Deep Reinforcement Learning for Adaptive Caching in Hierarchical Content Delivery Networks

Alireza Sadeghi, *Student Member, IEEE*, 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 (RL) framework is put forth. To handle the large continuous state space, a scalable deep RL approach is pursued. The novel approach relies on a hyper-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 RL, 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 [1], [2]. Caching popular content is envisioned to achieve major savings in terms of energy, bandwidth, and cost, in addition to user satisfaction [1].

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 goal, contemporary machine learning and artificial intelligence tools hold the promise to empower next-generation networks with 'smart' caching control units, 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 (DNNs) have lately boosted the notion of "learning from data" with field-changing performance improvements reported in diverse artificial intelligence tasks [3]. DNNs can cope with the 'curse of dimensionality' by providing compact low-dimensional representations of high-dimensional data [4]. Combining deep learning with RL, deep (D) RL has created the first artificial agents to achieve human-level performance across many challenging domains [5], [6]. As another example, a DNN system was built to operate Google's data centers, and shown able to consistently achieve a 40% reduction in energy consumption for cooling [7]. This system provides a general-purpose framework to understand complex dynamics, which has also been applied to address other challenges including, e.g., dynamic spectrum access [8], multiple access and handover control [9], [10], as well as resource allocation in fog-radio access networks [11], [12] or software-defined networks [13], [14].

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 but time-invariant content popularities, multi-armed bandit online learning was pursued in [15]. Yet, these methods are generally not amenable to online implementation. To serve delay-sensitive requests, a learning approach was developed in [16] using a pre-trained DNN to handle a non-convex problem reformulation.

In realistic networks however, popularities exhibit dynamics, which motivate well the so-termed *dynamic* caching. A

Manuscript received February 26, 2019; revised June 30, 2019; accepted August 10, 2019. This work was supported in part by NSF grants 1711471, 1514056, and 1901134. The associate editor coordinating the review of this article and approving it for publication was H. T. Dinh. *(Corresponding author: Gang Wang.)*

The authors are with the Digital Technology Center, University of Minnesota, Minneapolis, MN 55455 USA, and also with the Department of Electrical and Computer Engineering, University of Minnesota, Minneapolis, MN 55455 USA (e-mail: sadeghi@umn.edu; gangwang@umn.edu; georgios@umn.edu).

Digital Object Identifier 10.1109/TCCN.2019.2936193

Poisson shot noise model was adopted to approximate the evolution of popularities in [17], for which an age-based caching solution was developed in [18]. RL based methods have been pursued in [19], [20], [21], [22]. Specifically, a Q-learning based caching scheme was developed in [19] to model global and local content popularities as Markovian processes. Considering Poisson shot noise popularity dynamics, a policy gradient RL based caching scheme was devised in [20]. Assuming stationary file popularities and service costs, a dual-decomposition based Q-learning approach was pursued in [21]. Albeit reasonable for discrete states, these approaches cannot deal with large continuous state-action spaces. To cope with such spaces, DRL approaches have been considered for content caching in, e.g., [6], [22], [23], [24], [25], [26]. Encompassing finite-state time-varying Markov channels, a deep Q-network approach was devised in [22]. An actor-critic method with deep deterministic policy gradient updates was used in [23]. Boosted network performance using DRL was documented in several other applications, such as connected vehicular networks [24], and smart cities [25].

The aforementioned works focus on devising caching policies for a *single* caching entity. A more common setting in next-generation networks however, involves a network of interconnected caching nodes. It has been shown that considering a network of connected caches jointly can further improve performance [27], [28]. For instance, leveraging network topology and the broadcast nature of links, the coded caching strategy in [27] further reduces data traffic over a network. This idea has been extended in [29] to an online setting, where popularities are modeled Markov processes. Collaborative and distributed online learning approaches have been pursued [28], [30], [31]. Indeed, today's content delivery networks such as Akamai [32], have tree network structures. Accounting for the hierarchy of caches has become a common practice in recent works; see also [33], [34], [35]. Joint routing and in-network content caching in a hierarchical cache network was formulated in [33], for which greedy schemes with provable performance guarantees can be found in [34].

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

- **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.
- **c2)** *Complex Dynamics:* Content popularities are random and exhibit unknown space-time, heterogeneous, and often non-stationary dynamics over the entire network.
- **c3)** *Large Continuous State Space:* Due to the shear size of available content, caching nodes, and possible realizations of content requests, the decision space is huge.

B. This Work

Prompted by the recent interest in hierarchical caching, this paper focuses on a two-level network caching, where a parent node is connected to multiple leaf nodes to serve end-user file requests. Such a two-level network constitutes the building block of the popular tree hierarchical cache networks in, e.g., [32]. To model the interaction between caching decisions of parent and leaf nodes along with the space-time evolution of file requests, a scalable DRL approach based on hyper deep Q-networks (DQNs) is developed. As corroborated by extensive numerical tests, the novel caching policy for the parent node can adapt itself to local policies of leaf nodes and space-time evolution of file requests. Moreover, our approach is simple-to-implement, and performs close to the optimal policy.

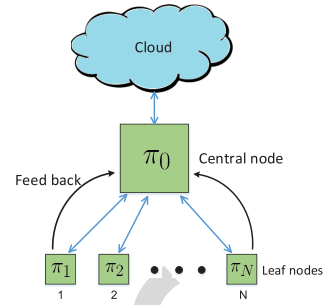

Fig. 1. A network of caching nodes.

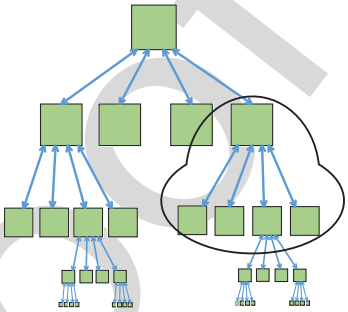

Fig. 2. A hierarchical tree network cache system.

II. 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) back-haul 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 [36, 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 collected in the set $\mathcal{F} = \{1, \ldots, F\}$. At the beginning of each slot t, every leaf node n selects a subset of files in $\mathcal{F}$ 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 π_n, to take (cache or no-cache) action $\boldsymbol{a}_n(t+1,\tau) = \pi_n(\boldsymbol{s}_n(t,\tau))$ at the beginning of slot $t+1$, based on its *state* vector $\boldsymbol{s}_n$ 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 $\boldsymbol{s}_n(t,\tau) := \boldsymbol{r}_n(t,\tau) := [r_n^1(t,\tau) \cdots r_n^F(t,\tau)]^\top$ to collect the number of requests received at leaf node n for individual files over the duration of slot t on interval τ. Likewise, to serve file requests that have not been served by leaf nodes, the parent node takes action $\boldsymbol{a}_0(\tau)$ to store files at the beginning of every interval τ, according to a certain policy π_0. Again, as aggregation smooths 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 τ

$$\begin{aligned}&\boldsymbol{c}_n(\pi_n(\boldsymbol{s}_n(t,\tau)), \boldsymbol{r}_n(t+1,\tau), \boldsymbol{a}_0(\tau))\\ &:= \boldsymbol{r}_n(t+1,\tau) \odot (\mathbf{1} - \boldsymbol{a}_0(\tau)) \odot (\mathbf{1} - \boldsymbol{a}_n(t+1,\tau))\\ &\quad + \boldsymbol{r}_n(t+1,\tau) \odot (\mathbf{1} - \boldsymbol{a}_n(t+1,\tau)) \qquad (1)\end{aligned}$$

where $\boldsymbol{c}_n(\cdot) := [c_n^1(\cdot) \ \cdots \ c_n^F(\cdot)]^\top$ concatenates the cost for serving individual files per node n; symbol $\odot$ denotes entry-wise vector multiplication; entries of $\boldsymbol{a}_0$ and $\boldsymbol{a}_n$ are either 1 (cache, hence no need to fetch), or, 0 (no-cache, hence fetch); and $\mathbf{1}$ stands for the all-one vector. Specifically, the second summand in (1) captures the cost of the leaf node fetching files for end users, while the first summand corresponds to that of the parent fetching files from the cloud.

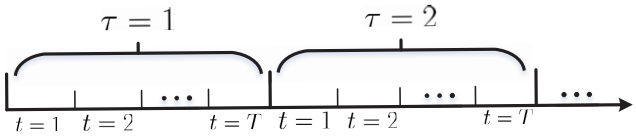

Fig. 3. Slow and fast time slots.

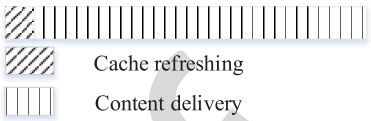

Fig. 4. Structure of slots and intervals.

We model user file requests as Markov processes with unknown transition probabilities [19]. Per interval τ, a reasonable caching scheme for leaf node $n \in \mathcal{N}$ could entail minimizing the expected cumulative cost; that is,

$$\pi_{n,\tau}^* := \arg\min_{\pi_n \in \Pi_n} \mathbb{E}\left[\sum_{t=1}^{T} \mathbf{1}^\top \boldsymbol{c}_n(\pi_n(\boldsymbol{s}_n(t,\tau)), \boldsymbol{r}_n(t+1,\tau), \boldsymbol{a}_0(\tau))\right] \qquad (2)$$

where Π_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., [15], [19], [20], and references therein. In particular, a RL based approach using tabular Q-learning was pursued in our precursor [19], which can be employed here to tackle this fast timescale optimization. The remainder of this paper will thus be on designing the caching policy π_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 τ, each leaf node $n \in \mathcal{N}$ first obtains the time-average state vector $\bar{\boldsymbol{s}}_n(\tau) := (1/T)\sum_{t=1}^{T} \boldsymbol{s}_n(t,\tau)$, and subsequently forms and forwards the per-node vector $\bar{\boldsymbol{s}}_n(\tau) \odot (\mathbf{1} - \pi_n(\bar{\boldsymbol{s}}_n(\tau)))$ to its parent node. This vector has nonzero entries the average number of file requests received during interval τ, and zero entries if π_n stores the corresponding files. Using the latter, the parent node forms its 'weighted' state vector as

$$\boldsymbol{s}_0(\tau) := \sum_{n=1}^{N} w_n \bar{\boldsymbol{s}}_n(\tau) \odot (\mathbf{1} - \pi_n(\bar{\boldsymbol{s}}_n(\tau))) \qquad (3)$$

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

$$\begin{aligned}&\bar{\boldsymbol{c}}_n\Big(\pi_0(\boldsymbol{s}_0(\tau)); \{\boldsymbol{a}_n(t,\tau+1), \boldsymbol{r}_n(t,\tau+1)\}_{t=1}^{T}\Big)\\ &= \frac{1}{T}\sum_{t=1}^{T} \boldsymbol{c}_n(\boldsymbol{a}_n(t,\tau+1), \boldsymbol{r}_n(t,\tau+1), \boldsymbol{a}_0(\tau+1)) \qquad (4)\end{aligned}$$

from all leaf nodes, the parent node determines its cost

$$\boldsymbol{c}_0(\boldsymbol{s}_0(\tau), \pi_0(\boldsymbol{s}_0(\tau))) = \sum_{n=1}^{N} w_n \bar{\boldsymbol{c}}_n \left(\boldsymbol{\pi}_0(\boldsymbol{s}_0(\tau)); \{\boldsymbol{a}_n(t, \tau+1), \boldsymbol{r}_n(t, \tau+1)\}_{t=1}^{T}\right). \tag{5}$$

Having observed $\{\boldsymbol{s}_0(\tau'), \boldsymbol{a}_0(\tau'), \boldsymbol{c}_0(\boldsymbol{s}_0(\tau'-1), \boldsymbol{a}_0(\tau')\}_{\tau'=1}^{\tau}$, the objective is to take a well-thought-out action $\boldsymbol{a}_0(\tau+1)$ for next interval. This is tantamount to finding an optimal policy function π_0^* to take a caching action $\boldsymbol{a}_0(\tau+1) = \pi_0^*(\boldsymbol{s}_0(\tau))$.

Since the requests $\{\boldsymbol{r}_n(t, \tau)\}_{n,t}$ are Markovian and present actions $\{\boldsymbol{a}_n(t, \tau)\}_{n,t}$ 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

$$\pi_0^* := \underset{\pi_0 \in \Pi_0}{\arg\min}\, \mathbb{E}\left[\sum_{\tau=1}^{\infty} \gamma^{\tau-1} \mathbf{1}^\top \boldsymbol{c}_0(\boldsymbol{s}_0(\tau), \pi_0(\boldsymbol{s}_0(\tau)))\right] \tag{6}$$

where Π_0 represents the set of all feasible policies; expectation is over $\{\boldsymbol{r}_n(t, \tau)\}_{n,t}$, as well as (possibly random) parent $\boldsymbol{a}_0(\tau)$ and leaf actions $\{\boldsymbol{a}_n(t, \tau)\}_{n,t}$; 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 $\boldsymbol{s}_0(\tau)$, namely $\boldsymbol{a}_0(\tau+1) = \pi(\boldsymbol{s}_0(\tau))$, influences the next state $\boldsymbol{s}_0(\tau+1)$ through $\pi_{n,\tau}(\cdot)$ in (3), as well as the cost $\boldsymbol{c}_0(\cdot)$ 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 [37]. To cope with this complexity of solving (6), an adaptive RL approach is pursued next.

IV. Adaptive RL-Based Caching

RL 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 states) to draw actions from, in order to minimize the average cumulative cost as in (6) [38]. To proceed, we start by giving some basics on Markov decision processes (MDPs) [38, p. 310].

A. MDP Formulation

MDPs provide a well-appreciated model for decision making tasks. It is represented by a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{C}, \mathcal{P} \rangle$, where $\mathcal{S}$ denotes a set of possible states, $\mathcal{A}$ a set of feasible actions, $\mathcal{C}$ a set of costs, and $\mathcal{P}$ the set of state transition probabilities. In our problem of interest, at the end of each interval τ, the parent node is in state $\boldsymbol{s}_0(\tau) \in \mathcal{S}$ and takes a caching decision $\boldsymbol{a}_0(\tau+1) \in \mathcal{A}$ for the next interval $\tau+1$, according to its local caching policy $\boldsymbol{a}_0(\tau+1) = \pi_0(\boldsymbol{s}_0(\tau))$. Upon proceeding to interval $\tau+1$, every leaf node *n* serves its locally connected users. Let vector $\boldsymbol{r}_n(t, \tau+1)$ collect the number of received requests at node *n* during slot *t* across files. We model the temporal evolution of this vector with Markov dynamics as $\boldsymbol{r}_n(t+1, \tau+1) = \lfloor \boldsymbol{r}_n(t, \tau+1) + \boldsymbol{\Delta}_n(t, \tau+1) \rfloor^+$, where $\boldsymbol{\Delta}_n(t, \tau+1)$ is a multivariate Gaussian random noise; $\lfloor \cdot \rfloor$ and $(\cdot)^+$ denote the entry-wise floor and $\max\{\cdot, 0\}$ operators. In this context, the incurred cost $\boldsymbol{c}_0(\boldsymbol{s}_0(\tau), \pi_0(\boldsymbol{s}_0(\tau))) \in \mathcal{C}$ (see (5)), is a random vector with an unknown distribution. As requests $\{\boldsymbol{r}_n\}$ are random, caching decisions $\{\boldsymbol{a}_n\}$ are also random. In addition, decision $\boldsymbol{a}_0$ of the parent node influences those of leaf nodes via (2), as well as the next state of the parent node in (3). That is, the current decision of parent node probabilistically influences its next state. To formalize this, we use $P_{\boldsymbol{s}\boldsymbol{s}'}^{\boldsymbol{a}} \in \mathcal{P}$ to denote the *unknown* state transition probability from state $\boldsymbol{s}$ to $\boldsymbol{s}'$ upon taking action $\boldsymbol{a}$

$$P_{\boldsymbol{s}\boldsymbol{s}'}^{\boldsymbol{a}} = \Pr\{\boldsymbol{s}(\tau+1) = \boldsymbol{s}' \mid \boldsymbol{s}(\tau) = \boldsymbol{s},\, \boldsymbol{a} = \pi(\boldsymbol{s})\} \tag{7}$$

where the subscript 0 referring to the parent node is dropped for brevity. As $\boldsymbol{a}_0$ probabilistically influences the next state as well as the incurred cost, our problem is indeed an MDP. The rest of this paper targets finding an optimal policy solving (6).

B. RL Based Caching

Towards developing an RL solver for (6), define the so-termed *value function* to indicate the quality of policy π_0, starting from initial state $\boldsymbol{s}_0(0)$ as

$$V_{\pi_0}(\boldsymbol{s}_0(0)) := \mathbb{E}\left[\sum_{\tau=1}^{\infty} \gamma^{\tau-1} \mathbf{1}^\top \boldsymbol{c}_0(\boldsymbol{s}_0(\tau), \pi_0(\boldsymbol{s}_0(\tau)))\right] \tag{8}$$

which represents the average cost of following policy π_0 to make caching decisions starting from $\boldsymbol{s}_0(0)$. Upon finding $V_{\pi_0}(\cdot)$ for all $\pi_0 \in \Pi_0$, one can readily obtain π_0^* that minimizes $V_{\pi_0}(\boldsymbol{s}_0(0))$ over all possible initial states $\boldsymbol{s}_0(0)$.

For brevity, the time index and the subscript 0 referring to the parent node will be dropped whenever it is clear from the context. To find $V_\pi(\cdot)$, one can rely on the Bellman equation, which basically relates the value of a policy at one state to values of the remaining states [38, p. 46]. Leveraging (8) and (7), the Bellman equation for value function $V_\pi(\cdot)$ is given by

$$V_\pi(\boldsymbol{s}) = \mathbb{E}\left[\mathbf{1}^\top \boldsymbol{c}(\boldsymbol{s}, \pi(\boldsymbol{s})) + \gamma \sum_{\boldsymbol{s}'} P_{\boldsymbol{s}\boldsymbol{s}'}^{\pi(\boldsymbol{s})} V_\pi(\boldsymbol{s}')\right] \tag{9}$$

where the average immediate cost can be found as

$$\mathbb{E}\left[\mathbf{1}^\top \boldsymbol{c}(\boldsymbol{s}, \pi(\boldsymbol{s}))\right] = \sum_{\boldsymbol{s}'} P_{\boldsymbol{s}\boldsymbol{s}'}^{\pi(\boldsymbol{s})} \mathbf{1}^\top \boldsymbol{c}\big(\boldsymbol{s}, \pi(\boldsymbol{s}) | \boldsymbol{s}'\big).$$

If $P_{\boldsymbol{s}\boldsymbol{s}'}^{\boldsymbol{a}}$ were known $\forall \boldsymbol{a} \in \mathcal{A}$, $\forall \boldsymbol{s}, \boldsymbol{s}' \in \mathcal{S}$, 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 π^* 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 [38, p. 64] outlined next. Define first the state-action value function, also called *Q*-function for policy π

$$Q_\pi(\boldsymbol{s}, \boldsymbol{a}) := \mathbb{E}\Big[\mathbf{1}^\top \boldsymbol{c}(\boldsymbol{s}, \boldsymbol{a})\Big] + \gamma \sum_{\boldsymbol{s}'} P_{\boldsymbol{s}\boldsymbol{s}'}^{\boldsymbol{a}} V_\pi\big(\boldsymbol{s}'\big). \tag{10}$$

This function captures the expected immediate cost of starting from state $\boldsymbol{s}$, taking the first action to be $\boldsymbol{a}$, and subsequently following policy π to take future actions onwards. The only difference between the value function in (8) and that

of Q-function in (10) is that the former takes the first action according to policy π, while the latter starts with $\boldsymbol{a}$ as the first action, which may not necessarily be taken when adhering to π. Having defined the Q-function, we are ready to present the policy iteration algorithm, in which every iteration i entails the following two steps:

Policy Evaluation: Find $V_{\pi^i}(\cdot)$ for the current policy π^i by solving the system of linear equations in (9).

Policy Improvement: Update the current policy greedily as

$$\pi^{i+1}(\boldsymbol{s}) = \arg\min_{\boldsymbol{\alpha}\in\mathcal{A}} \; Q_{\pi^i}(\boldsymbol{s},\boldsymbol{\alpha}).$$

To perform policy evaluation, we rely on knowledge of $P_{\boldsymbol{ss'}}^{\pi^i(\boldsymbol{s})}$. 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 π^*, without knowing $P_{\boldsymbol{ss'}}^{\boldsymbol{a}}$, $\forall \boldsymbol{a}, \boldsymbol{s}, \boldsymbol{s'}$. One such approach is Q-learning [38, 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 DQN.

C. Q-Learning Based Adaptive Caching

The Q-learning algorithm finds the optimal policy π^* by estimating $Q_{\pi^*}(\cdot,\cdot)$ 'on-the-fly.' It turns out that π^* is the greedy policy over the optimal Q-function [38, p. 64], that is

$$\pi^*(\boldsymbol{s}) = \arg\min_{\boldsymbol{\alpha}\in\mathcal{A}} \; Q_{\pi^*}(\boldsymbol{s},\boldsymbol{\alpha}), \quad \forall \boldsymbol{s}\in\mathcal{S} \tag{11}$$

where Q_{π^*} is estimated using Bellman's equation for the Q-function. This is possible because the V-function is linked with the Q-function under π^* through (see [38, p. 51] for details)

$$V_{\pi^*}(\boldsymbol{s}) = \min_{\boldsymbol{\alpha}\in\mathcal{A}} \; Q_{\pi^*}(\boldsymbol{s},\boldsymbol{\alpha}), \quad \forall \boldsymbol{s}. \tag{12}$$

Substituting (12) into (10), Bellman's equation for the Q-function under π^* is expressed as

$$Q_{\pi^*}(\boldsymbol{s},\boldsymbol{a}) = \mathbb{E}\Big[\mathbf{1}^\top \boldsymbol{c}(\boldsymbol{s},\boldsymbol{a})\Big] + \gamma \sum_{\boldsymbol{s'}} P_{\boldsymbol{ss'}}^{\boldsymbol{a}} \min_{\boldsymbol{\alpha}} \; Q_{\pi^*}(\boldsymbol{s'},\boldsymbol{\alpha}) \tag{13}$$

which plays a key role in many RL algorithms. Examples include Q-learning [39], and SARSA [38], where one relies on (13) 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 $\boldsymbol{a}$ in a given state $\boldsymbol{s}$. Specifically, it chooses the action minimizing the current estimate of $Q_{\pi^*}(\cdot,\cdot)$ denoted by $\hat{Q}_\tau(\cdot,\cdot)$, with probability (w.p.) $1-\epsilon_\tau$, or, it takes a random action $\boldsymbol{a}\in\mathcal{A}$ otherwise; that is,

$$\boldsymbol{a} = \begin{cases} \arg\min_{\boldsymbol{\alpha}\in\mathcal{A}} \; \hat{Q}_\tau(\boldsymbol{s},\boldsymbol{\alpha}), & \text{w.p. } 1-\epsilon_\tau \\ \text{random} \;\; \boldsymbol{a}\in\mathcal{A}, & \text{w.p.}\epsilon_\tau. \end{cases}$$

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

$$\mathcal{L}(\boldsymbol{s},\boldsymbol{a}) = \frac{1}{2}\left(\mathbf{1}^\top \boldsymbol{c}(\boldsymbol{s},\boldsymbol{a}) + \gamma \min_{\boldsymbol{\alpha}\in\mathcal{A}} \; \hat{Q}_\tau(\boldsymbol{s'},\boldsymbol{\alpha}) - \hat{Q}_\tau(\boldsymbol{s},\boldsymbol{a})\right)^2. \tag{14}$$

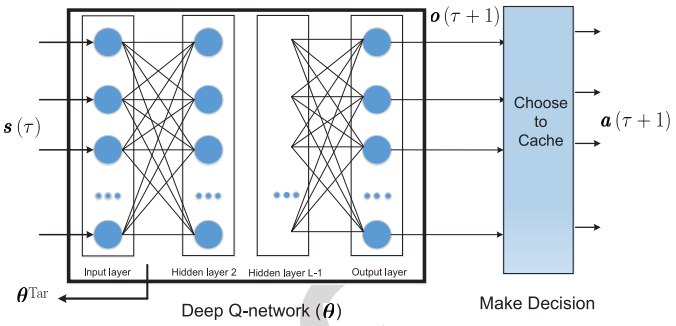

Fig. 5. Deep Q-network.

The estimated Q-function for a *single* state-action pair is subsequently updated, by following a gradient descent step to minimize the loss in (14), which yields the update

$$\hat{Q}_{\tau+1}(\boldsymbol{s},\boldsymbol{a}) = \hat{Q}_\tau(\boldsymbol{s},\boldsymbol{a}) - \beta \frac{\partial \mathcal{L}(\boldsymbol{s},\boldsymbol{a})}{\partial \hat{Q}_\tau(\boldsymbol{s},\boldsymbol{a})} \tag{15}$$

where $\beta > 0$ is some step size. Upon evaluating the gradient and merging terms, the update in (15) boils down to

$$\hat{Q}_{\tau+1}(\boldsymbol{s},\boldsymbol{a}) = (1-\beta)\hat{Q}_\tau(\boldsymbol{s},\boldsymbol{a}) + \beta\left[\mathbf{1}^\top \boldsymbol{c}(\boldsymbol{s},\boldsymbol{a}) + \gamma \min_{\boldsymbol{\alpha}\in\mathcal{A}} \; \hat{Q}_\tau(\boldsymbol{s'},\boldsymbol{\alpha})\right].$$

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, [40], the multi-armed bandit scheme [15], and even RL ones [19], [20].

Remark 2: The exploration-exploitation step for taking actions guarantees continuously visiting state-action pairs, and ensures convergence to the optimal Q-function [37]. Instead of the ϵ-greedy exploration-exploitation step, one can employ the upper confidence bound scheme [41]. Technically, any exploration-exploitation scheme should be greedy in the limit of infinite exploration (GLIE) [42, p. 840]. An example obeying GLIE is the ϵ-greedy algorithm [42, p. 840] with $\epsilon_\tau = 1/\tau$. It converges to an optimal policy, albeit at a very slow rate. On the other hand, using a constant $\epsilon_\tau = \epsilon$ approaches the optimal $Q^*(\cdot,\cdot)$ faster, but its exact convergence is not guaranteed as it is not GLIE.

Clearly, finding $Q_{\pi^*}(\boldsymbol{s},\boldsymbol{a})$ 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 [38]. 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 RL design good hand-crafted features that can properly approximate $V_{\pi^*}(\cdot)$, $Q_{\pi^*}(\cdot,\cdot)$, or, $\pi^*(\cdot)$ [5]. Their applicability has only been limited to domains, where such features can be discovered, or, to state spaces that are low dimensional [38].

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. This is because DNNs are capable of extracting compact low-dimensional features from high-dimensional data. Wedding DNNs with RL results in 'deep RL' 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. The most remarkable success in playing AI games adopted DNNs to estimate the Q-function, what is termed DQN in [5].

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 π^* for the parent node, the success of function approximators for RL (e.g., [5]), motivated us to pursue a parametric approach to estimating $Q(\boldsymbol{s}, \boldsymbol{a})$ with a DNN (see (11)). This DNN has as input pairs of vectors $(\boldsymbol{s}, \boldsymbol{a})$, and as scalar output the corresponding estimated $Q(\boldsymbol{s}, \boldsymbol{a})$ values. Clearly, the joint state-action space of the sought Q-function has cardinality $|\mathcal{A}\times\mathcal{S}| = |\mathcal{A}||\mathcal{S}|$. To reduce the search space, we shall take a parametric DQN approach [5] 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 $\boldsymbol{s}(\tau)$ as in (3), and $F \times 1$ output cost vector $\boldsymbol{o}(\tau+1)$ [5]. Note that input does not include the action vector $\boldsymbol{a}$. 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., [43]. Neurons of the L-th output layer use a softmax nonlinearity[1] to yield for the f-th entry of $\boldsymbol{o}$, 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 ($< F$) files at the parent node, the M largest entries of the DQN output $\boldsymbol{o}(\tau+1)$ are chosen by the decision module in Fig. 5 to obtain the action vector $\boldsymbol{a}(\tau+1)$. We can think of our DQN as a 'soft policy' function estimator, and $\boldsymbol{o}(\tau+1)$ as a 'predicted cost' or a 'soft action' vector for interval $\tau+1$, whose 'hard thresholding' yields $\boldsymbol{a}(\tau+1)$. It will turn out that excluding $\boldsymbol{a}$ 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 $\boldsymbol{\theta}$ that parameterizes the input-output relationship $\boldsymbol{o}(\tau+1) = Q(\boldsymbol{s}(\tau); \boldsymbol{\theta}_\tau)$. To recursively update $\boldsymbol{\theta}_\tau$ to $\boldsymbol{\theta}_{\tau+1}$, consider two successive intervals along with corresponding states $\boldsymbol{s}(\tau)$ and $\boldsymbol{s}(\tau+1)$; the action $\boldsymbol{a}(\tau+1)$ taken at the beginning of interval $\tau+1$; and, the cost $\boldsymbol{c}(\tau+1)$ revealed at the end of interval $\tau+1$. The instantaneous approximation of the optimal cost-to-go from interval $\tau+1$ is given by $\boldsymbol{c}(\tau+1) + \gamma Q(\boldsymbol{s}(\tau+1); \boldsymbol{\theta}_\tau)$, where $\boldsymbol{c}(\tau+1)$ is the immediate cost, and $Q(\boldsymbol{s}(\tau+1); \boldsymbol{\theta}_\tau)$ represents the predicted cost-to-go from interval $\tau+2$ that is provided by our DQN with $\boldsymbol{\theta}_\tau$, and discounted by γ. Since our DQN offers $Q(\boldsymbol{s}(\tau); \boldsymbol{\theta}_\tau)$ as the predicted cost for interval $\tau+1$, the prediction error of this cost as a function of $\boldsymbol{\theta}_\tau$ is given by

$$\boldsymbol{\delta}(\boldsymbol{\theta}_\tau) := \left[\overbrace{\boldsymbol{c}(\tau+1) + \gamma Q(\boldsymbol{s}(\tau+1); \boldsymbol{\theta}_\tau)}^{\text{target cost-to-go from interval } \tau+1} - Q(\boldsymbol{s}(\tau); \boldsymbol{\theta}_\tau)\right] \odot (\mathbf{1} - \boldsymbol{a}(\tau+1)) \tag{16}$$

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

Using the so-termed experience $\boldsymbol{E}_{\tau+1} := [\boldsymbol{s}(\tau), \boldsymbol{a}(\tau+1), \boldsymbol{c}(\tau+1), \boldsymbol{s}(\tau+1)]$, and the ℓ_2-norm of $\boldsymbol{\delta}(\boldsymbol{\theta}_\tau)$ as criterion

$$\mathcal{L}(\boldsymbol{\theta}_\tau) = \|\boldsymbol{\delta}(\boldsymbol{\theta}_\tau)\|_2^2 \tag{17}$$

the sought parameter update minimizing (17) is given by the stochastic gradient descent (SGD) iteration as

$$\boldsymbol{\theta}_{\tau+1} = \boldsymbol{\theta}_\tau - \beta_\tau \nabla \mathcal{L}(\boldsymbol{\theta})|_{\boldsymbol{\theta}=\boldsymbol{\theta}_\tau} \tag{18}$$

where $\beta_\tau > 0$ denotes the learning rate.

Since the dimensionality of $\boldsymbol{\theta}$ can be much smaller than $|\mathcal{S}||\mathcal{A}|$, 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 (16) that can cause instability in (18), which can lead to performance degradation, and even divergence [44]. Moreover, (18) leverages solely a single most recent experience $\boldsymbol{E}_{\tau+1}$. These limitations will be mitigated as elaborated next.

A. Target Network and Experience Replay

NN function approximation, along with the loss (17) and the update (18), often result in unstable RL algorithms [5]. This is due to: i) correlated experiences used to update the DQN parameters $\boldsymbol{\theta}$; 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 $\boldsymbol{E}_\tau$ in $\mathcal{E} := \{\boldsymbol{E}_1, \ldots, \boldsymbol{E}_\tau\}$, and utilizes a batch of B uniformly sampled experiences from this data set, namely $\{\boldsymbol{E}_{i_\tau}\}_{i=1}^B \sim U(\mathcal{E})$. By sampling and replaying previously observed experiences, experience replay can overcome the two challenges. On the other hand, to obtain decorrelated target values in (16), a second NN (called target network) with structure identical to the DQN is invoked with parameter vector $\boldsymbol{\theta}^{\mathrm{Tar}}$. Interestingly, $\boldsymbol{\theta}^{\mathrm{Tar}}$ can be periodically replaced with $\boldsymbol{\theta}_\tau$ every C training iterations of the DQN, which enables the target network to smooth out fluctuations in updating the DQN [5].

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

$$\boldsymbol{\delta}^{\mathrm{Tar}}(\boldsymbol{\theta}; \boldsymbol{E}_{i_\tau}) := \left[\boldsymbol{c}(i_\tau+1) + \gamma Q\big(\boldsymbol{s}(i_\tau+1); \boldsymbol{\theta}^{\mathrm{Tar}}\big) - Q(\boldsymbol{s}(i_\tau); \boldsymbol{\theta})\right] \odot (\mathbf{1} - \boldsymbol{a}(i_\tau+1)). \tag{19}$$

Different from (16), the target values here are found through the target network with weights $\boldsymbol{\theta}^{\mathrm{Tar}}$. In addition, the error in (16) is found by using the most recent experience, while

[1] Softmax is a function that takes as input a vector $\boldsymbol{z} \in \mathbb{R}^F$, and normalizes it into a probability distribution via $\sigma(\boldsymbol{z})_f = \mathrm{e}^{z_f} / (\sum_{f=1}^F \mathrm{e}^{z_f})$, $\forall f$.

Algorithm 1: Deep RL for Adaptive Caching

Initialize: $\boldsymbol{s}(0)$, $\boldsymbol{s}_n(t,\tau), \forall n$, $\boldsymbol{\theta}_\tau$, and $\boldsymbol{\theta}^{\text{Tar}}$
for $\tau = 1, 2, \ldots$ **do**
  Take action $\boldsymbol{a}(\tau)$ via exploration-exploitation
  $\boldsymbol{a}(\tau) = \begin{cases} \text{Best files via } Q(\boldsymbol{s}(\tau-1);\boldsymbol{\theta}_\tau) & \text{w.p. } 1-\epsilon_\tau \\ \text{random } \boldsymbol{a} \in \mathcal{A} & \text{w.p. } \epsilon_\tau \end{cases}$
  for $t = 1, \ldots, T$ **do**
    for $n = 1, \ldots, N$ **do**
      Take action $\boldsymbol{a}_n$ using local policy
      $\boldsymbol{a}_n(t,\tau) = \begin{cases} \pi_n(\boldsymbol{s}(t-1,\tau)) \text{ if } t \neq 1 \\ \pi_n(\boldsymbol{s}(T,\tau-1)) \text{ if } t = 1 \end{cases}$
      Requests $\boldsymbol{r}_n(t,\tau)$ are revealed
      Set $\boldsymbol{s}_n(t,\tau) = \boldsymbol{r}_n(t,\tau)$
      Incur $\boldsymbol{c}_n(\cdot)$, see (1)
    end
  end
  Leaf nodes
  Set $\bar{\boldsymbol{s}}_n(\tau) := (1/T)\sum_{t=1}^{T} \boldsymbol{s}_n(t,\tau)$
  Send $\bar{\boldsymbol{s}}_n(\tau) \odot (\mathbf{1} - \pi_n(\bar{\boldsymbol{s}}_n))$ to parent node
  Send $\bar{\boldsymbol{c}}_n(\cdot)$ see (4), to parent node
  Parent node
  Set $\boldsymbol{s}(\tau) := \sum_{n=1}^{N} w_n \bar{\boldsymbol{s}}_n(\tau) \odot (\mathbf{1} - \pi_n(\bar{\boldsymbol{s}}_n))$
  Find $\boldsymbol{c}(s(\tau-1), \boldsymbol{a}(\tau))$
  Save $(\boldsymbol{s}(\tau-1), \boldsymbol{a}(\tau), \boldsymbol{c}(s(\tau-1), \boldsymbol{a}(\tau)), \boldsymbol{s}(\tau))$ in $\mathcal{E}$
  Uniformly sample B experiences from $\mathcal{E}$
  Find $\nabla\mathcal{L}^{\text{Tar}}(\boldsymbol{\theta})$ for these samples, using (20)
  Update $\boldsymbol{\theta}_{\tau+1} = \boldsymbol{\theta}_\tau - \beta_\tau \nabla\mathcal{L}^{\text{Tar}}(\boldsymbol{\theta})$
  If $\text{mod}(\tau, \text{C}) = 0$, then update $\boldsymbol{\theta}^{\text{Tar}} = \boldsymbol{\theta}_\tau$
end

the experience here is randomly drawn from past experiences in $\mathcal{E}$. As a result, the loss function becomes

$$\mathcal{L}^{\text{Tar}}(\boldsymbol{\theta}) = \mathbb{E}\left\|\boldsymbol{\delta}^{\text{Tar}}(\boldsymbol{\theta}; \boldsymbol{E})\right\|_2^2 \tag{20}$$

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

$$\boldsymbol{\theta}_{\tau+1} = \boldsymbol{\theta}_\tau - \beta_\tau \nabla\mathcal{L}^{\text{Tar}}(\boldsymbol{\theta})|_{\boldsymbol{\theta}=\boldsymbol{\theta}_\tau}.$$

Both the experience replay and the target network help stabilize the DQN updates. Incorporating these remedies, Alg. 1 tabulates our deep RL 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 RL based caching schemes which is represented in Alg. 1.

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 $\mathcal{R}$ of most recent experiences, which is also known as the replay memory. It was set to $\mathcal{R} = 10$ in our test, along with mini-batch size $B = 1$ to update the target network every $C = 10$ iterations. In addition, hyper-parameter values $\gamma = 0.8$, learning rate $\beta_\tau = 0.01$, as well as $\epsilon_\tau = 0.4$ were used throughout our numerical tests.

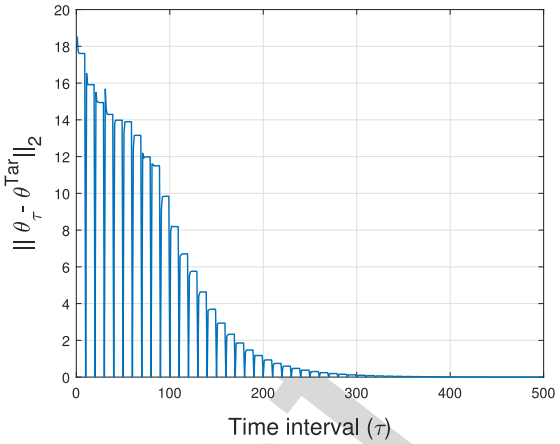

Fig. 6. Convergence of DQN to the target network.

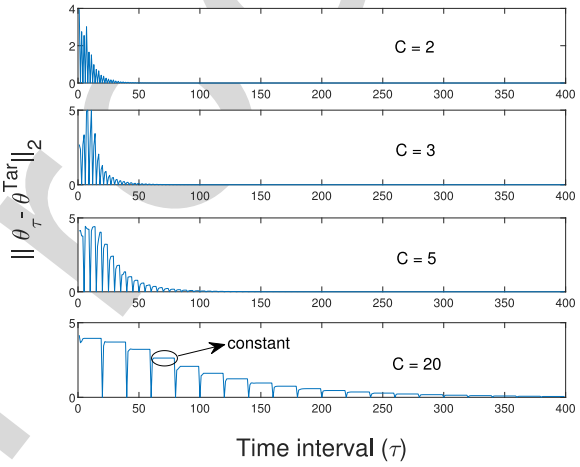

Fig. 7. Impact of C on DQN convergence in a static setting.

Figure 6 shows the convergence of the DQN parameter $\boldsymbol{\theta}_\tau$ to that of the target network $\boldsymbol{\theta}^{\text{Tar}}$. Since $\boldsymbol{\theta}^{\text{Tar}}$ is updated with $\boldsymbol{\theta}_\tau$ per C iteration, the error $\|\boldsymbol{\theta}_\tau - \boldsymbol{\theta}^{\text{Tar}}\|_2$ vanishes periodically. While the error changes just in a few iterations after the $\boldsymbol{\theta}^{\text{Tar}}$ update, it is constant in several iterations before the next $\boldsymbol{\theta}^{\text{Tar}}$ update, suggesting that $\boldsymbol{\theta}_\tau$ is fixed across those iterations. This motivates investigating the impact of C on $\boldsymbol{\theta}_\tau$'s convergence. Figure 7 shows the convergence of $\boldsymbol{\theta}_\tau$ to $\boldsymbol{\theta}^{\text{Tar}}$ for $C = 20, 5, 3, 2$. Starting with $C = 20$, vector $\boldsymbol{\theta}_\tau$ is not updated in most iterations between two consecutive updates of $\boldsymbol{\theta}^{\text{Tar}}$. Clearly, the smaller C is, the faster the convergence for $\boldsymbol{\theta}_\tau$ is achieved. Indeed, this is a direct result of having static

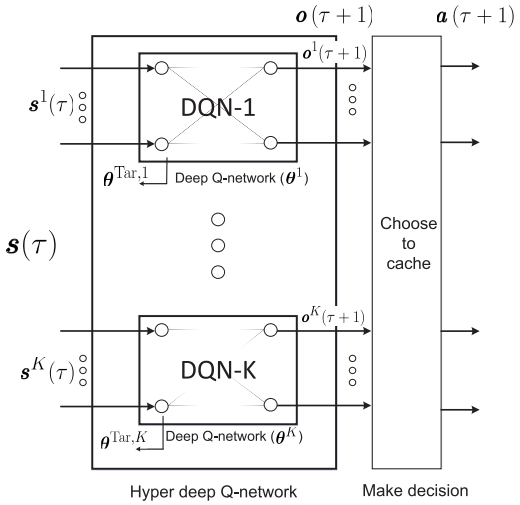

Fig. 8. Hyper deep-Q network for scalable caching.

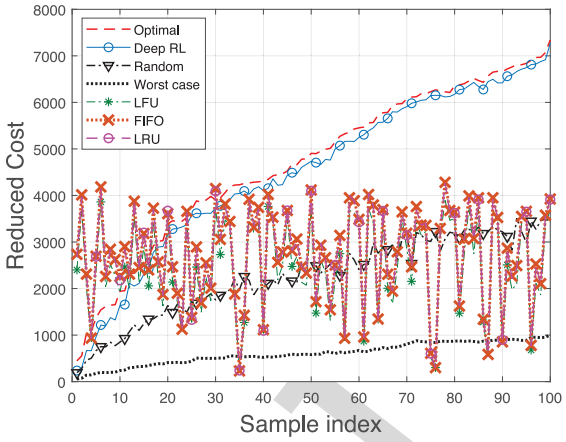

Fig. 9. Instantaneous reduced cost for different policies.

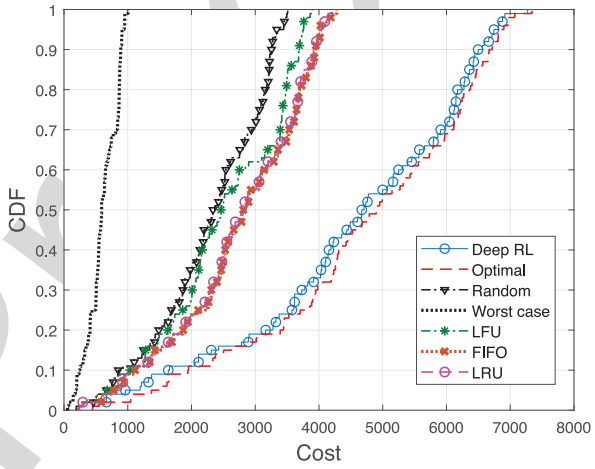

Fig. 10. Instantaneous reduced cost for different policies.

file popularity. Based on our numerical tests, having small C values is preferable in dynamic settings too.

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 process described earlier in the paper, and again they are unknown to the caching node.

We first consider that the parent node is connected to $N = 5$ leaf nodes, and every leaf node implements a local caching policy π_n with capacity of storing $M_n = 5$ files. Every leaf node receives user file requests within each fast time slot, and each slow-timescale interval consists of $T = 2$ slots. File popularity exhibits different Markovian dynamics locally at leaf nodes. Having no access to local policies $\{\pi_n\}$, the parent node not only should learn file popularities along with their temporal evolutions, but also learn the caching policies of leaf nodes. To endow our approach with scalability to handle $F \gg 1$, we advocate the following hyper Q-network implementation. Files are first split into K smaller groups of sizes $F_1, \ldots, F_K$ with $F_k \ll F$. This yields the representation $\boldsymbol{s}^\top(\tau) := [\boldsymbol{s}^{1\top}(\tau), \ldots, \boldsymbol{s}^{K\top}(\tau)]$, where $\boldsymbol{s}^k \in \mathbb{R}^{F_k}$. By running K DQNs in parallel, every DQN-k now outputs the associated predicted costs of input files through $\boldsymbol{o}^k(\tau) \in \mathbb{R}^{F^k}$. Concatenating all these outputs, one obtains the predicted output cost vector of all files as $\boldsymbol{o}^\top(\tau+1) := [\boldsymbol{o}^{1\top}(\tau+1), \ldots, \boldsymbol{o}^{K\top}(\tau+1)]$; see Section V for discussion on finding action $\boldsymbol{a}$ from vector $\boldsymbol{o}$, and also Fig. 8 for an illustration of our hyper Q-network.

The ensuing numerical tests consider $F = 1{,}000$ files with $K = 25$ and $\{F_k = 20\}$, where only $M_0 = 50$ can be stored. To further assess the performance, we adopt a *non-causal* optimal policy as a 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. Further, practically used cache schemes including, e.g., the LRU, LFU, and FIFO [45] are also simulated. A difference between LRU, LFU, FIFO, and our proposed approach is that they refresh the cache whenever a request is received, while our scheme refreshes the cache only at the end of every time slot.

By drawing 100 samples[2] randomly from all 2,000 time intervals, the instantaneous reduced cost and the empirical cumulative distribution function (CDF) obtained over these 100 random samples for different policies are plotted in Fig. 9 and Fig. 10, respectively. These plots further verify how the DRL policy performs relative to the alternatives, and in particular very close to the optimal policy.

LRU, LFU, and FIFO make caching decisions based on instantaneous observations, and can refresh the cache many times within each slot. Yet, our proposed policy as well as the optimal one here learns from all historical observations to cache, and refreshes the cache only once per slot. Because of this difference, the former policies outperform the latter at the very beginning of Fig. 9, but they do not adapt to the underlying popularity evolution and are outperformed by our learning-based approach after a number of slots. The merit of our approach is further illustrated by the CDF of the reduced cost depicted in Fig. 10.

[2]During these samples, DRL policy is enforced to exploit its learned caching policy.

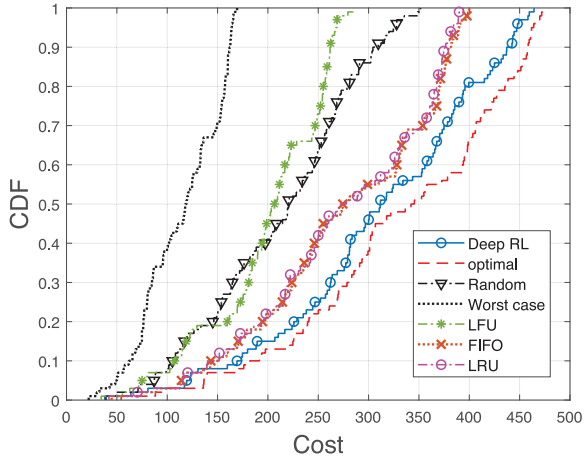

Fig. 11. Performance of all policies for $N = 10$ leaf nodes.

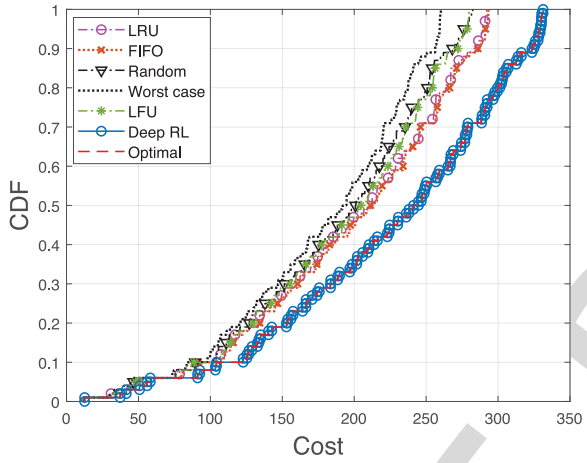

Fig. 12. Performance of all policies for $N = 1{,}000$ leaf nodes.

In the last test, we increased the number of leaf nodes from 10 in the previous experiment to $N = 1{,}000$. Figures 11 and 12 showcase that the DRL performance approaches that of the optimal policy as the number of nodes increases. This is likely because the more leaf nodes, the smoother the popularity fluctuations, and therefore the easier it becomes for DRL to learn the optimal policy.

VII. Conclusion

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. Such a two-timescale caching task was tackled in this paper using a RL approach. An efficient caching policy leveraging deep RL was introduced, and shown capable of learning-and-adapting to dynamic evolutions of file requests, and caching policies of leaf nodes. Simulated tests corroborated its impressive performance. This work also opens up several directions for future research, including multi-armed bandit online learning [46], and distributed deep RL using recurrent NNs [47] for future spatio-temporal file request prediction.

References

[1] G. S. Paschos, G. Iosifidis, M. Tao, D. Towsley, and G. Caire, "The role of caching in future communication systems and networks," *IEEE J. Sel. Areas Commun.*, vol. 36, no. 6, pp. 1111–1125, Jun. 2018.

[2] E. Bastug, M. Bennis, 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.

[3] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016.

[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] V. Mnih *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, p. 529, Feb. 2015.

[6] N. C. Luong *et al.*, "Applications of deep reinforcement learning in communications and networking: A survey," *IEEE Commun. Surveys Tuts.*, to be published.

[7] J. Gao and R. Jamidar, "Machine learning applications for data center optimization," Menlo Park, CA, USA, Google, White Paper, Oct. 2014.

[8] O. Naparstek and K. Cohen, "Deep multi-user reinforcement learning for dynamic spectrum access in multichannel wireless networks," in *Proc. Glob. Commun. Conf.*, Singapore, Dec. 2017, pp. 1–7.

[9] Y. Yu, T. Wang, and S. C. Liew, "Deep-reinforcement learning multiple access for heterogeneous wireless networks," in *Proc. Int. Conf. Commun.*, Kansas City, MO, USA, May 2018, pp. 1–7.

[10] Z. Wang, L. Li, Y. Xu, H. Tian, and S. Cui, "Handover control in wireless systems via asynchronous multiuser deep reinforcement learning," *IEEE Internet Things J.*, vol. 5, no. 6, pp. 4296–4307, Dec. 2018.

[11] Y. Sun, M. Peng, and S. Mao, "Deep reinforcement learning-based mode selection and resource management for green fog radio access networks," *IEEE Internet Things J.*, vol. 6, no. 2, pp. 1960–1971, Apr. 2019.

[12] Y. Dong, M. Z. Hassan, J. Cheng, M. J. Hossain, and V. C. M. Leung, "An edge computing empowered radio access network with UAV-mounted FSO fronthaul and backhaul: Key challenges and approaches," *IEEE Wireless Commun.*, vol. 25, no. 3, pp. 154–160, Jul. 2018.

[13] C. Yu, J. Lan, Z. Guo, and Y. Hu, "DROM: Optimizing the routing in software-defined networks with deep reinforcement learning," *IEEE Access*, vol. 6, pp. 64533–64539, 2018.

[14] Z. Guo, W. Chen, Y.-F. Liu, Y. Xu, and Z.-L. Zhang, "Joint switch upgrade and controller deployment in hybrid software-defined networks," *IEEE J. Sel. Areas Commun.*, vol. 37, no. 5, pp. 1012–1028, Mar. 2019.

[15] P. Blasco and D. Gündüz, "Learning-based optimization of cache content in a small cell base station," in *Proc. IEEE Int. Conf. Commun.*, Sydney, NSW, Australia, Jun. 2014, pp. 1897–1903.

[16] L. Lei, L. You, G. Dai, T. X. Vu, D. Yuan, and S. Chatzinotas, "A deep learning approach for optimizing content delivering in cache-enabled HetNet," in *Proc. Int. Symp. Wireless Commun. Syst.*, Bologna, Italy, Aug. 2017, pp. 449–453.

[17] S. Traverso, M. Ahmed, M. Garetto, P. Giaccone, E. Leonardi, and S. Niccolini, "Temporal locality in today's content caching: Why it matters and how to model it," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 5, pp. 5–12, Nov. 2013.

[18] M. Leconte, G. S. Paschos, L. Gkatzikis, M. Draief, S. Vassilaras, and S. Chouvardas, "Placing dynamic content in caches with small population," in *Proc. Int. Conf. Comput. Commun.*, San Francisco, CA, USA, Apr. 2016, pp. 1–9.

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

[20] S. O. Somuyiwa, A. György, and D. Gündüz, "A reinforcement-learning approach to proactive caching in wireless networks," *IEEE J. Sel. Areas Commun.*, vol. 36, no. 6, pp. 1331–1344, Jun. 2018.

[21] A. Sadeghi, F. Sheikholeslami, A. G. Marques, and G. B. Giannakis, "Reinforcement learning for adaptive caching with dynamic storage pricing," *IEEE J. Sel. Areas Commun.*, to be published.

[22] Y. He *et al.*, "Deep-reinforcement-learning-based optimization for cache-enabled opportunistic interference alignment wireless networks," *IEEE Trans. Veh. Technol.*, vol. 66, no. 11, pp. 10433–10445, Nov. 2017.
[23] C. Zhong, M. C. Gursoy, and S. Velipasalar, "A deep reinforcement learning-based framework for content caching," in *Proc. Conf. Inf. Sci. Syst.*, Princeton, NJ, USA, Mar. 2018, pp. 1–6.
[24] Y. He, N. Zhao, and H. Yin, "Integrated networking, caching, and computing for connected vehicles: A deep reinforcement learning approach," *IEEE Trans. Veh. Technol.*, vol. 67, no. 1, pp. 44–55, Jan. 2018.
[25] Y. He, F. R. Yu, N. Zhao, V. C. M. Leung, and H. Yin, "Software-defined networks with mobile edge computing and caching for smart cities: A big data deep reinforcement learning approach," *IEEE Commun. Mag.*, vol. 55, no. 12, pp. 31–37, Dec. 2017.
[26] H. Zhu, Y. Cao, W. Wang, T. Jiang, and S. Jin, "Deep reinforcement learning for mobile edge caching: Review, new features, and open issues," *IEEE Netw.*, vol. 32, no. 6, pp. 50–57, Nov./Dec. 2018.
[27] M. A. Maddah-Ali and U. Niesen, "Fundamental limits of caching," *IEEE Trans. Inf. Theory*, vol. 60, no. 5, pp. 2856–2867, May 2014.
[28] S. C. Borst, V. Gupta, and A. Walid, "Distributed caching algorithms for content distribution networks," in *Proc. Int. Conf. Comput. Commun.*, San Diego, CA, USA, Mar. 2010, pp. 1478–1486.
[29] R. Pedarsani, M. A. Maddah-Ali, and U. Niesen, "Online coded caching," *IEEE/ACM Trans. Netw.*, vol. 24, no. 2, pp. 836–845, Apr. 2016.
[30] J. Dai, Z. Hu, B. Li, J. Liu, and B. Li, "Collaborative hierarchical caching with dynamic request routing for massive content distribution," in *Proc. Int. Conf. Comput. Commun.*, Orlando, FL, USA, Mar. 2012, pp. 2444–2452.
[31] W. Wang, D. Niyato, P. Wang and A. Leshem, "Decentralized caching for content delivery based on blockchain: A game theoretic perspective," in *Proc. IEEE Intl. Conf. Commun.*, Kansas City, MO, USA, May 2018, pp. 1–6.
[32] E. Nygren, R. K. Sitaraman, and J. Sun, "The Akamai network: A platform for high-performance Internet applications," *ACM SIGOPS Oper. Syst. Rev.*, vol. 44, no. 3, pp. 2–19, 2010.
[33] M. Dehghan *et al.*, "On the complexity of optimal request routing and content caching in heterogeneous cache networks," *IEEE/ACM Trans. Netw.*, vol. 25, no. 3, pp. 1635–1648, Jun. 2017.
[34] S. Shukla, O. Bhardwaj, A. A. Abouzeid, T. Salonidis, and T. He, "Proactive retention-aware caching with multi-path routing for wireless edge networks," *IEEE J. Sel. Areas Commun.*, vol. 36, no. 6, pp. 1286–1299, Jun. 2018.
[35] L. Tong, Y. Li, and W. Gao, "A hierarchical edge cloud architecture for mobile computing," in *Proc. IEEE Int. Conf. Comput. Commun.*, 2016, pp. 1–9.
[36] E. Dahlman, S. Parkvall, and J. Skold, *4G: LTE/LTE-Advanced for Mobile Broadband*. Amsterdam, The Netherlands: Academic, 2013.
[37] C. H. Papadimitriou and J. N. Tsitsiklis, "The complexity of Markov decision processes," *Math. Oper. Res.*, vol. 12, no. 3, pp. 441–450, 1987.
[38] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: MIT Press, 2018.
[39] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Mach. Learn.*, vol. 8, nos. 3–4, pp. 279–292, May 1992.
[40] R. Fagin, "Asymptotic miss ratios over independent references," *J. Comput. Syst. Sci.*, vol. 14, no. 2, pp. 222–250, 1977.
[41] P. Auer, "Using confidence bounds for exploitation-exploration trade-offs," *J. Mach. Learn. Res.*, vol. 3, pp. 397–422, Nov. 2002.
[42] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Upper Saddle River, NJ, USA: Prentice-Hall, 2016.
[43] G. Wang, G. B. Giannakis, and J. Chen, "Learning ReLU networks on linearly separable data: Algorithm, optimality, and generalization," *IEEE Trans. Signal Process.*, vol. 67, no. 9, pp. 2357–2370, May 2019.
[44] D. Silver *et al.*, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, p. 484, Jan. 2016.
[45] A. Dan and D. F. Towsley, "An approximate analysis of the LRU and FIFO buffer replacement schemes," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 18, no. 1, pp. 143–152, 1990.
[46] B. Li, T. Chen, and G. B. Giannakis, "Bandit online learning with unknown delays," in *Proc. Int. Conf. Artif. Intell. Stat.*, Naha, Japan, Apr. 2019, pp. 1–10.
[47] A. Narayanan, S. Verma, E. Ramadan, P. Babaie, and Z.-L. Zhang, "DeepCache: A deep learning based framework for content caching," in *Proc. ACM Workshop Netw. Meets AI & ML*, Budapest, Hungary, Aug. 2018, pp. 48–53.

Alireza Sadeghi (S'16) received the B.Sc. degree (Hons.) in electrical engineering from the Iran University of Science and Technology, Tehran, Iran, in 2012, and the M.Sc. degree in electrical engineering from the University of Tehran in 2015. He is currently pursuing the Ph.D. degree with the Department of Electrical and Computer Engineering, University of Minnesota, Minneapolis, MN, USA. In 2015, he was a Visiting Scholar with the University of Padua, Padua, Italy. His research interests include machine learning, optimization, and signal processing with applications to networking. He was a recipient of the ADC Fellowship awarded by the Digital Technology Center of the University of Minnesota, and the Student Travel Awards from the IEEE Communications Society and the National Science Foundation.

Gang Wang (M'18) received the B.Eng. degree in electrical engineering and automation from the Beijing Institute of Technology, Beijing, China, in 2011, and the Ph.D. degree in electrical and computer engineering from the University of Minnesota, Minneapolis, MN, USA, in 2018.

He is currently a Post-Doctoral Associate with the Department of Electrical and Computer Engineering, University of Minnesota. His research interests focus on the areas of statistical signal processing, optimization, and deep learning with applications to data science and smart grids. He was a recipient of the National Scholarship in 2013, the Guo Rui Scholarship in 2015, and the Innovation Scholarship (First Place in 2017), all from China, as well as the Best Conference Papers at the 2017 European Signal Processing Conference and 2019 IEEE Power and Energy Society General Meeting.

Georgios B. Giannakis (F'97) received the Diploma degree in electrical engineering from the National Technical University of Athens, Greece, in 1981, and the first M.Sc. degree in electrical engineering, the second M.Sc. degree in mathematics, and the Ph.D. degree in electrical engineering from the University of Southern California in 1983, 1986, and 1986, respectively.

From 1982 to 1986, he was with the University of Southern California. He was a Faculty Member with the University of Virginia from 1987 to 1998. Since 1999, he has been a Professor with the University of Minnesota, where he holds an ADC Endowed Chair, a University of Minnesota McKnight Presidential Chair in ECE, and serves as the Director of the Digital Technology Center. His general interests span the areas of statistical learning, communications, and networking—subjects on which he has published over 450 journal papers, 750 conference papers, 25 book chapters, 2 edited books, and 2 research monographs (H-index 142). He has co-invented 32 patents. Current research focuses on data science and network science with applications to the Internet of Things, social, brain, and power networks with renewables. He was a (co-)recipient of nine best journal paper awards from the IEEE Signal Processing (SP) and Communications Societies, including the G. Marconi Prize Paper Award in *Wireless Communications*. He also received Technical Achievement Awards from the SP Society in 2000, the EURASIP in 2005, the Young Faculty Teaching Award, the G. W. Taylor Award for Distinguished Research from the University of Minnesota, and the IEEE Fourier Technical Field Award (inaugural recipient in 2015). He has served the IEEE in a number of posts, including that of a Distinguished Lecturer for the IEEE-SPS. He is a fellow of EURASIP.