This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/JSAC.2020.3000415, IEEE Journal on Selected Areas in Communications

RL-Cache: Learning-Based Cache Admission for Content Delivery

Vadim Kirilin, Aditya Sundarrajan, Sergey Gorinsky, *Member, IEEE,* and Ramesh K. Sitaraman, *Fellow, IEEE*

***Abstract*—Content delivery networks (CDNs) distribute much of the Internet content by caching and serving the objects requested by users. A major goal of a CDN is to maximize the hit rates of its caches, thereby enabling faster content downloads to the users. Content caching involves two components: an admission algorithm to decide whether to cache an object and an eviction algorithm to determine which object to evict from the cache when it is full. In this paper, we focus on cache admission and propose a novel algorithm called RL-Cache that uses model-free reinforcement learning (RL) to decide whether or not to admit a requested object into the CDN's cache. Unlike prior approaches that use a small set of criteria for decision making, RL-Cache weights a large set of features that include the object size, recency, and frequency of access. We develop a publicly available implementation of RL-Cache and perform an evaluation using production traces for the image, video, and web traffic classes from Akamai's CDN. The evaluation shows that RL-Cache improves the hit rate in comparison with the state of the art and imposes only a modest resource overhead on the CDN servers. Further, RL-Cache is robust enough that it can be trained in one location and executed on request traces of the same or different traffic classes in other locations of the same geographic region. The paper also reports extensive analyses of the RL-Cache sensitivity to its features and hyperparameter values. The analyses validate the made design choices and reveal interesting insights into the RL-Cache behavior.**

***Index Terms*—Content delivery network; caching; cache admission; hit rate; object feature; neural network; direct policy search; Monte Carlo sampling; stochastic optimization; traffic class; image; video; web; production trace.**

I. Introduction

Today's Internet heavily relies on content delivery networks (CDNs) to provide low-latency access to its content for billions of users around the globe. A large CDN deploys hundreds of thousands of servers worldwide so that at least some servers of the CDN lie in each user's network proximity. When a user requests an object such as an image, video, or web page, the user's request goes to a nearby server of the CDN [1]. If the cache of the CDN server stores the requested object, i.e., a *hit* happens, the user promptly receives the object from the server's cache. On the other hand, if the requested object is not in the server's cache, i.e., a *miss* occurs, the CDN server delivers the object to the user after fetching the object from the content provider's origin server, and the delivery might be slow because the origin server might be far away.

Decreasing the user-perceived latency of content delivery constitutes the main goal of the CDN. Hence, the CDN strives to maximize the server's *hit rate* defined as the percentage of requests that are served straight from the cache. When the CDN server receives an object request, the server might need to make *admission* and *eviction* decisions. If the request is a miss, the server must decide whether to admit the fetched object into the cache. Furthermore, if the server decides to cache the fetched object, and the cache is already full, the server must decide which object(s) it should evict from the cache to make space for the new arrival. For example, Least Recently Used (LRU) is a simple eviction policy that discards the least recently served object. Major CDNs employ LRU and its variants, such as Segmented LRU (SLRU) [2], for cache eviction. Researchers have proposed a large number of more sophisticated eviction algorithms that are more difficult to implement in practice, e.g., Greedy-Dual-Size-Frequency (GDSF) [3]. The work on admission algorithms is less extensive and includes SecondHit [4] and AdaptSize [5].

Our goal is to investigate whether Machine Learning (ML) techniques can increase cache hit rates in typical CDN production settings, without adding excessive overhead or requiring major software changes. This paper examines ML-based algorithms for cache admission, leaving the question of eviction improvement for future work. Despite the extensive prior research on cache eviction, nearly all production content caches – including Akamai caches [6], Varnish [7], Memcached [8], and Nginx [9] – use LRU variants as their default eviction algorithm. LRU's popularity arises due to easy implementation combined with very good hit rates in production settings. Consequently, similar to the state-of-the-art AdaptSize admission algorithm, we assume LRU as the eviction algorithm throughout our paper. Our work is complementary to recent ML-based caching proposals that learn popularity of objects and/or determine the cache eviction order, e.g., DeepCache [10] and PopCache [11].

Our Contributions. We formulate cache admission as a model-free Reinforcement Learning (RL) problem and solve it via direct policy search that combines Monte Carlo (MC) sampling and stochastic optimization. Unlike prior works that require complex object ordering and eviction strategies, our goal is to create a simple practicable cache-admission front end for an existing CDN server. This approach is easier to implement in a production setting because such cache-

The first two authors contributed equally to the paper.

Vadim Kirilin started this research at IMDEA Networks Institute, Spain and is currently with Yandex LLC, Russia (email: durrdurr@yandex-team.ru). Aditya Sundarrajan performed this research at the University of Massachusetts Amherst, USA and is currently with Facebook, Inc., USA (email: asundar@cs.umass.edu). Sergey Gorinsky is with IMDEA Networks Institute, Spain (email: sergey.gorinsky@imdea.org). Ramesh K. Sitaraman is with the University of Massachusetts Amherst and Akamai Technologies, Inc., USA (email: ramesh@cs.umass.edu).

This research was supported in part by the Regional Government of Madrid (grant P2018/TCS-4499, EdgeData-CM) and U.S. National Science Foundation (grants CNS-1763617 and CNS-1717179).

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/JSAC.2020.3000415, IEEE Journal on Selected Areas in Communications

admission front ends already exist in practice, e.g., Akamai's Bloom-filter implementation of SecondHit [4].

The proposed RL-Cache algorithm employs a feedforward neural network that computes probabilities for the binary decision of whether to admit or not to admit a requested object into the cache. The training of the neural network on request traces from CDN servers is computationally intensive and done offline in the cloud, periodically and not in real time. When a CDN server obtains the trained neural network, the server uses the network to make admission decisions in real time, by efficiently processing the object requests in batches with small computation overhead, e.g., the per-request processing time on the CPU (Central Processing Unit) and GPU (Graphics Processing Unit) platforms in our evaluation is about 16 μs and 4 μs respectively for batches of 4,096 requests. This additional processing does not impose a significant demand on the resources of CDN servers.

We develop a publicly available implementation of RL-Cache, provide open access to it [12], and perform an evaluation on real-world request traces from Akamai's production CDN. CDNs host traffic classes that have very different object-size distributions and object-access patterns, requiring different caching strategies [13]. We test our approach on three major CDN-traffic classes: web, images, and videos. We also define a notion of active bytes that characterizes the cache size needed to achieve a high hit rate on a particular trace. For the examined traffic classes and cache sizes, we show that RL-Cache successfully learns to outperform or at least match (e.g., when the cache is abundant for the needs of the trace) the hit rate achieved by state-of-the-art algorithms.

The paper also reports extensive robustness and sensitivity studies of RL-Cache. Specifically, we demonstrate that RL-Cache is robust in the sense that it can be trained in one location and executed, without a significant loss in the hit-rate performance, on traces of the same or different traffic classes in other locations of the same geographic region. After our assessments of feature correlation and feature importance, we evaluate RL-Cache on promising smaller subsets of its full feature set and conclude that the full set of eight features is critical for RL-Cache to maximize the hit-rate performance. The analyses of sensitivity of RL-Cache to its hyperparameter values justify the relevance of all hyperparameters and offer interesting insights into the RL-Cache behavior.

To sum up, RL-Cache represents a pioneering ML-based approach to improve cache hit rates by learning an admission policy and is the first such scheme to be validated on real-world CDN traces across multiple traffic classes. This article extends its preliminary short version [14].

Roadmap. This paper has the following structure. Section II provides background and discusses related work. Section III presents RL-Cache, including its feature selection, training, and implementation. Section IV empirically evaluates the algorithm. Finally, Section V concludes the paper with a summary of its contributions.

II. Background and Related Work

CDN architecture. Figure 1 depicts a typical CDN architecture that deploys clusters of edge servers in data centers around the world to host and serve content from thousands of providers to billions of end users. Upon receiving a user's request for content, the CDN directs the request to its closest edge server [4]. If this server caches the requested object, i.e., upon a cache hit, the server immediately sends the object to the user. Otherwise, the edge server fetches the object from either peer servers in the same cluster, or parent servers farther away, or over a large-latency Wide Area Network (WAN) from the content provider's origin server as a last resort. To minimize user-perceived latency, the CDN strives to serve the content from the lowest possible layer in this hierarchy of origin, parent, peer, and edge servers.

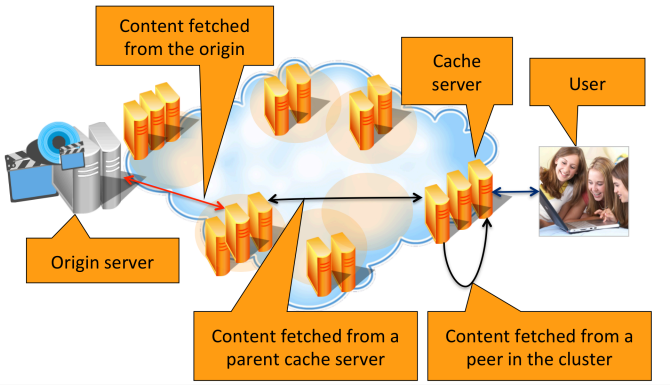


Fig. 1: Typical architecture of a CDN.

Every CDN server employs cache-admission and cache-eviction algorithms to maintain its cache of requested objects. The cache-admission algorithm determines whether to store the requested object upon a cache miss. The cache-eviction algorithm decides which objects to evict from the server when the cache is full. Typically, each server manages its cache independently. This paper proposes a new cache-admission algorithm for this kind of independent cache management. Our focus is on edge servers which constitute a vast majority of servers in a real-world CDN. We believe that the proposed approach is also extendable to peer and parent servers, as well as to cross-layer cache management. Such extensions represent a promising direction for future work.

Caching problem. Caching is related to the knapsack problem [15] which makes optimal caching computationally intractable, even in the offline setting where the entire request sequence is known beforehand. CDN caching faces additional online challenges due to uncertainty about future object requests. Further, CDNs host traffic classes with diverse object-size distributions and object-access patterns, making it hard for any particular caching policy to work well for all classes [13].

Existing work in caching predominantly focuses on design of eviction policies, e.g., LRU, SLRU, TLRU, S4LRU, GDSF, ARC, and Cliffhanger (see Table 2 in [5]). Such eviction-focused algorithms typically employ the basic admission policy of caching all requested objects. Recently, there has been an increased interest in more sophisticated cache-admission policies. SecondHit [4], an admission policy implemented by Akamai, uses the access frequency of an object and admits the object into the cache only upon a repeated request for the ob-

ject within a fixed time interval. SecondHit employs a Bloom filter as a front end of the cache to track objects that have been requested before. Another frequency-based approach is TinyLFU [16]. AdaptSize [5] is a size-based admission policy that uses a Markov model to adjust a threshold for the size of admitted objects. Unlike the previous work that considers one or two features of requested objects, our RL-Cache algorithm combines a broader set of eight features from the frequency, size, and recency classes to make an admission decision.

Previous ML-based caching solutions, which also commonly focus on eviction policies, optimize for proxy metrics of the hit rate. For example, DeepCache uses popularity prediction to prefetch popular objects into the cache [10]. PopCache caches objects with popularity-dependent probabilities [11]. FNN-based caching [17], NNPCR-2 [18], and KORA-2 [19] also rely on popularity prediction. LFO [20] uses supervised learning to make admission decisions by mapping object features to optimal decisions learned offline.

RL-Cache differs from some of the above ML-based schemes in optimizing for the hit rate directly, rather than via a proxy metric such as object popularity. Our design aligns perfectly with the RL paradigm because cache hits constitute a natural form of RL rewards. RL-Cache combines MC sampling with stochastic optimization to search directly for a policy that maximizes the hit rate. Further, RL-Cache focuses on cache admission, which is easier to implement as a front end for an existing CDN cache. In addition, much of the prior work uses less realistic traffic assumptions, such as uniform object sizes or synthetic workloads that do not accurately capture characteristics of real-world traffic classes in a CDN. Finally, some prior schemes require functionality that is hard to implement efficiently, such as creation of fake requests for popular objects [10] or modification of the eviction order based on object popularity [18].

III. RL-Cache

We follow the RL paradigm because of its natural fit with the cache-admission problem. In RL, an agent acts on the current state to maximize the sum of discounted future rewards that arise from the action [21]. The sequential decision making in model-free RL is highly suitable for networking problems in general because of the common necessity to make a sequence of online decisions in an uncertain environment where benefits from the made decisions become clearer only as time progresses further [22], [23]. Additionally, cache admission has special traits that make the problem more amenable to RL. Whereas formulation of some networking problems in terms of actions and rewards of the RL paradigm is far from straightforward [24], [25], cache admission submits itself to a natural RL formulation where admission decisions represent RL actions, and cache hits constitute RL rewards. In formulating and solving the problem, we keep close attention to the imposed computation overhead so that the derived solution is not only effective but also practical.

A. Feature Selection

Before formulating the cache-admission problem in RL terms, we select features u to characterize objects in a request trace. Caching algorithms typically describe a requested object with features belonging to the following three classes: (1) object size, (2) request recency, and (3) request frequency. The algorithms use these feature classes in either isolation or combination. For example, AdaptSize considers only the object size, LRU relies on recency, SecondHit is based on frequency, and GDSF combines the size and frequency. The strength of our approach is in simultaneously considering a broad set of eight features from these three classes, as defined in Table I. Compared to the object size, request recency is a vaguer notion amenable to diverse definitions. For instance, when exactly the same temporal gap separates two consecutive requests for object j, the number of all requests for objects between the two requests for object j can be very different depending on whether the other object requests arrive in the trace sparsely or in a burst. Besides, recency is different if defined with respect to the most recent request for object j as opposed to a series of such recent requests. Hence, we consider four recency features: temporal recency h_j, ordinal recency d_j, and their exponentially smoothed variants η_j and δ_j. Furthermore, a combination of primitive features might result in learning a dramatically different algorithmic behavior. Thus, we mix size s_j and request frequency f_j to also consider combined features f_j/s_j and $f_j \cdot s_j$. Later in the paper, we thoroughly evaluate the sensitivity of RL-Cache to the selection of features.

Feature	Meaning
s_j	Size of object j in bytes
h_j	Temporal recency, time in seconds since the previous request for object j
η_j	Exponential smoothing of h_j so far
d_j	Ordinal recency, the number of all requests for objects since the previous request for object j
δ_j	Exponential smoothing of d_j so far
f_j	Frequency, the fraction of requests for object j among all requests so far
f_j/s_j	Ratio of the frequency to size for object j
$f_j \cdot s_j$	Product of the frequency and size for object j

TABLE I: Features in our model.

B. RL Problem Formulation

In our RL formulation of the cache-admission problem, a state is a vector of object features. An action on the state refers to a decision whether to admit the requested object into the cache. To tremendously reduce the computation overhead, the state excludes cache occupancy because this allows RL-Cache to precompute the action probabilities for states and thereby avoid repeated computation of the probabilities for each runtime cache occupancy during the model training. Immediate reward r_i for an action signals whether the next request is a cache hit or miss. Then, we express the return as:

$$R = \sum_{i=1}^{\infty} \gamma^i r_i \tag{1}$$

where γ denotes the discount factor of future rewards. The objective is to design an RL algorithm so that its policy function of states and actions maximizes the expected return, which directly corresponds to maximizing the expected hit rate of the cache.

C. Narrowing Down the RL Approach

One can roughly classify model-free RL algorithms into TD (Temporal Difference), MC (Monte Carlo), and DPS (Direct Policy Search) types [26]. To narrow down our approach within the RL paradigm, we gear the design towards specifics of the formulated cache-admission problem. Because RL states are just the features of requested objects, an action on the current state has an extremely weak correlation with the immediate reward in the next state: the action of admitting the currently requested object triggers an immediate hit only when the next request is for the same object, which is extremely rare in real CDN request traces. Q-learning [27], and other TD algorithms that rely on bootstraping [21], might estimate action values too imprecisely under such noisy reward signals. Indeed, our experimentation with Q-learning reveals no significant improvements in the hit-rate performance compared to the state-of-art non-ML algorithms.

A potential direction for tackling the challenge of extremely noisy rewards is to adopt a different kind of rewards than cache hits. However, it is unclear how to design more informative rewards. The various rewards that we can think of require significant computation time, e.g., logarithmic in the number of cached objects. Due to the excessive computation overhead, we dismiss this direction and keep using cache hits as rewards.

An alternative is an MC learning algorithm that updates an action value based on longer sequences of state-action pairs. Such an algorithm observes the sequence-long returns of all sampled sequences and uses the average return for updating the action value. The drawbacks of MC learning algorithms include large overhead of computing the returns over long sequences to update only one state-action pair as well as vulnerability to high variance in the returns. Similarly to Q-learning, the MC learning algorithms in our experiments perform weakly.

Hence, we turn to and follow the DPS approach. Our RL-Cache algorithm also simulates long sequences of state-actions and records their sequence-long returns. Instead of averaging the returns to update one state-action pair as in the MC learning algorithms, RL-Cache utilizes the individual returns to segregate a subset of sequences with high returns and then leverages this subset to directly search for a better policy in the policy space. We represent a policy as a feedforward neural network that computes admission probability $A(u, w) \in [0, 1]$ as a function of features u of the requested object (which capture the state) and weights w of the neural network. Our training algorithm employs the neural network to simulate sufficiently many sequences of admission decisions (i.e., actions) and then adjusts the neural-network weights to learn a new policy on a high-performance subset of these sequences.

Note that RL-Cache is a DPS algorithm where MC sampling serves a different role than in MC learning algorithms. The latter consider all generated samples to update the state-action pair without bias. On the other hand, RL-Cache deliberately considers only high-return samples to steer its direct search towards a policy that reproduces actions leading to maximal rather than average returns. Later in the paper, we quantify the performance benefits of this intentional bias. Within the DPS paradigm, RL-Cache aligns closely with the CE (Cross-Entropy) method which also combines MC sampling and stochastic optimization [28].

D. Neural-Network Architecture

The network architecture in our solution is a fully connected ANN (Artificial Neural Network) with ELU (Exponential Linear Unit) activation functions in each of its five hidden layers. We select ELU over ReLu (Rectified Linear Unit) and Leaky ReLu to resolve the zero-gradient problem of ReLu on negative inputs without introducing the potential result inconsistency under Leaky ReLu [29]. With n denoting the number of neurons in the input layer, the l-th hidden layer contains $5(6 - l)n$ neurons, i.e., the hidden layers narrow linearly along the forward pass. To avoid overfitting, we apply L2 regularization as it can provide better generalization in RL than with dropout or batch normalization [30]. While the eight features selected in Subsection III-A are continuous, we quantize their value spaces into ten or less bins and use each bin value as an ANN input. For each feature, additional inputs similarly discretize its historical version that exponentially weights the feature value over the history of object requests (as we noticed later, benefits from this extension were relatively low, and it could be removed to make the network smaller). Thus, n in our network is at most $8 \cdot 10 \cdot 2 = 160$ inputs. The output layer uses the softmax activation function and contains two neurons that produce probabilities for the two respective outcomes of admitting or not admitting the requested object into the cache. Based on limited experimentation, our choice of the above ANN architecture strives to support effective learning on moderate computing resources, rather than to identify an optimal network design.

E. Training Algorithm

The objective of the training algorithm is to adjust weights w of the neural network so that the admission probabilities computed for object requests by the network realize a cache-admission policy with the maximum hit rate. Even though we envision training RL-Cache in the cloud, periodically and not in real time, the computation overhead remains a key consideration and guides our design choices. RL-Cache learns at the granularity of multiple consecutive requests, rather than a single request, simulates many sequences of admission decisions for every considered sequence of requests, and evaluates each of these admission-decision sequences by computing its long-term hit rate.

More specifically, we train the network on a window of K consecutive requests and slide the window along the training trace by K requests at a time. As the window slides from the beginning to the end of the training trace, the algorithm keeps updating weights w. To compute the return for any

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/JSAC.2020.3000415, IEEE Journal on Selected Areas in Communications

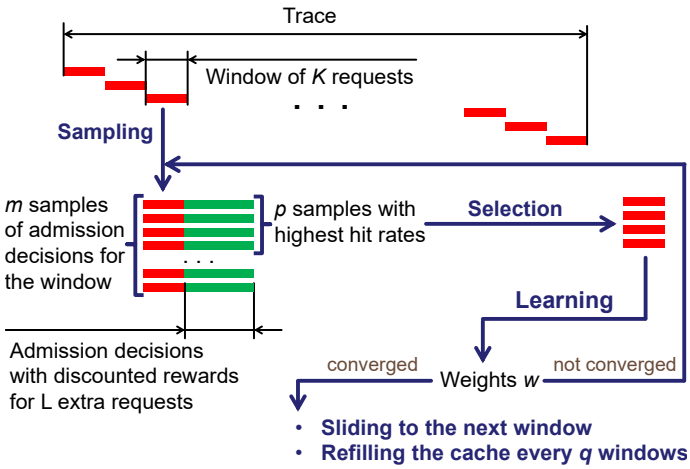


Fig. 2: Training algorithm of RL-Cache.

sample sequence of K admission decisions generated for a K-request window, we also simulate admission decisions for the L requests that immediately follow this window. Thus, the return computation for the sample of K admission decisions considers an extended sequence of $K+L$ admission decisions. Depending on whether the next request after the i-th decision in the sequence is a cache hit or miss, we express immediate reward r_i for this decision as:

$$r_i = \begin{cases} \frac{1}{K+L} \text{ for a cache hit,} \\ 0 \text{ for a cache miss.} \end{cases} \tag{2}$$

The first K immediate rewards in the extended sequence contribute to the return without a discount, i.e., one can view γ for these "native" immediate rewards as equal to 1. The return contributions by the last L rewards are discounted with factor γ between 0 and 1, which is common for future rewards in RL. We consider any rewards further in the future as negligible. With this, the return for the K-decision sample becomes:

$$R = \frac{\sum_{i=1}^{K} \mathbb{1}_{\text{hits}}(\text{reward } i) + \sum_{i=K+1}^{K+L} \gamma^{i-K} \mathbb{1}_{\text{hits}}(\text{reward } i)}{K+L} \tag{3}$$

where $\mathbb{1}$ is an indicator function. This return captures the long-term hit rate of the admission-decision sample. We refer to the γ^L factor of the last of these $K+L$ contributing rewards as hyperparameter c and derive γ from c. The training algorithm uses the extra L admission decisions in each extended admission-decision sequence solely to compute return R of the "native" K-decision sample.

Figure 2 describes how the training algorithm operates in each position of the K-request window. The operation consists in iterating over three steps. The first step uses MC sampling to generate m admission-decision sequences, with each sample containing K admission decisions. Then, the second step of the iteration selects the p-th percentile of these K-decision samples with the highest returns (computed over the extended sequences of $K+L$ admission decisions) to steer the policy search towards an optimal policy with the maximum hit rate. The final third step utilizes the selected K-decision samples for learning the new policy via the backpropagation algorithm that uses binary cross-entropy loss as the loss function [31]. The three-step iterations continue until the neural-network weights converge. Upon the convergence within a threshold (or when the number of iterations reaches an upper bound), the algorithm slides the window to the next K requests in the training trace. In the expected case when L exceeds 0, the extended admission-decision sequences of consecutive windows overlap, thereby connecting the consecutive learning episodes. Because the neural-network weights change as the window slides along the training trace, the cumulative effect of the weight changes might undermine the training effectiveness. Thus, the training algorithm simulates refilling the cache after every q windows under the current weights for all the requests from the beginning of the trace.

The presented training algorithm gears its design choices towards specifics of the cache-admission problem. Regardless of these adjustments, RL-Cache is essentially a CE algorithm and has the same theoretical properties as the general CE method [28]. Also despite the various design optimizations, the training algorithm is computationally intensive. We envision the training to be periodically performed in the cloud, with an updated version of the trained network provided to the CDN server at the end of each period.

F. Real-Time Operation

Whereas the training of RL-Cache is computationally intensive, the usage of RL-Cache to make online cache-admission decisions is simple and can be done efficiently in real time without a significant demand on the resources in the CDN server. Upon receiving a request for an object, the CDN server applies the trained neural network to the object features to compute the admission probability for the request and then rounds the computed probability to 1 or 0 to decide whether to admit or not to admit the requested object into the cache.

G. Implementation

We implement RL-Cache using the TensorFlow library [32] without requiring any extensions. Our implementation is publicly available with open access at its GitHub repository [12]. The implementation of RL-Cache in the cache server maintains a database with feature statistics, which are needed to compute the frequency and recency metrics, and applies the most recently obtained neural network to arriving requests. Upon receiving an object request, the cache computes the features of the object, updates the feature-statistics database, and uses the neural network to make an admission decision for the object. The usage of the neural network contributes the most to the processing overhead imposed by RL-Cache on the cache.

To keep the neural-net processing overhead low, our RL-Cache implementation leverages pipelining and batching. RL-Cache is invoked only for those requests that result in a cache miss and trigger fetching of the missed object from its origin over the WAN, with typical fetching latency above 100 ms. Further, RL-Cache makes admission decisions *asynchronously* with serving the requested object to the user, since the server

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/JSAC.2020.3000415, IEEE Journal on Selected Areas in Communications

Request trace US1-∗	image	video	web
Requests (10^6)	85.48	49.85	144.87
Unique objects (10^6)	33.20	7.13	11.11
Unique bytes (TB)	0.64	2.35	2.56
Traffic volume (Gbps)	0.06	0.21	1.69

TABLE II: Properties of the US1-image, US1-video, and US1-web request traces.

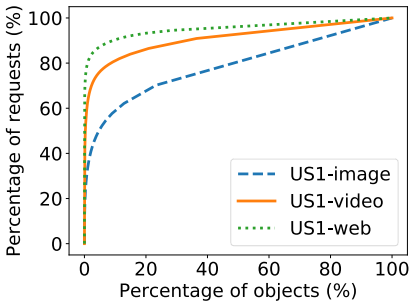


Fig. 3: Object-popularity distribution for US1-image, US1-video, US1-web.

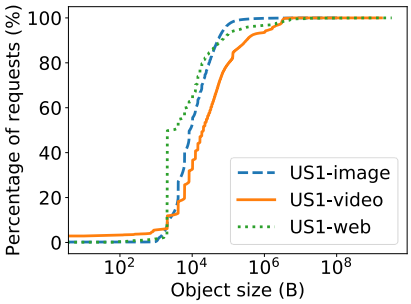


Fig. 4: Object-size distribution for US1-image, US1-video, and US1-web.

caches the object already after delivering it to the user. Hence, RL-Cache can be run in a batch mode where the cache accumulates arriving requests into a batch and sends them jointly, as one batch, for the neural-net processing. The batch mode exploits architectural properties of the multi-core processors in modern CDN servers. The parallel processing of the batch requests, further enhanced by potential sharing of memory banks among the processor cores, enables the modern CDN servers to adopt RL-Cache without reducing their request-processing rates.

IV. Empirical Evaluation

To evaluate RL-Cache, we start with three traces collected over a period of four days from a US-based edge server in Akamai's production network. Table II characterizes these US1-image, US1-video, and US1-web request traces that represent the image, video, and web traffic classes respectively.

We demonstrate the diversity of the considered traffic classes by plotting the object-popularity and object-size distributions for each of the traces. For US1-video and US1-web, Figure 3 shows that 80% of the requests are for less than 10% of the objects, indicating that a relatively small subset of such objects needs to be cached to achieve a high hit rate. On the other hand, 80% of the requests in the US1-image trace are for nearly 60% of the objects, suggesting that a larger fraction of objects belonging to the image traffic class should be cached to provide the high hit rate. Figure 4 reveals that objects of the examined traffic classes can vary in size by up to two orders of decimal magnitude. The extreme variability in the object-popularity and object-size distributions among traffic classes makes cache management challenging in production settings. We later show that RL-Cache is able to adapt to the varying popularity and size characteristics to achieve good hit rates.

Evaluation Methodology. Sizing the cache to achieve high hit-rate performance on a trace of object requests depends on properties of the specific trace. To characterize these properties, we introduce a notion of active bytes. First, we view an object as *active* at time t of a trace if this t lies, inclusively, between the first and last requests for the object in the trace. Then, we define *active bytes* as the total size of the objects active at time t. Active bytes are relevant because they capture the cache size sufficient to preclude any hot misses (i.e., misses upon the second and all subsequent requests for each object) by the offline algorithm that admits all requested

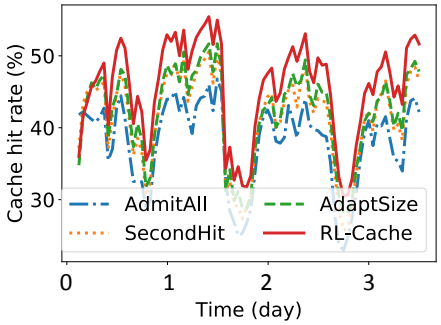


Fig. 5: Hit-rate dynamics on US1-image with the 16-GB cache.

objects and evicts every object upon its last request in the trace. The above policy of admitting all objects is common in eviction-focused caching algorithms, and we refer to this admission policy as AdmitAll. For an infinitely large cache, AdmitAll is an optimal admission policy.

Our evaluation compares RL-Cache with AdmitAll, frequency-based SecondHit, and adaptive size-based AdaptSize. Figure 5 illustrates hit-rate dynamics on the US1-image trace with the cache sized to 16 GB, where the hit rate is computed over 1-hour intervals. RL-Cache consistently outperforms the existing counterparts under the observed diurnal patterns of the image requests.

Guided by Figures 6, 7, and 8 that plot the active bytes for the US1-image, US1-video, and US1-web traces respectively, we choose to test the cache-admission algorithms on caches sized to 2 GB, 16 GB, and 128 GB. Because Figures 6 and 7 indicate that the active bytes remain below 128 GB throughout the US1-image and US1-video traces, we hypothesize that the 128-GB cache is plentiful for the needs of these two traces, and the basic AdmitAll admission policy should be nearly optimal on the 128-GB cache even when combined with a reasonable online eviction policy such as LRU. Furthermore, since the active bytes in Figures 6 and 7 exceed both 2 GB and 16 GB during most of these two traces (except the expected sharp rise and drop in the beginning and end of each trace respectively), our hypothesis is that a more sophisticated admission policy should be able to outperform AdmitAll in regard to the hit rates on the 2-GB and 16-GB caches. On the other hand, whereas Figure 8 shows that the active bytes surpass 128 GB during most of the US1-web trace, we expect AdmitAll to be suboptimal for this trace on the 128-GB, 16-GB, and 2-GB

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/JSAC.2020.3000415, IEEE Journal on Selected Areas in Communications

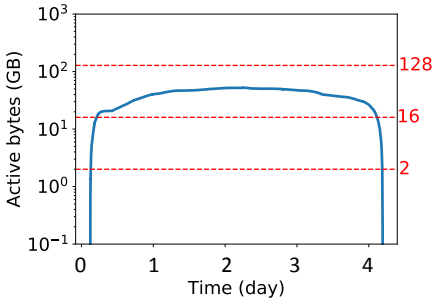


Fig. 6: Active bytes for US1-image.

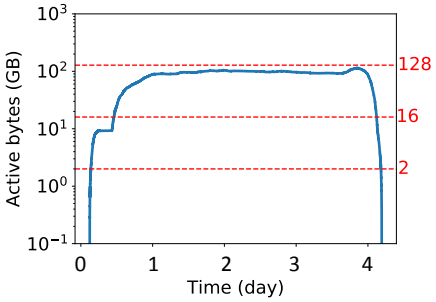


Fig. 7: Active bytes for US1-video.

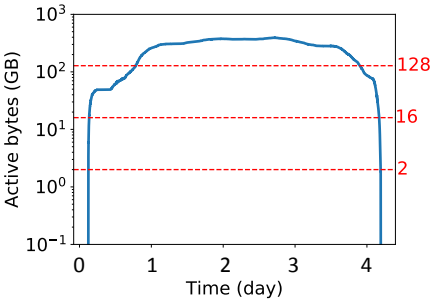


Fig. 8: Active bytes for US1-web.

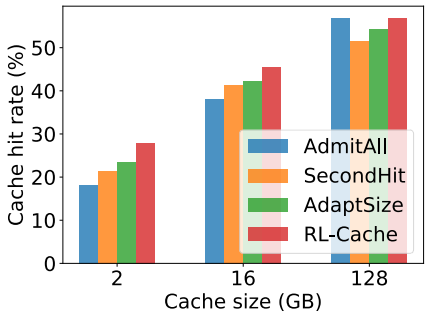


Fig. 9: Average hit rate on US1-image.

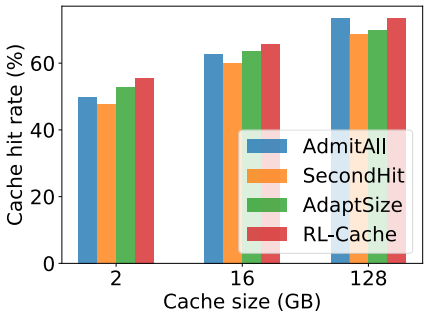


Fig. 10: Average hit rate on US1-video.

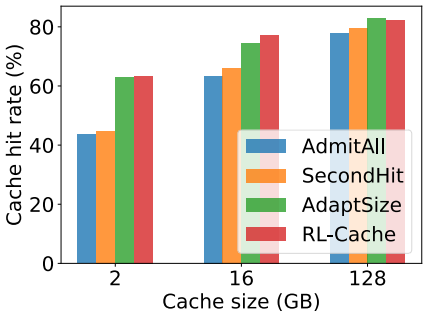


Fig. 11: Average hit rate on US1-web.

caches, creating the opportunity for a more intelligent policy to achieve higher hit rates. Thus, the choice of the 2-GB, 16-GB, and 128-GB cache sizes enables us to examine a wide range of achievable hit-rate improvements. We also note that these three sizes are typical for in-memory hot-object caches [5] and Solid-State Drive (SSD) caches in CDN servers.

We do both training and testing of RL-Cache in conjunction with LRU as the eviction policy, because most production systems employ LRU variants. The training of RL-Cache is done on caches of the same three sizes as for the testing: 2 GB, 16 GB, and 128 GB. The cache size used in the training can be thought of as an *aggressiveness knob* for an adaptive admission policy such as RL-Cache. Smaller cache sizes force the admission policy to admit objects with a lower probability, while the opposite happens with larger cache sizes. In the initial set of our experiments, we train and test on non-overlapping portions of a particular trace. Specifically, we train RL-Cache on the first ten million requests of each trace and test the trained model on the rest of the trace. When testing RL-Cache, we choose the model that gives the highest hit rate for every cache size.

Hyperparameter Settings. The evaluation uses the following default values of the RL-Cache hyperparameters: the cache is refilled every $q = 4$ windows, $m = 250$ decision samples are generated for each window, top $p = 10\%$ of the samples are selected for the learning step, each sample covers K = 50K requests, and extra L = 150K requests are used to compute the hit rate for each decision sample with contribution factor $c = 1\%$ for the last extra request. We select these default settings based on some theoretical considerations and limited experimental checks. Later in this section, we extensively study the sensitivity of RL-Cache to hyperparameter values.

A. Average Hit Rate

We evaluate the average hit rate of the algorithms over the entire testing portion of each trace. Recall that based on our earlier active-byte characterizations for the caching needs in Figures 6 and 7, we expect the 128-GB cache to be plentiful and AdmitAll to be nearly optimal for the US1-image and US1-video traces. For these two traces on the 128-GB cache, Figures 9 and 10 demonstrate that (a) AdmitAll indeed performs strongly, (b) SecondHit and AdaptSize counterproductively reject objects and thereby yield lower hit rates, and (c) RL-Cache successfully learns the near-optimality of admitting all objects and matches the AdmitAll performance when the cache is abundant.

Upon reducing the cache size to 16 GB, and further to 2 GB, admittance of all objects becomes increasingly suboptimal, again as predicted by our active-byte characterizations. With these smaller cache sizes, Figures 9 and 10 show that AdmitAll provides lower hit rates than AdaptSize. On the other hand, RL-Cache learns a different selective admission strategy and consistently outperforms AdmitAll, SecondHit, and AdaptSize on the smaller caches. In particular for the US1-image trace, Figure 9 shows that RL-Cache outperforms AdmitAll and AdaptSize by 9.7% and 4.5% respectively on the 2-GB cache, and by 7.5% and 3.4% respectively on the 16-GB cache. This corroborates the ability of RL-Cache to learn more effectively by using a more diverse set of object features, including recency and frequency characteristics, as opposed to AdaptSize which considers only object sizes.

Based on our active-byte characterization of the US1-web trace in Figure 8, even the 128-GB cache is expected to be insufficiently large to make AdmitAll a near-optimal admission policy. Indeed, Figure 11 confirms that RL-Cache

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/JSAC.2020.3000415, IEEE Journal on Selected Areas in Communications

Request trace	US2-web	EU-video
Requests (10^6)	436.69	69.06
Unique objects (10^6)	56.42	2.52
Unique bytes (TB)	8.90	24.16
Traffic volume (Gbps)	3.07	3.68

TABLE III: Properties of the US2-web and EU-video request traces.

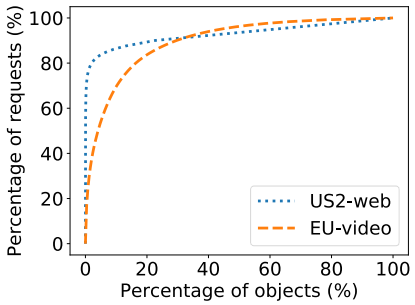


Fig. 12: Object-popularity distribution for the US2-web and EU-video traces.

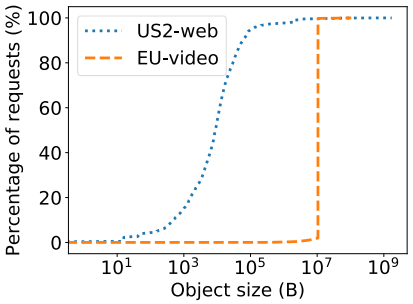


Fig. 13: Object-size distribution for the US2-web and EU-video traces.

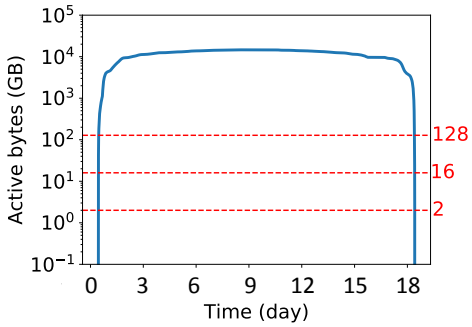


Fig. 14: Active bytes for EU-video.

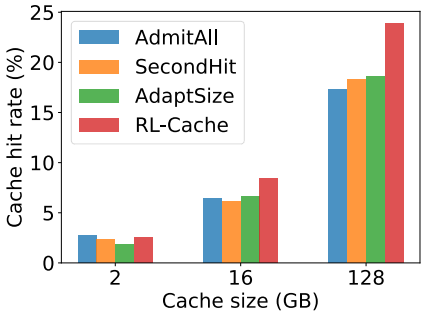


Fig. 15: Average hit rate on EU-video.

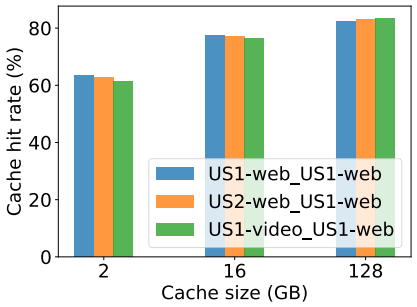
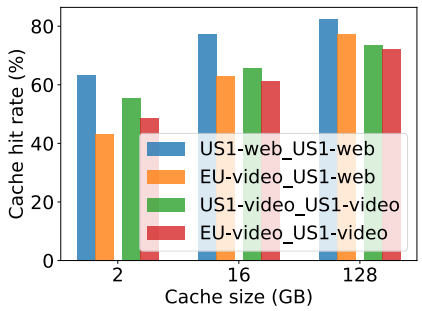


Fig. 16: Robustness within a region.



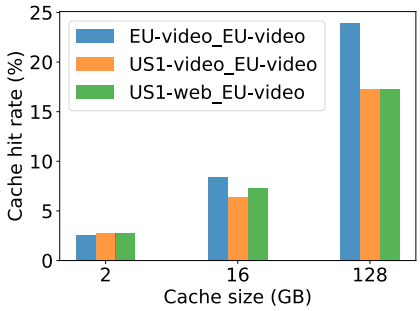


Fig. 17: Robustness of RL-Cache across geographic regions: (left) testing in the US1 location and (right) testing in the EU location.

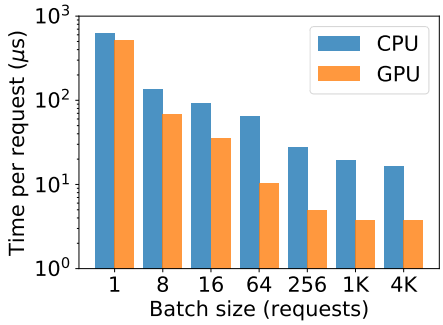


Fig. 18: Per-request neural-net processing of RL-Cache.

consistently outperforms AdmitAll for all three examined cache sizes: from 4.5% with the 128-GB cache to nearly 20% for the 2-GB cache. Figure 11 also shows that RL-Cache consistently outperforms SecondHit and performs at least as well as AdaptSize.

Overall, the above results for the three traces show that RL-Cache performs better than, or at least as well as, the state-of-the-art admission algorithms. Hence, RL-Cache is excellently suited for production settings where request patterns and cache partitions for traffic classes vary.

B. Robustness of RL-Cache

To assess the robustness of RL-Cache, we consider additional US2-web and EU-video traces characterized in Table III. Whereas US2-web is a four-day web trace from a different US-based data center than for US1-web, EU-video is an eighteen-day video trace from Ireland. Figures 12 and 13 depict respectively the object-popularity and object-size distributions for these two additional traces. In particular, Figure 13 reveals that EU-video contains mostly requests for 10-MB video chunks.

EU-video diversifies our study in regard to not only its geography and object-size distribution but also the cache size needed to achieve a good performance on a trace. Figure 14 shows that the active bytes during EU-video remain significantly above 128 GB, implying that the cache has to be much larger to achieve a high hit rate. Figure 15 corroborates this: the examined algorithms support hit rates of around 20% on the 128-GB cache, with RL-Cache outperforming the second-best AdaptSize algorithm by 6%. For the needs of EU-video, the 2-GB and 16-GB caches are too tiny as they support meaningless hit rates of just few percents.

Now, we examine the sensitivity of RL-Cache to being trained in a different geographic location and on a different traffic class. When labeling the plots for these experiments, we use format A_B where A and B refer to the training and testing traces respectively. Regardless of whether we train

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/JSAC.2020.3000415, IEEE Journal on Selected Areas in Communications

	s_j	f_j	$f_j \cdot s_j$	f_j/s_j	h_j	η_j	d_j	δ_j
s_j	1.00	-0.02	0.07	-0.20	0.03	0.01	0.02	0.00
f_j	-0.02	1.00	1.00	0.66	0.19	0.30	0.34	0.41
$f_j \cdot s_j$	0.07	1.00	1.00	0.64	0.19	0.30	0.35	0.41
f_j/s_j	-0.20	0.66	0.64	1.00	-0.14	-0.10	-0.04	0.00
h_j	0.03	0.19	0.19	-0.14	1.00	0.90	0.97	0.87
η_j	0.01	0.30	0.30	-0.10	0.90	1.00	0.93	0.98
d_j	0.02	0.34	0.35	-0.04	0.97	0.93	1.00	0.95
δ_j	0.00	0.41	0.41	0.00	0.87	0.98	0.95	1.00

A) EU-video trace

	s_j	$f_j \cdot s_j$	f_j	f_j/s_j	h_j	η_j	d_j	δ_j
s_j	1.00	0.6	0.08	-0.4	0.05	0.05	0.06	0.06
$f_j \cdot s_j$	0.6	1.00	0.84	0.47	0.35	0.42	0.47	0.50
f_j	0.08	0.84	1.00	0.86	0.41	0.49	0.55	0.58
f_j/s_j	-0.4	0.47	0.86	1.00	0.28	0.35	0.40	0.43
h_j	0.05	0.35	0.41	0.28	1.00	0.95	0.97	0.94
η_j	0.05	0.42	0.49	0.35	0.95	1.00	0.97	0.98
d_j	0.06	0.47	0.55	0.40	0.97	0.97	1.00	0.98
δ_j	0.06	0.50	0.58	0.43	0.94	0.98	0.98	1.00

B) US1-image trace

TABLE IV: Pearson correlation coefficients for all pairs of the eight features on the first 20M requests of two traces.

RL-Cache on US1-web, US2-web, or US1-video, Figure 16 shows that the hit rate on US1-web remains about the same. Hence, we can train RL-Cache in one location and run the algorithm on traces of the same or different traffic classes in other locations of the same geographic region.

We also consider scenarios where the training is done on a different continent. Figure 17 reveals that the hit rate on US1-web degrades significantly when RL-Cache is trained on EU-video rather than US1-web. The degradation is smaller when the traffic class is kept the same, as shown for the hit rate on US1-video when we train RL-Cache on EU-video rather than US1-video. Swapping the training and testing locations, Figure 17 also reports substantially lower hit rates on EU-video when RL-Cache is trained on US1-video or US1-web rather than EU-video. While the robustness across the continents is weak, the CDN can improve the scalability of its operation by training RL-Cache on a subset of the servers in the same geographic region, rather than across geographic regions.

C. Processing Overhead of RL-Cache

This section evaluates how effectively our RL-Cache implementation leverages modern multi-core CPUs and GPUs to keep the per-request neural-net processing overhead low. Figure 18 depicts the impact of the batch mode on the neural-net processing overhead. As the batch size increases, we use the same number of cores as the batch size until utilizing all the cores. Whereas the separate processing of each request takes 620 μs and 510 μs on an AMD Ryzen 7 1700X CPU (which has 16 cores with 64 threads) and GeForce GTX 1080 Ti GPU (with 3584 cores) respectively, the corresponding per-request overhead with 1024-request batches falls to 64 μs and 4 μs on the CPU and GPU. Such low per-request neural-net overhead already empowers modern cache servers to sustain their current rates of request processing. When batches are sized to 4096 requests, the per-request neural-net processing time becomes 16 μs and 4 μs for the CPU and GPU respectively.

D. Sensitivity to Features

In the above evaluation, RL-Cache uses the full set of its eight features presented in Table I. To understand how the selection of features affects the hit-rate improvements provided by RL-Cache, this section first examines correlation between the features, then quantifies feature importance, and finally assesses the RL-Cache performance on smaller sets of features.

1) Feature Correlation: Table IV-A reports Pearson correlation coefficients for all pairs of the eight features on the first 20M requests of the EU-video trace. Expectedly, frequency f_j has high positive correlation with features $f_j \cdot s_j$ and f_j/s_j that combine the frequency with size s_j of object j. Also as expected, each pair formed among four recency metrics h_j, η_j, d_j, and δ_j exhibits strong positive correlation.

Table IV-B similarly evaluates Pearson correlation coefficients for the US1-image trace. Frequency f_j is strongly correlated with either $f_j \cdot s_j$ or f_j/s_j. On the other hand, correlation between $f_j \cdot s_j$ and f_j/s_j is weaker than in the EU-video trace. Besides, size s_j is strongly correlated with $f_j \cdot s_j$ whereas such correlation does not exist in EU-video. To understand the latter result, we compare Figure 4 with Figure 13 and observe that while the object sizes in US1-image differ by two orders of decimal magnitude, almost all objects in EU-video have the same size. Hence, it is not surprising that $f_j \cdot s_j$ and s_j do not exhibit strong correlation in EU-video.

The above feature-correlation studies prompt a hypothesis that RL-Cache might be able to sustain its performance when operating with a reduced set of three features representing the size, frequency, and recency classes. Later in this section, we evaluate this hypothesis. Specifically, we consider a *basic feature set* that consists of size s_j, frequency f_j, and temporal recency h_j.

2) Feature Importance: We now assess importance of features for admission decisions made by RL-Cache. Again, we consider the EU-video and US1-image traces. The weights of the neural network unfortunately do not shed light on feature importance. To estimate the impact of each feature, we employ an algorithm that natively calculates feature importance. While decision trees constitute a reasonable choice for such estimation, their generalization ability might be insufficient for reproducing the output of the neural network, and we instead use random forests [33] as a proxy for the model. Because random forests depend on the initial seed, each run of the algorithm might produce different feature-importance values. To tackle this issue, we run random forests multiple times with different initial seeds on different portions of the trace.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/JSAC.2020.3000415, IEEE Journal on Selected Areas in Communications

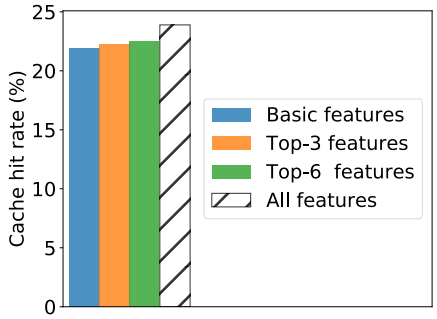


Fig. 19: Average hit rate on EU-video.

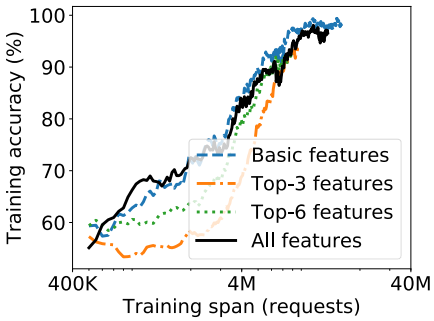


Fig. 20: Accuracy on EU-video.

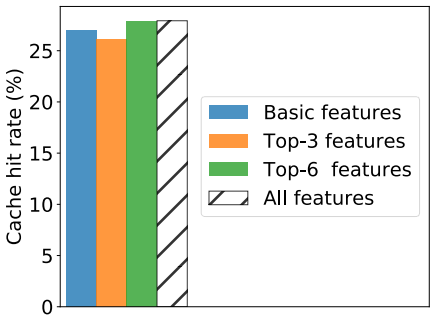


Fig. 21: Average hit rate on US1-image.

Feature	Importance (%)
η_j	15.6
d_j	12.9
f_j	12.6
h_j	12.4
δ_j	12.1
s_j	11.7
$f_j \cdot s_j$	11.5
f_j/s_j	11.2

A) EU-video trace

Feature	Importance (%)
f_j	20.9
$f_j \cdot s_j$	20.5
f_j/s_j	19.7
s_j	17.5
h_j	10.0
d_j	7.8
η_j	3.1
δ_j	0.5

B) US1-image trace

TABLE V: Usage of random forests to study feature importance on the first 1M requests of two traces.

In this classification usage of random forests, the algorithm offers a natively interpretable definition for the importance value of a feature as the average mutual information between the feature and the target. Although the importance values do not reveal which features should be selected for a high-performing feature set, these values allow us to compare features in regard to significance of their impact on made decisions.

Table V-A presents the average feature-importance values from multiple runs of random forests on the first 1M requests of the EU-video trace. For ease of interpretation, we scale the feature-importance values to add up to 100%. Because the full feature set contains eight features, the expected importance value for each feature would be 12.5% if all eight features were equally important. The actual feature-importance values for the eight features in Table V-A vary between 11.2% and 15.6%, suggesting that each of the features is fairly important. Table V-A also indicates that recency features η_j and d_j are more important for successful operation of RL-Cache on the EU-video trace than the features that involve frequency and/or size. The lowest importance of the size-based features is again not surprising because almost all objects in EU-video are of the same size.

Table V-B reports feature importance for RL-Cache on the US1-image trace. In stark contrast to the results for EU-video, the frequency-based features are the most important, and the recency-based features are the least important, for successful operation of RL-Cache. Also, the object size becomes an important feature with the US1-image trace. The unified perspective from Tables V-A and V-B justifies our design choice to consider a feature set that includes features from all three feature classes: frequency, recency, and size.

3) Reducing the Feature Set: While our feature set contains eight features, we explore the possibility of making the feature set smaller without degrading the RL-Cache performance. Because evaluating each subset of the full feature set is computationally infeasible, we leverage insights from the above feature-correlation and feature-importance studies to select and evaluate only some promising subsets. Once again, we assess how RL-Cache performs on the EU-video and US1-image traces. For either of the two traces, the assessment considers three reduced sets: (1) basic set that contains size s_j, frequency f_j, and temporal recency h_j, (2) top-3 set and, (3) top-6 set. The top-3 and top-6 subsets include features according to their rankings in the feature-importance studies, as reported in Tables V-A and V-B for EU-video and US1-image respectively. For each trace, its top-6 subset subsumes the basic feature set.

When evaluating sensitivity of the RL-Cache performance to the feature set, we track not only the hit-rate performance but also the training accuracy achieved with the feature set. A different feature set is likely to require a different effort to train the neural network with the same accuracy. We characterize the training effort via a notion of *training span* defined as the number of requests used to train the neural network. As described in Section III-E and Figure 2, the training algorithm of RL-Cache is iterative. For each window, the algorithm goes through some number of three-step iterations before the neural-network weights converge. Upon the convergence, the training algorithm slides to the next window. After each iteration for the current window of K requests, we apply the updated network to this window to predict the respective K admission decisions and compute the per-iteration accuracy of the prediction with respect to the admission-decision samples used at the learning step of the iteration. Because the number of iterations might vary from one window to another, we define *training accuracy* for a window as the average per-iteration accuracy over the same fixed number of the last iterations for this window. Because time spent per iteration, as well as the number of iterations per window, might be different in different settings, the same training span does not imply the same training time.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/JSAC.2020.3000415, IEEE Journal on Selected Areas in Communications

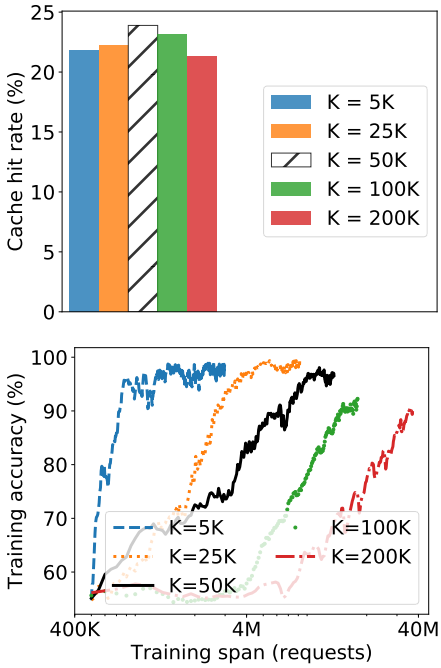


Fig. 22: Sensitivity to K, the size of a decision sample.

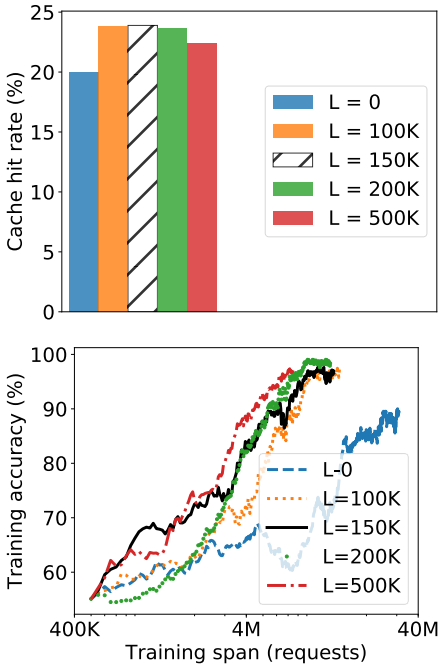


Fig. 23: Sensitivity to L, the number of extra requests per sample.

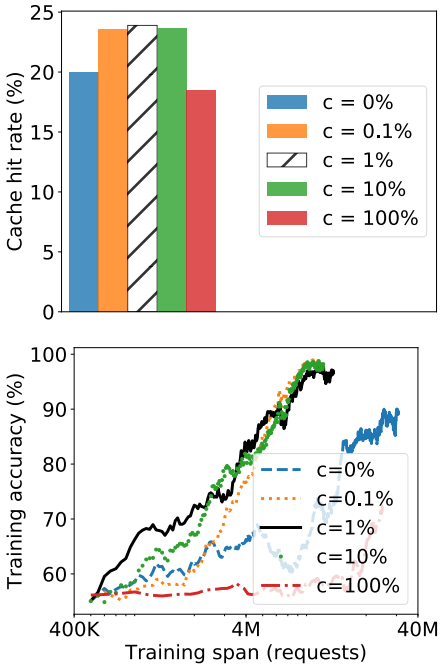


Fig. 24: Sensitivity to c, the contribution factor for the last extra request.

Figure 19 depicts sensitivity of the RL-Cache hit-rate performance on the EU-video trace to the feature set. The basic, top-3, and top-6 feature subsets support substantially lower performance than with the full set. Figure 20 shows that similar training spans are needed to achieve high training accuracy for all four examined sets of features. Thus, the full feature set provides the best performance without imposing a significantly higher training effort.

On the US1-image trace, sensitivity of the hit-rate performance and training accuracy to the feature set remains qualitatively the same. Compared to Figure 19, Figure 21 for US1-image exposes an interesting difference that RL-Cache performs better with the basic feature set than top-3 set. The result illustrates that when the size of the feature set is fixed, forming the set from the most important features does not assure the best performance.

Overall, the above sensitivity studies validate our design choice of using the broad set of eight features. On different traces, different subsets of the features become important for RL-Cache to sustain its high hit-rate performance. For each trace, RL-Cache automatically discerns the important features and focuses on them to maximize the performance. Furthermore, the training effort for the full feature set is comparable to those for reduced feature sets.

E. Sensitivity to Hyperparameter Values

After using a variety of real-world traces to show that RL-Cache outperforms existing cache admission algorithms in the default settings of its hyperparameters, we now evaluate the sensitivity of the RL-Cache performance to the choice of hyperparameter values. In particular, we examine how much the RL-Cache hit rate changes when each of its six hyper-parameters departs from its default value identified below: (1) $K = 50$K requests in a decision sample, (2) $L = 150$K extra subsequent requests for computing the hit rate per sample, (3) contribution factor $c = 1\%$ for the last extra request, (4) $m = 250$ generated decision samples, (5) $p = 10\%$ of the highest hit-rate samples selected for learning, and (6) $q = 4$ subsequent windows after which the cache is refilled according to the latest learned strategy. While changing one hyperparameter, we keep the other hyperparameters fixed at their default values. By analyzing the sensitivity, we explore both whether the default hyperparameter values constitute a reasonable configuration and whether the RL-Cache design really needs these hyperparameters. We experiment with the EU-video trace on the 128-GB cache and use bars with striped lines to depict the default hyperparameter settings.

(K) Decision-sample size. We depart from the default value of K = 50K requests to consider two smaller values of 5K and 25K requests, and two larger values of 100K and 200K requests. Figure 22 shows that whereas larger values of K consistently require longer training spans to achieve the same accuracy level, the intermediate default setting of K = 50K requests yields the highest hit rate. In particular, while using the smallest sample size of K = 5K requests enables accurate training of the neural network on the smallest number of requests, e.g., reaching the 95% accuracy on the training span of 800K requests, this desirable property does not translate into the best hit-rate performance.

The plots suggest that decision samples should be sufficiently large for the trained network to make near-optimal admission decisions in terms of the hit rate. On the other hand, even the long training span of 20M requests is too short for the larger K values of 100K and 200K requests to surpass even the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/JSAC.2020.3000415, IEEE Journal on Selected Areas in Communications

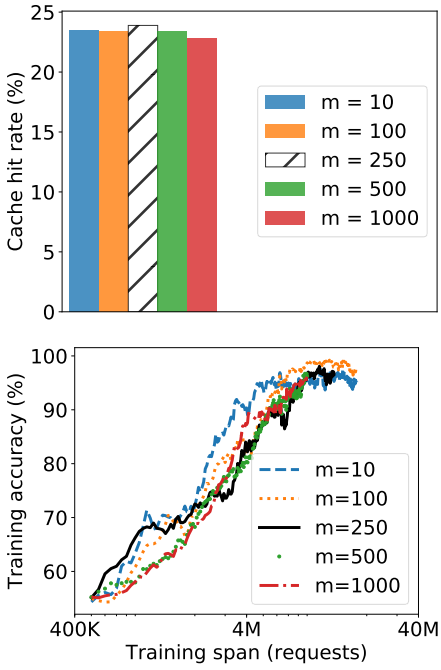


Fig. 25: Sensitivity to m, the number of generated decision samples.

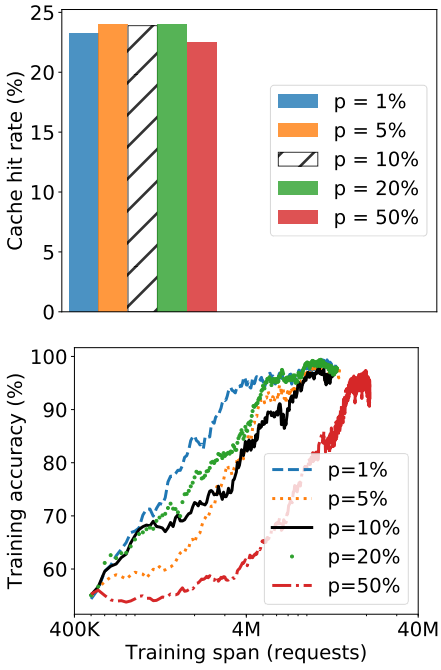


Fig. 26: Sensitivity to p, the percentage of the samples selected for learning.

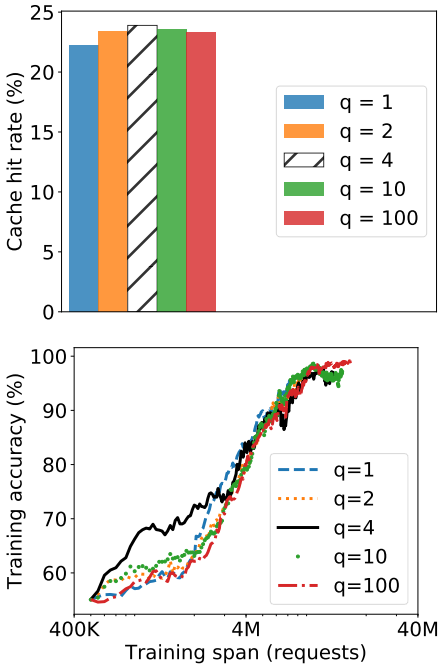


Fig. 27: Sensitivity to q, the number of subsequent windows for the cache refill.

90% accuracy, and the insufficient accuracy prevents the neural network from maximizing the hit rate. Hence, the choice of the decision-sample size poses a trade-off between the training effort and hit rate provided by the accurately trained network.

(L) Number of extra requests per sample. Figure 23 reports the hit-rate performance and training accuracy when the number of extra requests for computing the hit rate varies from 0 to 500K requests. The setting with L = 0 is especially interesting because it is equivalent to not having hyperparameter L at all. Compared to the other examined settings, L = 0 is inferior in terms of both hit-rate performance and training accuracy: Figure 23 reveals the lowest average hit rate and longest training span needed to reach the same accuracy level. Selecting the best samples based only on the K requests of the current window makes such myopic learning ineffective because it ignores future hit consequences of the current admission decisions. This result validates our design choice of introducing hyperparameter L and considering extra subsequent requests to calculate the hit rate for decision samples.

While the pattern of the accuracy convergence stays qualitatively similar across all considered positive values of L, Figure 23 exhibits the highest hit rate in the default setting of L = 150K requests. The lower hit rates for the larger L values of 200K and 500K requests indicate that not only being myopic but also looking too far ahead is counterproductive. We attribute this effect to the accumulating inability to reliably predict how the current admission decisions influence the hit consequences that lie too far into the future.

(c) Contribution factor for the last extra request. We examine the following five values of hyperparameter c: 0, 0.1%, 1% (default), 10%, and 100%. The case of $c = 0$ is equivalent to the setting with $c = 1\%$, K= 50K requests, and $L = 0$, for which we already observed poor hit-rate and accuracy-convergence behaviors in the sensitivity analysis for hyperparameter L above. The other extreme of $c = 100\%$ is the same as the setting with $c = 1\%$, K= 200K requests, and $L = 0$, which deviates from the default setting in two dimensions: increase of K from 50K to 200K requests and decrease of L from 150K requests to 0. The above sensitivity analyses for hyperparameters K and L already showed that each of these individual deviations was detrimental, creating the expectation that their combination should lead to even worse results. Indeed, Figure 24 shows that the setting of $c = 100\%$ exhibits the slowest growth of the training accuracy and results in the lowest hit-rate performance. On the other hand, the intermediate c values around the default setting of 1% demonstrate qualitatively similar accuracy-convergence profiles and deliver similarly high hit rates, with the default value of $c = 1\%$ providing the highest hit rate.

(m) Number of generated decision samples. As we increase the number of generated samples from 10 to 1K, Figure 25 unveils a surprisingly steady picture. Despite our expectation that larger values of m should diversify the pool of samples to select from and thereby make the learning more effective, the highest hit rate is achieved around the intermediate default setting of $m = 250$ samples. We explain the result by observing that generating more samples has the side effect of selecting more samples for learning: the additionally selected samples might have lower hit rates, which undermines the ability of the algorithm to learn admission behaviors optimizing the hit-rate performance.

For all examined values of m, Figure 25 also shows qualitatively similar profiles of the accuracy convergence. Because

a smaller number of generated decision samples induces lower computational overhead on each step of any sampling-selection-learning iteration, smaller m values are preferable. Overall, the sensitivity analysis points toward generating a relatively small number of decision samples.

(p) Percentage of the samples selected for learning. The selection step of the RL-Cache training algorithm segregates the top p^{th} percentile of the generated decision samples with the highest hit rates and utilizes the segregated samples for learning. Figure 26 reports the average hit rate and training-accuracy dynamics for the p values of 1%, 5%, 10% (default), 20%, and 50%. Selecting the top half of the generated samples yields the lowest average hit rate. This is consistent with our above observation for hyperparameter m: expanding the segregated pool with samples that have lower hit rates interferes with effective learning of an admission strategy that maximizes the average hit rate. Besides, the setting of $p = 50\%$ exhibits the slowest accuracy convergence: not only the training span needs to be the longest for the accuracy to rise to the same level but also the computational overhead of any sampling-selection-learning iteration is the highest due to dealing with the largest number of samples at the learning step.

On the other hand, the smallest examined setting of $p = 1\%$ results in a low average hit rate as well. We attribute this to overfitting to the small set of decision samples and, consequently, not being able to predict future near-optimal admission decisions reliably. The sensitivity study indicates that intermediate p settings around the default value of 10% are preferable because of providing a high average hit rate without imposing a large training overhead.

(q) Number of subsequent windows for the cache refill. Hyperparameter q specifies how frequently the RL-Cache training algorithm refills the cache to keep the cache state close to what it would have been if the most recently learned admission strategy were used from the very beginning. The design incorporates this hyperparameter with the expectation that smaller q values should yield higher hit rates. Figure 27 depicts a rather different outcome. While the larger q values of 10 and 100 windows indeed provide lower hit rates than the default q setting of 4 windows, the hit rate is the lowest with the smallest q value of 1 window. Across all examined q values, the qualitative pattern of the accuracy convergence remains similar. Because simulating a cache refill imposes an extra computational overhead, larger q values are preferable. Overall, refreshing the cache state every 4 windows or so seems appropriate.

To sum up, the above extensive sensitivity analyses justify the relevance of all six hyperparameters and corroborate that their default values constitute a near-optimal configuration for RL-Cache. The sensitivity studies also offer various interesting insights into the RL-Cache behavior. Sizing of the window presents a trade-off between the training effort and hit rate delivered by the accurately trained network. The consideration of additional subsequent requests to compute the hit rates of decision samples improves the average hit-rate performance. However, looking too far ahead into the future is counterproductive. The sampling and selection steps of the RL-Cache training algorithm are subject to a balance between (a) identifying samples with sufficiently high hit rates to enable effective learning that maximizes the average hit rate and (b) keeping the pool of such samples sufficiently diverse to avoid overfitting. While the cache refilling helps in general, it should not be done for every window.

V. Conclusion

This paper designed and evaluated RL-Cache, an algorithm that applies model-free RL to cache admission in an edge CDN server. RL-Cache relies on direct policy search that combines MC sampling with stochastic optimization to maximize the cache hit rate. The algorithm considers a broad set of features including the object's size, frequency, and recency characteristics. Our publicly available RL-Cache implementation supports batch processing of requests to keep the processing overhead low. Our evaluation used Akamai's production traces from the image, video, and web traffic classes. We introduced the notion of active bytes to characterize the cache size needed to achieve a high hit rate on a trace. Our results for different cache sizes showed that RL-Cache performed better than, or at least as well as, state-of-the-art admission algorithms. Thus, RL-Cache is highly suitable for production settings where request patterns and cache partitions for traffic classes vary.

We extensively evaluated sensitivity of RL-Cache to various factors, such as the location and traffic class used for its training and/or execution. The evaluation showed that the CDN can operate scalably by training RL-Cache in one location and running the algorithm on traces of the same or different traffic classes in other locations of the same geographic region. Also, after assessing feature correlation and feature importance, we considered promising reduced sets of object features and determined that RL-Cache achieves the best hit-rate performance when using the full set of its eight features.

Our analysis of the RL-Cache sensitivity to its hyperparameter values unveiled interesting insights into the algorithm behavior. We showed that choosing the window size was subject to a trade-off between the training effort and hit rate provided by the accurately trained network. While RL-Cache improved its average hit-rate performance by involving extra subsequent requests into calculation of the hit rates for decision samples, looking too far into the future had a negative effect. The training algorithm faced the tension between the need to learn from samples with sufficiently high hit rates and the danger of overfitting when the pool of samples selected for the learning step was insufficiently diverse. We also demonstrated that refilling of the cache according to the latest learned admission strategy was useful if not done too frequently.

This paper constitutes a good first step that opens ways for generalizing the RL-Cache algorithm to cache eviction, distributed caching in multiple servers, and joint optimization of caching and cache deployment [34]. Complementing the reported results on feature importance, we plan to work on better explainability of RL-Cache admission decisions via techniques used in [35] and more recent approaches [23].

REFERENCES

[1] F. Chen, R. K. Sitaraman, and M. Torres, "End-User Mapping: Next Generation Request Routing for Content Delivery," in *SIGCOMM*, 2015.

[2] K. Morales and B. K. Lee, "Fixed Segmented LRU Cache Replacement Scheme with Selective Caching," in *IPCCC*, 2012.

[3] L. Cherkasova, "Improving WWW Proxies Performance with Greedy-Dual-Size-Frequency Caching Policy," HP, Tech. Rep., 1998.

[4] B. M. Maggs and R. K. Sitaraman, "Algorithmic Nuggets in Content Delivery," in *SIGCOMM*, 2015.

[5] D. Berger, R. K. Sitaraman, and M. Harchol-Balter, "AdaptSize: Orchestrating the Hot Object Memory Cache in a Content Delivery Network," in *NSDI*, 2017.

[6] E. Nygren, R. K. Sitaraman, and J. Sun, "The Akamai Network: A Platform for High-performance Internet Applications," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 3, 2010.

[7] F. Velazquez, K. Lyngstol, T. F. Heen, and J. Renard, "The Varnish Book for Varnish 4.0." Varnish Software AS, 2016.

[8] B. Fitzpatrick and M. Community, "Memcached," GitHub, 2019, https://github.com/memcached/memcached.

[9] W. Reese, "Nginx: The High-Performance Web Server and Reverse Proxy," *Linux J.*, vol. 2008, no. 173, 2008.

[10] A. Narayanan, S. Verma, E. Ramadan, P. Babaie, and Z.-L. Zhang, "DeepCache: A Deep Learning Based Framework For Content Caching," in *NetAI*, 2018.

[11] K. Suksomboon *et al.*, "PopCache: Cache More or Less Based on Content Popularity for Information-Centric Networking," in *LCN*, 2013.

[12] V. Kirilin, "RL-Cache," GitHub, 2019, https://github.com/WVadim/RL-Cache.

[13] A. Sundarrajan, M. Feng, M. Kasbekar, and R. K. Sitaraman, "Footprint Descriptors: Theory and Practice of Cache Provisioning in a Global CDN," in *CoNEXT*, 2017.

[14] V. Kirilin, A. Sundarrajan, S. Gorinsky, and R. K. Sitaraman, "RL-Cache: Learning-Based Cache Admission for Content Delivery," in *NetAI*, 2019.

[15] G. B. Mathews, "On the Partition of Numbers," *Proceedings of the London Mathematical Society*, vol. 28, 1897.

[16] G. Einziger, R. Friedman, and B. Manes, "TinyLFU: A Highly Efficient Cache Admission Policy," arXiv:1512.00727v2, 2015.

[17] V. Fedchenko, G. Neglia, and B. Ribeiro, "Feedforward Neural Networks for Caching: Enough or Too Much?" arXiv:1810.06930v1, 2018.

[18] J. Cobb and H. ElAarag, "Web Proxy Cache Replacement Scheme Based on Backpropagation Neural Network," *J. of Systems and Software*, vol. 81, 2008.

[19] H. Khalid and M. S. Obaidat, "KORA-2: A New Cache Replacement Policy and Its Performance," in *ICECS*, 1999.

[20] D. Berger, "Towards Lightweight and Robust Machine Learning for CDN Caching," in *HotNets*, 2018.

[21] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. MIT Press, 2018.

[22] Y. Kazak, C. Barrett, G. Katz, and M. Schapira, "Verifying Deep-RL-Driven Systems," in *NetAI*, 2019.

[23] A. Dethise, M. Canini, and S. Kandula, "Cracking Open the Black Box: What Observations Can Tell Us About Reinforcement Learning Agents," in *NetAI*, 2019.

[24] E. Liang, H. Zhu, X. Jin, and I. Stoica, "Neural Packet Classification," in *SIGCOMM*, 2019.

[25] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh, "Learning Scheduling Algorithms for Data Processing Clusters," in *SIGCOMM*, 2019.

[26] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, "Deep Reinforcement Learning: A Brief Survey," *Signal Processing M.*, vol. 34, no. 6, 2017.

[27] C. J. C. H. Watkins, "Learning from Delayed Rewards," Ph.D. dissertation, University of Cambridge, 1989.

[28] R. Y. Rubinstein and D. P. Kroese, *The Cross-Entropy Method: A Unified Approach to Combinatorial Optimization, Monte-Carlo Simulation, and Machine Learning*. Springer, 2004.

[29] P. Ramachandran, B. Zoph, and Q. V. Le, "Searching for Activation Functions," arXiv:1710.05941, 2017.

[30] K. Cobbe, O. Klimov, C. Hesse, T. Kim, and J. Schulman, "Quantifying Generalization in Reinforcement Learning," arXiv:1812.02341v3, 2019.

[31] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning Representations by Back-Propagating Errors," *Nature*, vol. 323, no. 6088, 1986.

[32] M. Abadi *et al.*, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems," arXiv:1603.04467v2, 2016.

[33] T. K. Ho, "Random Decision Forests," in *ICDAR*, 1995.

[34] S. Hasan, S. Gorinsky, C. Dovrolis, and R. K. Sitaraman, "Trade-offs in Optimizing the Cache Deployments of CDNs," in *INFOCOM*, 2014.

[35] M. T. Ribeiro, S. Singh, and C. Guestrin, ""Why Should I Trust You?": Explaining the Predictions of Any Classifier," in *KDD*, 2016.

Vadim Kirilin is a Machine Learning Engineer at Yandex LLC, Russia. He received an M.S. degree from Carlos III University of Madrid, Spain in 2017 and B.S. degree from Lomonosov Moscow State University, Russia in 2015. His primary research interests are foundations of machine learning, neural network pruning, contextual bandits, and reinforcement learning.

Aditya Sundarrajan is a Research Scientist at Facebook, Inc., USA. Dr. Sundarrajan received his Ph.D. degree from the University of Massachusetts Amherst, USA in 2020, M.S. degree from the University of Arizona, USA in 2013 and B.E. degree from Anna University, India in 2010. His primary research interests are distributed systems, green computing, content delivery networks, and machine learning.

Sergey Gorinsky is a tenured Research Associate Professor at IMDEA Networks Institute, Spain, where he leads the NetEcon research group. Dr. Gorinsky received his Ph.D. and M.S. degrees from the University of Texas at Austin, USA in 2003 and 1999 respectively and Engineer degree from Moscow Institute of Electronic Technology, Russia in 1994. From 2003 to 2009, he served on the tenure-track faculty at Washington University in St. Louis, USA. Sergey Gorinsky graduated four Ph.D. students. The areas of his primary research interests are computer networking, distributed systems, and network economics. Sergey Gorinsky made research contributions to real-time scheduling, buffer sizing, economics of network interconnection, service differentiation, cache deployment, multicast, congestion control, networking education, routing, bulk data transfer, and machine learning for caching. He served as a TPC chair of ICNP 2017 and other conferences, as well as a TPC member for a much broader conference population including SIGCOMM, CoNEXT, and INFOCOM. Sergey Gorinsky contributed to conference organization in many roles, such as a general chair of SIGCOMM 2018. He also served as an evaluator of research proposals and projects for the European Research Council (ERC StG), European Commission (Horizon 2020, FP7), and numerous other funding agencies.

Ramesh K. Sitaraman is a Professor in the College of Information and Computer Sciences at the University of Massachusetts at Amherst. His research focuses on Internet-scale distributed systems, including algorithms, architectures, performance, energy efficiency, security, and economics. As a principal architect, he helped create the Akamai Content Delivery Network (CDN), the world's first major CDN that currently delivers a significant fraction of the Internet traffic. He retains a part-time role as Akamai's Chief Consulting Scientist. Prof. Sitaraman is a recipient of the inaugural ACM SIGCOMM Networking Systems Award for his work on the Akamai CDN, DASH-IF Excellence in DASH award for his work on ABR algorithms, an NSF CAREER Award, a College of Natural Sciences Outstanding Teacher Award, and a Lilly Fellowship. He received a B.Tech from the Indian Institute of Technology, Madras and a Ph.D. in computer science from Princeton University. He is a Fellow of the ACM and the IEEE.