RLCaR: Reinforcement Learning for Cache admission and Replacement

Isha Tarte, Syamantak Kumar
Department of Computer Science, UT Austin
{tarteisha, syamantak}@utexas.edu

Presentation Video • Github Link • Presentation Slides

Abstract

We propose to apply reinforcement learning on caching systems. The first problem we consider is to decide whether we want to admit an object in the cache, when an object request leads to a cache miss. While cache replacement policies have received significant traction over the years, most systems use LRU (Least Recently Used) for eviction, without explicit admission algorithms. The optimal algorithm for solving cache admission requires access to future requests, thus making it impractical. We train an RL agent to give a binary decision of admit/don't admit for each cache miss. We show that using our RL agent gives a higher byte hit rate compared to always (or randomly) admitting on a cache miss when LRU is used as the cache replacement policy. The next problem that we consider is the more common problem of cache replacement, i.e, deciding which object to evict from the cache on a miss. We model this as an adversarial bandit problem, treating LRU, LFU (Least Frequently Used) and FIFO (First In First Out) as experts, and solve it using the Hedge algorithm, assuming full feedback. We show that the algorithm eventually converges to the best expert. Our experiments are based on a simulated environment, where the cache traces are generated using a Zip-f distribution, which has been widely used in simulations.

1 Introduction

Caching is widely employed in computer systems to store data so that future requests for that data can be served faster. The cache size has to be relatively smaller than the number of unique object requests to be cost-effective and enable efficient use of data. Cache hits correspond to serving requests by reading objects that are present in the cache, which is considerably faster than reading from a secondary slow storage. Therefore, the more requests that can be served from the cache, the faster the system performs. One of the primary objectives of a caching system is to maximise the *byte hit rate*, which is the fraction of bytes that are served from the cache[1]. We deal with two of the major problems in the domain of caching systems, namely cache admission and cache replacement.

Given access to future requests, the optimal strategy for cache replacement is to evict items from the cache which are needed *farthest-in-the-future*. This was proposed and proven by Belady [2]. However, since it requires access to future requests, it is not possible to use this algorithm in a practical setting. Therefore, heuristic based algorithms have been proposed to solve this problem. Most of the algorithms rely either on *recency* (LRU, MRU, S4LRU, SLRU[3] etc.) or *frequency* (LFU, MFU, Tiny-LFU[4] etc.) of the request traces. Since these algorithms are human-engineered, they leave a significant room for improvement, and more recently, learning-based algorithms[[5], [6], [7]] have also been proposed to tackle this problem. We want to tackle the problem of caching[8] (both admission and replacement) using Reinforcement Learning (RL).

A lot of computer systems tasks are sequential decision making tasks which can be naturally expressed as Markov decision processes (MDP), and therefore, RL has been used to improve the performance

of computer systems such as datacenter scheduling[9], database query optimization[10] etc. Cache admission and replacement are examples of such tasks, and we explain their sequential nature in subsequent sections.

In this work, we design an RL agent which uses information of the existing objects in the cache and the incoming object to decide whether to admit the object into the cache or not. The goal is to maximise the *byte hit rate*(BHR). We experiment with a multitude of RL algorithms such as n-step Sarsa, Actor-Critic, Sarsa(λ) and REINFORCE [Chapter 10-13 [11]]. Given that the state space is very large, we use different function approximations for our state-space, including linear function approximation, stripe tile coding and neural network [Chapter 9-10 [11]]. We observe a significant improvement over the baselines in the BHR by using an RL agent when LRU is used for cache replacement.

The caching environment we use is built upon the Park project[12]. It uses object size, time since last hit, current cache occupancy as the state features. For generating simulated traces of caching requests, we use Zipfian access distributions as it has been used widely in prior literature to model the request distribution[13, 14], while assuming the object sizes to be the same for simplicity. We study the performance of the RL agents with varying feature space, cache size, and zipf parameters to understand their effects, and find that LFU performs the best compared to LRU and FIFO when the request trace is derived from a stationary zipf distribution.

We further model the cache replacement problem as an adversarial bandit, drawing inspiration from [6]. We treat different candidate replacement agents (LRU, LFU and FIFO) as experts, and assume a full-feedback setting, wherein we have access to the rewards of each expert. Under these assumptions, we employ the Hedge algorithm[15] and show that the algorithm converges to give maximum weight to the best expert.

2 Related Work

There have been different class of approaches in popular literature towards solving the problem of cache replacement using reinforcement learning. [16] models all possible objects as a library, with a hidden popularity distribution. Each item in the library is modelled as an arm of a stochastic bandit problem, and the algorithm aims to minimize the expected number of misses (regret) by selecting the most popular objects to be in the cache. Instead of the objects in the library, we model each candidate replacement policy as an arm, which makes the problem instance independent of the number of unique objects in the library, and greatly reduces the number of arms. [17] follows an imitation learning approach to automatically learn cache access patterns. They model the cache replacement problem as a Markov Decision Process (MDP). Their state space consists of the current cache access, the current contents of the cache, and a history of the past K cache accesses, encoded using an LSTM. We model the cache replacement problem as an *adversarial* bandit instead of an MDP, thereby greatly simplifying the state space. Our approach is similar to [6] which consider a group of known algorithms (for example, LRU, LFU, FIFO etc.) as experts, and select which expert to use at each given cache miss to decide which object to evict. This setting involves simulating a virtual cache for each of the replacement policies. However, they do not specify the details of the machine learning algorithms or reward function that they have used clearly. We solve the adversarial bandit problem assuming a full-feedback setting, using the standard Hedge algorithm[15]. [18] proposes a similar adaptive machine learning based approach, consisting of combining only LRU and LFU. They model regret of each of these policies using a FIFO-based eviction history, and express their weights as a function of the current associated regret. Our reward is similar to how they model the regret of each policy, using an eviction history for each arm. However, their reward is based on the time spent by the object in history, while we simply consider binary rewards based on presence in the history buffer. Additionally, each of the candidate policies rely on the canonical physical cache to decide which object to evict. However we maintain a virtual cache for each policy, and enforce consistency to make sure that the candidate object to be evicted is always present in the physical cache.

While cache replacement policies have received significant traction over the years, most systems use simple LRU for eviction, without any explicit admission algorithm[19]. [20] studies the problem to decide whether to cache a requested object using a feedforward neural network that computes admission probability. It uses a monte-carlo based approach to sample the requests and train the network on simulated data. However, the setting is not modeled as an MDP and there is no explicit

characterization of states and transitions. TinyLFU [4] uses prior statistics and a frequency-based cache admission policy (similar to LFU) to decide on object admission by building upon Bloom filter theory. The algorithm presented is a deterministic one, and is again not modelled as a Reinforcement Learning problem. [21] models the cache admission setting as an MDP, explicitly defining the state-space, and uses a Q-learning based algorithm to approach the problem. The most recent research on using RL directly for cache admission is [22]. They model the cache admission problem as an MDP, with a state space comprising of the object sizes, frequencies and access times. It uses standard advantage actor critic (A2C) on a reduced caching setting i.e. using a subsampling technique to shorten the MDP horizon by hashing the objects in a trace and reducing the cache size. These approaches are the most similar to our work, and we build upon them by experimenting with different algorithms instead of just actor-critic or Q-learning and also implement a cache replacement RL algorithm on top of the cache admission agent.

3 Problem Formulation

3.1 Cache Admission

We model the cache admission problem as a Markov Decision Process (MDP). We consider each sample trace as an episode, with requests coming in sequentially. Initially, the cache is assumed to be empty, and the cache size is fixed. On each cache miss, the admission agent takes a binary decision (i.e, $\in \{0, 1\}$), where 0 denotes *don't admit* and 1 denotes *admit*. The description of our state-space is built upon [22]. Consider the request at timestep t, for the object o_t which leads to a cache miss. Table 1 represents the features we use to build our state-space for the object o_t. The state transitions

Feature	Description
l_t	The time since the last access to the object o_t
f_t	The number of accesses until the timestep t for the object o_t
z_t	The size of the remaining cache at timestep t
i_t	The average interarrival time until the timestep t for the object o_t

Table 1: Description of the state-space for object o_t. We do not include object size as a feature since our trace assumes objects of the same size.

therefore occur only at each cache miss while processing the trace. The reward r_t at each step is the total byte hits until the next cache miss or the end of the episode. Therefore, the reward at each timestep is delayed until the next cache miss.

If the action by the agent is 1 i.e. the agent decides to admit the object in the cache, we will either use a deterministic algorithm such as LRU or an agent based algorithm defined in the next section to decide on which object to replace.

3.2 Cache Replacement

The cache replacement problem is modelled as an adversarial bandit, drawing inspiration from [6]. The adversarial bandit setting consists of a set of m experts (arms), undergoing a sequential process. We consider the setting where before the beginning of the process, an adversary sets the rewards for each expert at each timestep (i.e, the rewards are not adaptive to the decision of the agent at each timestep). Note that this is different from the setting of stochastic bandits, where the rewards are drawn from a fixed distribution for each expert. The objective of the agent is to maximise the total expected reward after T timesteps, at the end of the process by selecting an expert at each timstep. There are two popular settings for how rewards are provided to the agent at each timestep. The *bandit feedback* setting assumes that the agent only has access to the reward of the expert that it selects at each timestep. We focus on the *full-feedback* setting for this case, wherein the rewards of all experts are revealed at each timestep once the agent chooses an expert.

To model cache replacement as an adversarial bandit problem, we treat different candidate replacement policies (LRU, LFU and FIFO) as experts. At each cache miss, when the admission agent decides to admit the object into the cache, the agent now chooses an expert amongst the different candidates to decide on which object to evict. We use a concept of virtual cache which is the simulation of the

cache by each expert. On a cache miss, the object chosen by the expert to evict should be present in the actual physical cache (Note that there would be an actual physical cache which has the objects). Therefore, the state of all virtual cache should be consistent with the actual physical cache. To do so, whenever a page is evicted or admitted to the physical cache, the same operation is performed on the virtual cache as well.

The challenging aspect of this problem is to design the rewards for each expert as the cache state would be same for all the virtual caches. We take inspiration from [18] for our design. While simulating each of the experts along with their virtual caches, we store a FIFO-history of the objects each of the candidate policies would have evicted, upto a fixed length. Our reward is defined by checking the presence of the object requested in these histories, where if it is present, then the reward is 0, otherwise it is 1. Heuristically, this penalises experts for evicting an object in near past, which was going to be requested soon in the future. We also note that there are many different heuristics to design the rewards for these experts, but we leave such experimentation for future work.

4 Optimal Policies

In this section, we discuss the theoretically optimal algorithms for both the problems – cache replacement and cache admission – assuming access to future requests.

4.1 Cache Replacement

The optimal algorithm for cache replacement evicts the item that is going to be accessed farthest into the future. [2] proves optimality of this algorithm.

Algorithm 1 Belady's Algorithm

When object d_i needs to be bought into the cache,
evict the object that is needed farthest into the future.

4.2 Cache Admission

We devise an algorithm for cache admission that is based on 1. On a cache miss, it admits the object if the cache is not full. Otherwise, it compares the next access time of the object with that of all the objects in the cache, and admits if the incoming object would be accessed before any of the objects in the cache.

Algorithm 2 OPT-ADM

if cache is not full **or** $NextAccess(d_i) \leq max_{e \in cache}(NextAccess(e))$ **then**
  Admit object d_i into the cache
else
  Don't admit object d_i into the cache
end if

5 Methodology

5.1 Cache Admission

The state space and features are described in section 3. As the state features can take a large range of values, we use function approximation to represent the state-value and action-value functions. The different function approximations we tried include Tile Coding, Linear function approximation and Neural Networks discussed below.

- **Tile Coding** : We started with Tile Coding assuming interactions between all features. However, this led to a huge state space. On analyzing the features, we discovered that there is not much interaction and we therefore resorted to Stripe Tile Coding [Chapter 9

[11]] which treats each feature independently. The features such as size of remaining cache, number of accesses, time since last access could only take discrete values, therefore the tile width for those dimensions were integers and we tuned it to the maximum value possible which doesn't degrade the performance.

- **Linear Function Approximation**: As the number of features are small and there is not much interaction between the features, we implemented a linear combination of the features to approximation the state-value and action-value function minimizing the Mean Sqaured Error (MSE) loss.
- **Neural Network Approximation**: To observe whether adding further layers on top of the linear function approximation was beneficial, we used a 2-layer network similar to the one described in class programming assignments.

We experimented with a variety of RL algorithms including n-step Sarsa, Actor-Critic, REINFORCE and Sarsa(λ). We used the standard implementation of the algorithms as described in [11]. For Actor-Critic and REINFORCE, we used linear policy approximation. We provide further information about the performance and training details of these algorithms in Section 6.3.

5.2 Cache Replacement

We have described how to model cache replacement as an adversarial bandit problem in Section 3, along with the rewards for each experts. We now discuss a popular algorithm (Hedge[15]) to decide which expert to choose at any given timestep, in the full-feedback setting where arm i receives a profit $r^t(i) \in [0, R]$ during each time-step t for some bounded $R > 0$. ϵ is a parameter of the algorithm, that is typically problem dependent. The algorithm is described in Figure 3. It can be proved that the expected total reward of the algorithm, ALG, satisfies,

$$ALG \geq R_i(1-\epsilon) - \frac{R}{\epsilon} ln(n), \forall i \in [n] \tag{1}$$

where, $R_i = \sum_t r_i^t$ is the total reward accumulated by the i^{th} expert and n denotes the number of arms. Therefore, we are guaranteed to perform within some limit of the best expert, and regret is sub-linear with respect to number of timesteps τ.

Algorithm 3 The Hedge algorithm

Initialisation

$w_i^t = 1 \; \forall \; i \; \epsilon \; \{1, 2...n\}$

for `rounds t = 1, ...` τ **do**

Pick arm i with probability $\frac{w_i^t}{\sum_{i=1}^n w_i^t} = \frac{w_i^t}{W^t}$ and sample it's reward $r^t(i)$

Update the weights of expert i as :

$$w_i^{t+1} = w_i^t (1+\epsilon)^{\frac{r^t(i)}{R}}$$

end for

For our problem, arm i denotes a replacement policy. Each request access irrespective of whether it results in cache hit or miss is a round t. $r^t(i)$ is the reward received by the arm i at round t and $r^t(i) \in \{0, 1\}$. R is set to 1, the maximum reward an arm can get in a round t. The computation of $r^t(i)$ is explained in Section 3.

6 Experiments and Evaluation

6.1 Environment and Dataset

The caching environment we use is built upon the Park project[12]. It takes object trace as the input and simulates an environment for the cache admission setting, same as that described in Section 3. However, we significantly modify the codebase for our purposes. We add code to implement all the different RL agents, add additional state features as described in Section 5.1, and also implement

simulators for each of the different cache replacement policies (LRU, LFU and FIFO). We consider all objects to be of the same size for the purpose of our work.

The dataset that we use consists of simulated traces of caching requests, that are generated using Zipfian access distributions. Previous work [13, 14, 23] suggests that real-world access patterns for web-caching systems can be modelled by the Zip-f distribution, which ensures that the rank-frequency distribution has an inverse relation. Specifically, the distribution of page requests generally follows a Zipf-like distribution where the relative probability of a request for the i^{th} most popular page is proportional to $\frac{1}{i^{\alpha}}$, with α typically taking on some value less than unity.

We train on 50 episodes, and test on 20 episodes. We also maintain a validation dataset of 20 episodes for the purpose of hyperparameter tuning. Each episode consists of $10,000$ requests with approximately 1000 unique objects in each trace. We experimented with training on more number of episodes, but did not find a significant difference in the final byte hit rate. Figure 1 shows the frequency distribution of the objects in a particular trace. Observe that there are only a few objects that occur very frequently.

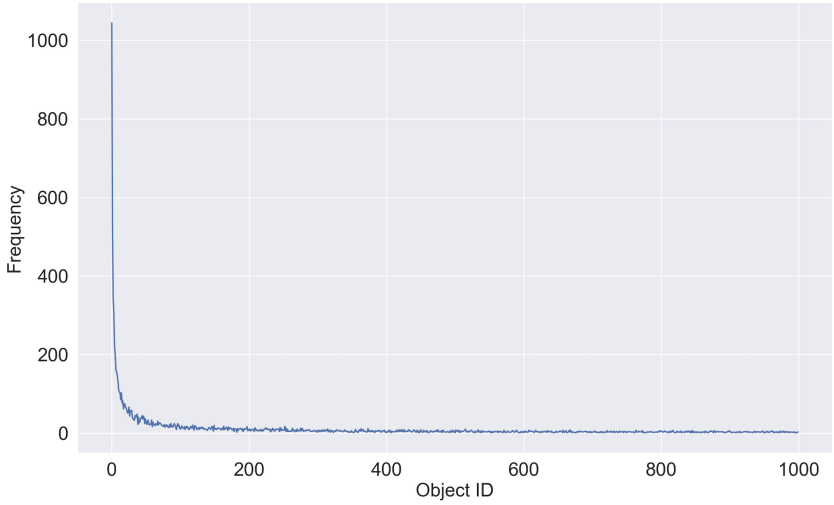


Figure 1: Variation of frequencies with object IDs for a sample trace in our dataset.

6.2 Baseline Performance

We use two baselines to compare against our RL agents. In most operating system level caches, no admission agent is typically used, and therefore, we use this as our first baseline. In this case, at each cache miss, we decide to admit the object into the cache and evict the item based on a fixed replacement agent (LRU, LFU, FIFO or a combination). We refer to this algorithm as *Always-Admit*. Our second baseline is a modification on top of this agent, and at each cache miss, it accepts the item into the cache with probability, $p = 0.5$. We refer to this algorithm as *Random-Admit*. Table 2 shows the variation of the object hit rate of each of the algorithms on the test episodes with different cache sizes, fixing the replacement agent to be LRU. We also provide the BHR achieved by the theoretical optimal algorithm for comparison and providing an upper-bound. Each value has been averaged over 10 repetitions, and we also provide the standard deviation of the values. We note that *Always-Admit* and *Random-Admit* have almost similar performance, with *Random-Admit* performing slightly better. The BHR increases with increasing cache size since on average more items can be accommodated in the cache.

6.3 Comparison between different RL agents

As mentioned in previous sections (Section 5.1), we experiment with a variety of RL algorithms for cache admission, in combination with different function approximations, fixing LRU as the replacement policy. In this section, we only focus on the best-performing function approximations for each algorithm, and discuss other alternatives in Section 6.4. The discount factor γ has been set to 1 as BHR is the fraction of total bytes hit across the episode and every hit across the entire episode

Cache Size	Always-Admit	Random-Admit	OPT-ADM
10	0.137 ± 0.001	0.151 ± 0.001	0.364 ± 0.001
20	0.221 ± 0.002	0.24 ± 0.002	0.457 ± 0.001
50	0.357 ± 0.002	0.375 ± 0.002	0.586 ± 0.001
75	0.424 ± 0.001	0.442 ± 0.001	0.644 ± 0.001
100	0.475 ± 0.001	0.492 ± 0.002	0.685 ± 0.001

Table 2: *Byte Hit Rate* (BHR) of various baseline policies with different cache size

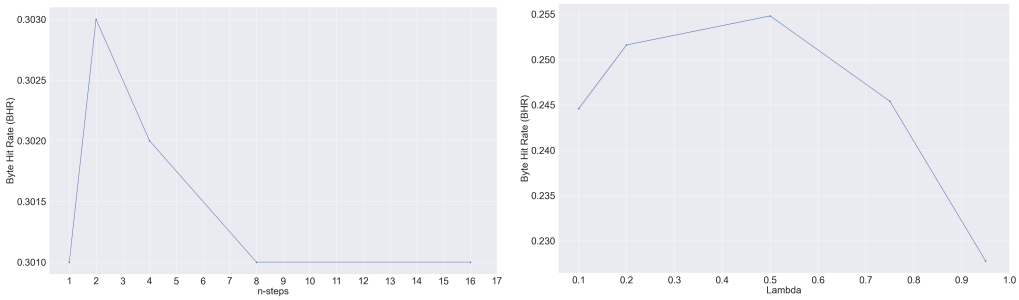
has the same impact on the final rate. We provide the training details and performance achieved by these algorithms below.

- **n-step Sarsa** : We implement the algorithm specified in [11][pg-147]. A 2-layered neural network was used as the function approximator for the action-value function. We experimented with different values for the number of steps, n, and tuned the learning rate, $\alpha \in \{0.001, 0.01, 0.1\}$ and the exploration parameter $\epsilon \in \{0.05, 0.1, 0.2, 0.5\}$. The parameters for which the algorithm achieved the highest validation BHR were $n = 2$, $\alpha = 0.01$ and $\epsilon = 0.1$. Figure 2a shows the variation of performance on the test set for different values of n, for cache size = 20. As expected, the BHR first increased with increasing n and peaked at $n = 2$, decreasing afterwards. We observed similar trends for other cache sizes as well.
- **True Online Sarsa(λ)** : We implement the True Online Sarsa(λ) algorithm described in [11][pg-307]. We used stripe tile coding for our function approximation. We experimented with different values of the eligibility-trace parameter $\lambda \in \{0.1, 0.2, 0.5, 0.75, 0.95\}$, the learning rate, $\alpha \in \{0.001, 0.01, 0.1\}$ and the exploration parameter $\epsilon \in \{0.05, 0.1, 0.2, 0.5\}$. The parameters for which the algorithm achieved the highest validation BHR were $\lambda = 0.5$, $\alpha = 0.01$ and $\epsilon = 0.1$. Figure 2b shows the variation of BHR on test set with different values of λ for cache size = 20. As expected, the BHR first increased with increasing λ and peaked at $\lambda = 0.5$, decreasing afterwards. Surprisingly, Sarsa(λ) didn't perform as well as n-step Sarsa in our case.
- **REINFORCE with Baseline** : The algorithm was implemented taking reference from [11][pg-330]. We set α^θ and α^w to 0.001. The baseline approximated the state value function and we used linear function approximation to represent it. The policy was also represented using a linear function approximation. REINFORCE didn't perform as well other algorithms as depicted in Table 3.
- **Actor-Critic with Eligibility Traces** : We implement the Actor-Critic with eligibility traces algorithm as mentioned in [11][pg-333]. We used stripe tile coding as our function approximation for the critic(state-value function), and linear function approximation for the actor(policy). The hyper-parameters we tuned includes the learning rates, $\alpha^\theta = \alpha^w \in \{0.001, 0.01, 0.1\}$ and eligibility-trace parameters $\lambda^\theta = \lambda^w \in \{0.2, 0.4, 0.6, 0.8\}$. The parameters that achieved the highest BHR on validation set were $\alpha^\theta = \alpha^w = 0.001$ and $\lambda^\theta = \lambda^w = 0.8$.

Cache Size	n-step Sarsa, $n = 2$	Sarsa(λ), $\lambda = 0.5$	REINFORCE	Actor-Critic
10	0.215 ± 0.002	0.165 ± 0.02	0.15 ± 0.005	**0.225** ± 0.01
20	0.302 ± 0.004	0.255 ± 0.01	0.24 ± 0.007	**0.305** ± 0.009
50	**0.41** ± 0.004	0.364 ± 0.02	0.369 ± 0.024	0.395 ± 0.03
75	0.453 ± 0.003	0.424 ± 0.01	0.434 ± 0.018	**0.462** ± 0.03
100	0.477 ± 0.004	0.472 ± 0.013	**0.490** ± 0.015	**0.491** ± 0.015

Table 3: *Byte Hit Rate* (BHR) of various RL algorithms with different cache sizes. Each value has been averaged over 10 repetitions, and we provide the mean and standard deviations across these runs.

Table 3 shows the BHR of the RL algorithms mentioned above with the best hyperparameters on different cache sizes. n-step sarsa and Actor-Critic perform the best for all cache sizes compared



(a) Performance of n-step Sarsa with varying n. (b) Performance of Sarsa(λ) with varying λ.

Figure 2: Variation of Sarsa performance at cache size=20

to Sarsa(λ) and Reinforce. We only observed minor difference in the performance of n-step sarsa and Actor-Critic which can even be due to the variance in input traces or random seeds set. Figure 3 shows the comparison of the best performing RL algorithms, i.e Actor-Critic and n-step Sarsa ($n = 2$), along with the *Random-Admit* baseline and the theoretical optimal algorithm, *OPT-ADM*, for reference. We observe that the RL algorithms clearly outperform the random baseline for smaller cache sizes. However for larger cache sizes, the difference is less apparent. This is most probably because at larger cache sizes, more items can be kept in the cache, and the overall byte hit rate increases even for the baseline, making it tough to improve upon.

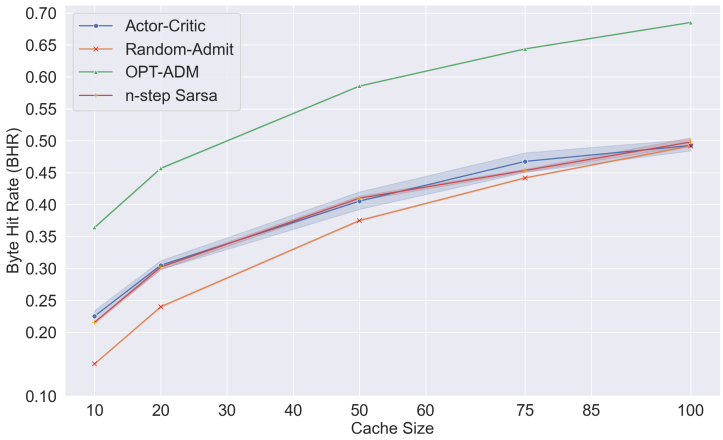


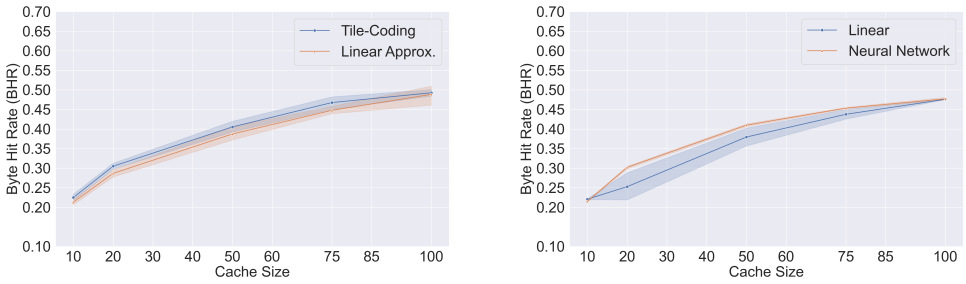
Figure 3: Comparison of Byte Hit Rate on Test data of various algorithms vs Cache Size. The shaded areas denote 95% confidence intervals over 10 observations.

6.4 Comparison between different function approximations

In this section, we discuss the different function approximations we experimented with, on the best-performing RL algorithms.

For **Actor-Critic with eligibility traces**, we experimented with two different function approximation for the state-value function.

- **Linear Function Approximation:** We used the standard linear function approximation with MSE as the loss function and Adam optimizer.
- **Tile Coding:** We used 1-D tile coding (stripe tile coding) as described previously (Section 5.1). Increasing the number of tilings didn't increase the validation BHR, so we kept the number of tilings as 1. We used the tiling width for the features (described in Table 1) as follows:



(a) Actor-Critic with tile-coding and linear function approximations

(b) n-step Sarsa with linear and neural-network function approximations

Figure 4: Comparison between different function approximations for Actor-Critic and n-step Sarsa. The shaded areas represent the 95% confidence intervals over 10 repetitions.

- l_t : 10
- f_t : 50
- z_t : 1
- i_t : 10

The tile widths were selected to be the maximum value which didn't decrease the BHR on validation set. We observed that Tile Coding performed better than linear function approximation. Figure 4a shows the BHR on test data for actor-critic with both the function approximations.

For n**-step Sarsa**, we experimented with linear function approximation and $2-$layered nerual network.

- **Linear Approximation:** We started with the standard linear approximation for the action-value function and used the Adam optimizer with MSE loss.
- $2-$**layer Neural Network:** We increased the number of layers to 2 to assess the performance gain. $2-$layer neural network performed better than linear approximation. Figure 4b shows the comparison of BHR on the test set for $n = 2$. We also observe that the confidence intervals for the BHR become narrower on using the 2-layered neural network. Further increasing the layers of neural network didn't cause any increase in performance. We set the number of nodes in each layer to 5, same as the feature vector length and used Adam optimizer with MSE loss.

6.5 Cache Replacement Policies

As mentioned in section 3.2, we analyzed the performance of three different cache replacement policies - LRU, LFU and FIFO. We observed that LFU achieved the highest BHR on all cache sizes when using *Always-Admit* as the admission agent. Using *Random-Admit* shows a similar trend with LFU performing best. Table 4 shows the test BHR of the replacement policies on different cache sizes.

Cache Size	LRU	LFU	FIFO	LRU + LFU + FIFO
10	0.137 ± 0.001	**0.262** ± 0.003	0.132 ± 0.009	0.257 ± 0.004
20	0.221 ± 0.002	**0.343** ± 0.002	0.212 ± 0.01	0.332 ± 0.002
50	0.357 ± 0.002	**0.456** ± 0.002	0.355 ± 0.008	**0.454** ± 0.001
75	0.424 ± 0.001	**0.514** ± 0.002	0.421 ± 0.006	**0.514** ± 0.002
100	0.475 ± 0.001	**0.555** ± 0.002	0.471 ± 0.006	**0.558** ± 0.001

Table 4: *Byte Hit Rate* (BHR) of various replacement policies with different cache sizes

We used the cache replacement algorithm described in Section 5.2 with the three policies mentioned above to show that the algorithm converges to the best performing replacement policy i.e. LFU.

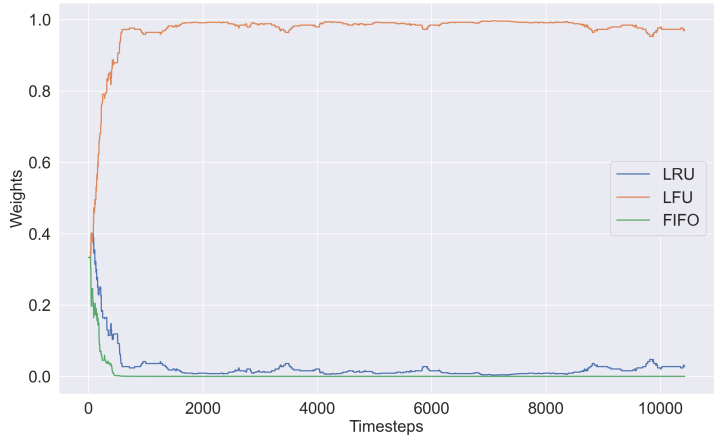


Figure 5: Variation of weights of LRU, LFU and FIFO with timesteps in an episode. The final values of the weights depict the relative importance of each of the algorithms, with the order $LFU > LRU > FIFO$.

The Hedge hyperparameter ϵ was set to 0.1. The graph 5 shows how the weights assigned to each of the policies change with the time-steps of an episode. The algorithm converges to LFU after approximately 1000 time-steps.

Next, we experimented with the RL algorithms described in Section 6.3 on LFU. We use the same set of hyperparameters that performed the best on LRU. The RL algorithms didn't achieve any performance improvement over the *Always-Admit* and *Random-Admit* policy. The only algorithm which performed on-par with the baselines was REINFORCE. The graph 6 shows the performance of REINFORCE and the baselines on different cache sizes. We observed that *Random-Admit*, *Always-Admit* and REINFORCE achieved almost similar performance with LFU on different cache sizes. This indicates that it is certainly difficult to improve upon LFU baselines for the cache admission setting.

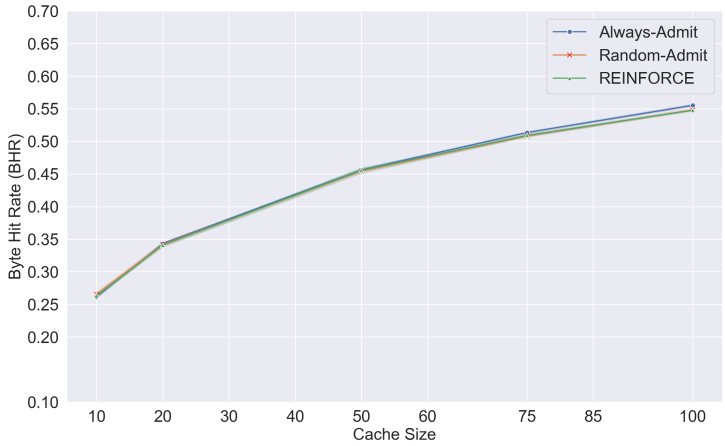


Figure 6: Comparison of BHR on Test data of various algorithms with LFU replacement

6.6 Cache Admission and Replacement

The only algorithm which performed on-par with baselines when LFU was used as the cache replacement policy was REINFORCE. So, we used REINFORCE as the admission agent and used

the cache replacement agent as described in section 5.2 with LRU, LFU and FIFO as replacement policies. We observed that the BHR obtained by using REINFORCE with the cache replacement agent was comparable to the baselines with LFU.

Cache Size	Always-Admit (LFU)	Random-Admit (LFU)	REINFORCE (LFU)	REINFORCE (LRU + LFU + FIFO)
10	0.262 ± 0.003	0.266 ± 0.004	0.263 ± 0.002	0.27 ± 0.006
20	0.343 ± 0.002	0.342 ± 0.004	0.341 ± 0.004	0.342 ± 0.003
50	0.456 ± 0.002	0.454 ± 0.002	0.455 ± 0.005	0.459 ± 0.002
75	0.514 ± 0.002	0.508 ± 0.002	0.509 ± 0.003	0.513 ± 0.005
100	0.555 ± 0.002	0.548 ± 0.002	0.548 ± 0.001	0.555 ± 0.003

Table 5: *Byte Hit Rate* (BHR) of baselines and REINFORCE with cache replacement agent

6.7 Feature Space Exploration

In addition to the features mentioned in Table 1, we experimented with the following features :

- **Frequency based rank of the incoming object**: The intuition was that this would help the agent decipher the underlying probability distribution. However, since we were already using the object frequency as a feature, we did not find much improvement in performance.
- **Difference between last access time of the incoming object and the minimum last access time of the cache items** : The reasoning behind using this feature was to provide features that capture the relative recency of the incoming object with the objects in the cache. However, this feature also did not lead to a significant lift in performance as the last access time of the incoming object was already included.

7 Conclusion

In this work, we use reinforcement learning to solve the problem of cache admission and cache replacement. We found that using Actor Critic with stripe tile coding and n-step Sarsa leads to significant increase in BHR over always (or randomly) admitting the object into the cache on a miss, while using LRU for replacement. However, we didn't observe any such improvement when using LFU as a cache replacement policy. We used an adversarial bandit setting to model cache replacement. The experiments showed that our algorithm converged to the best possible replacement policy among the pool of chosen policies (LFU, LRU, FIFO).

References

[1] Mukaddim Pathan, Ramesh K Sitaraman, and Dom Robinson. *Advanced content delivery, streaming, and cloud services*. John Wiley & Sons, 2014.

[2] Laszlo A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal*, 5(2):78–101, 1966.

[3] Ramakrishna Karedla, J Spencer Love, and Bradley G Wherry. Caching strategies to improve disk system performance. *Computer*, 27(3):38–46, 1994.

[4] Gil Einziger, Roy Friedman, and Ben Manes. Tinylfu: A highly efficient cache admission policy. *CoRR*, abs/1512.00727, 2015.

[5] Zhan Shi, Xiangru Huang, Akanksha Jain, and Calvin Lin. Applying deep learning to the cache replacement problem. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 413–425, 2019.

[6] Ismail Ari, Ahmed Amer, Robert Gramacy, Ethan Miller, Scott Brandt, and Darrell Long. Acme: Adaptive caching using multiple experts. *Proceedings in Informatics*, 01 2002.

[7] Yuanyuan Zhou, James Philbin, and Kai Li. The multi-queue replacement algorithm for second level buffer caches. In *USENIX Annual Technical Conference, General Track*, pages 91–104, 2001.

[8] Alan Jay Smith. Cache memories. *ACM Computing Surveys (CSUR)*, 14(3):473–530, 1982.

[9] Azalia Mirhoseini, Hieu Pham, Quoc V. Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean.

[10] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph Hellerstein, and Ion Stoica. Learning to optimize join queries with deep reinforcement learning. *arXiv preprint arXiv:1808.03196*, 2018.

[11] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[12] Hongzi Mao, Parimarjan Negi, Akshay Narayan, Hanrui Wang, Jiacheng Yang, Haonan Wang, Ryan Marcus, ravichandra addanki, Mehrdad Khani Shirkoohi, Songtao He, Vikram Nathan, Frank Cangialosi, Shaileshh Venkatakrishnan, Wei-Hung Weng, Song Han, Tim Kraska, and Dr.Mohammad Alizadeh. Park: An open platform for learning-augmented computer systems. volume 32. Curran Associates, Inc., 2019.

[13] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, pages 53–64, 2012.

[14] Benjamin Berg, Daniel S Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, et al. The {CacheLib} caching engine: Design and experiences at scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 753–768, 2020.

[15] Sanjeev Arora, Elad Hazan, and Satyen Kale. The multiplicative weights update method: a meta-algorithm and applications. *Theory of computing*, 8(1):121–164, 2012.

[16] Archana Bura, Desik Rengarajan, Dileep M. Kalathil, Srinivas Shakkottai, and Jean-François Chamberland-Tremblay. Learning to cache and caching to learn: Regret analysis of caching algorithms. *CoRR*, abs/2004.00472, 2020.

[17] Evan Zheran Liu, Milad Hashemi, Kevin Swersky, Parthasarathy Ranganathan, and Junwhan Ahn. An imitation learning approach for cache replacement. *CoRR*, abs/2006.16239, 2020.

[18] Giuseppe Vietri, Liana V. Rodriguez, Wendy A. Martinez, Steven Lyons, Jason Liu, Raju Rangaswami, Ming Zhao, and Giri Narasimhan. Driving cache replacement with ML-based LeCaR. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, Boston, MA, July 2018. USENIX Association.

[19] Daniel S Berger, Ramesh K Sitaraman, and Mor Harchol-Balter. Achieving high cache hit ratios for cdn memory caches with size-aware admission. 2016.

[20] Vadim Kirilin, Aditya Sundarrajan, Sergey Gorinsky, and Ramesh K. Sitaraman. Rl-cache: Learning-based cache admission for content delivery. *IEEE Journal on Selected Areas in Communications*, 38(10):2372–2385, 2020.

[21] Sami Alabed. Rlcache: Automated cache management using reinforcement learning. *CoRR*, abs/1909.13839, 2019.

[22] Haonan Wang, Hao He, Mohammad Alizadeh, and Hongzi Mao. Learning caching policies with subsampling. In *NeurIPS Machine Learning for Systems Workshop*, 2019.

[23] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and zipf-like distributions: Evidence and implications. In *IEEE INFOCOM'99. Conference on Computer Communications. Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. The Future is Now (Cat. No. 99CH36320)*, volume 1, pages 126–134. IEEE, 1999.