Learning Runtime Parameters in Computer Systems with Delayed Experience Injection

Michael Schaarschmidt
University of Cambridge
michael.schaarschmidt@cl.cam.ac.uk

Felix Gessert
University of Hamburg
gessert@informatik.uni-hamburg.de

Valentin Dalibard
University of Cambridge
valentin.dalibard@cl.cam.ac.uk

Eiko Yoneki
University of Cambridge
eiko.yoneki@cl.cam.ac.uk

Abstract

Learning effective configurations in computer systems without hand-crafting models for every parameter is a long-standing problem. This paper investigates the use of deep reinforcement learning for runtime parameters of cloud databases under latency constraints. Cloud services serve up to thousands of concurrent requests per second and can adjust critical parameters by leveraging performance metrics. In this work, we use continuous deep reinforcement learning to learn optimal cache expirations for HTTP caching in content delivery networks. To this end, we introduce a technique for asynchronous experience management called delayed experience injection, which facilitates delayed reward and next-state computation in concurrent environments where measurements are not immediately available. Evaluation results show that our approach based on normalized advantage functions and asynchronous CPU-only training outperforms a statistical estimator.

1 Introduction

In recent years, reinforcement learning (RL) algorithms have been successfully combined with deep neural networks as function approximators [1, 2, 3]. Neural networks can capture structure in the environment from high-dimensional raw inputs and efficiently generalize over large state spaces. Deep reinforcement learning (DRL) techniques hence provide a powerful end-to-end task learning model from sensory inputs without prior knowledge of environment dynamics. However, training value functions for complex tasks requires significant training times and substantial computational resources.

There is another set of control problems in the domain of computer systems which are characterized by smaller problem dimensions and strong latency constraints. The problem this paper addresses is the utilization of DRL to provide real-time controllers for such problems. The key idea of this paper is that for comparatively small state dimensions (< 100) and tasks with weaker structure, no extended offline training is necessary to implement effective controllers.

As an example application, we consider cloud database services (database-as-a-service; DBaaS), which manage data and automate the operations of distributed database infrastructures. They typically employ a convention-over-configuration paradigm and do not adjust request-level parameters unless specified by developers. Nonetheless, many configuration parameters have significant performance impact for clients. In order to adjust them at runtime, one must address several challenges.

First, the impact of individual actions taken in the system is difficult to measure due to serving many concurrent requests. Further, measuring system performance might only be possible after some time

Deep Reinforcement Learning Workshop, NIPS 2016, Barcelona, Spain.

has passed. Second, client-server architectures might prevent infrastructure providers from directly observing client performance. This work addresses the challenges of concurrent delayed credit assignment by introducing a mechanism for concurrent asynchronous experience management called delayed experience injection. Specifically, we modify normalized advantage functions [4], a recently introduced method for continuous deep reinforcement learning, to learn optimal cache expiration durations for dynamically changing query results. Results show that our controller outperforms a statistical estimator based on arrival processes.

2 Background and related work

2.1 Preliminaries

The parameter learning problem conforms to the setting of an infinite-horizon discounted Markov decision process where an agent interacts with an environment described by states $s \in \mathcal{S}$ and aims to learn a policy π that governs which action $a \in \mathcal{A}$ to take in each state [5]. At each discrete time step t, the agent takes an action a_t according to its current policy $\pi(a|s)$, transitions into a new state s_{t+1} according to the (often stochastic) environment dynamics, and observes a reward r_t. The goal of the agent is to maximize cumulative expected rewards $R = \mathbb{E}[\sum_t \gamma^t r_t]$, where future rewards are discounted by γ. This is often achieved by learning a Q-function $Q^\pi(s_t, a_t)$ which represents the expected return when starting from state s_t, taking action a_t with the highest Q-value and following π thereafter [6]. Mnih et al. have demonstrated how deep neural networks can be used as value functions for a variety of complex tasks by utilising a replay memory of stored experiences, and using a second value function to stabilize learning (fixed Q-target) [2].

In this work, we utilize normalized advantage functions (NAFs), which have recently been suggested as an effective method for continuous DRL [4]. The key problem in continuous RL is to efficiently select the action maximising the Q-function, i.e. $arg\ max_a Q(s, a)$ while avoiding to perform a costly numerical optimization at each step. Unlike other approaches in continuous DRL (e.g. deep deterministic policy gradients [7]), NAFs avoid the use of a second actor or policy network that needs to be trained separately. A single neural network $Q(s, a|\theta^Q)$ is used to output both a value function $V(s|\theta^V)$:

$$V^\pi(s_t|\theta^V) = \mathbb{E}_{r_{i\geq t}, s_{i>t} \sim E, a_{i\geq t} \sim \pi}[R_t|s_t, a_t] \tag{1}$$

and an an advantage term $A^\pi(s_t, a_t)$:

$$A^\pi(s_t, a_t|\theta^A) = Q^\pi(s_t, a_t|\theta^Q) - V^\pi(s_t|\theta^V) \tag{2}$$

Decomposing Q into a state-value term V and an advantage term A is a technique for variance reduction often used in policy gradient methods [8, 9]. Gu et al. suggest using a quadratic advantage term:

$$A(s, a|\theta^A) = -\frac{1}{2}(a - \mu(s|\theta^\mu))P(s|\theta^P)(a - \mu(s|\theta^\mu)), \tag{3}$$

where $P(s|\theta^P)$ is a positive-definite square matrix parametrized by a lower-triangular matrix $L(s|\theta^P)$, which is given by a linear output of the network ($P(s|\theta^P) = L(s|\theta^P)L(s|\theta^P)^T$), with the diagonal entries exponentiated. Hence, the maximizing action is always given by $\mu(s|\theta^\mu)$. Updates are computed by minimizing the mini batch loss $L = \frac{1}{N}\sum_i(\gamma_i - Q(s_i, a_i|\theta^Q))^2$ and using a replay memory, as well as a target network Q' (as described by Mnih et al.) to compute $y_i = r_i + \gamma V'(s_{i+1}|\theta^{Q'})$. NAFs are especially appealing in our context because using a single network simplifies asynchronous update semantics.

2.2 Related work

Our work is conceptually most similar to Tesauro et al.'s work on resource allocation in data centres [10, 11, 12]. They utilized a perceptron with a single hidden layer to make server allocation decisions for different applications. Their method aims to maximize the expected sum of service level agreement payments while minimizing penalties for unmet service-level objectives. Their state comprised the mean arrival rate of HTTP requests and the number of currently allocated servers. The same approach has also been successfully applied to power management of web servers [13].

Notably, their solution relies on a hybrid approach where initial values are improved by a parametric model. Our work similarly relies on arrival rates of certain events but shifts learning from global state and server-level decisions to per-request state and request-level decisions. RL has also been employed for auto-configuration of Xen virtual machines [14, 15].

For web caching, Candan et al. initially explored the notion of invalidation-based caching for web content [16], as opposed to treating web caches as static content stores or media distribution servers [17, 18].

Prior approaches on thread-parallel or distributed DRL such as A3C [19] or Gorila [20] accelerate training by having learners operate on separate copies of single-threaded environments (e.g. Atari simulator). Gu et al. have also recently applied distributed asynchronous NAFs to shared learning of 3D robot manipulation tasks [21]. In their work, distributed robot controllers asynchronously share their (sequentially) collected experiences with a central server. In contrast, our work considers thread-asynchronous training in a single node environment with a high degree of concurrency and delayed asynchronous reward assignment.

3 Problem overview

3.1 Estimating cache expirations

We consider the problem of learning parameters for cloud database services on a per-request level granularity. For each request, the database server can set response parameters affecting client performance. Multiple clients (e.g. mobile devices) can query and update the same entries in a single database. In this paper, we address the problem of estimating cache expiration times (time-to-live; TTL) for dynamically changing query results, which we now introduce.

A query q issued by a client is executed by a database and yields a set of result records of varying cardinality n, identified by their unique keys $k_1, .., k_n$. Query results can be cached for a specified time interval $t = TTL$ at server-controlled caches such as content delivery networks (CDNs) or reverse-proxy caches. If a key k is updated, all cached queries containing k become invalid and an invalidation request is sent to all caches. There are multiple reasons why estimating accurate TTLs for query results is critical.

First, the server has to store all cached queries and their expiration times to determine which queries need to be invalidated. Using indiscriminately large TTLs for dynamically changing database content would thus both strain cache capacities as well as create too much overhead to determine invalidations, as every update needs to be compared against all cached queries. Further, every invalidation creates the potential for stale reads, as clients can retrieve stale cached results while the invalidation is propagated to all cache edges [22]. In contrast, small TTLs increase client latencies significantly if the database server is physically remote since web performance is primarily governed by round-trip latency [23]. In the following subsection, we will introduce a Monte Carlo framework designed to analyze web request flows.

3.2 Simulation environment

We have implemented a Monte Carlo simulation inspired by the Yahoo! cloud serving benchmark (YCSB) [24, 25]. YCSB is a benchmark suite for cloud databases and defines a set of typical web workloads (e.g. read-dominant, scan-intensive, write-dominant). Custom workloads can be specified with properties such as request distribution, record count, operation count, and read/write/insert/scan/delete mixture. After running a workload, YCSB provides throughput and latency histograms. Our implementation provides the same workloads but instead of just providing a client interface and workloads, we stack together multiple layers (clients, caches, databases) and replicate web connection semantics. That is, YCSB operates on a synchronous thread-per-request model while web browsers typically use 6 HTTP connections and fetch multiple resources asynchronously.

Figure 1 gives a schematic overview of the request flow for queries. Clients sample query or update requests from the workload mixture. In the simulation, each entry has a single field with a numerical value. Operations read or modify a single key k drawn from an access distribution. For easier result size control, queries are defined as range queries that request all objects for which the corresponding database entry satisfies the range predicate on the numerical field.

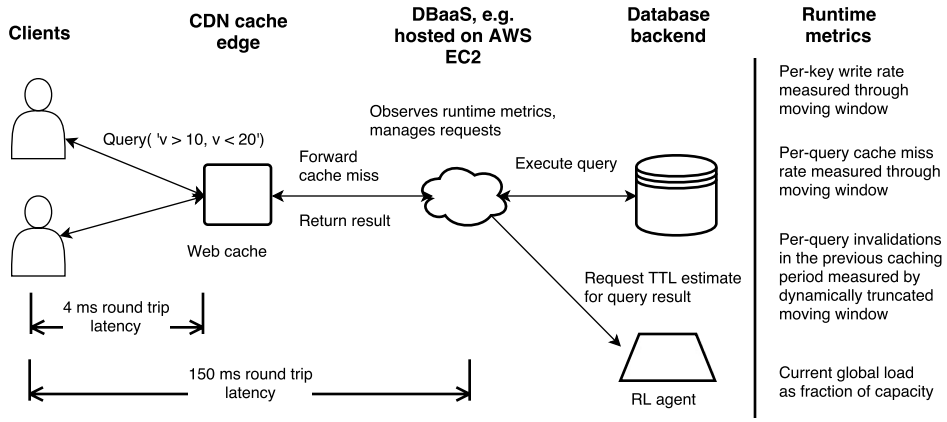


Figure 1: Overview of our Monte Carlo simulator. Clients issue requests which first go against local cache edges. On a cache miss, requests are forwarded to a backend in another geographical region, inducing much higher round-trip latency. The DBaaS middle ware collects metrics on writes, cache misses and invalidations.

We assume the setting of a geographically remote database server hosted in the California Amazon EC2 region with a client located in Europe. Clients can drastically reduce request latencies if dynamically changing query results are present in a near cache such as content delivery network (CDN) edge. Multiple clients may query and write the same data, e.g. by commenting on a social media post or refreshing their news feed. If a client executes an update operation on a key against the DBaaS, it determines which queries need to be invalidated by re-evaluating a maintained index of cached queries against the update (e.g. through stream processing). Entries are removed from the index once the respective TTL expires. In the simulation, we pre-construct an index of queries and initial result keys and can thus cheaply determine invalidated queries by incrementally updating this index at runtime. The DBaaS then sends out asynchronous invalidation requests to the CDN. We regard a read operation as a special case of query with result size one. If another client requests a cached entry before an invalidation has been completed, a stale read occurs [22]. For TTL estimation, the server can utilize update rates on records as well as cache miss rates and invalidations on queries. In the following section, we will discuss different TTL estimation strategies and explain how our approach leverages these metrics.

4 Estimating TTLs

4.1 True TTL

We begin by considering a hypothetical optimal strategy. Ideally, TTLs are estimated to expire right before an update invalidates the respective cached result. We define the *true TTL* as the interval between serving the query and the query result being invalidated by a write w. In our simulation, we can hence capture the optimal action for every step after the respective query has been invalidated. Since this would not capture true TTLs for queries which expire from the cache without invalidation, we further measure the "theoretical" true TTL for queries which are currently not cached by evaluating which queries would have been invalidated if they had not expired.

4.2 Baseline solution

We first introduce a baseline solution relying on the assumption of a Poisson process of incoming updates. For a Poisson process, the inter-arrival times of events have an exponential cumulative distribution function (CDF), i.e. each of the identically and independently distributed random variables X_i has the cumulative density $F(x;\lambda) = 1 - e^{(-\lambda x)}\ for\ x \geq 0$ and mean $1/\lambda$. For now, we make the impractical assumption that for each database record, there is an estimate of the rate of incoming writes λ_w over some time window.

The result set of a query of cardinality n can then be regarded as a set of independent exponentially distributed random variables $X_i, \ldots, X_n$ with different write-rates $\lambda_{w1}, \ldots, \lambda_{wn}$. Estimating the

TTL for the next update to any element of the result set requires a distribution that models the minimum time to the next write, i.e. $X_{min} = min\{X_1, \ldots, X_n\}$, which is again exponentially distributed with $\lambda_{min} = \lambda_{w1} + \ldots + \lambda_{wn}$. We can hence obtain an estimate of the TTL by using the expected value until the next write on any record present in the result set, which would invalidate the cached result: $TTL_{poisson} = \mathbb{E}[X_{min}] = 1/\lambda_{min}$. As we will show in the evaluation, the key problem of this approach is providing it with default write rates or default TTLs if no write-rate information is available.

4.3 TTL estimation with NAFs

Motivation. TTL estimation is an appealing problem for reinforcement learning solutions as they provide a natural way to deal with time-dependent and noisy feedback loops in control problems. We proceed to model the TTL estimation problem using NAFs. First, the previous solution does not distinguish between queries that are read often and queries that are requested very rarely, i.e. it does not incorporate cache miss rates. Second, the baseline solution cannot deal well with sparse information in the state: For most objects, write rate information might not be available. Given no (or often partial) information an estimator needs to fall back to default values.

State. We use individual record metrics to learn TTLs for query result sets. This is preferable to using an encoding of a query itself as the state, since many equivalent query strings lead to the same result. Using record-level metrics allows for an easier generalization when the result sets of seen and unseen queries overlap. Since update and query operations are generally independent, we also utilize query cache miss rates as part of the state to measure TTL impact by inputting the difference between current and last miss rate.

Query results can significantly vary in size but the contribution of records which are rarely updated to the TTL should be negligible. We hence set the number of inputs to the mean expected result size n and input a sorted vector of the top n available write rates to the network – other components in the case of $card(result) > n$ are discarded.

Reward. The reward needs to encode as much information as possible from what the DBaaS can observe. From a service provider's perspective, rewards should allow to trade off invalidations against cache misses. The server cannot observe direct reward measures such as cache hit rates for clients or CDN cache utilisation. We note that we expect most queries not to be invalidated frequently (or at all) due to the power-law nature of web workloads [26]. Hence, using the expected invalidation rate as a TTL estimation strategy is unlikely to be successful as there will be no information for most queries.

To punish invalidated queries, the agent needs to know which actions cause a query to be invalidated by a later write. This means the server can only sensibly measure a reward after some delay t_d. The same problem exists for state measurements relying on cache miss rates. If there is an invalidation at time t_{inv} before the expiration timestamp of a cached query t_{exp}, the reward can be computed as the difference between invalidation time and expiration time (in seconds), i.e. $r_t = t_{inv} - t_{exp}$. If there has been no invalidation, less informative metrics have to be used.

No invalidation before expiration means that the TTL could have been higher unless capacity constraints prohibit longer caching times. In this case, we hence use a static reward r and scale it by the current load c_t (current cached queries divided by capacity) to encourage longer TTLs when fewer queries are cached and shorter TTLs when load is close to capacity (by using $-c_t$ if larger than some threshold), i.e. $r_t = r \cdot (1 + c_t)$. The intuition behind this approach is that only using invalidation timestamps would not allow to give a reward for $80 - 90\%$ of queries (due to low invalidation rates, as shown in the evaluation), and would not give opportunity to globally up- or down-regulate estimates according to system-wide load. In the following section, we explain how we practically perform delayed reward and next-state measurements.

4.4 Delayed experience injection

In standard RL semantics, the agent sequentially moves through a Markov decision process by taking steps and recording transitions of state, action, reward and next-state. When using a replay memory, learning is decoupled from current state and actions by sampling transitions from the memory to perform mini-batch gradient descent. Consequently, if the desired runtime measurements for rewards

and next-states are not available immediately and the agent has to deal with many concurrent requests, the application needs to keep track of "incomplete" transitions and decide when to complete them.

Algorithm 1 Asynchronous NAF with delayed experience injection.

Initialize empty replay memory $\mathcal{R} \leftarrow \emptyset$
Initialize Q-network $Q(s, \mu|\theta^Q)$ with random weights
Initialize target network Q' with weight $\theta^{Q'} \leftarrow \theta^Q$
Initialize random process $\mathcal{N}$ for initial exploration
for $t = 1, T$ **do**
  Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$
  Create incomplete transition (s_t, a_t),
  Enqueue (s_t, a_t) in expiration queue with $exp_t = now() + t_d$
  At $t = exp_t$, asynchronously execute queue consumer:
    Compute r_t and s_{t+1}
    Insert complete transition (s_t, a_t, r_t, s_{t+1}) into $\mathcal{R}$.
  Submit asynchronous loss computation:
    Compute $y_i = r_i + \gamma V'(s_{i+1}|\theta^{Q'})$
    Minimize $L = \frac{1}{N}\sum_i(\gamma_i - Q(s_i, a_i|\theta^Q))^2$
    Periodically update $\theta^{Q'} \leftarrow \theta^Q$
end for

Algorithm 1 shows the control flow in our model. The DBaaS server computes the state from write rate and cache miss metrics for an incoming query and creates an incomplete transition (s_t, a_t). This is then enqueued into an *expiration queue* data structure which triggers an asynchronous consumer after the specified delay t_d, which we set to a_t in our experiments (i.e. the TTL). The consumer computes the reward and the next state as described above by requesting the last invalidation timestamp and cache miss rate. It then inserts the completed transition (s_t, a_t, r_t, s_{t+1}) into the replay memory $\mathcal{R}$, a mechanism we call *delayed experience injection* (DEI). DEI decouples not only current state from learning (as a replay memory does) but also decouples future state and reward computation for specific queries from the sequence of incoming states. Hence, NAF-DEI also solves a different problem than the recently introduced distributed asynchronous NAF [21], where multiple controllers sequentially collect experiences without delay in the experience computation itself. Further, the difference between DEI and the standard delayed reward assignment problem [27] is that DEI deals with concurrent delayed credit assignment.

Updates are performed similar to standard NAF except that the update step is also computed asynchronously by another thread. This is necessary because blocking incoming decision queries on the update step would result in latency spikes.

5 Evaluation

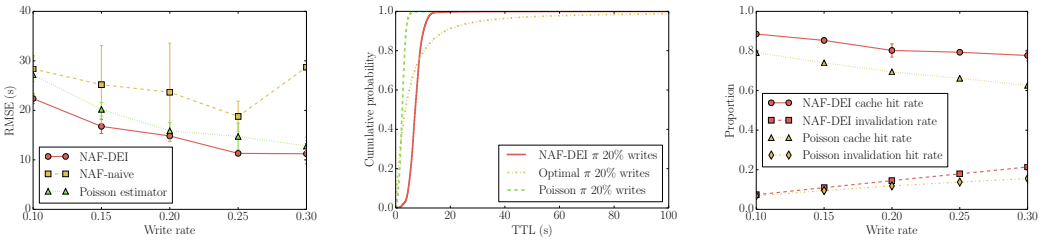
5.1 Setup

The goal of the evaluation is to demonstrate the principal feasibility of using deep reinforcement learning for request-level parameter learning. We begin by describing the experimental setup. We have implemented our simulator in Java 8 (for YCSB compatibility) and utilized deeplearning4j [28] (0.5.0) for the NAF implementation. We set 10 clients with each 6 concurrent connections to execute a combined target throughput of 1,000 (asynchronous) operations per second. They accessed 10,000 documents with 1,000 distinct queries under varying workloads. Updates and queries were drawn from a Zipfian distribution (Zipf constant 0.6). Each workload was run for 30 minutes on a commodity 4 core desktop machine and results were averaged over five runs. Query result sizes were set to be between $[1, 20]$ documents by sampling scan ranges from $\mathcal{N}(10, 5)$ (resp. $\mathcal{N}(5, 2)$ in smaller experiments). Simulated round-trip latencies reflected a client in Europe, a CDN edge in Europe (4 ms round trip latency), and a server in the EC2 California region (150 ms round trip latency.)

The NAF agent used 10 inputs (resp. half the maximal result size in other experiments) for write rates w and 1 input for cache miss differences for 11 inputs in total, followed by 2 hidden layers with each 30 neurons using rectified linear unit activations. Updates were performed using an Adam [29]

updater with mini-batches of size 10 (all training was executed by the CPU due to the small model size). Learning was non-episodic and fully on-policy after an initial exploration period. The learning rate was set to $\alpha = 0.0005$ with gradients clipped at 30, allowing for aggressive updates.

5.2 Results



(a) Errors against optimal policy for different workload mixtures.

(b) Comparison of learned, optimal and Poisson CDF for $w = 20\%$.

(c) Cache performance.

Figure 2: Evaluation of our approach against different baseline configurations.

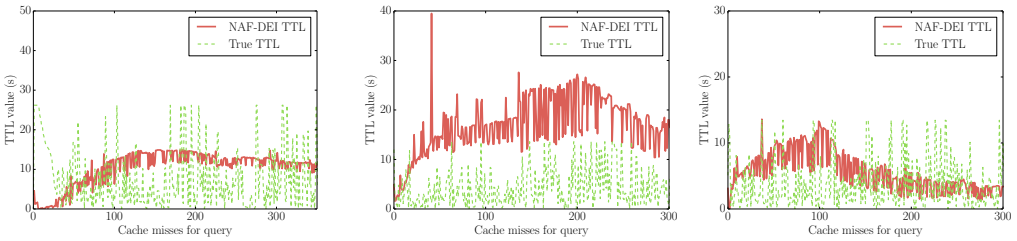
Figure 2a compares results for different workload mixtures as the root-mean-squared per-step error (RMSE) against the optimal policy (truncated at the 99th percentile to remove outliers). The Poisson estimator is limited by its need to specify some default action if no write rate estimates are available. A feasible strategy is to set a maximum for TTLs and presume that the write rate on unknown objects corresponds to the inverse maximum. This is preferable to setting a single default estimate because such a default value would not account for different result sizes. We ran a number of configurations for the Poisson estimator and report the error for the best configuration (max TTL 300 s). The NAF agent (using DEI) outperformed the Poisson estimator, as it is not dependent on a maximum value and uses 0 as input if no write rate is available instead of default per-key estimates.

Further, we show the impact of running NAF without DEI (NAF-naive). NAF-naive instantly computes rewards and will thus rely on invalidations caused a by prior action, creating much larger error and larger standard deviations (for some configurations, learning from the wrong rewards leads to good accidental performance). NAF-DEI outperformed naive NAF by 38.5% on average. While the TTL estimation problem is a special case due to the action being a time period where the delay is set to the action value, delayed asynchronous reward computation is likely beneficial to various other problems with high degrees of concurrency. We also observe that the absolute error is large for all solutions and improves with larger write rates, as more per-key information becomes available and the CDF turns steeper. Figure 2b compares CDFs from traces of NAF-learned, optimal and Poisson-estimated policies. Both approaches produce an overly steep CDF as the long tail of the optimal policy is principally difficult to predict.

It is important to recognize hat the theoretical optimal TTL we use to compute errors is not an ideal performance measure. It assigns an error to queries which expire without invalidation by computing the error until a future point in time when this query would have been invalidated if it had still been cached. In contrast, the reward function specifies a complex trade-off between invalidations, cache misses and global load. We nonetheless report this error since it allows for a more neutral comparison of strategies with different objective functions. We also compared learning performance in different settings to a hypothetical default-value predictor which knows in advance which single default value would give the lowest error per workload mixture. Our results show that the learner can outperform the default-predictor by over 60% for individual queries and up to 20% for the top 20% of queries (sorted by observed cache misses) for some workloads, performing better with smaller result sizes. With increasing write rates or increasing result sizes, true TTL distributions become more narrow and outperforming a default value is more difficult. However, the learner's mean error roughly matches (sometimes outperforms) the default-predictor due to large errors from the long tail of the access distribution. Note that the NAF-agent's error includes the online training period, as we wanted to evaluate the feasibility of a controller without prior training.

Figure 2c shows actual cache performance in terms of achieved cache hit rates (higher is better) and invalidation rates (lower is better). NAF-DEI accepts slightly higher invalidation rates to ensure high cache performance while the Poisson estimator tends to predict lower TTLs due to using default write rates. Assuming an equal weighting between cache hit rate and invalidation rate, NAF-DEI

offers much better cache performance with mean cache hit rate 88.5% for $w = 10\%$ with 7.3% invalidations versus 79.5% cache hit rate and 6.8% invalidations for the Poisson estimator. For $w = 30\%$, NAF-DEI achieved 77.7% cache hits and 21.4% invalidations versus 62.6% cache hits and 15.6% invalidations for the Poisson estimator. These results stress that simply estimating lower TTLs does not save many invalidations if estimates are imprecise on a per-query level.



(a) Query trace with $w = 10\%$.

(b) Query trace with $w = 30\%$.

(c) Reward-adjusted query trace with $w = 30\%$.

Figure 3: Individual query trace examples illustrate learning over time and noise in the optimal policy.

To better understand learning behavior, we examine traces of individual queries through experiment runs. Figure 3a shows learned and optimal actions for every instance an individual query is observed throughout one experiment with $w = 10\%$, illustrating both the online learning process and the noise in the optimal policy. Note that while learning seems fast, the plot does not show how much time (and hence learning from other queries) passes between each instance of the query. For a throughput of $1,000$ concurrent requests per second, most learning (by magnitude of error) took place in the first few minutes (about 20% of experiment duration). For $w = 30\%$, the agent uses the additional information from both more writes and faster changing cache miss rates to make more specific guesses, as seen in figure 3b. While the learner seems to match the general shape of the series of true TTLs, it systematically overestimates true values by $5 - 10$ s. This is because the reward was statically configured to always encourage high cache hit rates independent of higher write rates. In figure 3c, we adjusted rewards according to the workload mixture, i.e. we encouraged the load to stay below $1 - w$. Consequently, TTLs for the query decrease over time once the cache fills up. The Poisson estimator could be similarly tuned by using not the expected value until seeing the next write but some other quantile, e.g. 75% to encourage higher cache hit rates. However, RL-based solutions offer a natural interface to incorporate additional performance metrics without having to translate them into an analytical model, i.e. by determining which Poisson parameters correspond to the desired performance.

Our results allow some outlook on learning parameters at a much larger scale. Due to the Zipf nature of web workloads [26], most queries and updates will concentrate on a small sets of "hot" database records for which it might be feasible to track runtime information and use them for specific predictions. In conclusion, the combination of small model size and a high degree of concurrency allowed NAF-DEI to achieve an effective trade-off between avoiding invalidations and ensuring high cache hit rates without requiring prior training.

6 Conclusion and future work

To the best of your knowledge, this work presents the first application of deep reinforcement learning in predicting request-level parameters in computer systems. We introduced the concept of delayed experience injection to capture asynchronous reward/next-state semantics in concurrent environments where relevant metrics are only available later. The key idea of our work is that instead of learning global parameters from global metrics, DRL can facilitate per-request decisions based on fine-grained metrics. Results show that our NAF-based approach can outperform a statistical estimator on the TTL estimation problem by leveraging available runtime information.

In future work, we will explore in more detail how to relate dynamic reward adjustments to specific service level objectives in non-stationary environments. We have also made the simplifying assumption of a single node backend receiving all incoming requests. Future work in this domain hence needs to investigate coordination and distributed learning in infrastructures where each node only observes part of the environment, as opposed to each node observing a separate copy of the problem.

Acknowledgements

This work was supported by the EPSRC (grant reference EP/M508007/1) and a Computer Laboratory Premium Scholarship (Sansom scholarship).

References

[1] 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.

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[3] H. van Hasselt, A. Guez, and D. Silver. Deep Reinforcement Learning with Double Q-learning. *ArXiv e-prints*, September 2015.

[4] Shixiang Gu, Timothy Lillicrap, Ilya Sutskever, and Sergey Levine. Continuous deep q-learning with model-based acceleration. March 2016.

[5] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

[6] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3):279–292, 1992.

[7] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. September 2015.

[8] Advantage updating. Technical Report WL–TR-93-1146, Wright-Patterson Air Force Base Ohio: Wright Laboratory, 1993. URL `http://leemon.com/papers/1993b.pdf`.

[9] Advantage updating applied to a differential game. In editors Gerald Tesauro, et al, editor, *Advances in Neural Information Processing Systems 7*, pages 353–360. MIT Press, 1994. URL `http://leemon.com/papers/1994hbk.pdf`.

[10] G. Tesauro, R. Das, W.E. Walsh, and J.O. Kephart. Utility-function-driven resource allocation in autonomic systems. In *Autonomic Computing, 2005. ICAC 2005. Proceedings. Second International Conference on*, pages 342–343, June 2005. doi: 10.1109/ICAC.2005.65.

[11] G. Tesauro, N. K. Jong, R. Das, and M. N. Bennani. A hybrid reinforcement learning approach to autonomic resource allocation. In *Proceedings of the 2006 IEEE International Conference on Autonomic Computing*, ICAC '06, pages 65–73, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 1-4244-0175-5. doi: 10.1109/ICAC.2006.1662383. URL `http://dx.doi.org/10.1109/ICAC.2006.1662383`.

[12] Gerald Tesauro et al. Online resource allocation using decompositional reinforcement learning. In *AAAI*, volume 5, pages 886–891, 2005.

[13] Gerald Tesauro, Rajarshi Das, Hoi Chan, Jeffrey Kephart, David Levine, Freeman Rawson, and Charles Lefurgy. Managing power consumption and performance of computing systems using reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 1497–1504, 2007.

[14] Jia Rao, Xiangping Bu, Cheng-Zhong Xu, and Kun Wang. A distributed self-learning approach for elastic provisioning of virtualized cloud resources. In *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2011 IEEE 19th International Symposium on*, pages 45–54. IEEE, 2011.

[15] Cheng-Zhong Xu, Jia Rao, and Xiangping Bu. Url: A unified reinforcement learning approach for autonomic cloud management. *Journal of Parallel and Distributed Computing*, 72(2): 95–105, 2012.

[16] K. Selçuk Candan, Wen-Syan Li, Qiong Luo, Wang-Pin Hsiung, and Divyakant Agrawal. Enabling dynamic content caching for database-driven web sites. In *SIGMOD*, pages 532–543, New York, NY, USA, 2001. ACM. ISBN 1-58113-332-4. doi: 10.1145/375663.375736. URL `http://doi.acm.org/10.1145/375663.375736`.

[17] Michael J. Freedman. Experiences with coralcdn: A five-year operational view. In *In Proc NSDI*, 2010.

[18] Qi Huang, Ken Birman, Robbert van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C. Li. An analysis of facebook photo caching. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 167–181, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2388-8. doi: 10.1145/2517349.2522722. URL `http://doi.acm.org/10.1145/2517349.2522722`.

[19] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. February 2016.

[20] Arun Nair, Praveen Srinivasan, Sam Blackwell, Cagdas Alcicek, Rory Fearon, Alessandro De Maria, Vedavyas Panneershelvam, Mustafa Suleyman, Charles Beattie, Stig Petersen, et al. Massively parallel methods for deep reinforcement learning. *arXiv preprint arXiv:1507.04296*, 2015.

[21] S. Gu, E. Holly, T. Lillicrap, and S. Levine. Deep Reinforcement Learning for Robotic Manipulation. *ArXiv e-prints*, October 2016.

[22] Felix Gessert, Michael Schaarschmidt, Wolfram Wingerath, Steffen Friedrich, and Norbert Ritter. The cache sketch: Revisiting expiration-based caching in the age of cloud data management. BTW '15, Hamburg, Germany, March 2015.

[23] Ilya Grigorik. *High performance browser networking*. O'Reilly Media, [S.l.], 2013. ISBN 1449344763 9781449344764.

[24] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0036-0. doi: 10.1145/1807128.1807152. URL `http://doi.acm.org/10.1145/1807128.1807152`.

[25] Swapnil Patil, Milo Polte, Kai Ren, Wittawat Tantisiriroj, Lin Xiao, Julio López, Garth Gibson, Adam Fuchs, and Billie Rinaldi. Ycsb++: Benchmarking and performance debugging advanced features in scalable table stores. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, pages 9:1–9:14, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0976-9. doi: 10.1145/2038916.2038925. URL `http://doi.acm.org/10.1145/2038916.2038925`.

[26] L. Breslau, Pei Cao, Li Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: evidence and implications. In *INFOCOM '99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 1, pages 126–134 vol.1, Mar 1999. doi: 10.1109/INFCOM.1999.749260.

[27] Christopher John Cornish Hellaby Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge, UK, May 1989. URL `http://www.cs.rhul.ac.uk/~chrisw/new_thesis.pdf`.

[28] Deeplearning4j Development Team. Deeplearning4j: Open source distributed deep learning for the JVM, Apache Software Foundation License 2.0. http://deeplearning4j.org, 2016.

[29] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. December 2014.