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

Vadim Kirilin[§*], Aditya Sundarrajan[†*], Sergey Gorinsky[§], and Ramesh K. Sitaraman[†$]
[§]IMDEA Networks Institute, Spain [†]UMass Amherst, USA [$]Akamai Technologies, USA

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 decide 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.

CCS CONCEPTS

• **Networks** → **Network services**; • **Computing methodologies** → **Reinforcement learning**.

KEYWORDS

Content delivery network; caching; cache admission; hit rate; object feature; feedforward neural network; Monte Carlo method; batch processing; traffic class; image; video, web; production trace.

ACM Reference Format:
Vadim Kirilin, Aditya Sundarrajan, Sergey Gorinsky, and Ramesh K. Sitaraman. 2019. RL-Cache: Learning-Based Cache Admission for Content Delivery. In *NetAI 2019: ACM SIGCOMM 2019 Workshop on Network Meets AI & ML, August 23, 2019, Beijing, China.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3341216.3342214

*Both authors contributed equally to the paper. The first author is now with Yandex LLC, Russia.

NetAI 2019, August 23, 2019, Beijing, China

ACM ISBN 978-1-4503-6872-8/19/08…$15.00
https://doi.org/10.1145/3341216.3342214

1 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 [4]. 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 used object. Major CDNs employ LRU and its variants, such as Segmented LRU (SLRU) [15], 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) [5]. The work on admission algorithms is less extensive and includes SecondHit [13] and AdaptSize [3].

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 [17], Varnish [23], Memcached [9], and NGINX [18] – 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 [16] and PopCache [21].

Our Contributions. We formulate cache admission as a Reinforcement Learning (RL) problem solvable with Monte Carlo methods. 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-admission front ends already exist in practice, e.g., Akamai's Bloom-filter implementation of SecondHit [13].

We design RL-Cache, a cache-admission algorithm that trains a feedforward neural network which outputs a binary decision upon receiving a request for an object: 1 (admit) or 0 (do not admit). Training of the neural network on request traces from CDN servers is a non-real-time and computationally intensive task. We expect the training to be periodically performed in the cloud. However, after the training phase, RL-Cache can efficiently process batches of requests and make admission decisions with small computational overhead, e.g., per-request processing time of 16 μs and 4 μs on CPU and GPU platforms respectively with batches of 4096 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 [22]. 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, our evaluation shows 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. Our evaluation also demonstrates robustness of RL-Cache in the sense that RL-Cache 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.

In summary, RL-Cache is a promising approach to improving 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.

Roadmap. We organize the rest of this paper as follows. Section 2 provides background and discusses related work. Section 3 presents our RL-Cache algorithm. Section 4 empirically evaluates the proposed scheme. Section 5 concludes the paper with a summary of its contributions.

2 BACKGROUND AND RELATED WORK

Caching is related to the knapsack problem [14] 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 [22].

Existing work in caching predominantly focuses on design of eviction policies, e.g., LRU, SLRU, TLRU, S4LRU, GDSF, ARC, and Cliffhanger (cf. Table 2 in [3]). 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 [13], 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 object 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 [7]. AdaptSize [3] is a size-based admission policy that uses a Markov model to adjust a threshold for the size of admitted objects. In contrast to prior work that uses one or two object features, RL-Cache combines a broader set of eight features that use the object's size, request recency, and frequency characteristics 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 [16]. PopCache caches objects with popularity-dependent probabilities [21]. FNN-based caching [8], NNPCR-2 [6], and KORA-2 [11] also rely on popularity prediction. LFO [2] 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. Specifically, Monte Carlo methods of RL support optimization of the hit rate by learning directly from sample sequences of admission decisions. Further, RL-Cache focuses on cache admission that 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 [16] or modification of the eviction order based on object popularity [6].

3 RL-CACHE ADMISSION ALGORITHM

To decide whether to cache a requested object, RL-Cache uses a feedforward neural network that computes admission probability $A(u, w) \in [0, 1]$ as a function of features u of the object and weights w of the neural network. The features of an object include metrics related to its size as well as frequency and recency of its requests. Once the neural network is trained, RL-Cache computes admission probability $A(u, w)$ for each received request and then rounds it to make a binary decision of 1 (admit) or 0 (do not admit the requested object).

3.1 Feature Selection

First, we select features u that characterize each requested object. Traditional caching heuristics describe an object with its request

Notation	Meaning
s_j	Size of object j in bytes
f_j	Frequency, the fraction of requests for object j among all requests so far
r_j	Temporal recency, time in seconds since the previous request for object j
ρ_j	Exponential smoothing of r_j so far
d_j	Ordinal recency, the number of requests since the previous request for object j
δ_j	Exponential smoothing of d_j 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 1: Features of an object in our model.

recency, frequency, and object size and use these features in isolation or combination, e.g., LRU (recency), SecondHit (frequency), AdaptSize (size), and GDSF (frequency and size). The strength of our approach is in simultaneously considering a broad set of eight features (cf. Table 1) that capture various aspects of recency, frequency, and size to make admission decisions.

3.2 Training Algorithm

Now, we show how to train our feedforward neural network using a Monte Carlo approach. The training objective is to learn weights w of the network which computes admission probability function $A(u, w)$ for an object with features u. The training data set is a sequence of object requests i characterized by their features u_i. To keep the training overhead manageable, we traverse this sequence of requests by applying a sliding window. We initially position the window to start at the first request and iteratively slide the window forward by K requests at a time, until the training algorithm considers all requests in the trace. As the window slides along the trace, the algorithm learns weights w more accurately. Because the less accurate weights learned during previous windows affect the cache state, and the cumulative effect of these inaccuracies might undermine the training effectiveness, we refill the cache after every q windows by simulating admission decisions under the current weights for all the requests preceding the current window. For each window, our training algorithm performs the following four steps.

1) Sampling: Let w be the current weights and u_i denote the features of the i^{th} request in the current window, where $1 \le i \le K$. For each of these K requests, we admit the requested object with probability $A(u_i, w)$, i.e., the non-admission probability for request i is $1 - A(u_i, w)$. As a result, we receive a decision sample for the K requests. Altogether, we generate m such samples, where m is significantly smaller than 2^K which is the number of all possible samples.

2) Selection: Whereas our objective is to train the neural network to maximize the hit rate of the cache, we now select the p^{th} percentile of the m decision samples that produce the highest hit rates. Because the admission decisions in a K-request decision sample also affect the cache performance beyond the decision sample, we also simulate admission decisions for the L subsequent requests to compute the hit rate over the extended window of $K + L$ requests. For each request i such that $K < i \le K + L$, we multiply its contribution to the hit rate by γ^{i-K}, where $0 < \gamma < 1$, to give diminishing importance to requests that are farther into the future. We consider such extended windows of $K + L$ requests solely for the sample selection. Note that consecutive windows overlap because the window size exceeds the number of requests by which the window advances each time.

3) Learning: Utilizing the K-request samples selected at the previous step, we update weights w. Specifically, we train the neural network with a backpropagation algorithm [20] which uses binary cross-entropy loss as the loss function.

4) Termination check: If the absolute change in weights w falls below threshold ϵ, the algorithm terminates. Otherwise, we repeat the above three steps.

3.3 Implementation

We implement RL-Cache using the TensorFlow library [1] without requiring any extensions. Our implementation is publicly available with open access at its GitHub repository [12].

In the main mode envisioned for RL-Cache online operation, the CDN periodically trains the neural network on cloud infrastructure and sends the most recently trained network to the cache server. The server-side implementation of RL-Cache 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 allows for configurations that leverage 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 wide area network, with typical fetching latency above 100 ms. Further, RL-Cache can make admission decisions *asynchronously* with serving the requested object to the user, since the server can cache 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.

4 EMPIRICAL EVALUATION

To evaluate RL-Cache, we start with three traces collected over a period of 4 days from a US-based edge server in Akamai's production network. These US1-image, US1-video, and US1-web traces represent the image, video, and web traffic classes respectively. Table 2 sums up characteristics of the three request traces.

We demonstrate the diversity of the considered traffic classes by plotting the object-popularity and object-size distributions for

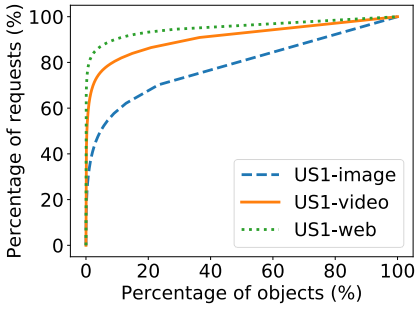

Figure 1: Object-popularity distribution for US1-image, US1-video, and US1-web.

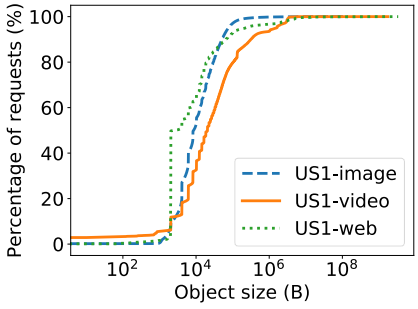

Figure 2: Distribution of object sizes for US1-image, US1-video, and US1-web.

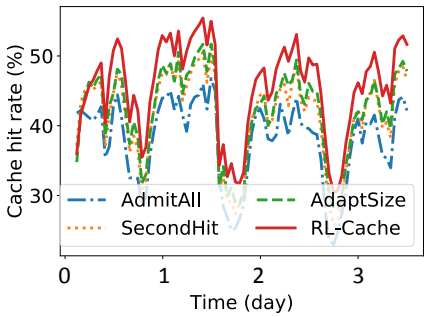

Figure 3: Dynamics of the hit rate for the US1-image trace on the 16-GB cache.

Request trace	US1-image	US1-video	US1-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 2: Characteristics of US1-image, US1-video, and US1-web.

each of the traces. For US1-video and US1-web, Figure 1 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 2 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. We train RL-Cache with LRU as the eviction algorithm, because most production systems employ LRU variants. RL-Cache is trained on three cache sizes: 2 GB, 16 GB, and 128 GB that correspond to typical in-memory hot-object caches [3] and Solid-State Drive (SSD) caches in CDN servers. The cache size used in the training can be thought of as an *aggressiveness knob* for the admission algorithm. Smaller cache sizes force the admission algorithm to admit objects with a lower probability, while the opposite happens with larger cache sizes. The optimal admission policy for an infinitely large cache is to admit all objects. We refer to this admission policy as AdmitAll; this is a common basic admission policy for eviction-focused caching algorithms. Thus, the advantages of any sophisticated admission policy decrease as the cache size increases. We train RL-Cache on the first ten million requests of each trace and test the trained model on the rest of the trace, i.e., the training and testing sets do not overlap. When testing RL-Cache, we choose the model that gives the highest hit rate for every cache size. The testing is done on caches sized to 2 GB, 16 GB, and 128 GB. We compare RL-Cache with AdmitAll, frequency-based SecondHit, and adaptive size-based AdaptSize.

Parameter Settings. The values of the RL-Cache parameters described in Section 3 are chosen as follows: $q = 4$ windows; $m = 250$ decision samples are generated for each window; top $p = 20\%$ of the samples are selected for the learning step; each sample covers $K = 25{,}000$ requests; extra $L = 175{,}000$ requests are used to compute the hit rate for each window; factor γ for the discounted hit-rate contributions is 0.99997; threshold ϵ for the per-window termination check is set to 10^{-6}.

4.1 Hit-Rate Performance

Figure 3 illustrates hit-rate dynamics for the US1-image trace on the cache sized to 16 GB, where the hit rate is computed over 1-hour intervals. RL-Cache successfully adapts to the diurnal patterns in image requests and consistently outperforms AdmitAll, SecondHit, and AdaptSize under the varying load.

We now turn our attention to the average hit rate of the algorithms over the entire testing trace. For US1-image, Figure 4 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 benefit from using a more diverse set of object features, including recency and frequency characteristics, as opposed to AdaptSize which considers only object sizes. With the cache sized to 128 GB, RL-Cache outperforms both SecondHit and AdaptSize and performs similarly to AdmitAll.

To understand the observed performance, 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 avoidable misses (i.e., those upon the second and all subsequent requests for each object) by the offline algorithm that uses AdmitAll for cache admission and evicts every object upon its last request in the trace. Figure 7 reveals that the active bytes remain significantly below 128 GB throughout the US1-image trace, suggesting that the AdmitAll admission should be nearly optimal on the 128-GB cache when combined with LRU eviction as well. Correspondingly, Figure 4 confirms that, unlike SecondHit or AdaptSize, RL-Cache successfully learns that admitting all objects is nearly optimal when the cache is abundant for the needs of US1-image. On the other hand, the active bytes in Figure 7 exceed 2 GB and even 16 GB during

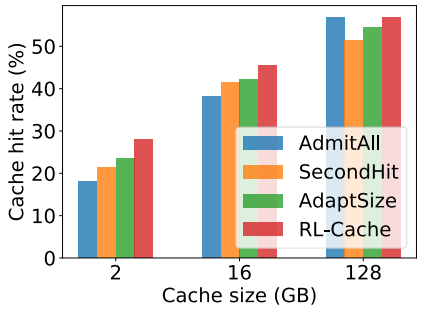

Figure 4: Average hit rate for US1-image.

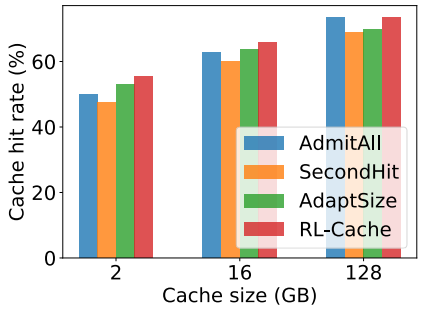

Figure 5: Average hit rate for US1-video.

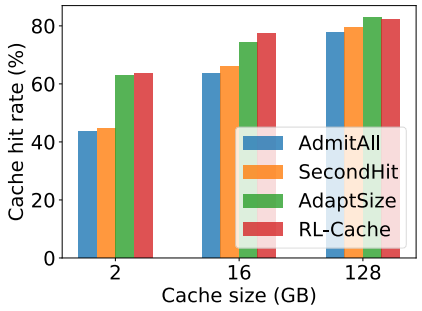

Figure 6: Average hit rate for US1-web.

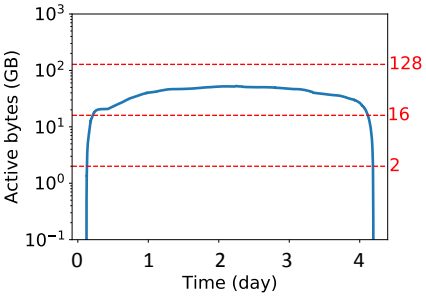

Figure 7: Active bytes for US1-image.

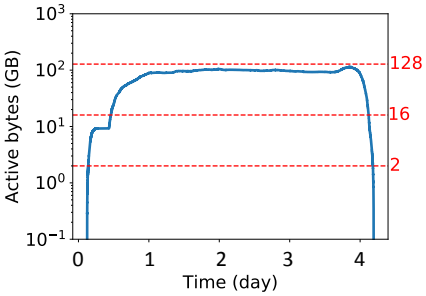

Figure 8: Active bytes for US1-video.

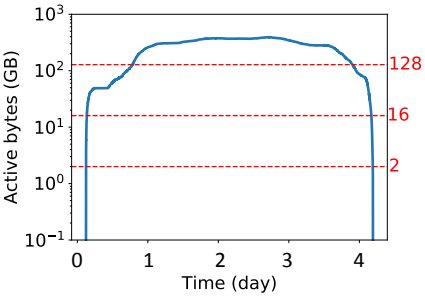

Figure 9: Active bytes for US1-web.

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 3: Properties of US2-web and EU-video.

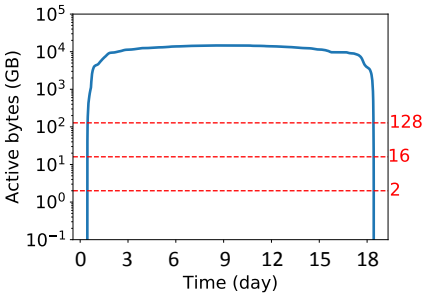

Figure 10: Active bytes for EU-video.

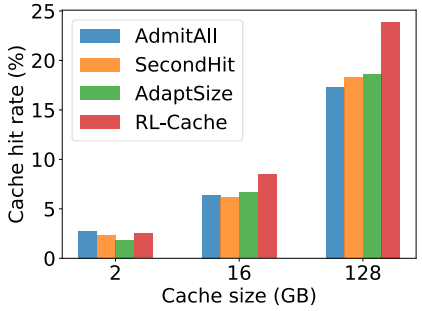

Figure 11: Average hit rate for EU-video.

most of the trace, except the expected sharp rise and drop at the start and finish respectively. Thus, when the cache size decreases to 16 GB and further to 2 GB, RL-Cache learns different admission strategies and outperforms not only SecondHit and AdaptSize but also AdmitAll, with relatively larger gains on smaller caches.

For US1-video, Figures 5 and 8 depict the same qualitative behavior as for US1-image. RL-Cache consistently outperforms SecondHit and AdaptSize. Also, RL-Cache beats AdmitAll by 5.6% and 2.9% on the 2-GB and 16-GB cache respectively and performs similarly to AdmitAll on the 128-GB cache which, as Figure 8 indicates, is plentiful for the needs of US1-video, making AdmitAll nearly optimal for the given combination of the request trace and cache size.

For US1-web, the qualitative picture changes. As Figures 6 and 9 show, the active bytes exceed 128 GB during most of the trace, and 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. RL-Cache also 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.

4.2 Robustness of RL-Cache

To assess the robustness of RL-Cache, we consider additional US2-web and EU-video traces characterized in Table 3. Whereas US2-web is a 4-day web trace from a different US-based data center than for US1-web, EU-video is a 18-day video trace from Ireland.

EU-video diversifies our study in regard to not only geography but also the cache size needed to achieve a good performance on a trace. Figure 10 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 11 corroborates this: the examined algorithms support hit rates of around 20% on the 128-GB cache, with RL-Cache outperforming the second-best AdaptSize

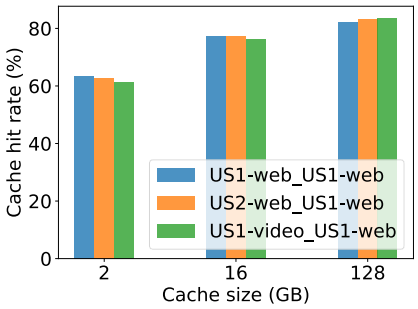

Figure 12: Robustness of RL-Cache within a geographic region.

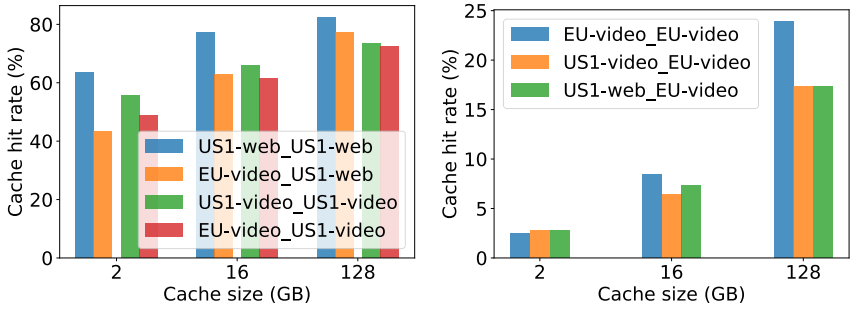

Figure 13: Robustness of RL-Cache across geographic regions: (left) testing in the US1 location and (right) testing in the EU location.

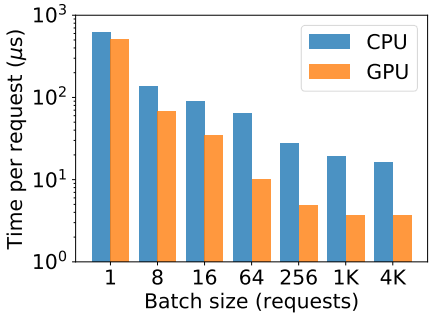

Figure 14: Per-request neural-net processing of RL-Cache.

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 RL-Cache on US1-web, US2-web, or US1-video, Figure 12 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 13 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 13 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.

4.3 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 14 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.

5 CONCLUSION

This paper proposed RL-Cache, an RL-based algorithm for cache admission in a CDN server. RL-Cache uses a Monte Carlo method to train a feedforward neural network for maximizing cache hit rates. 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 also studied robustness of RL-Cache and 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. This paper is a good first step that opens avenues for generalizing the RL-Cache approach to cache eviction, distributed caching in multiple servers, and joint optimization of caching and cache deployment [10]. Another direction for future work is to analyze feature contributions and explain RL-Cache decisions with tools such as LIME [19].

ACKNOWLEDGMENTS

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).

REFERENCES

[1] M. Abadi et al. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. URL : www.tensorflow.org.

[2] D. Berger. 2018. Towards Lightweight and Robust Machine Learning for CDN Caching. HotNets 2018.

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

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

[5] L. Cherkasova. 1998. Improving WWW Proxies Performance with Greedy-Dual-Size-Frequency Caching Policy. HP Technical Report.

[6] J. Cobb and H. ElAarag. 2008. Web Proxy Cache Replacement Scheme Based on Backpropagation Neural Network. *Journal of Systems and Software* 81, 1539–1558.

[7] G. Einziger, R. Friedman, and B. Manes. 2015. TinyLFU: A Highly Efficient Cache Admission Policy. CoRR 2015.

[8] V. Fedchenko, G. Neglia, and B. Ribeiro. 2018. Feedforward Neural Networks for Caching: Enough or Too Much? arxiv.org.

[9] B. Fitzpatrick and Memcached Community. 2019. Memcached. GitHub. URL : https://github.com/memcached/memcached.

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

[11] H. Khalid and M. S. Obaidat. 1999. KORA-2: A New Cache Replacement Policy and Its Performance. ICECS 1999.

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

[13] B. M. Maggs and R. K. Sitaraman. 2015. Algorithmic Nuggets in Content Delivery. SIGCOMM 2015.

[14] G. B. Mathews. 1897. On the Partition of Numbers. *Proceedings of the London Mathematical Society.* 28, 486–490.

[15] K. Morales and B. K. Lee. 2012. Fixed Segmented LRU Cache Replacement Scheme with Selective Caching. IPCCC 2012.

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

[17] E. Nygren, R. K. Sitaraman, and J. Sun. 2010. The Akamai Network: A Platform for High-performance Internet Applications. *SIGOPS Oper. Syst. Rev.* 44, 3, 2–19.

[18] W. Reese. 2008. Nginx: The High-Performance Web Server and Reverse Proxy. *Linux J.* 2008, 173, Article 2.

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

[20] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. 1986. Learning Representations by Back-Propagating Errors. *Nature* 323, 6088, 533–536.

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

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

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