Towards Meta-Reinforcement Learning Cache Replacement

Bryan Chan
University of Toronto

Abstract

Cache replacement is extensively researched on using classic algorithms with engineered heuristics. With the rise of machine learning, we question whether we can automatically leverage features to improve on hit rate performance. Specifically, we formulate the problem using reinforcement learning and verify the feasibility of using this approach. Previously, machine learning has only been applied to determine which existing policies to use for cache eviction, but we propose to directly determine the victim pages using the model instead. We also integrate logical block addresses as part of our features, which is not used in previous approaches. We find our approach to be a promising starting point for smaller cache sizes, as it is able to perform near optimal on average in similar workloads. Lastly, we identify possible problems with this approach and propose possible solutions to address them. One notable problem is the generalization across different workloads, which we plan to use meta-learning to address.

1 Introduction

Caching is an old technique that improves performance by keeping important data in memory locations that are faster. Megiddo and Modha describes the simplest system consisting of two memory levels: *main (cache)* and *auxiliary* memories [19]. The main memory is assumed to be significantly faster than auxiliary memory, but is significantly more costly. As a result, the size of the main memory is only a fraction of the size of the auxiliary memory. Similar to the setup of Megiddo and Modha, we assume there is a stream of requests for pages, fixed size units of data managed by the memories [19]. Depending on the workload pattern, some information about the pages may be crucial in determining the most important pages to keep in main memory. This is easy if the whole stream of requests can be fit into the main memory. However, this is not the case realistically. When the main memory is full and a requested page is not already in the main memory, we need to fetch the page from auxiliary memory and put it into the main memory. The question now is, *which page in the memory do we evict?* In this case, the page to be evicted is known as the *victim page*. A cache replacement policy determines the victim pages given a stream of requests. In practice, the policy does not have direct access to future requests but may perform some heuristic to deduce the pattern. This is important because it affects how quickly the users can retrieve/manipulate the data over time. To determine whether our choices of eviction by the end of a stream, we can compute the *hit rate*, which is defined by number of pages fetched directly from the main memory over the number of requests. The problem to address is simply, *how to design a cache replacement policy such that it maximizes the hit rate?*

Known methods such as Least Recently Used (LRU) [5, 9], Least Frequently Used (LFU) [5], and Adaptive Replacement Cache (ARC) [19] achieve great results by simply exploiting recency and/or frequency of the pages, in addition to having a feasible time and space complexities. However, one can ask the question, *are recency and frequency the only features that we can utilize?* One potential feature that is not used in any widely known cache replacement policy is spatial locality. We use deep neural networks to automatically extract this feature by simply providing logical block addresses.

To incorporate recency, frequency, and spatial locality, we propose to use machine learning techniques to automatically determine the importance of the features based on the previous states and the current state of the cache. There are previous work that attempts to address cache replacement by applying machine learning to decide which existing expert policies to use [4, 30]. Specifically, they learn a model to predict whether to use LFU, LRU, and other existing policies. This approach is only as good as the expert policies, assuming the model can choose the best policy at each request. In other words, if the set of expert policies is not optimal for specific workload patterns, their approaches will perform suboptimally regardless of the model's predictions. To address this problem, we learn a model to decide the victim page directly. Since the requests are not i.i.d., we apply reinforcement learning to verify the feasibility. Specifically, we formulate the cache replacement problem as a contextual multi-armed bandit problem. Then, since a model is only trained for a specific workload, the performance will suffer from workloads with different distributions. we plan to apply meta-learning, which is a paradigm that learns to learn, to allow generalization across workloads.

In the reinforcement learning setting, multi-armed bandit is studied extensively [1, 6, 7]. However, the traditional setting of multi-armed bandit assumes i.i.d. and hence its results are not directly applicable to the cache replacement problem. As a result, we formalize the cache replacement problem as a contextual bandit problem, which is also extensively explored [2, 8, 12]. However, these approaches have certain assumptions on the distribution of reward function and certain amount of correlated feedback.

The paper is organized as follows: we describe the required background that formulates our approach in section 2. Then, we describe our problem formulation and methods in section 3. Then, in section 4, we evaluate our methods on multiple workloads consisting of different patterns. Finally, we indicate the limitations of our method, provide possible future directions, and conclude our results in section 5.

2 Preliminary Background

2.1 Reinforcement Learning (RL)

Sutton and Barto introduces reinforcement learning as a framework to describe an agent interacting with an environment at discrete time steps [27]. At each timestep t, an agent receives environment state $s_t \in \mathcal{S}$ and reward signal $r_t \in \mathcal{R} \subset \mathbb{R}$. The agent selects an action $a_t \in \mathcal{A}$ according to a policy defined by a probability distribution $\pi(a|s) = P\{a_t = a | s_t = s\}$. At timestep $t+1$, the environment transitions to a new state s_{t+1} and emits a new reward signal r_{t+1} according to a transition probability distribution $p(s', r|s,a) = P\{s_{t+1} = s', r_{t+1} = r | s_t = s, a_t = a\}$. The goal of the agent is to find a policy π such that it maximizes the expected cumulative rewards (discounted return) $G_t = \sum_{k=t}^{\infty} \gamma^{k-t} r_{k+1}$, where $\gamma \in [0,1]$ is the discount factor. γ determines how much the agent focuses on short term reward.

2.2 Contextual Bandit

We provide a formal definition of the contextual bandit problem similar to Agrawal and Goyal [2]. There are N arms (i.e. $|\mathcal{A}| = N$). At each timestep t, a context vector $c_i(t) \in \mathbb{R}^d$ is revealed for each arm i. The context vectors $c_i(t)$ are determined in an adaptive manner after $t-1$ actions are taken along with the corresponding rewards, i.e., based on the history $\mathcal{H}_{t-1}$

$$\mathcal{H}_{t-1} = \{a(\tau), r(\tau, a(\tau)), c_i(\tau), i = 1, \dots, N, \tau = 1, \dots, t-1\}$$

, where $a(\tau)$ denotes the action taken at timestep τ, $r(\tau, a(\tau))$ is the reward received by taking action $a(\tau)$ at timestep τ, and $c_i(\tau)$ are the context vectors at timestep τ. Given $c_i(t)$, the reward for arm i at timestep t is generated from an (unknown) distribution that may depend on the history $\mathcal{H}_{t-1}$.

2.3 Meta-Learning

In machine learning, meta-learning is a paradigm that learns higher level reasoning of various tasks [18]. One focus in meta-learning is learning to learn. The goal is to avoid overfitting to a single task and generalize to similar tasks. One approach is to train a meta-learner on a distribution of tasks with the hope of generalizing to the entire task distribution [3]. Duan et al. developed a RL algorithm to capture higher level concepts through learning from multiple tasks [13]. Research also focuses on using deep neural networks to improve the performance of meta-learners [20, 23]. Another focus in meta-learning aims to learn new concepts with only few training examples. Finn et al. and Nichol and Schulman developed algorithms to achieve this goal by finding a good parameter initializer for the models [14, 22]. This approach, however, requires training under a new task, so it is not suitable for the cache replacement problem.

3 Methodology

3.1 Environment Formulation

As there is no established environment for the cache replacement problem in RL setting, we define our very own environment that reads from existing workload traces generated from blocktrace. Specifically, we define the environment similar to the contextual bandit problem. For a cache of fixed size C, we define the action set $\mathcal{A} = \{0, 1, \dots, C-2, C-1\}$ to be the slots in the cache (i.e., C arms). The action $a \in \mathcal{A}$ represents the page to be evicted at slot a. At each timestep t, a context vector $c_i(t) \in \mathbb{R}^3$ consisting of the features: logical block number, recency, and frequency, is revealed for each slot (arm) i. In this case, each timestep t corresponds to the tth eviction in the current workload based on previous evictions. Naturally, the context vectors $c_i(t)$ will depend on the entire history $\mathcal{H}_{t-1}$ as different action sequences can affect the current state of the cache. The context vectors are normalized such that: (1) Block numbers are divided by maximum block address, which is an accessible value from the operating system. (2) Recency and frequency are normalized based on the current cache. The reward $\mathcal{R}$ is defined to be the number of requests processed until the next eviction occurs. In other words, the more requests processed before an eviction, the better the action is. The trajectory terminates once the last request in the workload is processed.

3.2 REINFORCE

With the environment formulation, we can apply off-the-shelf RL algorithms to optimize the policy π. We use the vanilla REINFORCE algorithm as a proof of concept to verify the feasibility of using RL in this problem [28]. REINFORCE is a policy gradient approach, meaning it optimizes the parameters of π directly during training (See Algorithm 1). It uses gradient ascent to encourage actions that result in high returns and discourage low return actions. Both learning rate α and discount factor γ are hyperparameters that need to be tuned specifically for the task. Sutton et al. showed that REINFORCE converges to a local optimum asymptotically [28].

Algorithm 1 REINFORCE (Monte-Carlo Policy Gradient) algorithm. It performs updates the policy using gradient ascent when the agent finishes a trajectory [27].

```
Require: A differentiable policy $\pi$ parameterized by $\theta$,
    $\pi(a|s,\theta)$, learning rate $\alpha$, and discount factor $\gamma$
1: Initialize policy parameter $\theta$
2: loop until convergence
3:     Follow $\pi$ and generate a trajectory $T$,
4:     $T \leftarrow s_0, a_0, r_1, s_1, a_1, r_2, \ldots, s_{t-1}, a_{t-1}, r_t$
5:     for each step of $T$, $i = 0, 1, \ldots, t-1$ do
6:         $G \leftarrow$ discounted return from step $t$
7:         $\theta \leftarrow \theta + \alpha\gamma^i G \nabla_\theta \ln \pi(a_i|s_i,\theta)$
8:     end for
9: end loop
```

3.3 Policy Representation

Since the problem inherits a causal property, where the current context depends on history, and situations where same context vectors can be generated through different action sequences, we want our policy to capture such properties. A class of function approximators, recurrent neural networks (RNN), is suitable for this as it utilizes hidden states to capture the history. Specifically, we use a Gated-Recurrent Unit (GRU) architecture [10], which is shown to perform well in practice [11]. Our architecture first uses GRU, its output is then fed to fully connected layers followed by a softmax function, which forms a categorical distribution over actions. The architecture is not extensively fine-tuned and other architectures are not experimented on (See Figure 1).

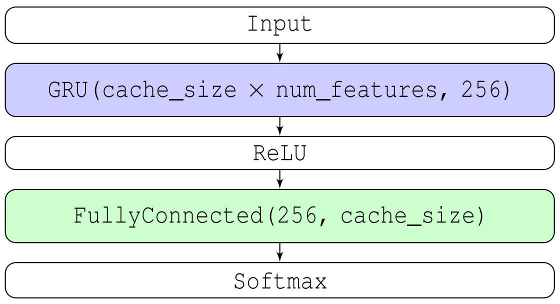

Input
GRU(cache_size × num_features, 256)
ReLU
FullyConnected(256, cache_size)
Softmax

Figure 1: Architecture of the cache replacement policy. The input is first fed into GRU layer `GRU(input_size, num_hidden)`. Its output is fed into ReLU non-linearity, followed by a fully connected layer `FullyConnected(input_size, output_size)`. Finally, a softmax function is applied on the output of the fully connected to form a categorical distribution over actions.

4 Evaluation

We now present our experiment results to answer the following questions:

1. Can we train a policy with RL such that it learns to perform similarly to the OPT algorithm?
2. How does the cache size affect the training?
3. Can the trained policy perform better than LRU and LFU in similar workloads?
4. Can the trained policy generalize to different workloads?
5. As opposed to existing methods, is the trained agent able to utilize the features provided to produce a better result?

4.1 Experiment Setup

Our experiments are tested on workloads from Koller and Rangaswami, which are collected from FIU computer science department [17]. Specifically, we used workloads from an email server (*mail* workload), and from a file server (*homes* workload). We will also refer the former to be *cheetah* and the latter to be *casa*. Since there are multiple blocktrace files, we used **cheetah-3**, **casa-4**, **casa-5**, **casa-6** to train the RL policy and evaluate our results. For the experiments, we used PyTorch and OpenAI Gym to implement the policy, algorithm, and environment described in Section 3 (See Availability for source code). Specifically, we implemented the policy and REINFORCE using Pytorch, and implemented our very own environment using OpenAI Gym.

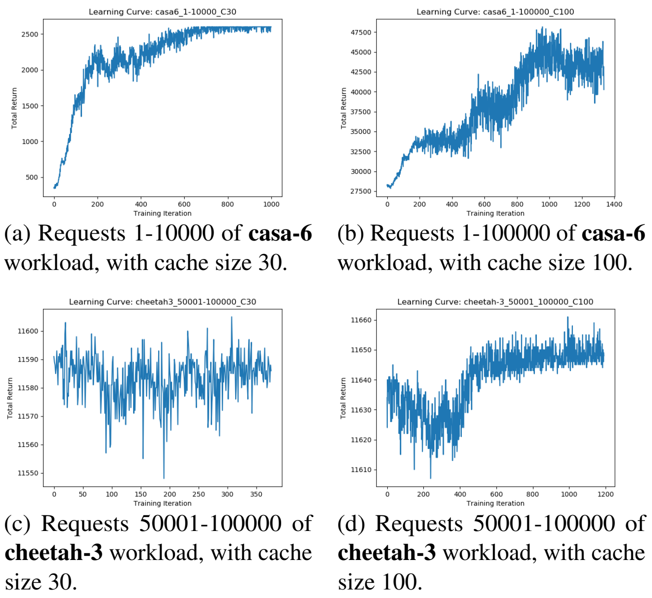
We first train separate models using **cheetah-3** and **casa-6** with different hyperparameters settings and cache sizes (See Tables 1 and 2). All models use Adam optimizer [16] for gradient updates. The input-hidden weights are initialized with Xavier initialization [15], the hidden-hidden weights are initialized with orthogonal initialization [24], and the biases are initialized with zeroes. Then, we evaluate the trained models (policies) by looking at the learning curve of the models, the average hit rate curve of the training workload request streams, and the average hit rates on unseen workload request streams. The first evaluation can determine

Cache Size	30	100
Learning Rate	10^{-3}	10^{-3}
Discount Factor	0.99	0.99
Training Epochs (# of trajectories)	1000	1340
Starting Request	1	1
Ending Request	10000	100000

Table 1: Hyperparameters for training **casa6** models.

Cache Size	30	100
Learning Rate	10^{-3}	10^{-3}
Discount Factor	0.99	0.99
Training Epochs (# of trajectories)	410	1200
Starting Request	50001	50001
Ending Request	100000	100000

Table 2: Hyperparameters for training **cheetah3** models.



(a) Requests 1-10000 of **casa-6** workload, with cache size 30.

(b) Requests 1-100000 of **casa-6** workload, with cache size 100.

(c) Requests 50001-100000 of **cheetah-3** workload, with cache size 30.

(d) Requests 50001-100000 of **cheetah-3** workload, with cache size 100.

Figure 2: The learning curves of the models trained. The total return is the number of hits within the training workload.

whether the model is learning and performing optimally after convergence. The second evaluation provides information on where the model performs differently to other algorithms and which features are utilized. The third evaluation provides insights on whether the trained policy can be applied on similar workloads and whether it can generalize to different workloads. For testing, we run each experiment 10 times and plot the average hit rate along with the standard deviation.

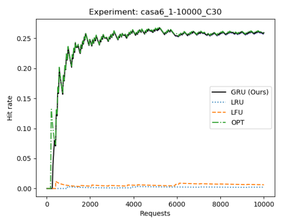
4.2 Results

We first look at the learning curves of the models trained (See Figure 2). Figure 2a shows that the total return plateaus at 2601, which is slightly lower than the optimal return. As we increase the cache size to 100 (See Figure 2b), the model fails to converge but is still gradually learning. With increasing number of samples, the model should converge to near optimal. The learning curve of Figure 2c is abnormal as the model fails to learn anything. We suspect two possible reasons: (1) The number of trajectories is not sufficient for the agent to learn. (2) By examining the workload itself, the optimal total return is not better than the learning agent by a large margin. Hence, this portion of the workload does not consist any meaningful information for the agent to learn from. This can be verified through Figure 3d, where all methods perform similarly. Another verification is from Figure 2d. We can see that the policy is almost converged but the total return is only at most 60 more than Figure 2c, which can be caused by having a larger cache size.

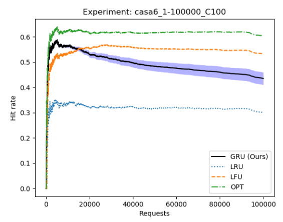
Next, we examine the average hit rate of the trained policies on the trained requests (See Figure 3). Looking at the training requests from **casa-6**, we can see that the policy fails to recognize the best option in the beginning (See Figure 3a). By observing OPT's hit rate, we can see that the peak follows by a drastic dip, meaning there is a sequence of requests that is not in the cache. Hence, the previous choices of victim pages can be decoupled from the choices after the dip. We suspect that our policy fell into a local optimum where it stopped exploring other options in the beginning after seeing a high return. However, the trained policy is able to perform similarly to OPT overall, with low standard deviation.

Cache size plays an important role on the performance of the trained policy. Looking at **casa-6** with cache size of 100, the policy performs worse than both OPT and LFU (See Figure 3b). This can be explained in multiple perspectives. First, since the length of the training stream is 100000 and the action space is 100, it requires more trajectories to identify a best sequence of actions. We emphasize that we only used 1340 trajectories to train the policy, which is a small fraction of possible trajectories. Second, the input dimension increases as the cache size increases. The model may not have the capacity to refine the features with the same hidden layers.

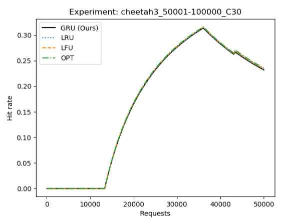
Lastly, we examine the performance of trained policies in new workloads (See Figures 4, 5, and 6). For **casa-6** with cache size 30 (See Figure 4), the trained policy seems to be able to perform relatively well on similar workloads with smaller cache size (See Figures 4a, 4b, 4c). However, we can clearly see that as the workload stream gets longer, the hit rate starts to degrade with larger standard deviation. We suspect that the workloads consist of some unseen patterns during training phase. Additionally, one suboptimal action



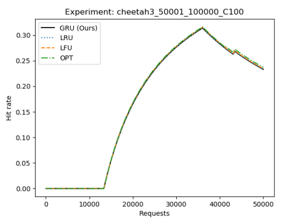
(a) Requests 1-10000 of **casa-6** workload, with cache size 30.



(b) Requests 1-100000 of **casa-6** workload, with cache size 100.



(c) Requests 50001-100000 of **cheetah-3** workload, with cache size 30.

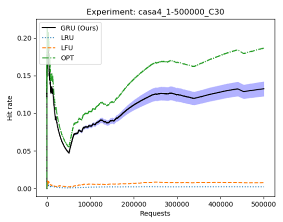


(d) Requests 50001-100000 of **cheetah-3** workload, with cache size 100.

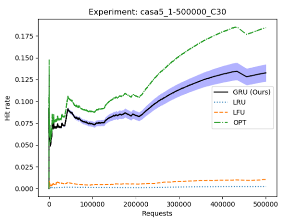
Figure 3: The hit rates of OPT, LRU, LFU, and GRU (Ours) over trained workloads. For GRU, we run the experiment 10 times as the policy is stochastic.

may greatly affect the proceeding hit rate. Hence, the policy might have been able to identify the pattern, but fail to achieve high hit rate nevertheless due to previous suboptimal actions. On the other hand, the policy fails to perform on a different workload (See Figure 4d). This is expected as workload patterns may differ, and the logical block addresses range can be drastically different as well. For the **cheetah-3** policy (See Figure 5), the performance of our policy is nowhere near OPT, regardless of the workload. However, it is better than both LFU and LRU. Since the standard deviations are small, we do not think our trained policy is randomly choosing a victim page, but it is unclear whether it learned a certain pattern due to the training workload. As we increase the cache size to 100 for **casa-6** (See Figure 6), our policy performs worse than OPT and LFU, but similarly to LRU. We can argue that since the policy is not converged yet so the performance is worse. However, it is promising to see that even without convergence, our policy is able to perform similarly to LRU.

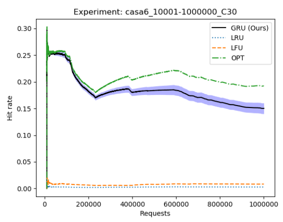
Based on the hit rate curves (See Figures 3a, 4a, 4b, 4c), we can clearly see that both LRU and LFU suffer in the **casa** workloads with cache size of 30. By accounting for recency, frequency, and logical block addresses, the policy was able to leverage the features to deduce a good victim page. In the future, ablation studies can verify whether logical block addresses play an important role in the workloads.



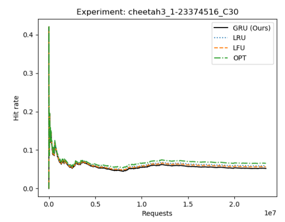
(a) Requests 1-500000 of **casa-4** workload.



(b) Requests 1-500000 of **casa-5** workload.



(c) Requests 10001-1000000 of **casa-6** workload.



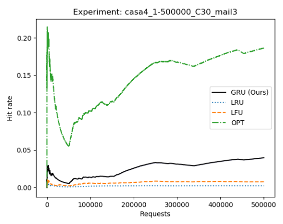
(d) All requests of **cheetah-3** workload.

Figure 4: The hit rates of OPT, LRU, LFU, and GRU (Ours) over unseen workloads. For GRU, we run the experiment 10 times as the policy is stochastic. The cache size is 30 and the policy is trained on requests 1-10000 of **casa-6**.

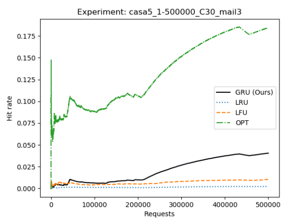
5 Conclusion and Discussion

We provide a new RL environment to formulate the cache replacement problem similar to contextual bandit. We utilize a variant of RNN to capture potential temporal connection within the workloads and we apply a classic RL algorithm to show that RL can be used to improve the hit rate for specific workloads. Specifically, without fine-tuning, the agents are able to perform near optimal without the access of the OPT algorithm. We also identify some potential problems with our approach such as convergence rate, generalization, and curse of dimensionality. However, some problems may be addressed using recent methods. We conclude that RL is a feasible approach to solving the cache replacement problem.

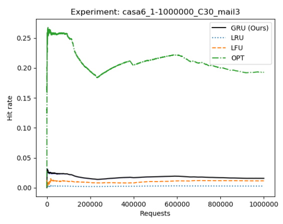
Using machine learning to address the cache replacement problem is appealing but has its own limitations. Current state of the art cache replacement algorithms use little space and decide victim pages rapidly. Based on the current advancement in machine learning, it is arguable that machine learning can achieve similar time and space complexity in both theory and practice. Additionally, with increasing cache size, the speed of training drastically decreases so it is not be feasible to train models with large memory size and small page size. In RL setting, the trade off between exploration and exploitation remains an active research problem and we have seen that in our experiments that the policy sometimes fall into local optimum. Another problem is that the same workload pattern may occur in



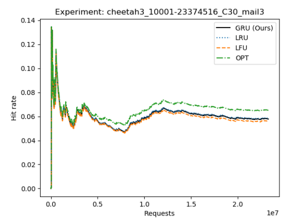
(a) Requests 1-500000 of **casa-4** workload.



(b) Requests 1-500000 of **casa-5** workload.



(c) Requests 1-1000000 of **casa-6** workload.



(d) All requests of **cheetah-3** workload.

Figure 5: The hit rates of OPT, LRU, LFU, and GRU (Ours) over unseen workloads. For GRU, we run the experiment 10 times as the policy is stochastic. The cache size is 30 and the policy is trained on requests 50001-100000 of **cheetah-3**.

different range of logical blocks. Our experiments are tested on workloads generated using the same machine, hence it may not perform well on workloads generated from a different machine even if the patterns are similar. Workloads can also inherit different patterns which means our approach is not applicable if we only train on one workload.

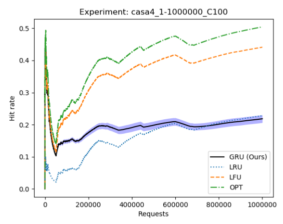
To address some of the above problems, we propose few solutions. One may try using a different RL algorithm that has been shown to converge quicker in practice. For example, A3C [21], TRPO [25] and PPO [26] are potential methods that can improve the results of our experiments. Additionally, better model architecture and hyperparameters can be chosen to improve the performance and convergence rate. For example, the Transformer architecture is shown to perform competitively against RNNs [29]. To address the generalization of the policy, we can apply meta-learning, specifically using the idea of RL^2 [13].

Acknowledgments

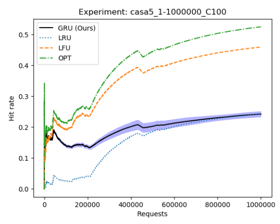
We thank Shehbaz Jaffer for providing the code for LRU, LFU, and OPT, and providing the approaches they attempted.

Availability

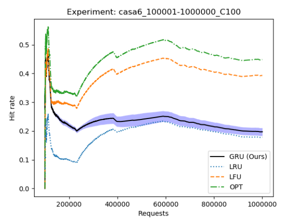
Our source code and trained models are available on https://github.com/chanb/MeLeCaR/.



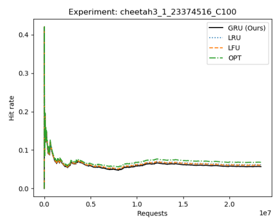
(a) Requests 1-1000000 of **casa-4** workload.



(b) Requests 1-1000000 of **casa-5** workload.



(c) Requests 100001-1000000 of **casa-6** workload.



(d) All requests of **cheetah-3** workload.

Figure 6: The hit rates of OPT, LRU, LFU, and GRU (Ours) over other workloads. For GRU, we run the experiment 10 times as the policy is stochastic. The cache size is 100 and the policy is trained on requests 1-100000 of **casa-6**.

References

[1] Shipra Agrawal and Navin Goyal. Analysis of thompson sampling for the multi-armed bandit problem. In *Conference on Learning Theory*, pages 39–1, 2012.

[2] Shipra Agrawal and Navin Goyal. Thompson sampling for contextual bandits with linear payoffs. In *International Conference on Machine Learning*, pages 127–135, 2013.

[3] Marcin Andrychowicz, Misha Denil, Sergio Gomez Colmenarejo, Matthew W. Hoffman, David Pfau, Tom Schaul, and Nando de Freitas. Learning to learn by gradient descent by gradient descent. *CoRR*, abs/1606.04474, 2016. URL http://arxiv.org/abs/1606.04474.

[4] Ismail Ari, Ahmed Amer, Robert B Gramacy, Ethan L Miller, Scott A Brandt, and Darrell DE Long. Acme: Adaptive caching using multiple experts. In *WDAS*, pages 143–158, 2002.

[5] Remzi H Arpaci-Dusseau and Andrea C Arpaci-Dusseau. *Operating systems: Three easy pieces*, volume 1. Arpaci-Dusseau Books, 2015.

[6] Peter Auer. Using confidence bounds for exploitation-exploration trade-offs. *Journal of Machine Learning Research*, 3(Nov):397–422, 2002.

[7] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer.

Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2-3):235–256, 2002.

[8] Mohammad Gheshlaghi Azar, Alessandro Lazaric, and Emma Brunskill. Online stochastic optimization under correlated bandit feedback. In *ICML*, pages 1557–1565, 2014.

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

[10] Kyunghyun Cho, Bart Van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259*, 2014.

[11] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

[12] Mark Collier and Hector Urdiales Llorens. Deep contextual multi-armed bandits. *arXiv preprint arXiv:1807.09809*, 2018.

[13] Yan Duan, John Schulman, Xi Chen, Peter L. Bartlett, Ilya Sutskever, and Pieter Abbeel. Rl^2: Fast reinforcement learning via slow reinforcement learning. *CoRR*, abs/1611.02779, 2016. URL http://arxiv.org/abs/1611.02779.

[14] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1126–1135. JMLR. org, 2017.

[15] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.

[16] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[17] Ricardo Koller and Raju Rangaswami. I/o deduplication: Utilizing content similarity to improve i/o performance. *ACM Transactions on Storage (TOS)*, 6(3):13, 2010.

[18] Christiane Lemke, Marcin Budka, and Bogdan Gabrys. Metalearning: a survey of trends and technologies. *Artificial intelligence review*, 44(1):117–130, 2015.

[19] Nimrod Megiddo and Dharmendra S Modha. Arc: A self-tuning, low overhead replacement cache. In *FAST*, volume 3, pages 115–130, 2003.

[20] Nikhil Mishra, Mostafa Rohaninejad, Xi Chen, and Pieter Abbeel. Meta-learning with temporal convolutions. *CoRR*, abs/1707.03141, 2017. URL http://arxiv.org/abs/1707.03141.

[21] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937, 2016.

[22] Alex Nichol and John Schulman. Reptile: a scalable metalearning algorithm. *arXiv preprint arXiv:1803.02999*, 2, 2018.

[23] Adam Santoro, Sergey Bartunov, Matthew Botvinick, Daan Wierstra, and Timothy Lillicrap. Meta-learning with memory-augmented neural networks. In *International conference on machine learning*, pages 1842–1850, 2016.

[24] Andrew M Saxe, James L McClelland, and Surya Ganguli. Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. *arXiv preprint arXiv:1312.6120*, 2013.

[25] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International Conference on Machine Learning*, pages 1889–1897, 2015.

[26] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

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

[28] Richard S Sutton, David A McAllester, Satinder P Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*, pages 1057–1063, 2000.

[29] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 5998–6008, 2017.

[30] 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.