacity of storing $M_n = 5$ files, and it implements a local caching policy $\pi_n$. But these local caching policies are unknown to the parent node. To find the optimal caching decision, the parent node not only learns the file popularities and their temporal evolution, but also should learn its leaf nodes’ caching policies. Figs. 15 and 16 compares the deep reinforcement learning based policy with the benchmarks. Again, these results corroborate the remarkable performance of our proposed deep reinforcement learning caching based scheme. Compared with previous results, the slight degradation of deep reinforcement learning performance, is a consequence of considering local caching policies at the leaf nodes. Nevertheless, given the results in Fig. 16, the deep reinforcement learning based policy performs remarkably well compared with the optimal policy.

VII. CONCLUSIONS

Caching highly popular contents across distributed caching entities can substantially reduce the heavy burden of content delivery in modern data networks. This paper considered a network caching setup, comprising a parent node connected to several leaf nodes to serve end user file requests. Since the leaf nodes are closer to end users, they observe file requests in a fast timescale, and could thus serve requests, either through their locally stored files or by fetching files from the parent node. Observing aggregate requests from all leaf nodes, the parent node makes caching decisions in a slower-time scale. This two-time scale caching task was cast in this paper using a reinforcement learning approach. An efficient caching policy leveraging deep reinforcement learning was introduced, and shown capable of learning-and-adapting to the
dynamic evolution of file requests, and caching policies of leaf nodes. Simulated tests demonstrated impressive performance of the novel approach.

This work also opens up several directions for future research, including multi-armed bandit online learning [18], and distributed deep reinforcement learning using recurrent NNs [22] for future spatio-temporal file request prediction.

**REFERENCES**

[1] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, “Deep reinforcement learning: A brief survey,” IEEE Signal Process. Mag., vol. 34, no. 6, pp. 26–38, Nov. 2017.

[2] P. Auer, “Using confidence bounds for exploration-exploitation trade-offs,” J. Mach. Learn. Res., vol. 3, no. Nov, pp. 397–422, 2002.

[3] E. Bastug, M.ennis, and M. Debbah, “Living on the edge: The role of proactive caching in 5G wireless networks,” IEEE Commun. Mag., vol. 52, no. 8, pp. 82–89, Aug. 2014.

[4] Y. Bengio, A. Courville, and P. Vincent, “Representation learning: A review and new perspectives,” IEEE Trans. Pattern Anal. Mach. Intell., vol. 35, no. 8, pp. 1798–1828, Aug. 2013.

[5] P. Blasco and D. Gündüz, “Learning-based optimization of cache content in a small cell base station,” in Proc. of IEEE Int’l Conf. on Commun., Sydney, Australia, June 10-14, 2014, pp. 1897–1903.

[6] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang et al., “End to end learning for self-driving cars;” arXiv:1604.07316, 2016.

[7] S. Borst, V. Gupta, and A. Walid, “Distributed caching algorithms for content distribution networks,” in Int. Conf. Comput. Commun., San Diego, CA, USA, Mar. 15-19, 2010, pp. 1–9.

[8] L. Busoniu, R. Babuska, B. De Schutter, and D. Ernst, Reinforcement Learning and Dynamic Programming Using Function Approximators. Boca Raton, FL, USA: CRC press, 2010.

[9] E. Dahlman, S. Parkvall, and J. Skold, “Deep RL for content delivery based on blockchain: A game theoretic perspective,” IEEE J. Sel. Areas Commun., vol. 36, no. 6, pp. 1331–1344, June 2018.

[10] J. Dai, Z. Hu, B. Li, J. Liu, and B. Li, “Collaborative hierarchical caching with dynamic request routing for massive content distribution,” in Intl. Conf. Comput. Commun., Orlando, FL, USA, Mar. 25-30, 2012, pp. 2444–2452.

[11] C. Diuk, A. Cohen, and M. L. Littman, “An object-oriented representation for efficient reinforcement learning,” in Proc. of Intl. Conf. on Mach. Learn. ACM, 2008, pp. 240–247.

[12] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, Deep Learning. Cambridge, MA, USA: MIT press, 2016.

[13] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, “Deep reinforcement learning: A brief survey,” IEEE Signal Process. Mag., vol. 34, no. 6, pp. 26–38, Nov. 2017.

[14] Y. Hu, Y. Jiang, M.ennis, and F.-C. Zheng, “Distributed edge caching in ultra-dense fog radio access networks: A mean field approach,” arXiv:1806.09076, 2018.