Towards Lightweight and Robust Machine Learning for CDN Caching

Daniel S. Berger
Carnegie Mellon University

ABSTRACT

Recent advances in the field of reinforcement learning promise a *general* approach to optimize networking systems. This paper argues against the recent trend for generalization by introducing a case study where domain-specific modeling enables the application of lightweight and robust learning techniques.

We study CDN caching systems, which make a good case for optimization as their performance directly affects operational costs, while currently relying on many hand-tuned parameters. In caching, reinforcement learning has been shown to perform suboptimally when compared to simple heuristics. A key challenge is that rewards (cache hits) manifest with large delays, which prevents timely feedback to the learning algorithm and introduces significant complexity.

This paper shows how to significantly simplify this problem by explicitly modeling optimal caching decisions (OPT). While prior work considered deriving OPT impractical, recent theoretical modeling advances change this assumption. Modeling OPT enables even lightweight decision trees to outperform state-of-the-art CDN caching heuristics.

1 INTRODUCTION

The majority of the Internet's content is delivered by global caching networks, also known as Content Delivery Networks (CDNs). CDNs enhance performance by caching content in servers located in user proximity. This proximity enables fast content delivery, but requires CDNs to operate servers in hundreds of networks around the world [58, 69].

A major operational cost factor is the bandwidth cost between CDN caching servers and data centers storing the original copies of web content. Hence, CDNs aim to maximize the fraction of bytes served locally from the cache [28, 51], which is also known as the byte hit ratio (BHR).

Growing content diversity introduces operational challenges. CDN traffic is rapidly growing and is estimated to soon account for two thirds of the Internet's traffic [19]. As their user base grows, CDNs serve an increasingly diverse set of content, e.g., web, social, and ecommerce sites, software downloads, and video streaming. Each type of content has unique demands on the CDN's caching systems, e.g., iOS software downloads are large in size with popularity spikes on iOS update days, whereas Facebook photos are small with a long tail of infrequently requested photos.

The growing diversity in CDN content complicates maintaining the high BHR of caching servers. To adapt a cache to the content mix it serves, CDNs have introduced various configuration parameters [12, 51]. For example, these parameters control which objects are placed into the RAM, SSD, and HDD caches on each CDN server.

The case for automatically tuning operational parameters. Tuning the many parameters in the CDN architecture is challenging. First, there are tens of thousands of caching servers, each with many parameters. Second, each content mix requires a different parameter choice [12, 51] and the content mix changes over time. In fact, content mix changes can happen within minutes, e.g., due to changes in how users are directed to caching servers to balance load [12, 51, 72]. Moreover, content providers increasingly rely on multi-CDN deployments and frequently shift traffic between CDNs [55].

In conclusion, the scale and requirements like fast reaction times are at odds with manually tuning parameters in CDNs.

The promise of general learning techniques. To improve efficiency, CDNs seek to remove their dependence on manual parameter tuning[1]. Fortunately, recent advances in reinforcement learning (RL) [31, 32, 54, 68] promise a general approach to systems that "manage resources on their own" [52]. So, one might consider RL a straightforward choice for automating CDN parameters. In fact, several groups propose to apply RL to caching, e.g., Google [22, 23], Microsoft [48], Facebook [16], and several academic papers [20, 29, 66, 78].

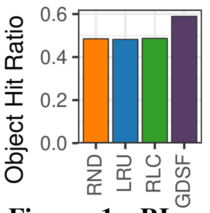


Figure 1: RL-based caching.

Despite these efforts, RL caching systems are not yet competitive with existing heuristics. For example, Figure 1 shows results from last year's HotNets workshop [48], where RL-based caching (RLC) performs similar to random (RND) and least-recently-used (LRU). All three are outperformed by a simple heuristic (GDSF [17]).

The cost of general learning techniques. Existing proposals for caching [20, 22, 23, 29, 48, 66, 78] rely on "model-free" RL [75] where the system starts without any knowledge (or bias) about the task at hand. Such systems learn to make

[1]Over the course of the last year, this topic was raised in conversions with leading figures at Akamai, Facebook, Verizon, and Wikipedia.org.

HotNets-XVII, November 15–16, 2018, Redmond, WA, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-6120-0/18/11...$15.00
https://doi.org/10.1145/3286062.3286082

decisions from experience interacting with its environment, where good behavior is reinforced via a reward function.

While model-free RL is very popular, recent discussions in the RL community highlight three key challenges [30–32, 34, 36–38]. First, millions of learning samples are typically required, which leads to slow reaction times in dynamic environments [30, 32]. Second, overfitting to past samples happens frequently, which complicates dealing with unexpected situations and can lead to unintended behaviors [36]. Third, debugging and maintenance is complicated due to high sensitivity to hyperparameters and random seeds [31, 37, 38].

For Internet facing systems these challenges are a significant roadblock. For example, CDN servers face quickly changing conditions that include unexpected (or even adversarial) traffic patterns [12, 51, 72]. CDN server also need to be easily maintainable while serving requests at 40+ Gbit/s.

While more sophisticated learning techniques, such as model-based RL [75] promise faster and more robust learning rates, they typically lead to significantly higher complexity and computational overhead (as discussed in Section 4).

A path towards lightweight and robust machine learning for CDNs. Our key insight is that the key challenge in caching lies in the fact that potential rewards (cache hits) manifest much later than the decisions that lead to this reward. These delays prevent timely feedback to RL algorithms and are the root cause of the high complexity of existing approaches. Instead of increasingly sophisticated learning techniques, we propose to explicitly address the root cause by calculating the sequence of optimal caching decisions (OPT) for the recent past. Until recently, calculating OPT was impractical [8, 18]. However, recent work in the theory community [8] has changed this assumption.

We develop a simulator prototype called Learning From OPT (LFO), which learns whether an object should be cached (Section 2). Knowing OPT enables LFO to use robust supervised learning techniques. Surprisingly, LFO achieves over 93% accuracy (Section 3) using lightweight boosted decision trees based on the LightGBM library [42]. On a CDN production trace, LFO outperforms state-of-the-art caching policies with regard to the BHR. and that LFO Furthermore, we find that LFO is robust to hyperparameters like random seeds and adapts to new request traffic with speeds comparable to state-of-the-art research systems [7, 12]. Finally, we verify LFO's scalability and find that it can fully utilize a 40 Gbit/s link.

From a machine learning perspective, LFO is essentially using *imitation learning* [34, 65] where the OPT algorithm [8] replaces the human expert. However, unlike in classical supervised imitation learning approaches (a.k.a behavior cloning) [63, 64], LFO's future observations do not depend on previous predictions. Thus, learning optimal cache admission policies becomes quite simple. Nevertheless, we find that building CDN servers that effectively exploit these admission policies remains challenging. We conclude this paper by discussing related work and prior cache learning approaches (Section 4), and these new challenges and open questions (Section 5).

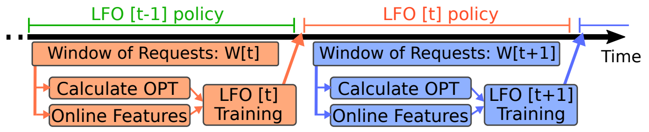


Figure 2: LFO learns from a window of past requests, t. After learning to map online features to OPT's decisions, we use the learned policy for the next time interval, $t+1$.

Object	a	b	c	b	d	a	c	d	a	b	b	a
Size	3	1	1	1	2	3	1	2	3	1	1	3
Cost	C_a	C_b	C_c	C_b	C_d	C_a	C_c	C_d	C_a	C_b	C_b	C_a

Figure 3: Example trace of requests to four objects.

2 LEARNING FROM OPT FOR CDNS

Figure 2 shows the high-level idea of LFO. As OPT is an offline algorithm, LFO records a sliding window of consecutive requests ($W[t]$). For the requests in $W[t]$, LFO calculates OPT's decisions and derives a vector of online features. LFO then trains a caching policy that maps the online features to OPT's decisions. The trained policy is then used over the next window, $t+1$, during which LFO again records the requests.

In this section, we discuss how to calculate OPT, LFO's online features, and LFO's learning lightweight algorithm.

2.1 Calculating OPT's Decisions

Given a request sequence $W[t]$, we calculate OPT's decisions using a variant of the approximation algorithms in [8].

Throughout this section, we will use the running example of a trace (Figure 3) that contains four objects, **a**, **b**, **c**, and **d**, with sizes 3, 1, 1, and 2, respectively. We assign a cost to each object: C_a, C_b, C_c, and C_d, respectively.

To optimize the BHR [33], we would set an object's costs equal to its size. To optimize the object hit ratio (OHR), we would set all costs to 1. We can also instantiate the costs from an object's average retrieval latency [17, 49].
OPT minimizes the total cost of cache misses ($\sum_{i \text{ not cached}} C_i$), subject to not exceeding the available cache space at any time. As calculating OPT directly is unfeasible, we approximate OPT using a min-cost flow model as proposed in [8].

The min-cost flow problem [2] consists of a graph with edges that are annotated with a capacity and a cost ($cap, cost$). Some graph nodes have an excess of "flow" (sources) and others have a demand for "flow" (sinks). We seek to route all units of flow from sources to sinks using the minimum cost edges. The power of this representation stems from the fact that there are fast algorithms that find this route, while maintaining edge capacities and minimizing the cost.

Figure 4 shows how we translate OPT for the trace in Figure 3 into a min-cost flow graph. The key idea is to use flow to represent the number of bytes of an object that have to be stored either in the cache, or have to be retrieved from another server. We add a node for each request, and label it with the object id (**a**, **b**, **c**, or **d**). There is excess flow, equal to the size in bytes of that object, at the node of the first request of an object, and equal demand for flow at the last request.

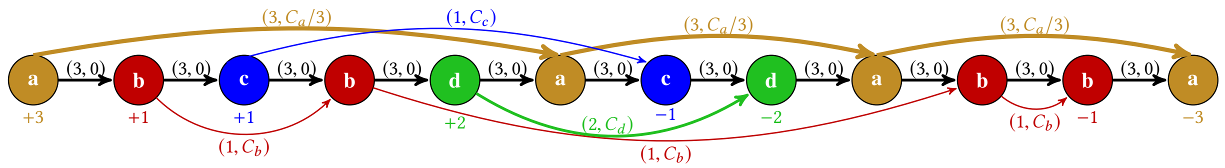


Figure 4: Min-cost flow representation of OPT for the short trace in Figure 3. Nodes represent requests, and are connected by central edges with capacity equal to the cache capacity and zero cost. Additional edges connect each pair of requests to the same object (e.g., a to a) and have cost equal to the retrieval cost (e.g,. C_a) scaled by the object size.

The flow can use a central path of horizontal edges, which connect all node pairs. Each byte of flow routed along this path translates into a byte stored in the cache. Thus, these edges have a capacity equal to the cache size and zero cost. The flow can also bypass the central path using edges that connect each pair of consecutive requests to the same object (e.g, a to a). Each byte of flow routed along these bypass edges translates into a byte of cache misses. Thus, these edges have a capacity equal to the object size in bytes, and they cost the per-byte retrieval cost (e.g., $C_a/3$).

By running a min-cost flow algorithm, we will get a solution that minimizes the cost, i.e., OPT. To derive whether OPT caches a request (e.g., the first a), we verify that all the request's bytes (starting at its node) are routed along the central path. If not, OPT does not cache this object[2].

While state-of-the-art min-cost flow algorithms are fast, solving graphs with millions of nodes can take hours [46, 61]. The authors in [8] suggest approximations that split the trace along its time axis and works on them sequentially[3]. As this technique is still slow, we propose to instead split the set of requests along a ranking axis, where higher ranked objects matter more for CDN performance. Specifically, we rank objects with the function $C_i/(S_i \times L_i)$, where S_i denotes object size and L_i is the distance to the object's next request. This ranking enables us to save 90% of the calculation time by running the algorithm only for popular requests.

In summary, we are able to quickly and accurately derive whether or not OPT caches the requests in $W[t]$.

2.2 LFO's Online Features

Our goal is to learn OPT's offline decisions from online features. Before we discuss LFO's training procedure (Section 2.3), we discuss LFO's four types of online features.

- Object size;
- Most recent retrieval cost;
- Currently free (available) bytes in the cache;
- Time gap between consecutive requests to this object.

The free-bytes feature is useful because evictions can temporarily free up lots of space (e.g., evicting a GB-large object [12]). If this happens, OPT and LFO are more likely to admit a new object. Finally, LFO uses the time gap between the last 50 requests to every object. The time "gap" is shift invariant (except for the most recent request), which is important for robustness [53]. LFO's gap approach is different from classical approaches which measure the time since each of the 50 past requests, like in LRU-K [60].

The overhead of a naive implementation that tracks all these features is 208 bytes per object. This is significantly more expensive than LRU, but only twice as much as recent caching systems [7, 12]. One might think that the overhead of storing all these feature is prohibitive. However, in practice, the feature space is very sparse (a large fraction of CDN objects receives fewer than 5 requests [51]). In addition, we can likely decrease the feature accuracy without affecting the learning results. In fact, it has been shown that adding small amounts of noise can actually be helpful in learning more robust models [27, 56, 76].

Our LFO prototype uses a sparse feature representation, but does not yet exploit lowering the accuracy.

2.3 Training LFO

The goal of LFO is to map its features to a likelihood that OPT would cache an object. We use a binary classifier and output its confidence for caching an object. Section 2.4 designs a caching policy based on this output.

The current prototype of LFO uses a gradient boosting decision tree learning algorithm, which achieves state-of-the-art performance in classification and ranking competitions on tabular features [15, 42]. We choose decision trees because of their computational efficiency and because they require few samples to learn non-linear relationships. They are also robust to outliers and skewed distributions (which are both common in caching) and interpretable.

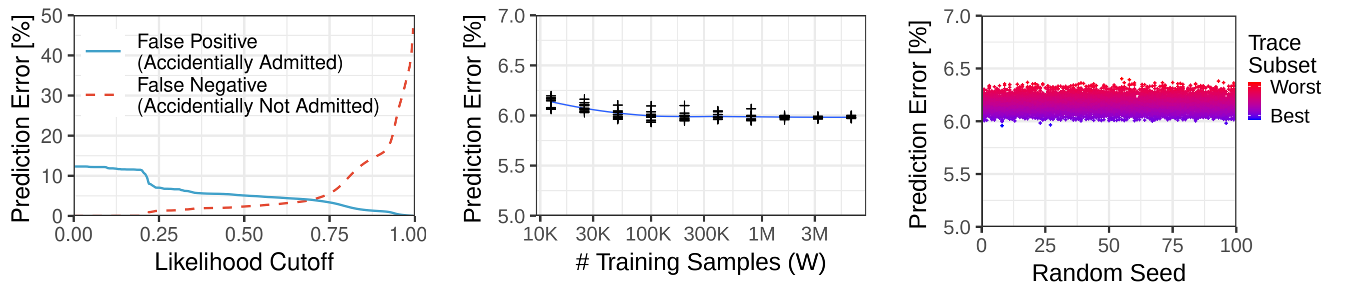
LFO currently uses LightGBM [42]. Throughout our evaluation, we use *LightGBM's default parameters* with one exception: we have decreased the number of iterations (the number of decision trees that are constructed) from 100 to 30 to further speed up our prototyping.

2.4 The LFO caching policy

We propose a simple caching policy. For every request, we call the LFO predictor to estimate how likely OPT is going to cache the object. If the confidence is $\geq .5$, we admit the object into the cache. Furthermore, we rank objects in the cache by their predicted likelihood. If we need to evict an object, we evict the one with the smallest predicted likelihood. Finally, we re-evaluate the likelihood of an object, when it is requested again. So, it may happen (unlike in existing systems), that a cache hit leads to the eviction of the hit object (which matches OPT frequently doing the same).

[2]This is a slight approximation because theoretically an object could be split into some bytes along the central path and the rest along the bypass. However, this happen rarely in practice and it is theoretically proved in [8] that min cost flow solutions route either all of an object's bytes along the central pass or all along the bypass, assuming that the cache holds many objects.

[3]This improves the calculation time because the algorithm does not have to check dependencies between distant requests of the trace.



(a) False Positives/Negatives **(b) Impact of Training Set Size** **(c) Impact of Random Seeds**

Figure 5: Preliminary results on the accuracy of our proposal, LFO, measured in terms of prediction error (requests where OPT and LFO's prediction disagree). (a) The false positive rate and false negative rate depend on the cutoff parameter, but are roughly stable between $.25$ and $.75$. (b) The prediction error quickly decays with increasing training set size and stabilizes after around $60K$ samples. (c) Random seeds, trace subsets, and hyperparameters have only a small impact on LFO's accuracy.

3 PRELIMINARY RESULTS

Our evaluation is based on an early prototype of LFO. Our implementation contains three parts: approximately 400 lines of C++ code to compute OPT (Section 2.1) using the Lemon numerical library [59], approximately 700 lines of C++ code to track object features (Section 2.2) and train the decision trees (Section 2.3) using the LightGBM learning library [42], and approximately 1500 lines of C++ code for replaying traces and simulating various CDN caching policies (only 50 lines of simulator code are dedicated to LFO).

We use a 2016 request trace from the CDN of an anonymous top-ten US website[4]. Recorded on a San Francisco CDN server, the trace spans about a week (500 million requests). Requests are anonymized to a sequence number, an object identifier, and the object size in bytes. We split the CDN trace chronologically into parts with one million requests each. LFO is trained on one part (e.g., requests 0-1 million), and evaluated on the ensuing part (e.g., requests 1-2 million). We repeat this for all parts of the trace.

We use our prototype[5] to evaluate five key properties of our proposal: the accuracy, learning speed, robustness, byte hit ratio improvements, and throughput of LFO.

What is LFO's prediction accuracy? LFO matches OPT's prediction for over 93% of the requests.

To more accurately quantify the errors made by LFO, we measure the false positive rate (LFO admits when OPT does not) and the false negative rate (LFO does not admit when OPT does). Figure 5a shows both error rates as a function of LFO's cutoff point. Recall that LFO predict OPT's likelihood to cache a given request. LFO places an object into the cache if this likelihood is greater than .5 (Section 2.4).

Figure 5a shows that false positive and false negative rates plateau between cutoff values .25 and .75. Below a .25 cutoff the false negative rate increases quickly. Above a .75 cutoff the false positive rate increases quickly. The plateau between .25 and .75 shows that LFO achieves a good separation between objects that should and should not be admitted.

We also observe that LFO is conservative in that it is biased towards admitting objects (false positives). We could make LFO more aggressive by raising the cutoff to about .65, equalizing false negative and false positive rate.

In summary, even our naive implementation of LFO is surprisingly accurate in prediction OPT's decisions, and there remains potential for further improving the accuracy.

How quickly does LFO learn OPT? We find that LFO converges within a few tens of thousands of requests. Thus, LFO can be frequently updated to match OPT.

To quantify LFO's learning speed, we measure the fraction of prediction errors as a function of the number of training samples. We repeat this comparison for ten random subsets of the trace. Figure 5b shows the samples including a local regression curve fit. The error is below 6.5% even for a few thousand training samples (10K), and decreases slightly until 100K. As we further increase the training set, prediction accuracy becomes more predictable.

In summary, LFO requires as few samples as recent caching systems that also take request history into account [7, 12].

How sensitive is LFO to random seeds and hyperparameters? We find that LFO is not sensitive to either.

Figure 5c shows how LFO's error varies across 100 seeds on 100 different trace subsets. LFO's accuracy remains within a range of .5% and is thus not sensitive to random seeds.

So far, our evaluation uses LightGBM's default parameters (Section 2.3). Through additional experiments (not shown), we verify the impact of other parameter choices. For larger iteration counts and lower learning rates, LFO's accuracy improves somewhat (to 95%). For larger tree sizes, LFO is prone to overfitting, which decreases the accuracy (to 88%).

In summary, LFO is not highly sensitive to either random seeds or hyperparameters. We plan to further explore the parameter space in the future.

[4]Our request trace is publicly available as a link from our code repository at https://github.com/dasebe/optimalwebcaching.

[5]Our evaluation uses a 2017 Super Micro Server with two Intel Xeon E5-2699 v4 at 2.20GHz CPUs. This gives us a total of 44 cores/ 88 threads and 512GB of memory. The size of the cache managed by LFO has size 256GB.

How does LFO compare to existing caching systems? We find that LFO achieves higher BHRs than state-of-the-art caching systems and about 80% of OPT's BHR.

We compare LFO to our own implementations of LRU, LRU-K [60], LFUDA [4, 24, 67], S4LRU [33], GD-Wheel [49], AdaptSize [12], Hyperbolic [13], LHD [7], and OPT.

Figure 6 shows that LFO improves the BHR by 6% over the next best system, S4LRU. We have also evaluated the OHR of these caching policies as AdaptSize, Hyperbolic, and LHD all focus on the OHR, which leads to very low BHRs. Surprisingly, LFO achieves almost the same OHR as LHD, which is the next best system. This indicates that sacrificing BHR to gain OHR is not necessary.

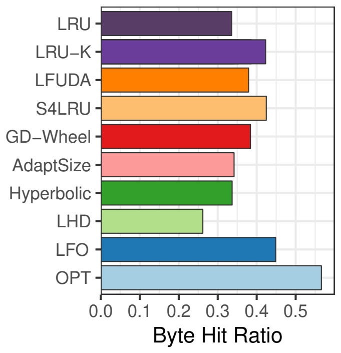


Figure 6: Comparison of LFO to state-of-the-art caching systems.

Compared to OPT, LFO achieves only about 80% of either BHR or OHR, which is significantly lower than LFO's accuracy. We defer a discussion of this fact to Section 5.

In summary, LFO performs surprisingly well by beating much more complex recent caching systems.

Can LFO predict fast enough for production use? We find that LFO's choice of lightweight decision trees is highly scalable and adds negligible delay associated with evaluating its prediction trees.

Figure 7 shows the throughput in million requests per second achieved by our naive LFO predictor. A single thread can serve predictions for just below 300K requests per second. For 12 threads (44 threads), prediction speed scales almost linearly reaching more than 3 million (11 million) requests per second. To utilize a 40 GBit/s network, LFO needs only two threads, assuming an average object size of 32KB [12, 33, 51]. To serve tiny 500B objects, LFO would need to use all 44 threads. We remark that we have not included the training overhead and that a production implementation would need to carefully optimize priorities such that training tasks do not interfere with the request traffic.

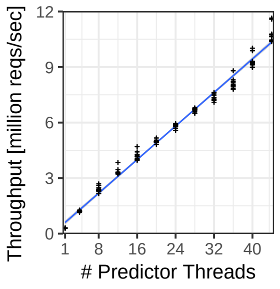


Figure 7: LFO's prediction throughput scales well.

In summary, LFO does not come for free, but it is highly scalable and meets the fundamental performance requirements of production CDN servers.

Which features matter most for LFO's predictions? We find that LFO most relies on object size (similar to other recent systems) [7, 12], free cache space, and the first few gaps.

LFO's learned models are composed of a large set of "if-then-else" tree branches. Each split depends on a single feature. To determine the role of LFO's features, we count how often each feature occurs in a split.

Figure 8 shows the percentage of branches that each feature occurs in. We observe that LFO heavily relies on the object size (28% of branches, bar extends outside of axis limits). This is consistent with recent caching systems that also put the object size into focus. LFO does not use the cost feature. This makes sense, as it is redundant with the object size when optimizing BHRs (Section 2.1). LFO uses the free cache space feature in almost 10% of branches. This is surprising and likely due to the highly variable object sizes in our trace (Section 2.2). LFO makes most use of time gaps 1 to 4. However, up to time gap 16, LFO still makes significant use of these features. In addition, several higher time gaps also see significant use (e.g., 20, 24, 32, 36, 48).

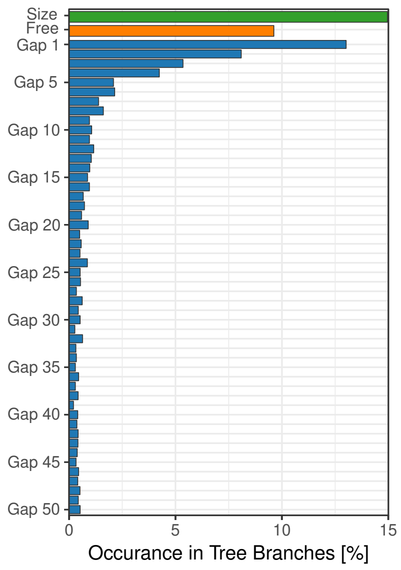


Figure 8: Relative importance of LFO's features.

Our observations about the time gap feature suggests two possible improvements. First, we can speed up the model by artificially thinning out the time gap feature space (e.g., only using time gaps 1, 2, 4, 8, 16, etc.). Second, as high time gaps are still being used, keeping track of an even larger history might allow us to further improve LFO's accuracy.

In summary, LFO behaves consistent with other caching systems and includes potential for further improvements.

4 RELATED WORK

While the literature on CDNs and web caching is extensive, we are not aware of prior work that either proposes or evaluates learning from the optimal caching policy[6].

The likely reason why few have considered this direction is that calculating OPT is strongly NP complete [18] for Internet traces (which have variable object sizes) and that approximation algorithms [3, 5, 35] were impractical until recently. As discussed in Section 2.1, LFO is enabled by recent theoretical work [8], which derives OPT for the OHR metric. This work differs from works on approximation algorithms [3, 5, 8, 35] by focusing on learning OPT's decisions from online features.

[6]In computer architecture, where computing OPT is simple, a recent work [39] exploits the fact that a side-channel (the program counter) predict OPT's behavior. That work is not applicable to CDNs and does not evaluate learning OPT's behavior from information that is available to a CDN server.

The most common research direction in caching involves clever heuristics [9, 13, 17, 33, 49, 60] and Markovian modeling of future hits [7, 11, 12, 24]. None of these works learn from OPT. Our unique approach enables LFO to outperform all of these systems in our preliminary experiments.

Several recent works apply model-free RL techniques to caching [16, 20, 22, 23, 29, 48, 66, 78]. Currently, these model-free RL caching systems are not competitive to existing caching systems, whereas LFO outperforms existing systems. Additionally, systems based on model-free RL are prone to slow convergence speeds [30, 32], overfitting [36] and sensitivity to hyperparameters [31, 37, 38].

There are more sophisticated learning techniques, e.g., "model-based" RL techniques such as Dyna [73, 74], E^3 [43, 44], and R-max [14]. However, these techniques typically focus on small and stationary state spaces, whereas CDN servers face a highly dynamic environment. Unfortunately, even simple caching policies create an intractable state space explosion [10, 11, 21, 25, 26, 40, 45]. More sophisticated approximative learning techniques [75] can solve this problem — at the cost of significantly higher complexity and computational overhead. We remark that current advanced learning systems for caches [66, 78] are already limited to tiny caches due to their computational complexity. Our key contribution is to reduce the learning problem for caching to a simple supervised learning problem. This enables the use of robust and lightweight boosted decision trees [42]. In the future, our reduction may also enable the design of better RL caching systems using techniques from inverse reinforcement learning that learn optimal rewards from OPT [1, 57, 62].

Finally, there is an emerging literature on successful machine learning applications in networking research such as traffic classification [77], video rate adaptation [41, 71], request routing [50], resource allocation [52], and congestion control [70]. Our work pursues similar goals, but for a different set of problems and with unique insights.

5 DISCUSSION AND OPEN QUESTIONS

While there are several avenues for future work, we seek to highlight and discuss three key questions.

What is the real reason for the gap to optimality, and how to bridge it? We started this project with the assumption that OPT's advantage is its wealth of information that cannot be matched by any online policy. However, our preliminary results show that LFO correctly predicts 93% of OPT's decisions using a limited set of online features.

Somewhat surprisingly, we also find that this near-optimal prediction performance does not translate into near-optimal caching performance. In fact, the gap between LFO and OPT is 20% — much larger than 7%. Consequently, it must be that incorrect admission choices (e.g., false positives) have a knock-on effect: objects that should receive hits end up being evicted before they do receive a hit.

One angle to partially address this could be to adjust LFO's cutoff value (we do see some improvement), or to work on better prediction performance (using advanced "deep learning" approaches [47] is likely fruitful).

However, we argue that to bridge the gap to OPT we should focus our efforts on how to translate a ranking of objects into a caching policy, an algorithmic problem which we call policy design — a field which has not seen much innovation in many years. Given the potential cost and energy savings that are enabled by better caching policies, we hope to renew interest in this fundamental research question.

Is model-based learning extensible? How would we apply it to a whole CDN? The main message of this paper is that we can build robust machine learning systems using a model that predicts the actual intended behavior (e.g., optimal caching decisions). While we have only studied the example of a single cache, we believe that the resulting robustness represents a significant advantage over model-free reinforcement learning approaches.

The obvious challenge in generalizing our approach is that modeling requires significant time and effort. However, modeling requires us to learn about our system (instead of tuning a black box) and this knowledge can directly benefit operators of Internet-scale systems. From this perspective, we argue that time invested in modeling is time well spent.

We also argue that CDN-scale modeling does not need to be a daunting task. A key idea to simplify this problem is to use hierarchical models. For example, we could apply our "single cache" model to the aggregate cache space of a CDN server (RAM, SSD, HDD) or to a whole rack of servers. We first learn whether to cache an object at all. A second level of the model then learns rules on where to place the object, e.g., based on storage characteristics such as write endurance, read delay/throughput, or utilization. Finally, recent work on modeling CDN cache provisioning [72] gives us hope that we can identify and learn the optimal behavior across many servers and CDN points-of-presence.

What is the impact of user feedback loops on LFO's trace-driven approach? LFO may be subject to fundamental biases of trace-driven techniques [6]. For example, it is conceivable that LFO's improvement of CDN server performance increases the number of user requests per hour. A larger number of user requests may increase the bandwidth usage of the CDN server — despite an increased BHR — defeating the purpose of the CDN to deploy LFO in the first case. The impact of such feedback loops is very hard to predict or measure through experiments. We thus leave approaches to quantify this effect as an open question for future work.

ACKNOWLEDGEMENTS

We thank Nathan Beckmann, Anwar Hithnawi, our shepherd Mohammad Alizadeh, and our anonymous reviewers for their helpful comments. Daniel is supported by a CMU-CS Mark Stehlik Postdoctoral Teaching Fellowship. This paper was partially supported by NSF-CSR-180341 and wikipedia.org.

REFERENCES

[1] Pieter Abbeel and Andrew Y Ng. 2004. Apprenticeship learning via inverse reinforcement learning. In *ACM ICML*.

[2] Ravindra K Ahuja, Thomas L Magnanti, and James B Orlin. 1993. *Network flows: theory, algorithms, and applications*. Prentice hall.

[3] Susanne Albers, Sanjeev Arora, and Sanjeev Khanna. 1999. Page replacement for general caching problems. In *SODA*. 31–40.

[4] Martin Arlitt, Ludmila Cherkasova, John Dilley, Rich Friedrich, and Tai Jin. 2000. Evaluating content management techniques for web proxy caches. *Performance Evaluation Review* 27, 4 (2000), 3–11.

[5] Amotz Bar-Noy, Reuven Bar-Yehuda, Ari Freund, Joseph Naor, and Baruch Schieber. 2001. A unified approach to approximating resource allocation and scheduling. *J. ACM* 48, 5 (2001), 1069–1090.

[6] Mihovil Bartulovic, Junchen Jiang, Sivaraman Balakrishnan, Vyas Sekar, and Bruno Sinopoli. 2017. Biases in Data-Driven Networking, and What to Do About Them. In *ACM HotNets*. 192–198.

[7] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. 2018. LHD: Improving Hit Rate by Maximizing Hit Density. In *USENIX NSDI*. 1–14.

[8] Daniel S. Berger, Nathan Beckmann, and Mor Harchol-Balter. 2018. Practical Bounds on Optimal Caching with Variable Object Sizes. *Proc. ACM Meas. Anal. Comput. Syst.* 2, 2, Article 32 (June 2018), 38 pages.

[9] Daniel S. Berger, Benjamin Berg, Timothy Zhu, Mor Harchol-Balter, and Siddhartha Sen. 2018. RobinHood: Tail Latency-Aware Caching - Dynamically Reallocating from Cache-Rich to Cache-Poor.. In *USENIX OSDI*.

[10] Daniel S. Berger, Philipp Gland, Sahil Singla, and Florin Ciucu. 2014. Exact analysis of TTL cache networks. *Perform. Eval.* 79 (2014), 2 – 23. Special Issue: Performance 2014.

[11] Daniel S Berger, Sebastian Henningsen, Florin Ciucu, and Jens B Schmitt. 2015. Maximizing cache hit ratios by variance reduction. *ACM SIGMETRICS Performance Evaluation Review* 43, 2 (2015), 57–59.

[12] Daniel S. Berger, Ramesh Sitaraman, and Mor Harchol-Balter. 2017. AdaptSize: Orchestrating the Hot Object Memory Cache in a CDN. In *USENIX NSDI*. 483–498.

[13] Aaron Blankstein, Siddhartha Sen, and Michael J Freedman. 2017. Hyperbolic Caching: Flexible Caching for Web Applications. In *USENIX ATC*. 499–511.

[14] Ronen I Brafman and Moshe Tennenholtz. 2002. R-max-a general polynomial time algorithm for near-optimal reinforcement learning. *Journal of Machine Learning Research* 3, Oct (2002), 213–231.

[15] Christopher J.C. Burges. 2010. *From RankNet to LambdaRank to LambdaMART: An Overview*. Technical Report. Microsoft Research Technical Report MSR-TR-2010-82.

[16] Vladimir Bychkovsky, Jim Cipar, Alvin Wen, Lili Hu, and Saurav Mohapatra. 2018. Spiral: Self-tuning services via real-time machine learning. Available at https://code.fb.com/data-infrastructure/spiral-self-tuning-services-via-real-time-machine-learning/, accessed 07/10/18.

[17] Ludmila Cherkasova. 1998. *Improving WWW proxies performance with greedy-dual-size-frequency caching policy*. Technical Report. Hewlett-Packard Laboratories.

[18] Marek Chrobak, Gerhard J Woeginger, Kazuhisa Makino, and Haifeng Xu. 2012. Caching is hard—even in the fault model. *Algorithmica* 63 (2012), 781–794.

[19] CISCO. 2017. VNI Global IP Traffic Forecast: The Zettabyte Era—Trends and Analysis. Available at https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/vni-hyperconnectivity-wp.pdf, accessed 24/09/17.

[20] Renato Costa and Jose Pazos. 2017. *MLCache: A Multi-Armed Bandit Policy for an Operating System Page Cache*. Technical Report. University of British Columbia.

[21] Asit Dan and Don Towsley. 1990. An Approximate Analysis of the LRU and FIFO Buffer Replacement Schemes. In *ACM SIGMETRICS*. 143–152.

[22] Jeff Dean. 2017. Machine Learning for Systems and Systems for Machine Learning. Presentation at NIPS Systems for ML Workshop. Available at http://goo.gl/wxuvVk, accessed 10/10/18.

[23] Jeff Dean. 2018. Is Google Using Reinforcement Learning to Improve Caching? Personal communication on 2018-09-27.

[24] Gil Einziger and Roy Friedman. 2014. Tinylfu: A highly efficient cache admission policy. In *IEEE Euromicro PDP*. 146–153.

[25] Philippe Flajolet, Daniele Gardy, and Loÿs Thimonier. 1992. Birthday paradox, coupon collectors, caching algorithms and self-organizing search. *Discrete Applied Mathematics* 39 (1992), 207–229.

[26] Erol Gelenbe. 1973. A unified approach to the evaluation of a class of replacement algorithms. *IEEE Trans. Comput.* 100 (1973), 611–618.

[27] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. 2015. Explaining and harnessing adversarial examples (2014). In *ICLR*.

[28] Syed Hasan, Sergey Gorinsky, Constantine Dovrolis, and Ramesh K Sitaraman. 2014. Trade-offs in optimizing the cache deployments of CDNs. In *IEEE INFOCOM*. 460–468.

[29] Ying He, F Richard Yu, Nan Zhao, Victor CM Leung, and Hongxi Yin. 2017. Software-defined networks with mobile edge computing and caching for smart cities: A big data deep reinforcement learning approach. *IEEE Communications Magazine* 55, 12 (2017), 31–37.

[30] Nicolas Heess, Dhruva TB, Srinivasan Sriram, Jay Lemmon, Josh Merel, Greg Wayne, Yuval Tassa, Tom Erez, Ziyu Wang, S. M. Ali Eslami, Martin A. Riedmiller, and David Silver. 2017. Emergence of Locomotion Behaviours in Rich Environments. *CoRR* abs/1707.02286 (2017).

[31] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. 2018. Deep Reinforcement Learning that Matters. In *AAAI (Conference on Artificial Intelligence)*.

[32] Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. 2018. Rainbow: Combining Improvements in Deep Reinforcement Learning. In *AAAI (Conference on Artificial Intelligence)*.

[33] Qi Huang, Ken Birman, Robbert van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C Li. 2013. An analysis of Facebook photo caching. In *ACM SOSP*. 167–181.

[34] Ahmed Hussein, Mohamed Medhat Gaber, Eyad Elyan, and Chrisina Jayne. 2017. Imitation learning: A survey of learning methods. *ACM Computing Surveys (CSUR)* 50, 2 (2017), 21.

[35] Sandy Irani. 1997. Page replacement with multi-size pages and applications to web caching. In *ACM STOC*. 701–710.

[36] Alex Irpan. 2016. Faulty Reward Functions in the Wild. OpenAI Blog https://blog.openai.com/faulty-reward-functions/.

[37] Alex Irpan. 2018. Deep Reinforcement Learning Doesn't Work Yet. https://www.alexirpan.com/2018/02/14/rl-hard.html.

[38] Riashat Islam, Peter Henderson, Maziar Gomrokchi, and Doina Precup. 2017. Reproducibility of benchmarked deep reinforcement learning tasks for continuous control. In *ACM ICML Reproducibility in Machine Learning Workshop*.

[39] Akanksha Jain and Calvin Lin. 2016. Back to the future: leveraging Belady's algorithm for improved cache replacement. In *ACM/IEEE ISCA*. 78–89.

[40] Predrag R Jelenković. 1999. Asymptotic approximation of the move-to-front search cost distribution and least-recently used caching fault probabilities. *The Annals of Applied Probability* 9 (1999), 430–464.

[41] Junchen Jiang, Vyas Sekar, Henry Milner, Davis Shepherd, Ion Stoica, and Hui Zhang. 2016. CFA: a practical prediction system for video QoE optimization. In *USENIX NSDI*. 137–150.

[42] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. Lightgbm: A highly efficient gradient boosting decision tree. In *Advances in Neural Information Processing Systems*. 3146–3154.

[43] Michael Kearns and Daphne Koller. 1999. Efficient reinforcement learning in factored MDPs. In *IJCAI*, Vol. 16. 740–747.

[44] Michael Kearns and Satinder Singh. 2002. Near-optimal reinforcement learning in polynomial time. *Machine learning* 49, 2-3 (2002), 209–232.

[45] W. Frank King. 1971. Analysis of Demand Paging Algorithms. In *IFIP Congress (1)*. 485–490.

[46] Péter Kovács. 2015. Minimum-cost flow algorithms: an experimental evaluation. *Optimization Methods and Software* 30, 1 (2015), 94–127.

[47] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *Nature* 521, 7553 (2015), 436.

[48] Mathias Lecuyer, Joshua Lockerman, Lamont Nelson, Siddhartha Sen, Amit Sharma, and Aleksandrs Slivkins. 2017. Harvesting Randomness to Optimize Distributed Systems. In *ACM HotNets*. 178–184.

[49] Conglong Li and Alan L Cox. 2015. GD-Wheel: a cost-aware replacement policy for key-value stores. In *EUROSYS*. 1–15.

[50] Hongqiang Harry Liu and Raajay Viswanathan. 2016. Efficiently Delivering Online Services over Integrated Infrastructure.. In *USENIX NSDI*.

[51] Bruce M Maggs and Ramesh K Sitaraman. 2015. Algorithmic nuggets in content delivery. *ACM SIGCOMM CCR* 45 (2015), 52–66.

[52] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. 2016. Resource management with deep reinforcement learning. In *ACM HotNets*. 50–56.

[53] Sanjit K Mitra and James F Kaiser. 1993. *Handbook for digital signal processing*. John Wiley & Sons, Inc.

[54] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. 2015. Human-level control through deep reinforcement learning. *Nature* 518, 7540 (2015), 529.

[55] Matthew K Mukerjee, Ilker Nadi Bozkurt, Bruce Maggs, Srinivasan Seshan, and Hui Zhang. 2016. The impact of brokers on the future of content delivery. In *ACM HotNets*. 127–133.

[56] A Neelakantan, L Vilnis, QV Le, I Sutskever, L Kaiser, K Kurach, and J Martens. 2016. Adding Gradient Noise Improves Learning for Very Deep Networks. In *ICLR Workshop*.

[57] Andrew Y Ng, Stuart J Russell, et al. 2000. Algorithms for inverse reinforcement learning.. In *ACM ICML*. 663–670.

[58] E. Nygren, Ramesh K. Sitaraman, and J. Sun. 2010. The Akamai Network: A platform for high-performance Internet applications. *ACM SIGOPS Operating Systems Review* 44, 3 (2010), 2–19.

[59] Egerváry Research Group on Combinatorial Optimization. 2015. COIN-OR::LEMON Library. Available at http://lemon.cs.elte.hu/trac/lemon, accessed 5/5/18.

[60] Elizabeth J O'Neil, Patrick E O'Neil, and Gerhard Weikum. 1993. The LRU-K page replacement algorithm for database disk buffering. *ACM SIGMOD* 22, 2 (1993), 297–306.

[61] James B Orlin. 1997. A polynomial time primal network simplex algorithm for minimum cost flows. *Mathematical Programming* 78, 2 (1997), 109–129.

[62] Nathan D Ratliff, J Andrew Bagnell, and Martin A Zinkevich. 2006. Maximum margin planning. In *ACM ICML*. 729–736.

[63] Stéphane Ross and Drew Bagnell. 2010. Efficient reductions for imitation learning. In *AISTATS*. 661–668.

[64] Stéphane Ross, Geoffrey Gordon, and Drew Bagnell. 2011. A reduction of imitation learning and structured prediction to no-regret online learning. In *AISTATS*. 627–635.

[65] Stefan Schaal. 1999. Is imitation learning the route to humanoid robots? *Trends in cognitive sciences* 3, 6 (1999), 233–242.

[66] Avik Sengupta, SaiDhiraj Amuru, Ravi Tandon, R Michael Buehrer, and T Charles Clancy. 2014. Learning distributed caching strategies in small cell networks. In *IEEE ISWCS*. 917–921.

[67] Ketan Shah, Anirban Mitra, and Dhruv Matani. 2010. *An O(1) algorithm for implementing the LFU cache eviction scheme*. Technical Report. Stony Brook University.

[68] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. 2017. Mastering the game of Go without human knowledge. *Nature* 550, 7676 (2017), 354.

[69] Ramesh K. Sitaraman, Mangesh Kasbekar, Woody Lichtenstein, and Manish Jain. 2014. Overlay networks: An Akamai perspective. In *Advanced Content Delivery, Streaming, and Cloud Services*. John Wiley & Sons.

[70] Anirudh Sivaraman, Keith Winstein, Pratiksha Thaker, and Hari Balakrishnan. 2014. An experimental study of the learnability of congestion control. In *ACM SIGCOMM*, Vol. 44. 479–490.

[71] Yi Sun, Xiaoqi Yin, Junchen Jiang, Vyas Sekar, Fuyuan Lin, Nanshu Wang, Tao Liu, and Bruno Sinopoli. 2016. CS2P: Improving video bitrate selection and adaptation with data-driven throughput prediction. In *ACM SIGCOMM*. 272–285.

[72] Aditya Sundarrajan, Mingdong Feng, Mangesh Kasbekar, and Ramesh K Sitaraman. 2017. Footprint Descriptors: Theory and Practice of Cache Provisioning in a Global CDN. In *ACM CoNEXT*. 55–67.

[73] Richard S Sutton. 1990. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Machine Learning Proceedings*. 216–224.

[74] Richard S Sutton. 1991. Planning by incremental dynamic programming. In *Machine Learning Proceedings*. 353–357.

[75] Richard S. Sutton and Andrew G. Barto. 2018. *Reinforcement learning: An introduction* (2 ed.). MIT press.

[76] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. 2013. Intriguing properties of neural networks. In *ICLR*.

[77] Jun Zhang, Xiao Chen, Yang Xiang, Wanlei Zhou, and Jie Wu. 2015. Robust network traffic classification. *IEEE/ACM TON* 23, 4 (2015), 1257–1270.

[78] Chen Zhong, M Cenk Gursoy, and Senem Velipasalar. 2018. A deep reinforcement learning-based framework for content caching. In *IEEE CISS (Annual Conference on Information Sciences and Systems)*. 1–6.