LeCaR Caching with Multi Armed Bandits

Dwaipayan Saha[†] **Ansh Bhargava**[†] **Garner Thompson**[†]
[†]Department of Computer Science, Princeton University
`{dsaha, anshb, mgt2}@princeton.edu`

1 Project Overview

Cache eviction is a field studied by many systems researchers and is core to many large scale systems. Classical cache eviction techniques include both LRU (Least Recently Used) and LFU (least frequently used) both built from FIFO models. Recently, dynamic cache eviction policies have become increasingly popular, where the goal is to leverage probabilistic properties to find the best caching approach for a given workload. There are many frameworks of machine learning that are used and in this paper we consider LeCaR, a non-queue based eviction algorithm. This algorithm deploys a reinforcement learning approach which uses online learning, regret minimization and multi-arm bandits.

2 Related Works

Our implementation, presented in section 3, is inspired by the original 2018 LeCaR algorithm: [5]

Other related works include ACME, a 2002 algorithm that uses weight updates to determine a particular caching algorithm. [1] This method does not invent a new cache eviction policy. Rather, it instantiates existing cache replacement policies as experts with all of them having equal weights initially. As queries are made the weights are updated based off of metrics such as hit rate or byte hit rate and these weights are used to evaluate what should be evicted. Note that this is very similar to our approach with LeCaR except maintaining a collection of experts is expensive and, as it turns out, the conjunction of LFU and LRU already does sufficiently well. In general, such a technique is known as formulating the problem as online regret minimization problem and generalization of the Multiplicative Weights algorithm[4].

Adaptive Replacement Cache (ARC) is a 2003 algorithm that chooses whether LRU or LFU is a better caching policy using learning rules. It separates the cache directory into two lists, one for recently used items and one for frequently used items (items used more than once). This differs from LeCaR, since LeCaR maintains one singular unified cache and does not split recently used items from frequently used items. ARC also keeps track of elements recently evicted from both lists, which it uses to determine whether LRU or LFU is a better caching algorithm in a given scenario[3].

3 Design Overview and Implementation

The cache eviction algorithm we implement, LeCaR, uses a multi-arm bandit approach. Specifically, we have 2 arms, each of which defines whether we do LFU or LRU based eviction. This is because as shown in [5] often it is the case that using a distribution of the 2 eviction strategies does empirically better due to it taking advantage of the weaknesses in each and patching them with their differences. We use LRU and LFU specifically because both policies perform rather well on their own due to each policy, in practice, having good temporal and spatial locality. Furthermore, this requires an implementation of an online algorithm in the sense that we sequentially update the distribution/policy we are playing in hopes of converging onto an optimal policy (getting as close as possible to a clairvoyant cache).

We begin by selecting some hyper-parameter $\lambda \in [0, 1)$. We select 0.27. Furthermore, we initialize weights $(w_{\text{LFU}}, w_{\text{LRU}}) = (0.5, 0.5)$ forming a valid probability distribution.

For each request in our trace data, we run `get(query)`. This executes the following algorithm:

If the query is in the cache, we update the number of hits by 1 and continue. Since we do not need to evict, there is no need to update the history or the weights. However, we do update our recency and frequency data structures.

In the case that `query` is not in the cache, we first wish to update our weights in order to "converge" to what is the optimal policy. At this point we check if `query` is in H_{LRU} or H_{LFU}. We define $r = d^t$, where d is the discount rate and t is the time elapsed between the last miss of `query` in the cache. If `query` is not in either of the histories, we do not consider either r or t. We will perform the following algorithm:

1. If `query` was in neither history then the weights remain the same.

2. If `query` was in either H_{LRU} or H_{LFU} we wish to penalize the weight associated the eviction system that evicted `query` when they should not have (which is why it is in the history). We do so by boosting the weight of the other. Notate i to be the history it was in, and j to be one it was not in. Thus in our update $w_i = w_i$ and $w_j = w_j \cdot e^{\lambda r}$. We finish by normalizing in order to ensure that $(w_{\text{LFU}}, w_{\text{LRU}})$ still forms a probability distribution.

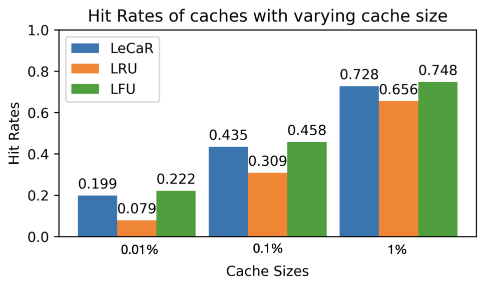
Now based off this probability distribution, we randomly choose an eviction policy with probability of enacting LRU equal to w_{LRU}, and probability of enacting LFU equal to w_{LFU}. Continuing this for many rounds, we utilize this policy, adapting weights as we go.

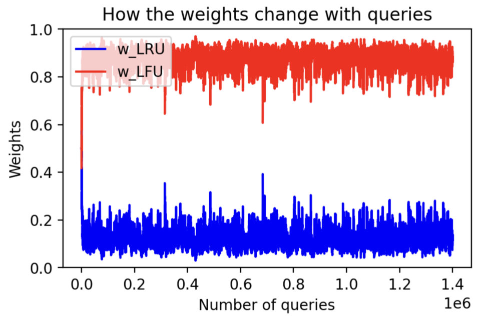
4 Testing Methodology

We evaluated LeCar against both a LRU and LFU cache. Our key metric was hit rate and we also analyzed the weight distribution between the two arms in the LeCar cache as well as sensitivity to the λ parameter. To evaluate, we ran the cache algorithms on a trace generated by `webcacheism` [2] which generates a synthetic trace mimicking a CDN server. The length of traces we tested on are 10,000 and 100,000 requests. Our results are drawn from the 100,000 request trace.

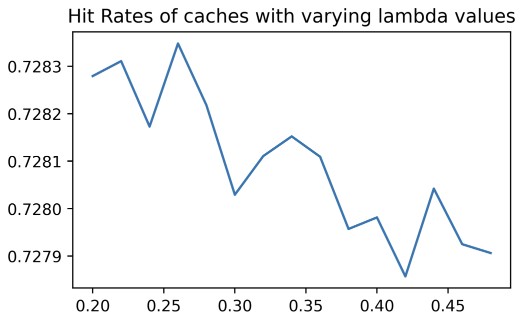
5 Outcome and Results

The figure below shows the hit rates for the LeCar cache versus LRU and LFU across different cache sizes. It shows that, over our trace data (100k requests), the LeCar cache consistently performs better than LRU but slightly worse than LFU.





The figure above shows the how the weights evolve through the online learning process. One can see that there are approximately 100k requests. The initial weights as you can see are both 0.5. As we see requests come in at each round, our weights are penalized to minmize regret (cache misses). In fact, one can see that the weights do generally converge to $w_{\text{LFU}} \approx 0.85$ and $w_{\text{LRU}} \approx 0.15$.



The figure above shows the sensitivity of the LeCar cache to different λ values. The original paper claims that the cache is robust to different λ values and performs best at $\lambda = 0.45$ for most data. Our findings agree with the consistency in the range of 0.2 to 0.5 but have an optimal $\lambda = 0.27$.

6 Concluding Remarks

Our results present a few interesting findings. First, we agree with the paper that the LeCaR algorithm tends to weigh the LFU arm higher, at around 0.85. Second, we also agree with the paper that the LeCar algorithm is robust across the λ hyper-parameter.

We also found that the LeCaR algorithm outperforms the LRU cache on our trace data, which simulates requests to a CDN server, but contrary to the paper, we found that the LFU cache performs better. However, this is not necessarily a general result and the LeCaR algorithm may still benefit from the orthogonality of recency and frequency in other use cases.

References

[1] Ismail Ari et al. *ACME: Adaptive Caching Using Multiple Experts*. 2002.

[2] Mor Harchol-Balter Daniel S. Berger Ramesh K. Sitaraman. *AdaptSize: Orchestrating the Hot Object Memory Cache in a CDN*. 2017.

[3] Nimrod Megiddo and Dharmendra S. Modha. *ARC: A Self-Tuning, Low Overhead Replacement Cache*. 2003.

[4] S. Kale S. Arora E. Hazan. *The multiplicative weights update method: A meta algorithm and its applications. Theory of Computing, Volume 8 (2012), pp. 121–164.*

[5] Giuseppe Vietri et al. *Driving Cache Replacement with ML-based LeCaR*. 2018.