A Reinforcement Learning-Based Replacement Strategy for Content Caching

Peihao Wang, Yuehao Wang & Rui Wang
School of Information Science and Technology
ShanghaiTech University
{wangph,wangyh3,wangrui3}@shanghaitech.edu.cn

Abstract

We adopt a learning-based method to cache replacement strategy, aiming to improve the miss rate of existing traditional cache replacement policies. The main idea of modeling is to regard the strategy as a MDP so that we can employ DRL to learn how to make decision. We refer to Zhong et al. (2018) and design a similar MDP model. The learning backbone, however, is value-based DQN. Our main effort is to use short-term reward to optimize the long term miss rate, and further adopt the network content caching system to file system, testing the DRL agent with realistic file accesses.

1 Introduction

Caching is a very commonly used mechanism in contemporary computer systems, which was introduced to computer architecture as a repository for copies that can be accessed by CPU more quickly than by direct addressing. In addition to the CPU cache as a module in memory management unit (MMU), it also exists between memory and disk, base station and coordinators. With the development of networking technology, content caching becomes general and has been adopted to network server where frequent requests require a relatively high performance of reading or writing resources. An essential part of caching mechanism is the replacement policy, since limited size buffer cannot accommodate all requested resource. Usually, the performance of a cache system is evaluated by miss rate: the proportion of miss times of the total access times, and hit rate: $1 -$ miss rate. Cache performance can be simply improved by increasing cache capacity but this solution is costly and even infeasible for some devices with a high cohesion architecture. Since the early 1970s, computer scientists have started focusing on study of replacement strategies. Many traditional deterministic replacement algorithms have been invented aiming to minimize cache miss rate. However, each of these algorithms has flaws in different aspects and cannot produce substantial hit rate under various circumstances, since it is unrealistic for these traditional algorithms to be adaptive to all requested cases.

In this report, we adopt Reinforcement Learning to replacement strategy, which brings in adaptability and prediction to caching system. Once the cache is filled and our Reinforcement Learning agent receives a request, it will choose to evict a certain cache entry based on a Markov Decision Process (MDP) learnt via previous experience.

2 Related Work

The big idea of replacement strategy is that for a full cache, when a requested content needs to be cached in, the cache entry whose next request will occur farthest in the future should be cached out. This idea is also called Optimal Replacement Policy, which is clairvoyant and demands predictions of the future requests. Although Optimal Replacement Policy is an ideal approach without any actual implementation, many traditional replacement algorithms like LRU, FIFO, LFU, MRU, etc. attempt to perform a reasonable and deterministic policy which maximizes the hit rate under certain assumptions. According to analysis on different replacement algorithms

Ahmed et al. (2013), LRU has a higher performance over most of traditional algorithms in common use cases.

Least Recently Used algorithm, a.k.a LRU, is a very popular and traditional replacement strategy applied in caching mechanism. This algorithm will track each cache entry and record their last used time. Once a missing cache access reaches, if the cache is filled, the algorithm will evict the entry which has the earliest last used time and a new entry will be added to cache the requested content. Moreover, many approximate algorithms of LRU, like Clock Algorithm and The Second-Chance Algorithm, improves caching efficiency and memory overhead at the expense of losing certain hit rate. However, this sort of algorithms also face a big issue, where request arrival pattern is close to sequential flooding. In the case of sequential flooding, the series of requests has a repeatedly consecutive access order, e.g. 1, 2, 3, 4, 1, 2, 3, 4 ..., thus LRU will cause a 100% miss rate theoretically in this way. On the contrary, MRU, short for Most Recently Used algorithm, manages to handle sequential flooding but lacks hit rate in many other cases. Another policy named Least Frequently Used (LFU) also gives a good performance even better than LRU in some cases. LFU tracks the access times of cache content and evicts the one which has the least access times. The major problem of LFU is that new cache entries that was just cached are subject to being replaced very soon, since new cache entries always start with a low access times Zivkov & Smith (1997). Traditional replacement strategies are deterministic and lack adaptability. This is why most of cache systems have their own bottlenecks in terms of replacement performance.

Rapid advances in machine learning technology during the last decade provide various methods for relatively accurate prediction tasks, e.g. stock trend prediction and weather prediction. Due to cache replacement performance can benefit a lot from accurate prediction of future requests, machine learning methods are expected to be feasible and promising in cache replacement. Many related work have tried different learning-based methods and models, aiming to refine cache replacement strategies.

Vietri et al. (2018) introduces a machine learning-based cache replacement method, named LeCaR. The work harness Reinforcement Learning to perform an online learning process. The action space is composed of two different traditional replacement policies: LFU and LRU. The learning agent is supposed to learn a set of weights for the two policies and choose the policy with a higher weight once a cache replacement is demanded. According to its given results, the work shows a timely and adaptive behavior when requests arrive in an alternating LRU pattern and LFU pattern. Nevertheless, if the pattern changes too fast, its adaptive behavior will be delayed and produce considerably low performance in terms of miss rate. In addition, since the learning agent in this method only makes decision between LRU and LFU, its adaptability is limited even though its decision-making is light-weight and fast.

Zhong et al. (2018) presents another Reinforcement Learning-based approach for content caching. Its modeling of state space and action space are much more complex than Vietri et al. (2018). Its framework is based on Wolpertinger Policy Dulac-Arnold et al. (2015) and consists of three main parts: actor network, KNN, and critic network. This work introduces a concept of short-term, middle-term and long-term requests, which indicate the last 10, 100, 1000 requests, respectively. The state is represented by a set of features according to cache access frequencies in short-term, middle-term and long-term. Since the state space is very large in its design, the work trains the replacement strategy using Deep Deterministic Policy Gradient (DDPG). The learning agent is expected to learn which cache entry should be evicted once a replacement is brought about, i.e. the actions are represented by eviction choices of cache entries. In this way, the Actor-Critic model is used here to avoid the listing and consideration of the entire action space, which is large as well. Reward of each decision epoch is a weighted sum of short-term and long-term hit rate. With these complex designs, the framework can adapt to data in an online process and enables the agent to develop a long-term policy. In accordance with its experimental results, cache hit rate can be raised by over 10% compared with LRU.

Learning-based decision-making process is fairly exhausted in both time and space utilization compared with traditional approaches, Wang et al. provides a subsampling method to reduce the size of caching environment resulting in an MDP with shorter time horizon. Differentiating from those methods mentioned above, this work does not focus on cache replacement strategy, while its learning agent learns whether the incoming content is worth caching. Its action space only contains two actions: "cache" or "not cache". Once action "cache" is chosen and the cache is filed, LRU replacement will be performed to make room for the new content. Moreover, this work brings in content size into feature vectors of states. Therefore, the learning agent is also obligated to make decision according to the size of requested content.

Our method mostly bases on those concepts introduced by Zhong et al. (2018). Differently, we adopt Deep Q Network Mnih et al. (2013) to train our learning agent. Besides, we also design a new reward function which will be elaborated in Section 3.1.

3 Methodology

There are basically two schemes of replacement policy: The one is that, on cache miss, system must swap data into the cache buffer to proceed the current request. This scheme is applied to CPU cache, translation lookaside buffer (TLB), etc. The other does not compulsorily require swapping-in. If the requested data is not cached, system can alternatively guide the user to the destination without retrieving data. This scheme works on file systems and network stations. In our work, we experiment both schemes, but the result shows the minimal difference, see Section 5.

Moreover, to simplify the complexity of our problem, we assume the caching mechanism is fully associative and works on a file system, which can support both two schemes of replacement policy mentioned above. We denote the cache size as C, which means C slots for buffering data blocks. Typically, the file system cache stores data blocks with fixed size rather than a whole file content into an entry, therefore, we can consider the backend caching system only observes requests on blocks, and every read and write has the same access time due to the fixed size property. Notice that, we only care non-compulsory miss, hence we should fill up the cache initially. And all the cache misses mentioned in this report should be interpreted as non-compulsory misses.

We give each block on disk driver an ID number, which ranges from 1 to N, where $N \gg C$ is the total count of blocks on the device. Afterward, we can regard the disk accesses as a temporal sequence of the block IDs, which can be represented as $\mathcal{R} = \{R_1, R_2, ..., R_\tau, ..., R_\mathcal{T}\}$, where $R_\tau \in \{1, ..., N\}$. Similarly, we can consider that the cache content at t-th requests is a sequence with fixed size of the block IDs, which is denoted as $\mathcal{C}_\tau = \{E_1, E_2, ..., E_k, ..., E_C\}$, where $E_k \in \{1, ..., N\}$. We can see $\mathcal{C}_\tau$ only varys when requests are coming, hence our definition w.r.t. request time τ makes sense. Cache miss happens when $R_\tau \notin \mathcal{C}_\tau$ and $\mathcal{C}_\tau$ will differ from $\mathcal{C}_{\tau+1}$ after replacement policy is taken.

3.1 MDP Formulation

In our work, we consider the cache replacement policy as a Markov decision problem (MDP), as the current cached content is determined only by the previous state and the action taken during replacement. A typical MDP consists of finite state space $\mathcal{S}$, action space $\mathcal{A}$, reward utility function $R(\cdot)$ and transition probability $T(\cdot)$. Stating from an initial state $s_1 \in \mathcal{S}$, the agent takes an action $a_t \in \mathcal{A}$ at t-th decision epoch and transits its current state $s_t \in \mathcal{S}$ to $s_{t+1} \in \mathcal{S}$ following a probability distribution. Meanwhile, the agent will gain a reward r to evaluate this action. Therefore, the transition probability and the reward are considered to be a function of the two states and the taken action: $T(s_t, a_t, s_{t+1})$ and $R(s_t, a_t, s_{t+1})$.

As for our cache model, we define the decision epoch falls on cache miss, i.e. every cache miss should be handled on decision epoch and decisions are only made on cache miss. We consider both currently requested data and cached resources as the state space $\mathcal{S}$. We apply the second replacement scheme by default, so the actions $\mathcal{A}$ taken on decision epoch is which cache resource should be evicted and skipping this epoch is prohibited. The transitions should be non-deterministic. Although caching systems are commonly considered as a denoised environment (i.e. a swapping process will not cause side effects, evicting unexpected items out), the next requested data cannot be predicted.

Another non-deterministic factor is the reward. Even though we observe identical transitions, the reward could be distinct. This is difference from the typical definition of MDP, but we notice that model-free reinforcement learning can conquer this issue that will be introduced in Section 3.2.

State Space We consider both currently requested block and currently cached block as the state space $\mathcal{S}$. The initial state s_1 is the first non-compulsory cache miss. Basically, every state s_t at decision epoch t can be represented by $(\mathcal{C}_\tau, R_\tau)$, where $\tau = 1, ..., \mathcal{T}$ and $R_\tau \notin \mathcal{C}_\tau$. But the space would be very large as $\binom{C+1}{N}$, which makes exploration so far to be sufficient to and our learning model hard to converge. Therefore, we employ numerical feature vectors to encode each state as $\boldsymbol{s}_t$. We use $\boldsymbol{\mathcal{F}}_k$ to represent the features of the each accounted resource, and the state vector turns out to be $\boldsymbol{s}_t = \{\boldsymbol{\mathcal{F}}_0, \boldsymbol{\mathcal{F}}_1, ..., \boldsymbol{\mathcal{F}}_C\}$, where $\boldsymbol{\mathcal{F}}_0$ is the feature vector of the currently requested blocks, and $\boldsymbol{\mathcal{F}}_i$ is the feature vector of i-th cached resource, $i = 1, ..., C$. We will discuss our feature extraction in Section 4.

Action Space Our definition of the action space is straight-forward. First, to limit the action space $\mathcal{A}$, we assume on each decision epoch, at most one cached item can be swapped out. Let $a_t \in \mathcal{A}$ is the action selected on decision epoch t, transiting states from s_t to s_{t+1}. For the first replacement scheme, let $\mathcal{A} = \{0, 1, ..., C\}$, where $a_t = 0$ means skipping eviction and ignoring the request block, and $a_t = 1, ..., C$ means swapping out cached item at a_t and caching new block into that slot. For the second scheme, let $\mathcal{A} = \{1, ..., C\}$, which enforces buffering the currently request data. By default, we follow the second scheme, and the action state is $\mathcal{A} = \{1, ..., C\}$ unless special statements.

Reward Reward should be given immediately after taking an action due to our learning method present in Section in 3.2. Therefore, our DRL agent can only receive the short-term feedback, at most as long as the interval between two decision epoch. Therefore, our goal lies in utilizing short-term feedback to optimize the overall cache miss, i.e. long-term cache miss. Compared with Zhong et al. (2018), our cache system only provide short-term reward but no long-term, considering exposing long-term information is unrealistic, since computer systems never foresees the subsequent requests in practice. We define the most basic reward of transition (s, a, s') as Equation 1.

$$r(s, a, s') = H(s, a, s') + \lambda P(s, a, s') \cdot \mathbf{1}(a \text{ causes miss}) \tag{1}$$

where $H(s, a, s')$ is the reward term and $P(s, a, s')$ is the penalty term. λ balances the proportion between two terms. $\mathbf{1}(*) = 1$ if $*$ takes place, otherwise $\mathbf{1}(*) = 0$. The reward term is supposed to be large when the selected action is judicious and results in relatively high hit count before the next decision epoch. Hence, it is defined as a weighted sum of hit count on each cache slot, formulated as Equation 2.

$$H(s, a, s') = \sum_{i}^{C} w_i(h_i(t+1) - h_i(t)) = \boldsymbol{w}(\boldsymbol{h}(t+1) - \boldsymbol{h}(t)) \tag{2}$$

where $\boldsymbol{h}(t)$ is the histogram vector of cumulative hit count of each cached content at t-th decision epoch. We employ a weight $\boldsymbol{w}$ here to highlight difference on each cache slot. For instance, we can increase the weight on newly cached item to endorse the latest selected actions. The concrete parameters will be present in Section 4. As for the penalty term, it is supposed to bring a reduction to the final reward when the next cache miss is exactly caused by the previously taken action. It should also hold the property that the penalty decreases as the hit count accumulated between two decision epochs. Thus, we simply use Equation 3 to represent our penalty.

$$P(s, a, s') = \mu + \frac{\psi}{H(s, a, s')^\kappa + 1} \tag{3}$$

where μ, ψ, κ are the hyperparameters of penalty. Especially, they should satisfy $\mu \leq 0$, $\psi < 0$ and $\kappa \geq 1$, Since we only observe the short-term states, we need to be cautious about the penalty in case that the previous action causes the next miss but results in the lower overall cache miss.

3.2 Reinforcement Learning

In addition to our MDP settings, we need to adopt reinforcement learning to learn these reward along with the transition probabilistic distribution. Our learning agent is model-free, which means it only takes account of the expected utility of each action taken on each state, instead of compute the reward and find probabilistic distribution finally. It can also explore the system environment actively by taking actions based on currently learned model. These properties inspire us to leverage Q-Learning, a value-based method, to empower our adaptive caching system.

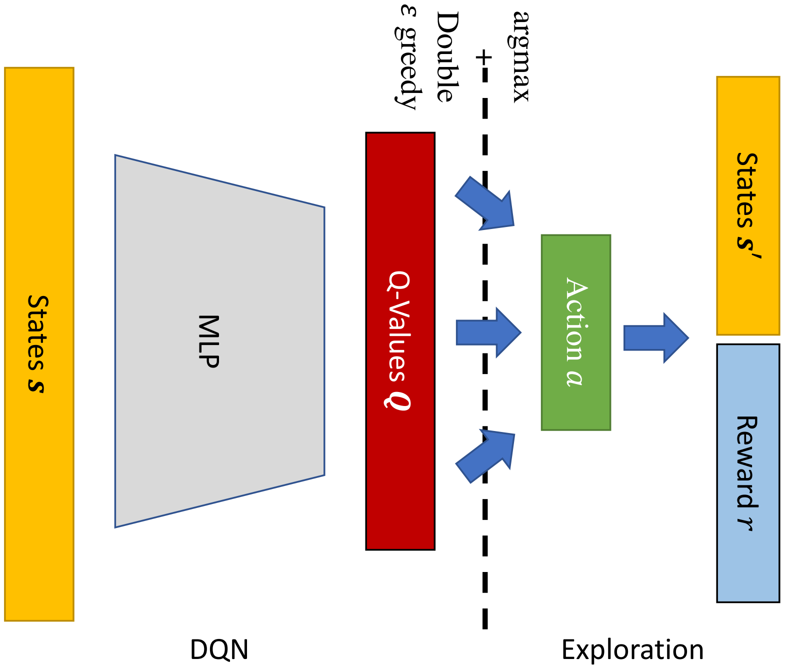

Figure 1: The whole pipeline in one decision epoch. Taking an old state s and transiting to the next state s' with a feedback r. **Left:** Our DQN architecture. Input is a state feature vector and output is a Q-value vector, each element of which is the Q-value by taking the action corresponding to the index. **Right:** The decision making process. We adopt ε-greedy here to balance exploration and exploitation.

Deep Q-Network Traditional Q-Learning updates Q-values $Q(s, a)$ iteratively after experiencing a transition. Due to the huge state space and action space, we cannot use the traditional Q-Learning to store every Q-value in a hash table. Instead, we exploit feature-based Q-Learning, a.k.a approximate Q-Learning. The Q-values are computed by feeding real-time state features $\boldsymbol{s}$ and applying function $f_{\boldsymbol{\theta}} : \boldsymbol{s} \mapsto \boldsymbol{Q} \in \Re^{|\mathcal{A}|}$, where each element $\boldsymbol{Q}_a$ of $\boldsymbol{Q}$ is the Q-value $Q(s, a)$ by taking action a.

By exploring the environment, suppose the DRL agent experiences a new transition (s, a, s') and gains reward r, the Q-values can be updated on-the-fly through these observations. The target Q-values $Q'_k(s, a)$ is calculated from the current Q-values $Q_{k-1}(s, a)$ in Equation 4

$$Q'_k(s, a) = r + \gamma \max_{a'} Q_{k-1}(s', a') \tag{4}$$

where γ is known as the discount factor in reinforcement learning, k is the number of parameter update. If we update $\boldsymbol{\theta}$ at T_0 period, $T_0 \cdot k = t$, where t is the current decision epoch, which will be introduced further in Section 3.2. The traditional Q-Learning chooses to update Q-values by optimizing the temporal difference error (a.k.a. TD difference) between current Q-values and target Q-values. But with the feature representation, the update on Q-values can only be done on the parameters $\boldsymbol{\theta}$.

One of the most popular method to optimize the temporal difference is gradient descent. We adopt the squared form of TD difference as the loss function, formulated by Equation 5.

$$\mathcal{L} = \|Q'_k - Q_k\|^2 \tag{5}$$

Afterward, the update on parameters $\boldsymbol{\theta}$ can represented by Equation 7.

$$\begin{aligned}\boldsymbol{\theta} &= \boldsymbol{\theta} - \alpha \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\boldsymbol{\theta}} & (6)\\ &= \boldsymbol{\theta} + \alpha(Q'_k - Q_k)\frac{\mathrm{d}Q_k}{\mathrm{d}\boldsymbol{\theta}} & (7)\end{aligned}$$

where α is known as learning rate, and $Q'_k - Q_k$ again appears as TD difference.

In DRL, multi-layer perceptron (MLP) is chosen to be the mapping function $f_{\boldsymbol{\theta}}$. Therefore, it can utilize back propagation to compute gradient on parameters per layer and optimize the overall loss. Compared with simple linear weighted sum, MLP is more effective and widely-used to solve regression and prediction problems. Since we prefer value-based method, this learning framework is also called Deep Q-Network (DQN), whose workflow is present in the left part of Figure 1.

Exploration Our DRL agent is also integrated with active exploration. We use revised ε-greedy, formulated in Equation 8, to determine whether to follow the currently optimal policy or risk exploring another manner. With probability (w.p.) ε_1, the agent may take a stochastic step, while with another probability ε_2, the agent acts as if a reflex agent, taking a effective yet method, e.g. LRU or LFU. Otherwise, the agent strictly takes the optimal action evaluated by the Q-values. The exploration process is shown by the right part of Figure 1.

$$a_t = \begin{cases} \arg\max_{a\in\mathcal{A}} Q(\boldsymbol{s_t}, a), & \text{w.p. } 1-\varepsilon_1-\varepsilon_2 \\ \text{random } a \in \mathcal{A}, & \text{w.p. } \varepsilon_1 \\ \text{LRU or LFU } a \in \mathcal{A}, & \text{w.p. } \varepsilon_2 \end{cases} \tag{8}$$

The ε_1 and ε_2 are also supposed to be decreased over time. However, due to the system environment always varys sharply, we should not decrease ε_1 and ε_2 too quickly or even monotonically. When DRL agent starts to perform poorly, we will enforce more exploration. We simply keep track with a small set $\mathcal{H}$ of recent $|\mathcal{H}|$ rewards, and evaluate the agent by the median within the history set. The ε's update function is defined as Equation 9.

$$\varepsilon = \begin{cases} \varepsilon + \delta^+ & median(\mathcal{H}) < \rho \\ \varepsilon - \delta^-, & median(\mathcal{H}) \geq \rho \end{cases} \tag{9}$$

where δ^+ and δ^- are the length of step, both are larger than 0. ρ is a threshold to evaluate the reward. We find introducing two ε's in our model significantly improves our results in miss rate. The DRL agent is able to learn from a suboptimal policy and explore with lower overall regret. Moreover, with varying request patterns, adaptive ε's can also instruct our agent to become more flexible.

Learning Process Different from the traditional approximate Q-Learning, DQN updates Q-values in batch. To store every transition for delayed training, we set a memory with size of N_β. When the memory is full, we follow FIFO policy to evict old memory for incoming memory since we prefer recent transition. Every T_1 decision epochs, we sample a batch of size N_α from the memory list and exploit mini-batch gradient descent to train the MLP neural network.

Notice that, when we compute the loss on $Q(s, a)$, the network will output a vector $\boldsymbol{Q}$ of all actions. We should not back-propagate the gradient to those Q-values of unselected actions. Furthermore, to keep two sets of Q-values, we have to use double MLPs to record parameters θ_k and θ_{k-1} which generate Q_k and Q_{k-1}, respectively. When evaluating actions, f_{θ_k} is invoked. Every T_0 decision epochs, we synchronize θ_{k-1} with θ_k. We said that, $T_0 \cdot k = t$, where t is the current decision epoch on update. The double MLP architecture is shown as Figure 2. The hyperparameters will be present in Section 4.

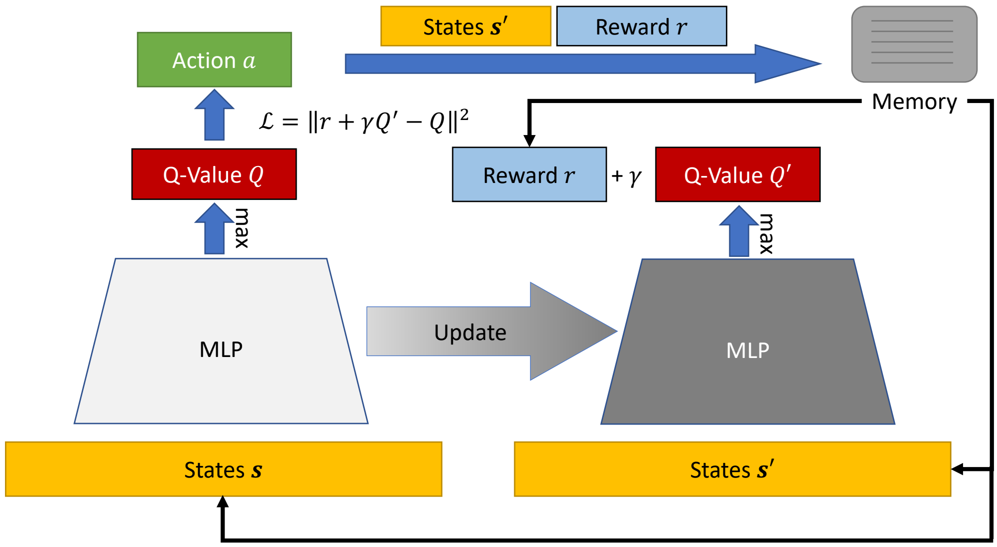

Figure 2: The training process of DQN. We maintain two sets of network parameters and synchronize them periodically. The left MLP is updated on-the-fly and used to evaluate actions. The right MLP is mainly to obtain out-of-date Q-values to calculate the TD difference. The loss function is identical to Equation 5.

4 Implementation Details

In this section, we will introduce our implementation details for the following experiments. The learning part of our source code is adopted from Zhou (2019), and we implmented the backend environment model of our file cache system. Our source code will be available on https://github.com/peihaowang/ReinforcementCache.

Feature Extraction In our basic model, we use frequency as the features of states. Recall that, each state is encoded by $s_t = \{\mathcal{F}_0, \mathcal{F}_1, ..., \mathcal{F}_C\}$, where $\mathcal{F}_0$, concatenating the features of cached blocks and currently requested block. Let $\mathcal{F}_i = \{f_i^s, f_i^m, f_i^l\}$, where the three elements represent short-term, middle-term and long-term recent access times. The motivation to use frequency information is that we observe the temporality of data access as well as the good adaptability of LFU, as we can see in section . But keeping all-time access times requires memory cost of $O(N)$ which is extremely high. Therefore, we only keep a small subset of history instead and adopt a neural network to regress and explore the latent information from it. This method reaches high effectiveness and avoid high memory cost. In our experiments, we take past 10, 100, 1000 accesses into account.

Reward In our experiments, we set $\mu = 10$, $\psi = 50$, $\kappa = 1$ and $\lambda = 1.0$ in penalty term. We further explore the importance of the two terms in our reward function. Penalty often makes less difference, but can assist our DRL significantly in some extreme cases. Notice that, when reaching the end state, i.e. all accesses are handled, the reward will still obey the formulation as Equation 1, but without the penalty term.

DQN Hyperparameters Our DQN contains a large set of hyperparameters. First we present the basic learning parameters: learning rate $\alpha = 0.01$, batch size $N_\alpha = 32$. Our MLP consists of one hidden layer, containing 256 perceptrons. We do not deepen the network, because we find small-scale network is much more easy to converge. Since our cache system world varys over time, we should keep the convergence of the network as quick as possible. We utilize normal initializer to initialize the parameters and employ RMSProp as the optimizer.

Our the memory list has size $N_\beta = 10000$ and we train our network every $T_1 = 5$ transition. The parameters will be synchronized every $T_0 = 100$ decision epoch. As for the parameters of Q-Learning with ε-greedy, we set the discount factor $\gamma = 0.9$ and ε_1, ε_2 are initialized with 0.1, 0.5 respectively. For ε_1, $\delta^+ = 0.005$ and $\delta^- = 0.005$. For ε_2, $\delta^+ = 0.1$ and $\delta^- = 0.001$. A maximum value 0.5 is set for every ε. The threshold ρ ranges from 5 to 50. We tunes this value while experiment to acquire better results.

5 Experiments

In order to evaluate our method's performance, we have designed 3 experiments for miss rate assessment and 1 experiment for model tuning. As comparisons, we also evaluate the performance of FIFO, LRU, LFU, MRU, Clock and Random in the first three experiments.

5.1 Data Generation and Simulation

The dataset in our experiment has two parts. The first part of dataset is used for simulation. We attempt to find a probabilistic distribution and draw samples from the distribution as our requested content IDs. Zipf is a simple but general content popularity distribution Suksomboon et al. (2013), which is usually used in many simulations of networking content requests. We set the distribution parameter to 1.3, which is reused from Zhong et al. (2018). We generate 10,000 requests to blocks in file system such that enough misses are produced.

The second part of the dataset is collected from Pintos Pfaff et al. (2009). We run a data collecting process in Pintos kernel so that every accessed block ID will be written to the output file. Then we can extract these accessed block IDs from the output file. We choose 6 typical user programs given in the Pintos testcases to produce this part of data in a practice environment. There are over 10,000 requests in these real testcases, which are convenient for our learning agent to observe and learn from.

5.2 Miss Rate Assessment

Experiment 1 We use the simulation data in this experiment. For the first part of this experiment, we observe the tendency of miss rate for every comparing methods under varying cache capacity. For the second part, we divide the sequence of requests into 100 slices and records average miss rate over each time slice. We never inform our agent of any knowledge about the request distribution in order to ensure fairness.

Figure 3 shows that the miss rate drops down with the increasing of cache capacity for all the comparing methods. Generally speaking, our method always produce lower miss rate than others. Additionally, decreasing tendency of the 7 comparing methods are similar: when the capacity varies from 1 to 25, miss rates of the 7 methods drop very quickly and afterward their drops pace in a slow rate. During the quick drop, our method has the fastest average descent rate over other methods. As the capacity becomes larger, their miss rates gradually converge around 0.2 at C = 300. This is because the cache capacity is large enough to hold on most of the popular requested content generated from our content popularity distribution. The miss rate of LRU is relatively close to our method, which indicates that this deterministic replacement policy may work very well in most request cases.

In Figure 4 [1], we plot curves of each method's miss rate changing over time with the cache capacity fixed to 50. Miss rate of each policy starts at about 0.43 due to compulsory misses before the first 100 accesses. With the following requests arriving, our method drops sharply compared with other methods, suggesting that our agent starts to study from memory and become

[1] There was a defect in this figure in our presentation. We downsampled some points and then made the curves inconsistent at the beginning by mistake. The figure in this report has been revised.

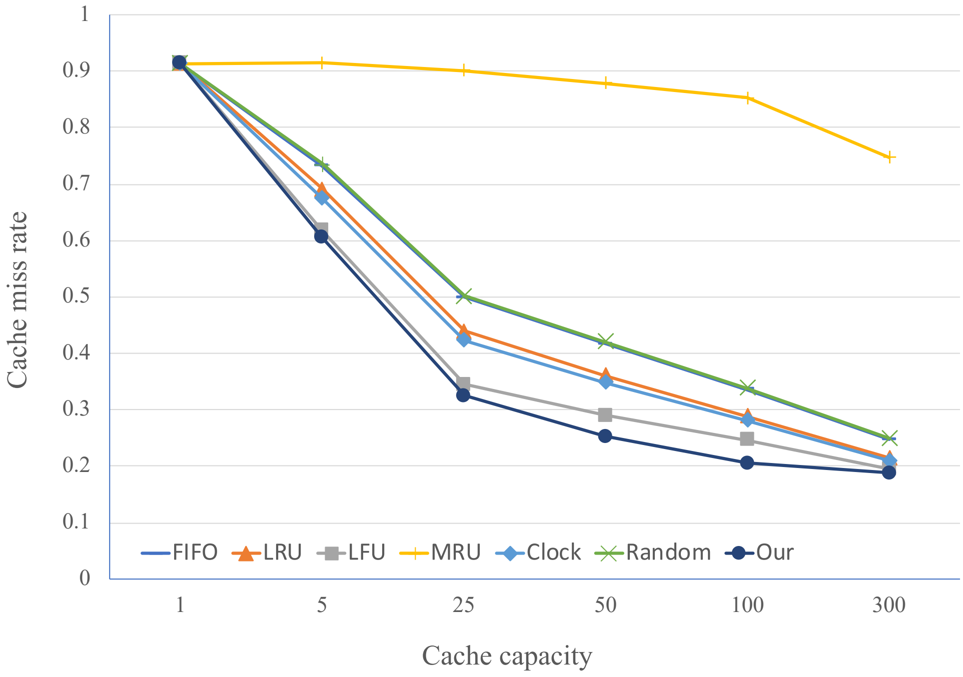

Figure 3: Cache miss rate vs. cache capacity. The cache capacity varies as 1, 5, 25, 50, 100, 300.

adaptive to the given environment. Our curve looks similar to the LFU curve. This is caused by that the features we used in our model is similar to the "features" used by LFU.

Experiment 2 We use the practice data from Pintos in this experiment. As the Table 1 shows, our method wins 4 of 6 cases and perform at an average level in the other 2 cases. However, this tables shows that the gap between LRU and aour method is seldom higher than 1%, even though our method spends lots of time training DQN.

	FIFO	LRU	LFU	MRU	Clock	Random	Our
dir-open	20.38%	19.91%	19.56%	42.83%	19.91%	19.41%	**18.03%**
dir-vine	1.49%	**1.44%**	57.67%	62.16%	**1.44%**	1.55%	1.80%
grow-create	21.46%	20.96%	20.59%	42.29%	20.96%	20.88%	**19.17%**
grow-file-size	17.80%	17.40%	17.09%	39.97%	17.40%	16.66%	**16.12%**
grow-seq-sm	20.45%	19.99%	19.65%	43.14%	19.99%	19.61%	**18.25%**
syn-rw-persistence	15.74%	14.61%	**14.13%**	68.08%	14.71%	15.72%	14.94%

Table 1: Miss rate in practice Pintos environment. We choose six typical testcases from it and observe slight advantage of our method over others.

Experiment 3 We also prepare sequential flooding tests to examine if our method is adaptive to some extreme cases. In this experiment, the cache capacity is fixed to 5 and the request is a repeated sequence of 1, 2, 3, 4, 5, 6. In the Table 2, LRU, the widely-used policy, reaches 100% miss rate (consistent to theoretical analysis), while MLU, which is proved of the optimal policy in this case, reaches 20%. Our method works slightly better than Random policy. After 40 episodes of training, the miss rate declines to 27% approximately, indicating that the agent gradually learns the pattern of this case but still cannot predict at a considerably high accuracy.

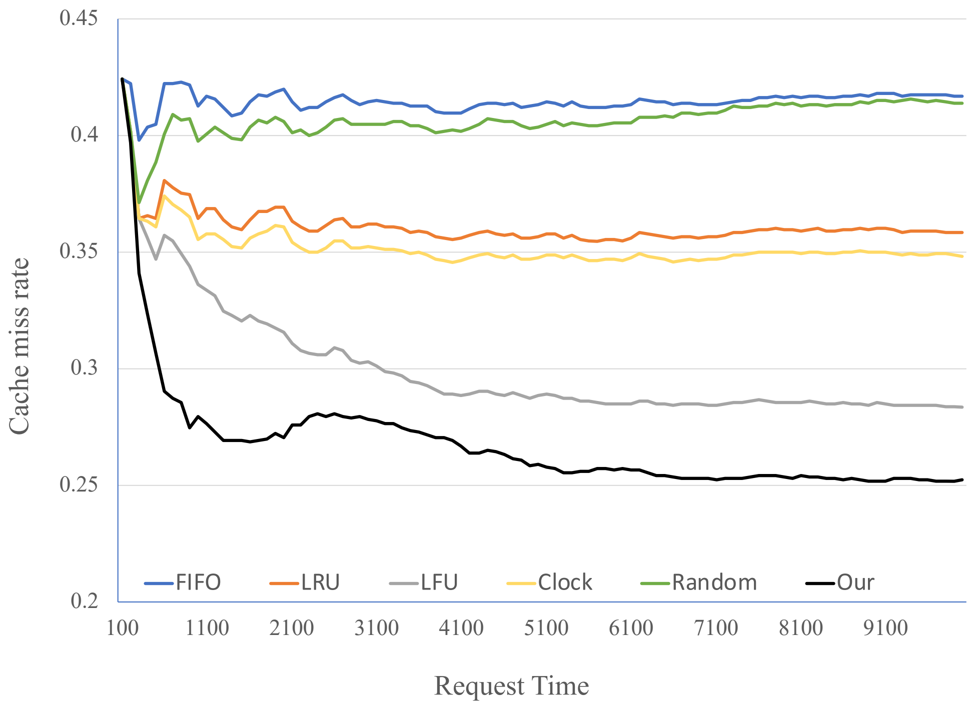

Figure 4: Cache miss rate over time. We start to record the miss rate from the 100-th request and make total 10,000 requests in total.

Method	FIFO	LRU	LFU	MRU	Clock	Random	Our (1 ep)	Our (40 eps)
Miss Rate (%)	100	100	100	20	100	33.41	31.32	27.22

Table 2: Miss rate in sequential flooding. We run an extreme case and observe our method can perform much better than other methods except for the MRU.

5.3 Model Tuning

Feature Engineering In addition to the frequency feature mentioned in Section 4, we also consider recent used time (UT), cached time (CT) and cached content (CS) these three features.

UT features are used by LRU policy, which chooses the cache item with smallest used time to evict. CT features are used by FIFO policy, which chooses the first cached item to evict. These two policies because typically commit relative high performance, therefore, we are inspired to engineer with these two features. The cache content is the most straightforward feature of cache content. However, it belongs to categorical data because each slot can only select value from integers $1, ..., N$. We use one-hot encoder to represent this category into a sparse feature. We set the cache capacity $C = 50$, and run our DRL agent on the generated Zipf data. Table 3 shows the experiments result.

We find that adding UT features reduces the miss rate slightly by 1%, which shows the good insight of LRU policy. However, CT features of FIFO bring drawbacks to our model. The combination of both additonal features does not reach the level of our original features. We can draw conclusion that, LRU can perform well due to UT features, but CT features seem useless in raising hit rate. However, the overall improvement keeps slight, thus, it may not worth being

Features	Miss rate (%)
Base	25.22
Base + UT	24.60
Base + CT	27.83
Base + UT + CT	26.39
Base + CS	34.14

Table 3: The results of our feature engineering. Base is the original features mentioned in Section 4 UT = most recent used time, CT = cached time, CS = cache content state.

applied in addition to our basic features, since the increment of dimensionality of features will bring more computational cost.

The performance of CS features perform frustratingly. It commits the miss rate as high as the random policy. We think it is because our neural network cannot process these sparse data, as it only contains one hidden layer with 256 perceptrons. To summarize, we think our original feature is expressive enough and commits fairly high performance, thus, to avoid more computational burden, the frequency only feature might be the best choice.

Reward Function We also tested different reward function in the following experiment. First, we remove the penalty term and then we experiment with the short term + long term reward proposed by Zhong et al. (2018). We set the cache capacity $C = 50$ and run our DRL agent on the generated Zipf data. Table 4 shows the experiment results.

Reward	Miss rate (%)
# Hit Only	26.76
# Hit + Penalty	25.22
Short + Long Term	27.54

Table 4: The results of our reward tuning. = the number of, eps = episodes. The first two reward is proposed by us and the last reward is proposed by Zhong et al. (2018).

We observe slight improvement when taking penalty term into account. However, we witnesse that the penalty term can instruct our DRL agent to make more judicious decision when the temporality turns lower in the request sequence. Since penalty only reflects the short term utility, we do not think it is useful enough. When we apply the features proposed by Zhong et al. (2018), we surprisingly observe the side effects. The short + long term reward reveals the future information to the DRL agent, but it cannot be implemented in realistic computer systems. But Zhong et al. (2018) exploit policy-based method, which can involve delayed rewards into the learning process, thus, the reward can be reasonable in their work.

6 Conclusion

In this report, we have proposed and implemented a reinforcement learning based cache replacement strategy. We trained our agent via a Deep Q Network and evaluated its performance under both simulation and practice operating system environment. Compared with LRU, LFU and other traditional replacement policies, our method produces a substantial hit rate under common circumstances and has better adaptability under many extreme cases.

Our future work will mainly focus on confirming the effectiveness of the proposed method in real operation systems. This will demand more study on underlying architecture design. Additionally, our current study only considers the scenario that a single thread is requesting. We would either design a new thread context with corresponding learning model or refine our learning framework to be adaptive to other complex content popularity distributions, such as a mixture model of Zipf.

REFERENCES

Mohamed Ahmed, Stefano Traverso, Paolo Giaccone, Emilio Leonardi, and Saverio Niccolini. Analyzing the performance of lru caches under non-stationary traffic patterns. *arXiv preprint arXiv:1301.4909*, 2013.

Gabriel Dulac-Arnold, Richard Evans, Hado van Hasselt, Peter Sunehag, Timothy Lillicrap, Jonathan Hunt, Timothy Mann, Theophane Weber, Thomas Degris, and Ben Coppin. Deep reinforcement learning in large discrete action spaces. *arXiv preprint arXiv:1512.07679*, 2015.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

Ben Pfaff, Anthony Romano, and Godmar Back. The pintos instructional operating system kernel. In *ACM SIGCSE Bulletin*, volume 41, pp. 453–457. ACM, 2009.

Kalika Suksomboon, Saran Tarnoi, Yusheng Ji, Michihiro Koibuchi, Kensuke Fukuda, Shunji Abe, Nakamura Motonori, Michihiro Aoki, Shigeo Urushidani, and Shigeki Yamada. Popcache: Cache more or less based on content popularity for information-centric networking. In *38th Annual IEEE Conference on Local Computer Networks*, pp. 236–243. IEEE, 2013.

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)*, 2018.

Haonan Wang, Hao He, Mohammad Alizadeh, and Hongzi Mao. Learning caching policies with subsampling.

Chen Zhong, M Cenk Gursoy, and Senem Velipasalar. A deep reinforcement learning-based framework for content caching. In *2018 52nd Annual Conference on Information Sciences and Systems (CISS)*, pp. 1–6. IEEE, 2018.

Morvan Zhou. Reinforcement learning with tensorflow. `https://github.com/MorvanZhou/Reinforcement-learning-with-tensorflow`, 2019.

Barbara Tockey Zivkov and Alan Jay Smith. Disk caching in large database and timeshared systems. In *Proceedings Fifth International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pp. 184–195. IEEE, 1997.