DeepChunk: Deep Q-Learning for Chunk-based Caching in Data Processing Networks

Yimeng Wang, Yongbo Li, Tian Lan, and Vaneet Aggarwal

***Abstract*—A Data Processing Network (DPN) streams massive volumes of data collected and stored by the network to multiple processing units to compute desired results in a timely fashion. Due to ever-increasing traffic, distributed cache nodes can be deployed to store hot data and rapidly deliver them for consumption. However, prior work on caching policies has primarily focused on the potential gains in network performance, e.g., cache hit ratio and download latency, while neglecting the impact of cache on data processing and consumption.**

In this paper, we propose a novel framework, DeepChunk, which leverages deep Q-learning for chunk-based caching in DPN. We show that cache policies must be optimized for both network performance during data delivery and processing efficiency during data consumption. Specifically, DeepChunk utilizes a model-free approach by jointly learning limited network, data streaming, and processing statistics at runtime and making cache update decisions under the guidance of powerful deep Q-learning. It enables a joint optimization of multiple objectives including chunk hit ratio, processing stall time, and object download time while being self-adaptive under the time-varying workload and network conditions. We build a prototype implementation of DeepChunk with Ceph, a popular distributed object storage system. Our extensive experiments and evaluation demonstrate significant improvement, i.e., 43% in total reward and 39% in processing stall time, over a number of baseline caching policies.

***Index Terms*—Data streaming and processing, caching, reinforcement learning.**

I. Introduction

Data collection, streaming, and processing are essential tasks for modern networks due to the rapid development in areas such as Internet of Things, sensor networks, online data analytics and edge computing [1], [2], [3]. In such applications, massive volumes of data collected (and stored) by the network need to be streamed to multiple processing units to compute desired results in a timely fashion. One example of this type of application is intelligent transportation, where large volumes of sensor data and video footage are recorded from various monitoring points, and then fetched on-demand into distributed computing nodes, for objectives from vehicle identification to traffic analysis. Due to ever-increasing traffic in Data Processing Networks (DPN), [4], [5], [6], they often enhance the performance by caching data in nodes close to computing units and rapidly delivering those data for consumption.

The design of caching policies, however, is primarily focused on the potential gains in network performance, e.g., cache hit ratio and download latency, while neglecting the impact of data processing and consumption. These include the Least Frequently Used [7] and Most Popular Object [8] strategies that achieve a high cache hit ratio, and the Least Recently Used (LRU), qLRU, and kLRU [9] strategies that use request recency for cache update. Yet, for data processing networks, a data stream is simultaneously being fetched and consumed on the fly. Existing caching policies only considering network performance during data delivery fall short on addressing end-to-end tuple processing time [2], which depends on both data delivery and consumption. In practice, when available network bandwidth cannot fully sustain all data processing needs [1], [2], stalls during data processing become inevitable. As a result, the design of caching policies should be made aware of data consumption needs, to mitigate processing stalls and fulfill the timely online processing requirements.

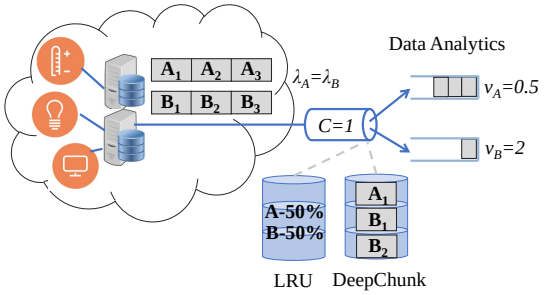


Fig. 1. An illustrative example of DeepChunk approach for data processing.

In this paper, we propose a novel framework, DeepChunk, which leverages deep Q-learning [10] for chunk-based caching in DPN. Our key idea is that cache policies must be optimized for both (i) network performance during data delivery and (ii) processing efficiency during data consumption. Since the classical hit ratio does not account for partial files in the cache, we use the notion of *chunk hit ratio* that accounts for the existence of partial files in the cache. The necessity of taking data processing into account in caching can be illustrated via a simple example in Figure 1. Two data objects, A and B (each consisting of 3 chunks), are recorded on network edge and fetched by different data processing units through a link that has a capacity of 1 chunk per second and is equipped with a cache of size $C = 3$. If A and B have the same request rate, they are equally likely to be stored in the cache under LRU and LFU policies. Suppose that two applications with processing speed of $\nu_A = 1/2$ and $\nu_B = 2$ chunks per second start to request A and B at $t = 0$, respectively. It is easy to see that if A is cached, the processing of B stalls for a total of 2 seconds (i.e., 1 second waiting time for the first chunk and 1/2 second idle time before processing each subsequent chunks), and similarly the processing of A stalls for 1 second if B is cached. However, we show that stall-free data processing is indeed possible under an optimized, chunk-based caching policy. If the first 2 chunks of B are cached, we can fetch the last chunk of B by $t = 1$, so that the processing of B is stall-free. At the same time, caching

the first chunk of A allows it to continue processing without interruption until the next two chunks are fetched by $t = 2$ and $t = 3$, respective. This caching policy not only achieves the same chunk hit ratio (i.e., 50%), but also minimizes end-to-end tuple processing time (i.e., 6 and 1.5 seconds for A and B respectively) with stall-free data processing. Thus, to fulfill the real-time data processing requirements, it is necessary to consider both network performance and data processing objectives.

A fundamental problem in our chunk-based cache system is the cache update policy, which entails two types of decisions at chunk level – how many chunks of a data object to admit (cache admission) and which chunks to evict from the cache (cache eviction), if it is already full – with the objective of jointly optimizing average chunk hit ratio, processing stall time, and object download time, in DPN. This optimization may be solvable if we can accurately model the correlation between the objective values and underlying variables, e.g., request arrival patterns, data popularity distributions and network conditions. However, this is very challenging and has not yet been well studied in the context of data processing networks, which represent a fairly complicated multi-point to multi-point system with the dynamics of data streaming and processing (from multiple data objects) closely-coupled and jointly impacting the design objectives.

Hence, DeepChunk aims to develop a model-free approach by jointly learning data streaming, processing, and network statistics and making decisions under the guidance of powerful deep Q-learning [10]. We believe the approach is especially promising for chunk-based cache in data processing networks because: (i) it does not rely on precise and mathematically solvable models, hard to obtain in practical DPN, (ii) it is capable of supporting an enormously large state space, and (iii) it is self-adaptive to the dynamic environment, e.g., evolving data popularity/arrivals and time-varying network conditions. In particular, the state space in DeepChunk's Q-learning include data popularity distribution, cache states, request arrival statistics, network conditions and current request information, while its reward captures chunk hit ratio, processing stall time, and object download time. The output action determines DeepChunk's cache update policy, and the resulting reward is further fed-back to the neural network for learning.

We build a prototype implementation of DeepChunk with Ceph [11], a popular distributed object storage system. Ceph's cache node is modified to implement a Q-learning engine and a chunk-based cache module. Upon a request arrival, the cache module immediately streams all cached chunks and request the remaining chunks from the Ceph storage cluster, where all data objects are stored. The Q-learning engine obtains state updates $s(t)$ via UDP messages, decides an action $a(t)$ based on the trained neural network, and then sends the action $a(t)$ through a UDP message to the cache module, which performs cache updates accordingly, calculate the resulting reward, and send it back to the Q-learning engine through another UDP message. To evaluate DeepChunk, we generate data popularities from Zipf distribution and utilize Linux TC traffic control [12] to set up different network configurations. We run extensive experiments to compare DeepChunk with a number of baselines including No-Cache, LRU, kLRU, gLRU. DeepChunk achieves up to 43% reward improvements compared with the baselines, and in all scenarios, it has the ability to balance different design objects, illuminating an interesting tradeoff between chunk hit ratio, processing stall time, and object download time.

The main contributions of this paper are as follows:

- We propose DeepChunk for chunk-based caching in DPN with the objective of jointly optimizing both data processing and network performance.
- DeepChunk leverages a model-free approach by jointly learning limited statistics on the fly and making cache update decisions under the guidance of powerful Q-learning.
- We build a prototype of DeepChunk using Ceph and compare its performance with a number of baseline caching policies. Significant improvement, i.e., 43% in total reward and 39% in processing stall time, is observed.

Related work. LRU-based caching mechanisms have been widely studied [13], [14]. One of the key issue in LRU based caching strategy is that a large file arrival can evict multiple small files [15]. In order to have better performance with realistic file sizes, multiple approaches have been proposed, see [15] and the references therein. Caching has been applied in many applications including radio-access network [16], mobile 5G networks [17], web applications [18], BigData applications [19], and video delivery [20]. In this paper, we propose a chunk-based caching framework that leverages deep Q-learning [10]. Q-learning has been applied to address network optimization tasks such as traffic engineering [21], video bitrate control [22], LTE femtocell configuration [23], and cell outage management [24].

II. Background and Problem Statement

We consider a DPN as a set of data sources and processing units, connected through a communication network to apply various data operations and computations. Massive volumes of data collected (and stored) by the network often need to be streamed to multiple processing units and processed in real time. This feature is also known as stream data processing [2] and can be found in many existing and future applications such as the Internet of Things, sensor networks, online data analytics and edge computing [1], [2]. In such DPN, cache nodes can be deployed between the data source and the processing units, to store and reuse hot data objects passing through the network. Most of the existing cache replacement policies, such as LRU and LFU [25], focus on network performance metrics such as cache hit ratio. However, in DPN, as demonstrated in the previous example in Figure 1, higher hit ratio does not necessarily lead to more efficient data processing, as measured by processing stall time. Motivated by this phenomenon, we acknowledge the new cache design challenges arising from DPN and develop a chunk-based caching framework. guided by deep Q-learning.

Chunk-based caching. To reduce data processing stall, caching the starting chunks of different data objects is crucial, while storing the complete data objects does not offer any additional benefits, as illustrated by the example in Figure 1. It mandates us to consider chunk-based caching in DeepChunk.

At the core of chunk-based caching policies is the need to make cache admission and eviction decisions for each individual data chunk, resulting in higher decision complexity on the fly. In object-based policies, such as kLRU and qLRU [26], cache admission and eviction decisions are typically binary, as to whether or not to add a new object to replace the least-recently-used one in the cache. A chunk-based caching policy, however, must decide *how many* new chunks to admit into the cache when a requested data object traverses the cache node, even-though the cache eviction can employ a least-recently-used strategy.

Consider the example in Figure 1. Suppose that the cache currently contains 2 chunks of A and 1 chunk B, i.e., A_1, A_2, B_1. If a new request of A arrives with a data processing speed of $\nu_A = 0.5$, stall-free processing can already be achieved with the 2 chunks of A in the cache (ignoring a fixed initial waiting time to start streaming). There is no need to admit any more chunks of A into the cache, as it only negatively impacts the processing stall time of other objects that must be evicted as a result. On the other hand, if a new request of B arrives with a data processing speed of $\nu_B = 2$, the processing will stall, demanding new chunks of B to be admitted into the cache. However, we need to carefully choose the number of object-B chunks to admit, since an overly aggressive policy (e.g., adding both B_2 and B_3) would superfluously stall future processing requests of A. In DeepChunk, we collect runtime statistics from the DPN – including cache state, request rates, data processing speeds, chunk sizes, and network configuration – and leverage powerful Q-learning to guide the chunk-based decision making.

III. Proposed DeepChunk Framework

We assume that there are N data objects to be streamed and processed by different processing units. Each object $i \in \{1, 2, ..., N\}$ is partitioned into F_i identical-sized chunks. A cache node is located in close proximity to the processing units and can store up to C data chunks. We consider a time-slotted system model, in which each time bin contains exactly one request arrival. At a given time bin t, let $C_i(t)$ denote the number of chunks of data object i stored in the cache node, satisfying $0 \leq C_i(t) \leq F_i$. The cache size constraint requires $\sum_{i=1}^{N} C_i(t) \leq C$ for any time bin t. For the proposed caching strategy, in steady state the above cache size constraint will hold with equality since we will not waste any cache capacity in the steady state.

Our chunk-based caching policy is formulated as follows. In time bin t, when data object i is requested, $C_i(t)$ chunks stored in the cache are directly streamed to the processing unit, which then immediately begin data processing. At the same time, the remaining $F_i - C_i(t)$ requested chunks will be delivered from the data source and through the network. When the complete data object i is not yet in the cache, i.e., $C_i(t) < F_i$, E_c additional chunks of object i satisfying $0 \leq E_c \leq F_i - C_i(t)$ will be added to the cache. Thus, the number of object-i chunks in cache increases from $C_i(t)$ to $C_i(t+1) = C_i(t) + E_c$ in the next time bin $t+1$. Further, these $C_i(t+1)$ chunks of object i are moved to the head of line in the cache since they become most-recently-used. It is easy to see that to mitigate processing stall, we should always place the first

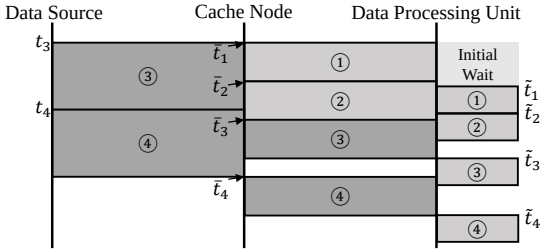


Fig. 2. Calculation of processing stall time.

$C_i(t+1)$ chunks of object i in cache and stream them to jump-start data processing. Finally, when the cache is already full, to make space for the added chunks, an equivalent number of chunks must be removed from the cache. In DeepChunk, we adopt a policy similar to least-recently-used and remove a necessary number of chunks from the tail of the cache line.

Our goal is to design a cache policy to optimize both network performance and data processing objectives. Let $h(t)$ be the chunk hit ratio in time bin t (defined formally later in this section), $T_d(t)$ the average object download time, and $T_s(t)$ the average processing stall time. Specifically, DeepChunk aims to optimize the following objective in steady state (for sufficiently large τ) over feasible cache policies $\mathcal{P}$:

$$\max_{\mathcal{P}} \frac{1}{\tau} \sum_{t=1}^{\tau} \alpha_1 h(t) - \alpha_2 T_s(t) - \alpha_3 T_d(t), \tag{1}$$

where $\alpha_1, \alpha_2, \alpha_3$ are non-negative weights assigned to chunk hit ratio, processing stall time, and object download time, respectively. In practice, we can adjust these weights to achieve different tradeoffs between the network performance and data processing objectives. For instance, when $\alpha_1 = \alpha_3 = 0$, the resulting cache policy minimizes processing stall time $T_s(t)$. We use Reinforcement Learning (RL) to solve the above optimization, and for each network state, to determine the optimal cache replacement, i.e., the number of chunks E_c to replace. We note that existing caching policies typically rely on constant cache replacement strategies, e.g., $E_c = F_i$ for the LRU policy and $E_c = 1$ for the gLRU policy [14], thus lacking the ability to adapt cache replacement on the fly with respect to dynamics in DPN.

Next, we derive different network performance and data processing metrics that will be computed from limited runtime statistics on the fly and leveraged by the RL agent to optimize DeepChunk.

Process stall time. To find processing stall time, we assume that the data processing unit is equipped with a sufficiently large buffer, so all streamed data chunks are consumed by the processing unit without the need for retransmission. We consider a request in time bin t and drop the index t in the following derivations for the simplicity of notations. Since the first C_i out of F_i chunks of data object i are already stored in the cache and the remaining $F_i - C_i$ chunks need to be streamed from the data source, we find the processing stall time by considering a two-stage buffer problem.

As shown in Figure 2, let r_2 denote the available bandwidth (i.e., data streaming speed) from data source to cache node, r_1 denotes the speed from cache node to processing unit, and

ν_i denotes the data processing/consumption speed of object i. We consider 2 stages in data streaming, t_k and $\bar{t}_k$ to denote the time when the kth chunk starts streaming from data source to cache node and from cache node to processing unit, respectively. Since the first C_i chunks are already stored in the cache and the other chunks are streamed one-by-one from data source, we have

$$t_k = \begin{cases} t_{k-1} + \frac{1}{r_2}, & k \in [C_i + 1, K], \\ 0, & k \in [1, C_i]. \end{cases} \quad (2)$$

Next, each data chunk k can be streamed from the cache node to the processing unit, when it becomes available (i.e., at $t = 0$ for any cached chunk and $t_k + 1/r_2$ if it needs to be fetched from data source) and after the preceding chunk $k-1$ is delivered (i.e., at $\bar{t}_{k-1} + 1/r_1$). Combining these, we have

$$\bar{t}_k = \begin{cases} \max\{\bar{t}_{k-1} + \frac{1}{r_1}, t_k + \frac{1}{r_2}\}, & k \in [C_i + 1, F_i], \\ \bar{t}_{k-1} + \frac{1}{r_1}, & k \in [2, C_i], \end{cases} \quad (3)$$

except that $\bar{t}_1$ of the first chunk depends on whether any chunks of object i are cached, i.e.,

$$\bar{t}_1 = \begin{cases} t_1 + \frac{1}{r_2}, & C_i = 0, \\ 0, & \text{otherwise.} \end{cases} \quad (4)$$

Finally, the processing unit consumes data at speed ν_i. Then the processing start time of chunk k can be recursively computed from $\bar{t}_k$'s as

$$\tilde{t}_k = max\{\tilde{t}_{k-1} + \frac{1}{\nu_i}, \bar{t}_k + \frac{1}{r_1}\}, \text{ and } \tilde{t}_1 = t_{ini}, \quad (5)$$

where t_{ini} is the initial wait time (or startup delay) mentioned in Section II. When $\tilde{t}_k > (k-1)/\nu_i + t_{ini}$, the processing unit would experience stall time waiting for chunk k. The total processing stall time of object i is then given by the difference between actual play time and expected play time of the last chunk F_i (since stall time accumulates during processing), that is

$$T_s = (\tilde{t}_{F_i} - \frac{F_i - 1}{\nu_i} - t_{ini})^+. \quad (6)$$

Object download time. The download time of each data chunk k can be found through $\bar{t}_k$, which is the time to starting streaming chunk k from the cache node to the processing unit and is already given in the above analysis of processing stall time. Thus the download time T_d of data object i that is equivalent to the arrival time of last chunk F_i can be derived directly from $\bar{t}_{F_i}$ as follows:

$$T_d = \bar{t}_{F_i} + \frac{1}{r_1}. \quad (7)$$

Chunk hit ratio. We note that for object-based cache systems, a "hit" occurs if the requested data object is found in the cache. Thus, the hit ratio equals to the probability that the requested data object is stored in the cache. For chunk-based caching considered in this paper, we define a similar *chunk hit ratio* to quantify the performance of the proposed caching policy. Specifically, we denote $h_i = C_i/F_i$ as the chunk hit ratio of data object i, which is the percentage of object-i chunks that are stored in the cache and can be directly used to serve a request. This notion of chunk hit ratio generalizes the existing object-based definition, which is binary – $h_i = 0$ for a cache miss and $h_i = 1$ for a cache hit – and cannot be used to evaluate chunk-based caching systems. Finally, the overall chunk hit ratio is simply the average of chunk hit ratio of individual data objects weighted by request arrival rates λ_i, i.e., $(\sum_i \lambda_i h_i)/(\sum_i \lambda_i)$.

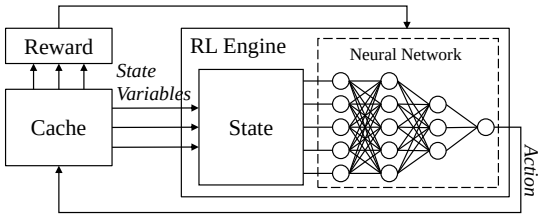


Fig. 3. Reinforcement learning mechanism of DeepChunk decision making.

IV. Reinforcement Learning

A. Q-Learning

To make caching decisions in different system states, we utilize Q-Learning to dynamically generate optimized values.

With the development of the neural networks, Q-Learning is commonly used in modern decision making tasks. The advantages of RL decision making process are: i) supporting enormously large state space, ii) Scalable to different input dimensions, and iii) self-adapting to the environment including evolving data popularity and varying network conditions. We take advantage of RL to adapt to these dynamic features in the caching problem.

Figure 3 illustrates how Q-Learning is solving our cache decision problem. At each time bin t, the cache node monitors the current state $s(t)$ of the system. When a request $q(t)$ is observed at the cache node, the RL Engine feeds the current state $s(t)$ into a neural network to generate an action $a(t)$. Further, according to the state $s(t)$ and the action $a(t)$, the state will be pushed to the next state $s(t+1)$. When the next request arrives, the reward $r(t)$ of the previous action $a(t)$ can be observed, further fed back to train the neural network.

To dynamically improve the reward, a Q-table is maintained. Q-values are correlated with state-action pairs, represent the quality of decisions. It is made of the immediate reward, and an expected optimal future reward which is discounted by a factor $\gamma, (0 \leq \gamma \leq 1)$:

$$Q(s(t), a(t)) = r(t) + \gamma \max_a Q(s(t+1), a). \quad (8)$$

The learning starts with zero-knowledge. At time t, when the detected state $s(t)$ does not exist in the Q-table, it is added with arbitrary Q-values. Actions are chosen following the *Epsilon Greedy* scheme: with probability $1 - \epsilon$, the agent will choose the action that results in the highest Q-value, otherwise, select a random action. The ϵ reduces linearly from 1 to 0.1 over iterations.

After the action is selected, according to the corresponding reward $r(t)$, the Q-value is updated with a learning rate β:

$$\begin{aligned} Q'(s(t), a(t)) \leftarrow & (1 - \beta) Q(s(t), a(t)) \\ & + \beta[r(t) + \gamma \max_a Q(s(t+1), a)]. \end{aligned} \quad (9)$$

Similar to ϵ, the learning rate β is also reduced linearly.

To maintain a large system state space, an artificial neural network is utilized. Different from the Q-table, a neural network produces a vector of Q-values for all actions. When updating the Q-values, a loss function is used to compute the difference between the predicted Q-values and the target Q-values obtained by Equation (8).

$$loss = \sum_{a}(Q' - Q)^2, \tag{10}$$

where Q' represents the target Q-values.

B. Reward, States, and Actions

To use Q-Learning for decision making, we define the three crucial elements - **state**, **action** and **reward** - as follows:

1) Reward: We consider both data processing and network performance to be optimized as the reward. As mentioned in Section III, *chunk hit ratio*, *processing stall time* and *object download time* are defined as the reward variables. We apply three weight factors α_1, α_2 and α_3 to adjust the importances of the three reward variables. Thus, the immediate reward is defined as:

$$r = \alpha_1 h - \alpha_2 T_s - \alpha_3 T_d. \tag{11}$$

Note that when the algorithm is running, all three objectives – h, T_s, T_d – can be measured from the cache node or the data processing units, while Equations (6) and (7) can be utilized to pre-train the neural network from zero knowledge, to improve the speed of convergence.

2) States: The state variables should reflect the system status, further affect the reward feedback of different actions. We measure the system state in our caching model as a four-tuple: $(\overrightarrow{p}, \overrightarrow{c}, \overrightarrow{o}, f)$.

$\overrightarrow{p}$ denotes the *popularity distribution* over all objects. In practical networks, the popularities of objects are changing over time. In order to adapt to the latest popularities, instead of using a static distribution, we maintain a sliding window to monitor popularities of all objects during the past n requests. The currently *cached chunks* for all objects are denoted by $\overrightarrow{c}$. This is important for decision making to keep an appropriate amount of chunks in the cache, and further optimize the reward. The *order* of requested objects is expressed by an array $\overrightarrow{o}$. In our DeepChunk policy, we make decisions to replace the least-recently-used object chunks by new chunks. This state variable will indicate chunks from which object/objects will be removed when making decisions. Finally, f denotes the *currently requested object.* As we designed, only this object will be cached in the same time iteration. Decisions will result in different $c[f]$ in the next time bin.

All the elements in this four-tuple can be obtained at the cache node when a request arrives. For cache decision, the four-tuple is pushed into the input layer of the neural network.

3) Actions: Current RL applications can have a large state space, but the action space is always limited to be small. So, we define an unsophisticated action to fit our problem to this feature. The action $a(t)$ represents *the number of chunks of the requested object f to be added to the cache*. When the action is made, E_c additional chunks of object f will be stored in the cache storage. When the cache storage is full, chunks from the LRU objects will be removed.

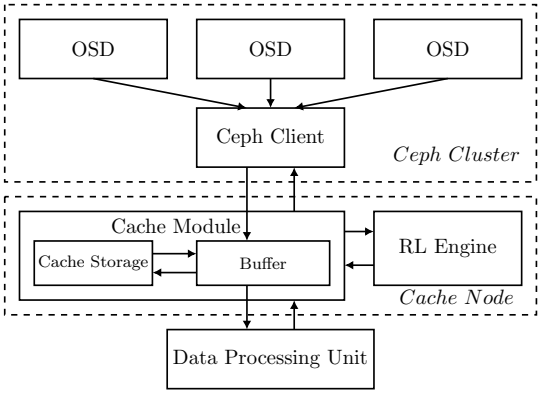


Fig. 4. Implemented DPN diagram.

V. Implementation

In this section, we will describe the prototype implementation details. In specific, we will discuss the DPN setup in terms of the following aspects: data source, cache node, data processing unit, and the links between nodes. The system diagram is depicted in Figure 4.

A. Data Source

Three virtual machines running a Ceph [11] cluster are implemented as the data source. All files used in the experiments are divided into chunks. The files are stored in the three Object Storage Daemons (OSDs). Another virtual machine within the same cluster configuration is set as a Ceph client, which is responsible to collect chunks (Ceph objects) from the storage cluster and send them to the cache node. A Tornado server [27] is implemented on the Ceph client, where it handles the fetch requests from the cache node with responses from the data source. The transmission latency and processing delay within the Ceph cluster machines (those for cloud storage, Ceph client, and Tornado server) are ignored.

B. Cache Node

The cache node aims to simulate an edge node between the data processing unit and the source. When the processing unit sends a request to the cache node, the cache node fetches the chunks of the object that do not exist in the cache from the Tornado server at the source, and delivers the chunks that obtained (previously cached and newly downloaded) to the processing unit simultaneously. Further, with the arriving chunks from the source, the cache node runs an RL Engine to determine the updated placement of the chunks of different objects in the cache. Cache node has two modules - Cache Module and Reinforcement Learning Engine, which together achieve the functions explained above. In our experiments, one virtual machine is utilized to run both the modules.

1) Cache Module: The Cache Module manages the placement of the chunks on the cache node, fetching the chunks from the data source, and delivery of contents to the processing unit. A Tornado server is used to build the connection with the data processing unit, and a Tornado client is paired with the

server on the data source. When an object request is received from the processing unit, the cache module starts to push the cached chunks and fetch the missing chunks from the source simultaneously.

A buffer is used to store the fetched data bytes from the data source. The fetched contents will stay in buffer till all the chunk are sent to the processing unit. The cache policy will determine how many of these contents will be transferred to the cache storage. After the transmission of the contents to the processing unit and the transfer of required contents to the cache storage, the contents are removed from the buffer.

We note that the cache node knows how many contents are in the cache, and what was fetched from the data source. Thus, the metric of chunk hit ratio is collected from the cache module. We will next describe the Reinforcement Learning Engine that makes the decision for the update of caching policy which will influence how many of the chunks fetched from the data source will go to cache storage and which contents will be removed from the cache storage.

2) Reinforcement Learning Engine: The Reinforcement Learning (RL Engine) implements the cache update policy. To accelerate the training process, the neural network is pre-trained using simulated inputs. A serial of Zipf-random requests are used to activate the evolution of the system state. After the action is made, the reward is calculated by the monitored chunk hit ratio and calculated stall/download time from Equations (6) and (7). The pre-trained neural network is then stored in the engine. When a cache decision is needed, the Cache Module will consult the RL Engine. In our implementation, the RL Engine and Cache Module are located on the same node and communicate using UDP messages.

A state listener queries a UDP message from the Cache Module which contains all the required state information, including current cache storage status, request history, and currently requested object. By feeding the state variables to the neural network, the RL Engine obtains the action. The action is then sent to the Cache Module via a UDP message to transmit the cache update decision.

For the real-time training of RL Engine, the reward variables (the chunk hit ratio from the cache module, the response delay and stall time from the data processing unit) are collected after all chunks are received by the processing unit. We note that the obtained reward is dependent on the previous states and actions. In the evaluations, it takes 60 minutes to train the neural network with a million preset samples. The training process is controlled by a linear control signal, which decreases the epsilon greedy parameter ϵ and the learning rate β linearly.

C. Data Processing Unit

A Tornado client is set up as the data processing unit. The processing unit continuously sends object requests to the cache node following a Zipf distribution. A Zipf "seed" is utilized to order the popularities of objects. By changing the "seed", we can set the popularity distributions and object sizes positively or negatively correlation. Each request is sent to the cache node, and the object is received from the cache module. The data processing unit records the reward attributes, including processing time, initial waiting time, and stall time. These reward variables are sent to the cache node, which will be used by the RL engine for online learning and improving of the caching policy.

D. Data/Signal Flow

In the above, we introduced the function modules. Now, we will show an example to trace the data flow, and further describe how the system works.

When initiated, the Tornado servers in the Ceph client and the cache module, and the state listener in the RL Engine are activated. The data processing unit generates a random integer following Zipf distribution and decides which object to be requested according to the Zipf seed. Then a request of the object is sent to the cache module.

The cache module maintains awareness of cache status real-timely. Once the new request is detected, it is able to build up the system state by cache storage status (cached chunks of all objects), updated request history, and the currently requested object. We denote this state as $s(t)$ for clearer demonstration. $s(t)$ is further pushed to the RL Engine via a UDP message. Then, the action listener is activated.

The RL Engine captures state message by the state listener and makes an action $a(t)$ based on previously trained neural network. The RL Engine then sends $a(t)$ to the cache module through a UDP message and starts to listen for the reward.

The cache module obtains the action message $a(t)$, and begins the data flushing process. It first reads all cached chunks of the requested object to its buffer, then sends a request for the missing chunks from its Tornado client to the server in Ceph cluster.

At the Ceph cluster, the chunks stored in Ceph OSDs are fetched as the response data. When sending the response, the Tornado server will flush the chunks into the network interface one by one to achieve a streaming feature.

The cache module starts two threads simultaneously. The fetching thread pushes the fetched bytes into the buffer. It is also responsible to write a part of the chunks into files according to $a(t)$, and further store them into the cache storage. The flushing thread keeps flushing the existing bytes in the buffer to the network interface. When the buffer is empty, it waits for the fetching thread until another chunk is ready in the buffer. After all chunks are fetched and flushed, the connection finishes. Then, the reward listener enters standby mode.

The data processing unit receives the data stream. A timer will measure the download time from the moment when the request is sent, to the moment when the connection is finished. By subtracting the preset initial wait time and processing time, it obtains the processing stall time. A UDP message carrying stall time and download time is reported to the cache module.

The reward value is calculated at the cache module. Note that three terms: chunk hit ratio, processing stall time, and total delay. The chunk hit ratio is obtained from the cache status which can be found in $s(t)$. Further, the reward $r(t-1)$ is sent to the RL Engine.

At the RL Engine, $r(t)$ is utilized to train the neural network. In our Q-Learning algorithm, the reward $r(t-1)$ is correlated with the previous state and action, $s(t-1)$ and $a(t-1)$. After the whole process is complete, the data processing unit loops.

VI. Evaluation

In this section, we present our evaluation results based on the experimental setup described in the last section.

A. Configuration

1) Machine Setup: Virtual machines running the Ceph cluster have identical disk space which is $10GB$. The virtual machine acts as the cache node has $256GB$ disk, $16GB$ memory, and has a core of Intel Xeon CPU E5-2630 v3 (2.40Ghz).

2) Link Setup: All experiment machines are within the same local network. We use the Linux TC traffic control feature [12] to throttle the bandwidths between nodes. The link between the user client and the cache node is set to 5MB/s, while the link between the cache node and the cloud server node is 1MB/s. The bandwidths between Ceph nodes (OSDs and Ceph client) are not a bottleneck, so no additional bandwidth restriction is imposed.

3) Reward Weights: We adjust the weights in the reward function, given in Equation (11). Based to the range of the reward variables, we set α_1, α_2, and α_3 as 10, 10, and 1, respectively. Using these weights, we make processing stall time the most important reward metric in data processing networks.

4) Objects: We use video files to run the experiments. Each file consists of multiple chunks, where each chunk is 1MB. Since the file sizes are not multiples of 1024^2, the last chunk of each file can be less than 1MB. Further, the processing speed of the files is divided into two groups – 1 MB/s and 3 MB/s – to represent different data processing applications.

We generate file popularities using Zipf's distribution [28] with the parameter $z = 1$. We sort the files by their sizes as the popularity ranks. In general, the file sizes and the popularities are dependent on each other. We consider both positive and negative correlation between the file size and the popularity. In positive correlation, larger file sizes have higher popularity and the reverse holds for negative correlation. The popularity numbers are taken as the arrival rates of the requests at the user client.

5) Evaluated Policies: We compare the proposed DeepChunk policy with four baseline strategies, as described below.

No Cache: This caching policy does not store anything in the cache.

LRU: The LRU caching policy [29] moves the requested file to the head of the cache, if already in the cache. If it is not in the cache, the file is added to the head of the cache and the files are removed from the tail to make space for the incoming file. Due to different file sizes, multiple files can be evicted to make space of a large incoming file.

kLRU: Instead of caching every missed request, kLRU [26], [9] deploys k virtual LRU caches ahead of the physical LRU cache to achieve a selective caching scheme. The virtual caches cache only file pointers instead of the data. A virtual/physical cache will store the pointer/data only if there is a hit in the LRU cache ahead of it, and replace the object following the LRU policy. In our experiments, two kLRU policies are tested. kLRU-1 and kLRU-2 policies apply 1 and 2 virtual caches ahead of their physical cache, respectively.

TABLE I
Reward summary for policies in different popularity and cache size settings.

Policy	Reward	
	Positive correlation, cache size = 80	Negative correlation, cache size = 40
DeepChunk	-38.1650385015	-9.4738189784
gLRU	-50.824600919	-12.6091608098
kLRU-1	-58.5616000914	-9.8270152744
kLRU-2	-43.7540371966	-15.4748794595
LRU	-56.9513351377	-16.6214642859
No cache	-119.5670992208	-27.7861073589

gLRU: Distinct from the previous policies, the generalized LRU [14] extends the LRU caching algorithm from file-level to a chunk-level algorithm. Upon a request arrival, if the requested file is not completely in the cache, one additional chunk of that file will be stored. When the cache storage is full, one chunk of the LRU object will be replaced. Further, all chunks of the requested file will be moved to the head of the cache (in order, such that the earlier chunks are towards the head so that they are evicted later).

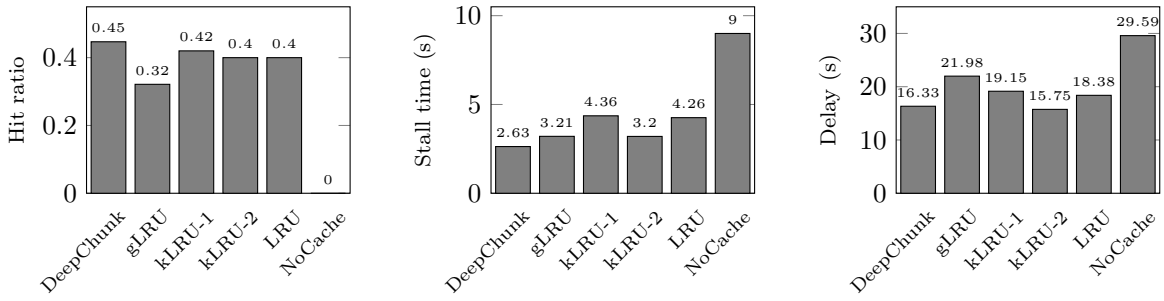
B. Evaluation Results

In this subsection, we compare the proposed DeepChunk policy with the baseline policies stated above. The main results are depicted in Figures 5 and 6.

Figure 5 shows the reward breakdown when popularities and file sizes are positively correlated, and the cache size is set to 80 chunks. From the figure, we observe that our DeepChunk policy outperforms all other policies in terms of both the chunk hit ratio and the processing stall time. As compared to the file-level policies, our improvements are up to 11.71% and 39.70%, respectively. Since the reward weight of response delay is set low, DeepChunk policy has a slightly higher delay than kLRU-2 while being better in the other two metrics. As compared with the other chunk-level policy gLRU, DeepChunk is better for all three reward factors. We note that one of the down-sides of gLRU is that it takes a long time to fill the cache since only one chunk is added for each request. This startup phase is alleviated in other schemes, including ours, which helps provide improved gains as the system evolves or when the popularities of contents change. This phase makes the rewards of the gLRU conservative, while this issue is alleviated with the proposed DeepChunk policy.

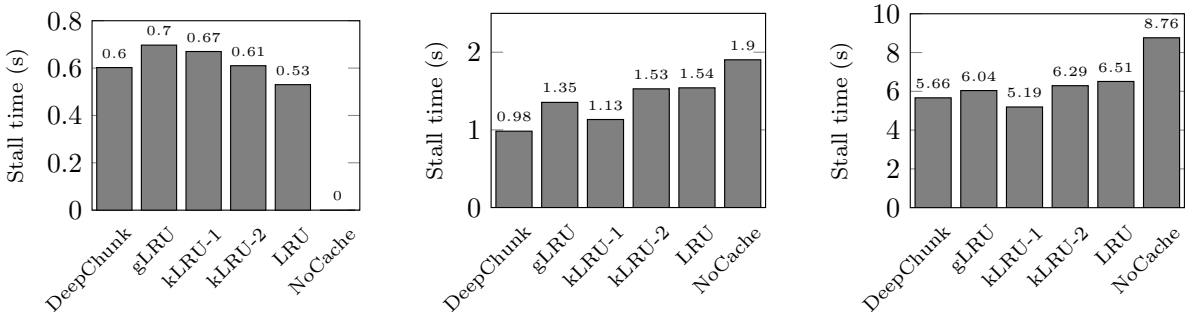
As seen in Table I, DeepChunk policy improves the total reward by 12.77 - 34.83% from other caching algorithms. The chunk-level gLRU policy has 16.16% lower reward than kLRU-2, which indicates that without optimizing the chunk cache decision, chunk-level policies do not always gain a better performance than file-level schemes in a data streaming scenario.

Similar results are concluded from Figure 6. When the file popularity and size are negatively correlated, DeepChunk improves the total reward by up to 43.00%. Note that unlike the previous popularity/cache size configuration, in this case, kLRU-1 has a better performance than kLRU-2. Thus when using the kLRU policy, it is important to optimize the number of virtual caches to be deployed.



(a) Hit ratio (b) Stall time (c) Delay

Fig. 5. Reward breakdown for different policies. Cache capability is 80 chunks, file popularity and size are positively correlated.



(a) Hit ratio (b) Stall time (c) Delay

Fig. 6. Reward breakdown for different policies. Cache capability is 40 chunks, file popularity and size are negatively correlated.

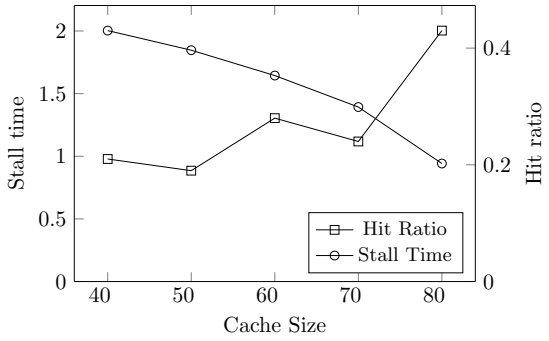


Fig. 7. LRU performance affected by cache size.

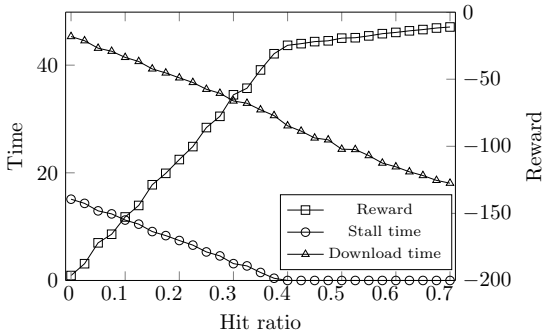


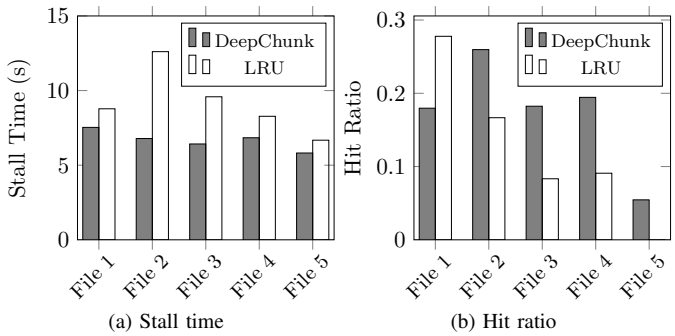
Fig. 8. Reinforcement learning mechanism of DeepChunk decision making.

Intuitively, smaller cache size results in lower rewards. We observed that even with cache size halved, all three reward factors will be better when popularity and file size are negatively correlated. In this experiment, although DeepChunk compromises a lower chunk hit ratio, it has 13.27 - 36.36% advantages to other caching policies on processing stall time, resulting in an improvement of the overall reward.

In Figure 7 we show the improvement obtained by increasing the cache size. Take the LRU policy as an example, both stall time and chunk hit ratio are improved by 50% when the cache size is increased from 40 chunks to 80 chunks. With the increasing cache size, the average stall time improves smoothly, while the chunk hit ratio does not. It is because a slight increase in cache size is not able to make the storage capacity for one additional big file.

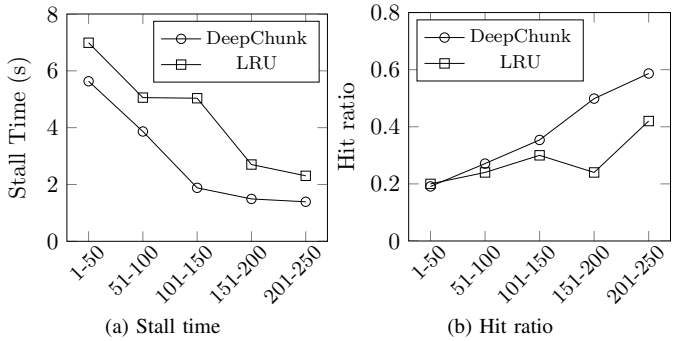
Figure 8 describes the importance of cache chunk decision optimization. We run the DeepChunk policy at the cache node with an empty cache storage, while a file with 40 chunks is repeatedly requested. As the time evolves, the chunk hit ratio of individual requests will grow from 0 to 1 linearly. The x-axis in the figure is marked by the chunk hit ratio of the request. As it grows, the download time decreases linearly. However, with the chunk hit ratio at 0.4 (iteration 10), the stall time hits 0 which is its minimum. Thus at this state, caching more chunks of this file will no longer gain rewards from the stall time, which has a heavy weight. The reward curve shows the same information, the growth slows down after the 10th iteration. Since DeepChunk is a state-aware policy, it tends to cache fewer chunks to save cache space for other files when the stall time can no longer be improved.

We extract the reward variables for individual files while running the experiments. In Figure 9, we compare the average stall time and chunk hit ratio for each file when different caching policies are applied. According to Figure 9 (a), the five files shown experience similar stall time under the DeepChunk policy. The standard deviation is 0.268, that is lower than LRU's standard deviation which is 2.189. As depicted in Figure 9 (b), although chunk hit ratio is not the largest



(a) Stall time (b) Hit ratio

Fig. 9. Reward breakdown for different files. The standard deviations of stall time are 0.628 for CLRU and 2.189 for LRU. The standard deviations of chunk hit ratio are 0.074 for CLRU and 0.104 for LRU.



(a) Stall time (b) Hit ratio

Fig. 10. Reward breakdown comparison for time bins, when the popularity changes for every 50 requests.

weighted term in the reward function, DeepChunk still has a lower standard deviation (0.074) as compared to LRU (0.104). We can conclude that under the DeepChunk policy, more space is saved to reduce the overall stall time among all files.

Finally, we measure the performance of the policies when the popularity distribution suddenly changes. At the user client node, we change the popularity ranking for each of the 50 requests. From Figure 10, we observe that since the DeepCache keeps monitoring the historical probability distribution of the requests in a sliding window, and its performance (both processing stall time and chunk hit ratio) is better than the static LRU algorithm. Thus this verifies that our DeepChunk is more robust and has the ability to adapt to time-varying data popularity in a dynamic environment.

VII. Conclusion

We propose DeepChunk to leverage powerful Q-learning to make chunk-based cache update decisions on the fly in Data Processing Networks, and to jointly optimize both network performance and data processing objectives, including the chunk hit ratio, processing stall time, and object download time. Our prototype using Ceph demonstrates significant improvement, i.e., 43% in total reward and 39% in processing stall time, over a number of baseline caching policies, as well as DeepChunk's ability to adapt to time-varying workload and network conditions. As a future research, we plan to incorporate other metrics such as the Age of Information into DeepChunk and investigate the performance of DeepChunk in a setting with network of caches.

References

[1] M. D. de Assunção, A. D. S. Veith, and R. Buyya, "Distributed data stream processing and edge computing: A survey on resource elasticity and future directions," *J. Network and Computer Applications*, vol. 103, pp. 1–17, 2018.

[2] T. Li, Z. Xu, J. Tang, and Y. Wang, "Model-free control for distributed stream data processing using deep reinforcement learning," *CoRR*, vol. abs/1803.01016, 2018.

[3] A. Elgabli, V. Aggarwal, S. Hao, F. Qian, and S. Sen, "Lbp: Robust rate adaptation algorithm for svc video streaming," *IEEE/ACM Transactions on Networking*, pp. 1–13, 2018.

[4] D. Jiang, B. C. Ooi, L. Shi, and S. Wu, "The performance of mapreduce: An in-depth study," *Proceedings of the VLDB Endowment*, vol. 3.

[5] N. Alliance, "5g white paper," *Next generation mobile networks, white paper*, pp. 1–125, 2015.

[6] Youtube help. https://support.google.com/youtube/answer/1722171.

[7] Y. Kim and I. Yeom, "Performance analysis of in-network caching for content-centric networking," *Comput. Netw.*, vol. 57, no. 13, pp. 2465–2482, Sep. 2013.

[8] D. K. Krishnappa, S. Khemmarat, L. Gao, and M. Zink, "On the feasibility of prefetching and caching for online tv services: a measurement study on hulu," in *International Conference on Passive and Active Network Measurement*. Springer, 2011, pp. 72–80.

[9] D. Shasha and T. Johnson, "2q: A low overhead high performance buffer management replacement algoritm," in *VLDB*, 1994, pp. 439–450.

[10] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[11] Ceph. https://ceph.com/.

[12] B. Hubert *et al.*, "Linux advanced routing & traffic control howto," *Netherlabs BV*, vol. 1, 2002.

[13] H. Che, Y. Tung, and Z. Wang, "Hierarchical web caching systems: Modeling, design and experimental results," *IEEE Journal on Selected Areas in Communications*, vol. 20, no. 7, pp. 1305–1314, 2002.

[14] E. Friedlander and V. Aggarwal, "Generalization of lru cache replacement policy with applications to video streaming," *arXiv preprint arXiv:1806.10853*, 2018.

[15] D. S. Berger, R. K. Sitaraman, and M. Harchol-Balter, "Adaptsize: Orchestrating the hot object memory cache in a content delivery network." in *NSDI*, 2017, pp. 483–498.

[16] H. Ahlehagh and S. Dey, "Hierarchical video caching in wireless cloud: Approaches and algorithms," in *ICC 2012*. IEEE, 2012, pp. 7082–7087.

[17] E. Baştuğ, M. Bennis, and M. Debbah, "Living on the edge: The role of proactive caching in 5g wireless networks," *arXiv preprint arXiv:1405.5974*, 2014.

[18] S. Sivasubramanian, G. Pierre, M. Van Steen, and G. Alonso, "Analysis of caching and replication strategies for web applications," *IEEE Internet Computing*, vol. 11, no. 1, 2007.

[19] Y. Zhao, J. Wu, and C. Liu, "Dache: A data aware caching for big-data applications using the mapreduce framework," *Tsinghua science and technology*, vol. 19, no. 1, pp. 39–50, 2014.

[20] A. Alabbasi, V. Aggarwal, T. Lan, Y. Xiang, M.-R. Ra, and Y.-F. R. Chen, "Fasttrack: Minimizing stalls for cdn-based over-the-top video streaming systems," *arXiv preprint arXiv:1807.01147*, 2018.

[21] Z. Xu, J. Tang, J. Meng, W. Zhang, Y. Wang, C. H. Liu, and D. Yang, "Experience-driven networking: A deep reinforcement learning based approach," *arXiv preprint arXiv:1801.05757*, 2018.

[22] H. Mao, R. Netravali, and M. Alizadeh, "Neural adaptive video streaming with pensieve," in *SIGCOMM*. ACM, 2017, pp. 197–210.

[23] G. Alnwaimi, S. Vahid, and K. Moessner, "Dynamic heterogeneous learning games for opportunistic access in lte-based macro/femtocell deployments," *IEEE Transactions on Wireless Communications*, vol. 14, no. 4, pp. 2294–2308, 2015.

[24] O. Onireti, A. Zoha, J. Moysen, A. Imran, L. Giupponi, M. A. Imran, and A. Abu-Dayya, "A cell outage management framework for dense heterogeneous networks," *IEEE Transactions on Vehicular Technology*, vol. 65, no. 4, pp. 2097–2113, 2016.

[25] R. Fagin, "Asymptotic miss ratios over independent references," *Journal of Computer and System Sciences*, vol. 14, no. 2, pp. 222–250, 1977.

[26] M. Garetto, E. Leonardi, and V. Martina, "A unified approach to the performance analysis of caching systems," *ACM TOMPECS*, vol. 1, no. 3, p. 12, 2016.

[27] Tornado web server. http://www.tornadoweb.org/en/stable/.

[28] L. A. Adamic and B. A. Huberman, "Zipf's law and the internet." *Glottometrics*, vol. 3, no. 1, pp. 143–150, 2002.

[29] A. V. Aho, P. J. Denning, and J. D. Ullman, "Principles of optimal page replacement," *JACM*, vol. 18, no. 1, pp. 80–93, 1971.