An Imitation Learning Approach for Cache Replacement

Evan Zheran Liu [1 2] Milad Hashemi [2] Kevin Swersky [2] Parthasarathy Ranganathan [2] Junwhan Ahn [2]

Abstract

Program execution speed critically depends on increasing cache hits, as cache hits are orders of magnitude faster than misses. To increase cache hits, we focus on the problem of cache replacement: choosing which cache line to evict upon inserting a new line. This is challenging because it requires planning far ahead and currently there is no known practical solution. As a result, current replacement policies typically resort to heuristics designed for specific common access patterns, which fail on more diverse and complex access patterns. In contrast, we propose an imitation learning approach to automatically learn cache access patterns by leveraging Belady's, an oracle policy that computes the optimal eviction decision given the future cache accesses. While directly applying Belady's is infeasible since the future is unknown, we train a policy conditioned only on *past* accesses that accurately approximates Belady's even on diverse and complex access patterns, and call this approach PARROT. When evaluated on 13 of the most memory-intensive SPEC applications, PARROT increases cache miss rates by 20% over the current state of the art. In addition, on a large-scale web search benchmark, PARROT increases cache hit rates by 61% over a conventional LRU policy. We release a Gym environment to facilitate research in this area, as data is plentiful, and further advancements can have significant real-world impact.

1. Introduction

Caching is a universal concept in computer systems that bridges the performance gap between different levels of data storage hierarchies, found everywhere from databases to operating systems to CPUs (Jouppi, 1990; Harty & Cheriton, 1992; Xu et al., 2013; Cidon et al., 2016). Correctly selecting what data is stored in caches is critical for latency, as accessing the data directly from the cache (a *cache hit*) is orders of magnitude faster than retrieving the data from a lower level in the storage hierarchy (a *cache miss*). For example, Cidon et al. (2016) show that improving cache hit rates of web-scale applications by just 1% can decrease total latency by as much as 35%.

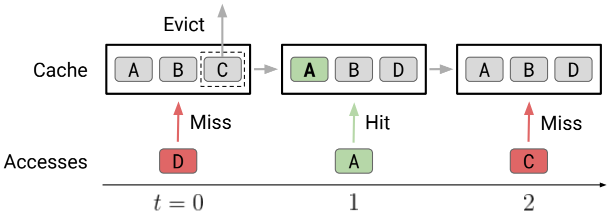

Figure 1. Cache replacement. At $t = 0$, line D is accessed, causing a cache miss. The replacement policy chooses between lines A, B, and C in the cache and in this case evicts C. At $t = 1$, line A is accessed and is already in the cache, causing a cache hit. No action from the replacement policy is needed. At $t = 2$, line C is accessed, causing another cache miss. The replacement policy could have avoided this miss by evicting a different line at $t = 0$.

Thus, general techniques for increasing cache hit rates would significantly improve performance at all levels of the software stack. Broadly, two main avenues for increasing cache hit rates exist: (i) avoiding future cache misses by proactively prefetching the appropriate data into the cache beforehand; and (ii) strategically selecting which data to evict from the cache when making space for new data (cache replacement). Simply increasing cache sizes is a tempting third avenue, but is generally prohibitively expensive.

This work focuses on single-level cache replacement (Figure 1). When a new block of data (referred to as a *line*) is added to the cache (i.e., due to a cache miss), an existing cache line must be evicted from the cache to make space for the new line. To do this, during cache misses, a cache replacement policy takes as inputs the currently accessed line and the lines in the cache and outputs which of the cache lines to evict.

Prior work frequently relies on manually-engineered heuristics to capture the most common cache access patterns, such as evicting the most recently used (MRU) or least recently used (LRU) cache lines, or trying to identify the cache

[1] Department of Computer Science, Stanford University, California, USA [2] Google Research, Sunnyvale, California, USA. Correspondence to: Evan Z. Liu <evanliu@cs.stanford.edu>.

Proceedings of the 37^{th} International Conference on Machine Learning, Online, PMLR 119, 2020. Copyright 2020 by the author(s).

lines that are cache-friendly vs. cache-averse (Qureshi et al., 2007; Jaleel et al., 2010; Jain & Lin, 2016; Shi et al., 2019). These heuristics perform well on the specific simple access patterns they target, but they only target a small fraction of all possible access patterns, and consequently they perform poorly on programs with more diverse and complex access patterns. Current cache replacement policies resort to heuristics as practical theoretical foundations have not yet been developed (Beckmann & Sanchez, 2017).

We propose a new approach for learning cache replacement policies by leveraging Belady's optimal policy (Belady, 1966) in the framework of imitation learning (IL), and name this approach PARROT.[1] Belady's optimal policy (Belady's for short) is an oracle policy that computes the theoretically optimal cache eviction decision based on knowledge of future cache accesses, which we propose to approximate with a policy that only conditions on the *past* accesses. While our main goal is to establish (imitation) learned replacement policies as a proof-of-concept, we note that deploying such learned policies requires solving practical challenges, e.g., model latency may overshadow gains due to better cache replacement. We address some of these challenges in Section 4.5 and highlight promising future directions in Section 7.

Hawkeye (Jain & Lin, 2016) and Glider (Shi et al., 2019) were the first to propose learning from Belady's. They train a binary classifier to predict if a cache line will soon be reused (cache-friendly) or not (cache-averse), evicting the cache-averse lines before the cache-friendly ones and relying on a traditional heuristic to determine which lines are evicted first within the cache-friendly and cache-averse groups. Training such a binary classifier avoids the challenges (e.g., *compounding errors*) of directly learning a policy, but relying on the traditional heuristic heavily limits the expressivity of the policy class that these methods optimize over, which prevents them from accurately approximating Belady's. In contrast, our work is the first to propose cache replacement as an IL problem, which allows us to directly train a replacement policy end-to-end over a much more expressive policy class to approximate Belady's. This represents a novel way of leveraging Belady's and provides a new framework for learning end-to-end replacement policies.

Concretely, this paper makes the following contributions:

- We cast cache replacement as an imitation learning problem, leveraging Belady's in a new way (Section 3).
- We develop a neural architecture for end-to-end cache replacement and several supervised tasks that further improve its performance over standard IL (Section 4).
- Our proposed approach, PARROT, exceeds the state-of-the-art replacement policy's hit rates by over 20% on memory-intensive CPU benchmarks. On an industrial-scale web search workload, PARROT improves cache hit rates by 61% over a commonly implemented LRU policy (Section 5).
- We propose cache replacement as a challenging new IL/RL (reinforcement learning) benchmark involving dynamically changing action spaces, delayed rewards, and significant real-world impact. To that end, we release an associated Gym environment (Section 7).

2. Cache Preliminaries

We begin with cache preliminaries before formulating cache replacement as learning a policy over a Markov decision process in Section 3. We describe the details relevant to CPU caches, which we evaluate our approach on, but as caching is a general concept, our approach can be extended towards other cache structures as well.

A cache is a memory structure that maintains a portion of the data from a larger memory. If the desired data is located in the cache when it is required, this is advantageous, as smaller memories are faster to access than larger memories. Provided a memory structure, there is a question of how to best organize it into a cache. In CPUs, caches operate in terms of atomic blocks of memory or *cache lines* (typically 64-bytes large). This is the minimum granularity of data that can be accessed from the cache.

During a memory access, the cache must be searched for the requested data. *Fully-associative* caches layout all data in a single flat structure, but this is generally prohibitively expensive, as locating the requested data requires searching through all data in the cache. Instead, CPU caches are often W-way *set-associative* caches of size $N \times W$, consisting of N *cache sets*, where each cache set holds W *cache lines* $\{l_1, \ldots, l_W\}$. Each line maps to a particular cache set (typically determined by the lower order bits of line's address), so only the W lines within that set must be searched.

During execution, programs read from and write to *memory addresses* by executing load or store instructions. These load/store instructions have unique identifiers known as *program counters* (PCs). If the address is located in the cache, this is called a *cache hit*. Otherwise, this is a *cache miss*, and the data at that address must be retrieved from a larger memory. Once the data is retrieved, it is generally added to the appropriate cache set (as recently accessed lines could be accessed again). Since each cache set can only hold W lines, if a new line is added to a cache set already containing W lines, the cache replacement policy must choose an existing line to replace. This is called a *cache eviction* and selecting the optimal line to evict is the cache replacement problem.

[1] Parrots are known for their ability to *imitate* others.

Belady's Optimal Policy. Given knowledge of future cache accesses, Belady's computes the *optimal* cache eviction decision. Specifically, at each timestep t, Belady's computes the *reuse distance* $d_t(l_w)$ for each line l_w in the cache set, which is defined as the number of total cache accesses until the next access to l_w. Then, Belady's chooses to evict the line with the highest reuse distance, effectively the line used furthest in the future, i.e., $\arg\max_{w=1,\ldots,W} d_t(l_w)$.

3. Casting Cache Replacement as Imitation Learning

We cast cache replacement as learning a policy on an episodic Markov decision process $\langle \mathcal{S}, \mathcal{A}_s, R, P\rangle$ in order to leverage techniques from imitation learning. Specifically, the state at the t-th timestep $s_t = (s_t^c, s_t^a, s_t^h) \in \mathcal{S}$ consists of three components, where:

- $s_t^a = (m_t, pc_t)$ is the current cache access, consisting of the currently accessed cache line address m_t and the unique program counter pc_t of the access.
- $s_t^c = \{l_1, \ldots, l_W\}$ is the cache state consisting of the W cache line addresses currently in the cache set accessed by s_t^a (the replacement policy does not require the whole cache state including other cache sets to make a decision).[2]
- $s_t^h = (\{m_1, \ldots, m_{t-1}\}, \{pc_1, \ldots, pc_{t-1}\})$ is the history of all past cache accesses. In practice, we effectively only condition on the past H accesses.

The action set $\mathcal{A}_{s_t}$ available at a state $s_t = (s_t^c, s_t^a, s_t^h)$ is defined as follows: During cache misses, i.e., $m_t \notin s_t^c$, the action set $\mathcal{A}_{s_t}$ consists of the integers $\{1, \ldots, W\}$, where action w corresponds to evicting line l_w. Otherwise, during cache hits, the action set $\mathcal{A}_{s_t}$ consists of a single no-op action $a_{\text{no-op}}$, since no line must be evicted.

The transition dynamics $P(s_{t+1} \mid a_t, s_t)$ are given by the dynamics of the three parts of the state. The dynamics of the next cache access s_{t+1}^a and the cache access history s_{t+1}^h are independent of the action a_t and are defined by the program being executed. Specifically, the next access $s_{t+1}^a = (m_{t+1}, pc_{t+1})$ is simply the next memory address the program accesses and its associated PC. The t-th access is appended to s_{t+1}^h, i.e., $s_{t+1}^h = (\{m_1, \ldots, m_{t-1}, m_t\}, \{pc_1, \ldots, pc_{t-1}, pc_t\})$.

The dynamics of the cache state are determined by the actions taken by the replacement policy. At state s_t with $s_t^c = \{l_1, \ldots, l_W\}$ and $s_t^a = (m_t, pc_t)$: A cache hit does not change the cache state, i.e., $s_{t+1}^c = s_t^c$, as the accessed line is already available in the cache. A cache miss replaces the selected line with the newly accessed line, i.e., $s_{t+1}^c = \{l_1, \ldots, l_{w-1}, l_{w+1}, \ldots, l_W, m_t\}$ where $a_t = w$.

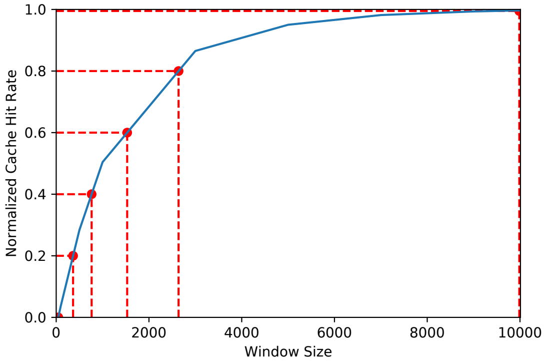

Figure 2. Normalized cache hit rates of Belady's vs. the number of accesses it looks into the future. Achieving 80% the performance of Belady's with an infinite window size requires accurately computing reuse distances for lines 2600 accesses into the future.

The reward $R(s_t)$ is 0 for a cache miss (i.e., $m_t \notin s_t^c$) and is 1 otherwise for a cache hit. The goal is to learn a policy $\pi_\theta(a_t \mid s_t)$ that maximizes the undiscounted total number of cache hits (the reward), $\sum_{t=0}^{T} R(s_t)$, for a sequence of T cache accesses $(m_1, pc_1), \ldots, (m_T, pc_T)$.

In this paper, we formulate this task as an imitation learning problem. During training, we can compute the optimal policy (Belady's) $\pi^*(a_t \mid s_t, (m_{t+1}, pc_{t+1}), \ldots, (m_T, pc_T))$, by leveraging that the future accesses are fixed. Then, our approach learns a policy $\pi_\theta(a_t \mid s_t)$ to approximate the optimal policy without using the future accesses, as future accesses are unknown during test time.

To demonstrate the difficulty of the problem, similar to the figure from Jain & Lin (2016), Figure 2 shows the amount of future information required to match the performance of Belady's on a common computer architecture benchmark (*omnetpp*, Section 5). We compute this by imposing a future window of size x on Belady's, which we call Belady$_x$, Within the window (reuse distances $\leq x$), Belady$_x$ observes exact reuse distances, and sets the reuse distances of the remaining cache lines (with reuse distance $> x$) to ∞. Then, Belady$_x$ evicts the line with the highest reuse distance, breaking ties randomly. The cache hit rate of Belady$_x$ is plotted on the y-axis, normalized so that 0 and 1 correspond to the cache hit rate of LRU and Belady$_\infty$ (the normal unconstrained version of Belady's), respectively. As the figure shows, a significant amount of future information is required to fully match Belady's performance.

4. Parrot: Learning to Imitate Belady's

4.1. Model and Training Overview

Model. Below, we overview the basic architecture of the Parrot policy $\pi_\theta(a_t \mid s_t)$ (Figure 3), which draws on the

[2] A cache set can have less than W cache lines for the first $W - 1$ cache accesses (small fraction of program execution). In this case, no eviction is needed to insert the line.

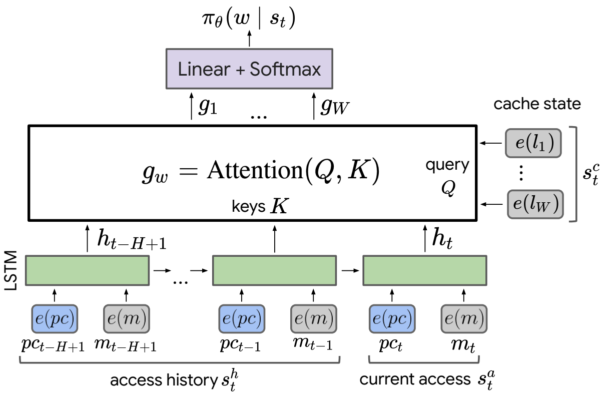

Figure 3. Neural architecture of PARROT.

Algorithm 1 PARROT training algorithm

```
1:  Initialize policy $\pi_\theta$
2:  for step $= 0$ to $K$ do
3:      if step $\equiv 0 \pmod{5000}$ then
4:          Collect data set of visited states $B = \{s_t\}_{t=0}^T$ by
            following $\pi_\theta$ on all accesses $(m_1, pc_1), \ldots, (m_T, pc_T)$
5:      end if
6:      Sample contiguous accesses $\{s_t\}_{t=l-H}^{l+H}$ from $B$
7:      Warm up policy $\pi_\theta$ on initial $H$ accesses
        $(m_{l-H}, pc_{l-H}), \ldots, (m_l, pc_l)$
8:      Compute loss $\mathcal{L} = \sum_{t=l}^{l+H} \mathcal{L}_\theta(s_t, \pi^*)$
9:      Update policy parameters $\theta$ based on loss $\mathcal{L}$
10: end for
```

Transformer (Vaswani et al., 2017) and BiDAF (Seo et al., 2016) architectures. See Appendix A for the full details.

1. Embed the current cache access $s_t^a = (m_t, pc_t)$ to obtain memory address embedding $e(m_t)$ and PC embedding $e(pc_t)$ and pass them through an LSTM to obtain cell state c_t and hidden state h_t:
$$c_t, h_t = \text{LSTM}([e(m_t); e(pc_t)], c_{t-1}, h_{t-1})$$
2. Keep the past H hidden states, $[h_{t-H+1}, \ldots, h_t]$, representing an embedding of the cache access history s_t^h and current cache access s_t^a.
3. Form a context g_w for each cache line l_w in the cache state s_t^c by embedding each line as $e(l_w)$ and attending over the past H hidden states with $e(l_w)$:
$$g_w = \text{Attention}(Q, K)$$
$$\text{where query } Q = e(l_w), \text{keys } K = [h_{t-H+1}, \ldots, h_t]$$
4. Apply a final dense layer and softmax on top of these line contexts to obtain the policy:
$$\pi_\theta(a_t = w \mid s_t) = \text{softmax}(\text{dense}(g_w))$$
5. Choose $\arg\max_{a \in \mathcal{A}_{s_t}} \pi_\theta(a \mid s_t)$ as the replacement action to take at timestep t.

Training. Algorithm 1 summarizes the training algorithm for the PARROT policy π_θ. The high-level strategy is to visit a set of states B and then update the parameters θ to make the same eviction decision as the optimal policy π^* on each state $s \in B$ via the loss function $\mathcal{L}_\theta(s, \pi^*)$.

First, we convert a given sequence of consecutive cache accesses $(m_1, pc_1), \ldots, (m_T, pc_T)$ into states $s_0, \ldots, s_T$ (Section 4.2), on which we can compute the optimal action with Belady's (lines 3–5). Given the states, we train PARROT with truncated backpropagation through time (lines 6–9). We sample batches of consecutive states $s_{l-H}, \ldots, s_{l+H}$ and initialize the LSTM hidden state of our policy on the cache accesses of s_{l-H} to s_{l-1}. Then, we apply our replacement policy π_θ to each of the remaining states $s_l, \ldots, s_{l+H-1}$ in order to compute the loss $\mathcal{L}_\theta(s_t, \pi^*)$ (Sections 4.3 and 4.4), which encourages the learned replacement policy to make the same decisions as Belady's.

4.2. Avoiding Compounding Errors

Since we are only given the cache accesses and not the states, we must determine which replacement policy to follow on these cache accesses to obtain the states B. Naively, one natural policy to follow is the optimal policy π^*. However, this leads to *compounding errors* (Ross et al., 2011; Daumé et al., 2009; Bengio et al., 2015), where the distribution of states seen during test time (when following the learned policy) differs from the distribution of states seen during training (when following the oracle policy). At test time, since PARROT learns an imperfect approximation of the oracle policy, it will eventually make a mistake and evict a suboptimal cache line. This leads to cache states that are different from those seen during training, which the learned policy has not trained on, leading to further mistakes.

To address this problem, we leverage the DAgger algorithm (Ross et al., 2011). DAgger avoids compounding errors by also following the current learned policy π_θ instead of the oracle policy π^* to collect B during training, which forces the distribution of training states to match that of test states. As PARROT updates the policy, the current policy becomes increasingly different from the policy used to collect B, causing the training state distribution B to drift from the test state distribution. To mitigate this, we periodically update B every 5000 parameter updates by recollecting B again under the current policy. Based on the recommendation in (Ross et al., 2011), we follow the oracle policy the first time we collect B, since at that point, the policy π_θ is still random and likely to make poor eviction decisions.

Notably, this approach is possible because we can compute our oracle policy (Belady's) at any state during training, as long as the future accesses are known. This differs from many IL tasks (Hosu & Rebedea, 2016; Vecerik et al., 2017), where querying the expert is expensive and limited.

4.3. Ranking Loss

Once the states B are collected, we update our policy π_θ to better approximate Belady's π^* on these states via the loss function $\mathcal{L}_\theta(s, \pi^*)$. A simple log-likelihood (LL) behavior cloning loss (Pomerleau, 1989) $\mathcal{L}_\theta(s, \pi^*) = \log \pi_\theta(\pi^*(s) \mid s)$ encourages the learned policy to place probability mass on the optimal action $\pi^*(s)$. However, in the setting where the *distribution* $\pi^*(a \mid s)$ is known, instead of just the optimal action $\pi^*(s)$, optimizing to match this distribution can provide more supervision, similar to the intuition of distillation (Hinton et al., 2015). Thus, we propose an alternate ranking loss to leverage this additional supervision.

Concretely, PARROT uses a differentiable approximation (Qin et al., 2010) of normalized discounted cumulative gain (NDCG) with reuse distance as the relevancy metric:

$$\mathcal{L}_\theta^{\text{rank}}(s_t, \pi^*) = -\frac{\text{DCG}}{\text{IDCG}}$$

$$\text{where DCG} = \sum_{w=1}^{W} \frac{d_t(l_w) - 1}{\log(\text{pos}(l_w) + 1)}$$

$$\text{pos}(l_w) = \sum_{i \neq w} \sigma(-\alpha(\pi_\theta(i \mid s_t) - \pi_\theta(w \mid s_t)).$$

Here, $\text{pos}(l_w)$ is a differentiable approximation of the rank of line l_w, ranked by how much probability the policy π_θ places on evicting l_w, where $\alpha = 10$ is a hyperparameter and σ is the sigmoid function. IDCG is a normalization constant set so that $-1 \leq \mathcal{L}_\theta^{\text{rank}} \leq 0$, equal to the value of DCG when the policy π_θ correctly places probability mass on the lines in descending order of reuse distance. This loss function improves cache hit rates by heavily penalizing π_θ for placing probability on lines with low reuse distance, which will likely lead to cache misses, and only lightly penalizing π_θ for placing probability on lines with higher reuse distance, which are closer to being optimal and are less likely to lead to cache misses.

Optimizing our loss function is similar to optimizing the Kullback-Liebler (KL) divergence (Kullback & Leibler, 1951) between a smoothed version of Belady's, which evicts line l_w with probability proportional to its exponentiated reuse distance $e^{d_t(l_w)}$, and our policy π_θ. Directly optimizing the KL between the non-smoothed oracle policy and our policy just recovers the normal LL loss, since Belady's actually places all of its probability on a single line.

4.4. Predicting Reuse Distance

To add further supervision during training, we propose to predict the reuse distances of each cache line as an auxiliary task (Jaderberg et al., 2016; Mirowski et al., 2016; Lample & Chaplot, 2017). Concretely, we add a second fully-connected head on PARROT's network that takes as inputs the per-line context embeddings g_w and outputs predictions of the log-reuse distance $\hat{d}(g_w)$. We train this head with a mean-squared error loss $\mathcal{L}_\theta^{\text{reuse}}(s, \pi^*) = \frac{1}{W} \sum_{w=1}^{W} (\hat{d}(g_w) - \log d_t(l_w))^2$. Intuitively, since the reuse distance predicting head shares the same body as the policy head π_θ, learning to predict reuse distances helps learn better representations in the rest of the network. Overall, we train our policy with loss $\mathcal{L}_\theta(s, \pi^*) = \mathcal{L}_\theta^{\text{rank}}(s, \pi^*) + \mathcal{L}_\theta^{\text{reuse}}(s, \pi^*)$.

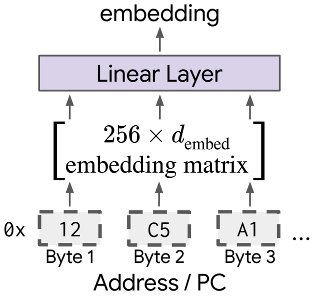

Figure 4. Byte embedder, taking only a few kilobytes of memory.

4.5. Towards Practicality

The goal of this work is to establish directly imitating Belady's as a proof-of-concept. Applying approaches like PARROT to real-world systems requires reducing model size and latency to prevent overshadowing improved cache replacement. We leave these challenges to future work, but highlight one way to reduce model size in this section, and discuss further promising directions in Section 7.

In the full-sized PARROT model, we learn a separate embedding for each PC and memory address, akin to word vectors (Mikolov et al., 2013) in natural language processing. While this approach performs well, these embeddings can require tens of megabytes to store for real-world programs that access hundreds of thousands of unique memory addresses.

To reduce model size, we propose learning a byte embedder shared across all memory addresses, only requiring several kilobytes of storage. This byte embedder embeds each memory address (or PC) by embedding each byte separately and then passing a small linear layer over their concatenated outputs (Figure 4). In principle, this can learn a hierarchical representation, that separately represents large memory regions (upper bytes of an address) and finer-grained objects (lower bytes).

5. Experiments

5.1. Experimental Setup

Following Shi et al. (2019), we evaluate our approach on a three-level cache hierarchy with a 4-way 32 KB L1 cache, a 8-way 256 KB L2 cache, and a 16-way 2 MB last-level cache. We apply our approach to the last-level cache while using the LRU replacement policy for L1/L2 caches.

For benchmark workloads, we evaluate on the memory-intensive SPEC CPU2006 (Henning, 2006) applications used by Shi et al. (2019). In addition, we evaluate on Google Web Search, an industrial-scale application that serves billions of queries per day, to further evaluate the effectiveness of PARROT on real-world applications with complex access patterns and large working sets.

For each of these programs, we run them and collect raw memory access traces over a 50 second interval using dynamic binary instrumentation tools (Bruening et al., 2003). This produces the sequence of all memory accesses that the program makes during that interval. Last-level cache access traces are obtained from this sequence by passing the raw memory accesses through the L1 and L2 caches using an LRU replacement policy.

As this produces a large amount of data, we then sample the resultant trace for our training data (Qureshi et al., 2007). We randomly choose 64 sets and collect the accesses to those sets on the last-level cache, totaling an average of about 5M accesses per program. Concretely, this yields a sequence of accesses $(m_1, pc_1), ..., (m_T, pc_T)$. We train replacement policies on the first 80% of this sequence, validate on the next 10%, and report test results on the final 10%.

Our evaluation focuses on two key metrics representing the efficiency of cache replacement policies. First, as increasing cache hit rates is highly correlated to decreasing program latency (Qureshi et al., 2007; Shi et al., 2019; Jain & Lin, 2016), we evaluate our policies using raw cache hit rates. Second, we report *normalized cache hit rates*, representing the gap between LRU (the most common replacement policy) and Belady's (the optimal replacement policy). For a policy with hit rate r, we define the normalized cache hit rate as $\frac{(r - r_{\text{LRU}})}{(r_{\text{opt}} - r_{\text{LRU}})}$, where r_{LRU} and r_{opt} are the hit rates of LRU and Belady's, respectively. The normalized hit rate represents the effectiveness of a given policy with respect to the two baselines, LRU (normalized hit rate of 0) and Belady's (normalized hit rate of 1).

We compare the following four approaches:

1. PARROT: trained with the full-sized model, learning a separate embedding for each PC and address.
2. PARROT (byte): trained with the much smaller byte embedder (Section 4.5).
3. Glider (Shi et al., 2019): the state-of-the-art cache replacement policy, based on the results reported in their paper.
4. Nearest Neighbor: a nearest neighbors version of Belady's, which finds the longest matching PC and memory address suffix in the training data and follows the Belady's decision of that.

The SPEC2006 program accesses we evaluate on may slightly differ from those used by Shi et al. (2019) in evaluating Glider, as the latter is not publicly available. However, to ensure a fair comparison, we verified that the measured hit rates for LRU and Belady's on our cache accesses are close to the numbers reported by Shi et al. (2019), and we only compare on normalized cache hit rates. Since Glider's hit rates are not available on Web Search, we compare PARROT against LRU, the policy frequently used in production CPU caches. The reported hit rates for PARROT, LRU, Belady's, and Nearest Neighbors are measured on the test sets. We apply early stopping on PARROT, based on the cache hit rate on the validation set. For PARROT, we report results averaged over 3 random seeds, using the same minimally-tuned hyperparameters in all domains. These hyperparameters were tuned exclusively on the validation set of omnetpp (full details in Appendix B).

5.2. Main Results

Table 1 compares the raw cache hit rate of PARROT with that of Belady's and LRU. PARROT achieves significantly higher cache hit rates than LRU on every program, ranging from 2% to 30%. Averaged over all programs, PARROT achieves 16% higher cache hit rates than LRU. According to prior study on cache sensitivity of SPEC2006 workloads (Jaleel, 2010), achieving the same level of cache hit rates as PARROT with LRU would require increasing the cache capacity by 2–3x (e.g., omnetpp and mcf) to 16x (e.g., libquantum).

On the Web Search benchmark, PARROT achieves a 61% higher normalized cache hit rate and 13.5% higher raw cache hit rate than LRU, demonstrating PARROT's practical ability to scale to the complex memory access patterns found in datacenter-scale workloads.

Figure 5 compares the normalized cache hit rates of PARROT and Glider. With the full-sized model, PARROT outperforms Glider on 10 of the 13 SPEC2006 programs, achieving a 20% higher normalized cache hit rate averaged over all programs; on the remaining 3 programs (bzip, bwaves, and mcf), Glider performs marginally better. Additionally, PARROT achieves consistent performance with low variance across seeds.

Reducing model size. Though learning PARROT from scratch with the byte embedder does not perform as well as the full-sized model, the byte embedder model is significantly smaller and still achieves an average of 8% higher normalized cache hit rate than Glider (Figure 5). In Section 7, we highlight promising future directions to reduce the performance gap and further reduce model size and latency.

Generalization. An effective cache replacement policy must be able to generalize to unseen *code paths* (i.e., sequences of accesses) from the same program, as there are exponentially many code paths and encountering them all

Table 1. Raw cache hit rates. Optimal is the hit rate of Belady's. Averaged over all programs, PARROT (3 seeds) outperforms LRU by 16%.

	astar	bwaves	bzip	cactusadm	gems	lbm	leslie3d	libq	mcf	milc	omnetpp	sphinx3	xalanc	Web Search
Optimal	43.5%	8.7%	78.4%	38.8%	26.5%	31.3%	31.9%	5.8%	46.8%	2.4%	45.1%	38.2%	33.3%	67.5%
LRU	20.0%	4.5%	56.1%	7.4%	9.9%	0.0%	12.7%	0.0%	25.3%	0.1%	26.1%	9.5%	6.6%	45.5%
PARROT	34.4%	7.8%	64.5%	38.6%	26.0%	30.8%	31.7%	5.4%	41.4%	2.1%	41.4%	36.7%	30.4%	59.0%

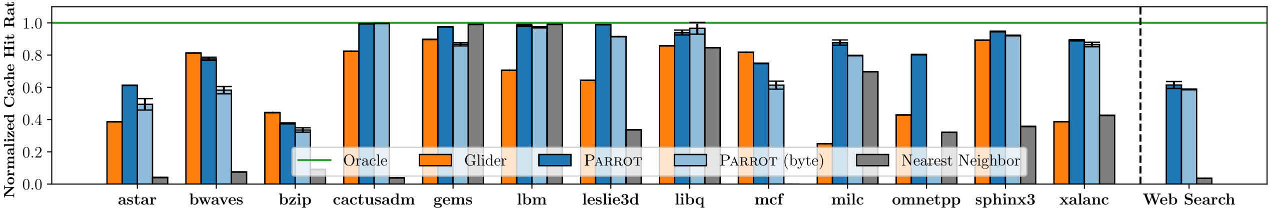

Figure 5. Comparison of PARROT with the state-of-the-art replacement policy, Glider. We evaluate two versions of PARROT, the full-sized model (PARROT) and the byte embedder model (PARROT (byte)), and report the mean performance over 3 seeds with 1-standard deviation error bars. On the SPEC2006 programs (left), PARROT with the full-sized model improves hit rates over Glider by 20% on average.

during training is infeasible. We test PARROT's ability to generalize to new code paths by comparing it to the nearest neighbors baseline (Figure 5). The performance of the nearest neighbors baseline shows that merely memorizing training code paths seen achieves near-optimal cache hit rates on simpler programs (e.g., gems, lbm), which just repeatedly execute the same code paths, but fails for more complex programs (e.g., mcf, Web Search), which exhibit highly varied code paths. In contrast, PARROT maintains high cache hit rates even on these more complex programs, showing that it can generalize to new code paths not seen during training.

Additionally, some of the programs require generalizing to new memory addresses and program counters at test time. In mcf, 21.6% of the test-time memory addresses did not appear in the training data, and in Web Search, 5.3% of the test-time memory addresses and 6% of the test-time PCs did not appear in the training data (full details in Appendix B), but PARROT performs well despite this.

5.3. Ablations

Below, we ablate each of the following from PARROT: predicting reuse distance, on-policy training (DAgger), and ranking loss. We evaluate on four of the most memory-intensive SPEC2006 applications (lbm, libq, mcf, and omnetpp) and Web Search and compare each ablation with Glider, Belady's, and two versions of PARROT. PARROT is the full-sized model with no ablations. PARROT (base) is PARROT's neural architecture, with all three additions ablated. Comparing PARROT (base) to Glider (e.g., Figure 6) shows that in some programs (e.g., omnetpp and lbm), simply casting cache replacement as an IL problem with PARROT's neural architecture is sufficient to obtain competitive performance, while in other programs, our additions are required to achieve state-of-the-art cache hit rates.

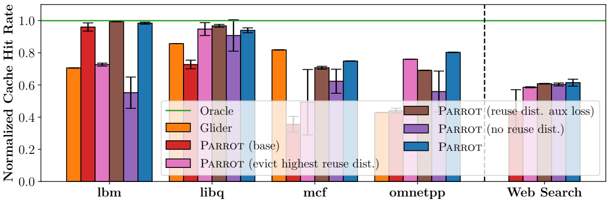

Figure 6. Comparison between different mechanisms of incorporating reuse distance into PARROT. Including reuse distance prediction in our full model (PARROT) achieves 16.8% higher normalized cache hit rates than ablating reuse distance prediction (PARROT (no reuse dist.)).

Predicting Reuse Distance. Figure 6 compares the following three configurations to show the effect of incorporating reuse distance information: (i) PARROT (no reuse dist.), where reuse distance prediction is ablated, (ii) PARROT (evict highest reuse dist.), where our fully ablated model (PARROT (base)) predicts reuse distance and directly evicts the line with the highest predicted reuse distance, and (iii) PARROT (reuse dist. aux loss), where our fully ablated model learns to predict reuse distance as an auxiliary task.

Comparing PARROT (no reuse dist.) to PARROT shows that incorporating reuse distance greatly improves cache hit rates. Between different ways to incorporate reuse distance into PARROT, using reuse distance prediction indirectly as an auxiliary loss function (PARROT (reuse dist. aux loss)) leads to higher cache hit rates than using the reuse distance predictor directly to choose which cache line to evict (PARROT (evict highest reuse dist.)). We hypothesize that in some cache states, accurately predicting the reuse distance for each line may be challenging, but ranking the lines may be relatively easy. Since our reuse distance predictor predicts log reuse distances, small errors may drastically affect which line is evicted when the reuse distance predictor is used directly.

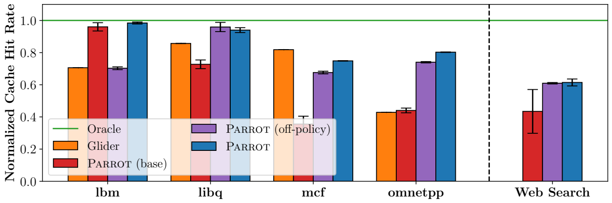

Figure 7. Ablation study for training with DAgger. Training with DAgger achieves 9.8% higher normalized cache hit rates than training off-policy on the states visited by the oracle policy.

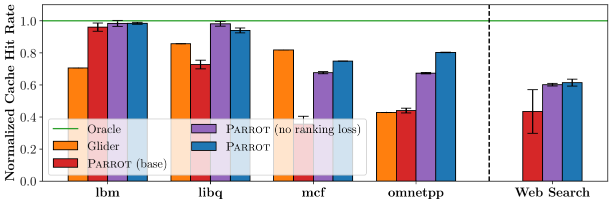

Figure 8. Ablation study for our ranking loss. Using our ranking loss improves normalized cache hit rate by 3.5% over a LL loss.

Training with DAgger. Figure 7 summarizes the results when ablating training on-policy with DAgger. In theory, training off-policy on roll-outs of Belady's should lead to compounding errors, as the states visited during training under Belady's differ from those visited during test time. Empirically, we observe that this is highly program-dependent. In some programs, like mcf or Web Search, training off-policy performs as well or better than training on-policy, but in other programs, training on-policy is crucial. Overall, training on-policy leads to an average 9.8% normalized cache hit rate improvement over off-policy training.

Ranking Loss. Figure 8 summarizes the results when ablating our ranking loss. Using our ranking loss over a log-likelihood (LL) loss introduces some bias, as the true optimal policy places all its probability on the line with the highest reuse distance. However, our ranking loss better optimizes cache hit rates, as it more heavily penalizes evicting lines with lower reuse distances, which lead to misses. In addition, a distillation perspective of our loss, where the teacher network is an exponentially-smoothed version of Belady's with the probability of evicting a line set as proportional to $\exp(\text{reuse distance})$, suggests that our ranking loss provides greater supervision than LL. Tuning a temperature on the exponential smoothing of Belady's could interpolate between less bias and greater supervision. Empirically, we observe that our ranking loss leads to an average 3.5% normalized cache hit rate improvement over LL.

5.4. History Length

One key question is: how much past information is needed to accurately approximate Belady's? We study this by varying the number of past accesses that PARROT attends over

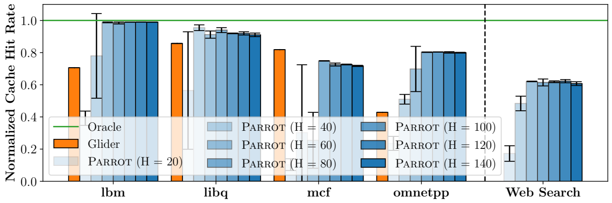

Figure 9. Performance of PARROT trained with different numbers of past accesses (H). As the number of past accesses increases, normalized cache hit rates improve, until reaching a history length of 80. At that point, additional past accesses have little impact.

(H) from 20 to 140. In theory, PARROT's LSTM hidden state could contain information about *all* past accesses, but the LSTM's memory is limited in practice.

The results are summarized in Figure 9. We observe that the past accesses become an increasingly better predictor of the future as the number past accesses increase, until about 80. After that point, more past information doesn't appear to help approximate Belady's. Interestingly, Shi et al. (2019) show that Glider experiences a similar saturation in improvement from additional past accesses, but at around 30 past accesses. This suggests that learning a replacement policy end-to-end with PARROT can effectively leverage more past information than simply predicting whether a cache line is cache-friendly or cache-averse.

6. Related Work

Cache Replacement. Traditional approaches to cache replacement rely on heuristics built upon intuition for cache access behavior. LRU is based on the assumption that most recently used lines are more likely to be reused. More sophisticated policies target a handful of manually classified access patterns based on simple counters (Qureshi et al., 2007; Jaleel et al., 2010) or try to predict instructions that tend to load zero-reuse lines based on a table of saturating counters (Wu et al., 2011; Khan et al., 2010).

Several recent approaches instead focus on *learning* cache replacement policies. Wang et al. (2019) also cast cache replacement as learning over a Markov decision process, but apply reinforcement learning instead of imitation learning, which results in lower performance. More closely related to ours are Hawkeye (Jain & Lin, 2016) and Glider (Shi et al., 2019), which also learn from Belady's. They train a binary classification model based on Belady's to predict if a line is cache-friendly or cache-averse, but rely on a traditional replacement heuristic to determine which line to evict when several lines are cache-averse. Relying on the traditional heuristic to produce the final eviction decisions heavily constrains the expressivity of the policy class they learn over, so that even the best policy within their class of learnable policies may not accurately approximate Belady's, yielding high cache miss rates for some access patterns.

In contrast, our work is the first to propose learning a cache replacement policy end-to-end with imitation learning. Framing cache replacement in this principled framework is important as much prior research has resorted to heuristics for hill climbing specific benchmarks. In addition, learning end-to-end enables us to optimize over a highly expressive policy class, achieving high cache hit rates even on complex and diverse access patterns.

Imitation Learning. Our work builds on imitation learning (IL) techniques (Ross & Bagnell, 2014; Sun et al., 2017), where the goal is to approximate an expert policy. Our setting exhibits two distinctive properties: First, in our setting, the expert policy (Belady's) can be queried at any state during training. the oracle policy (Belady's) can be cheaply queried at any state during training, which differs from a body of IL work (Vecerik et al., 2017; Hosu & Rebedea, 2016; Hester et al., 2018) focusing on learning with limited samples from an expensive expert (e.g., a human). The ability to arbitrarily query the oracle enables us to avoid compounding errors with DAgger (Ross et al., 2011). Second, the distribution over actions of the oracle policy is available, enabling more sophisticated loss functions. Prior work (Sabour et al., 2018; Choudhury et al., 2017) also studies settings with these two properties, although in different domains. Sabour et al. (2018) shows that an approximate oracle can be computed in some natural-language sequence generation tasks; Choudhury et al. (2017) learns to imitate an oracle computed from data only available during training, similar to Belady's, which requires future information.

7. Conclusion and Future Directions

We develop a foundation for learning end-to-end cache replacement policies with imitation learning, which significantly bridges the gap between prior work and Belady's optimal replacement policy. Although we evaluate our approach on CPU caches, due to the popularity of SPEC2006 as a caching benchmark, we emphasize that our approach applies to other caches as well, such as software caches, databases, and operating systems. Software caches may be an especially promising area for applying our approach, as they tolerate higher latency in the replacement policy and implementing more complex replacement policies is easier in software. We highlight two promising future directions:

First, this work focuses on the ML challenges of training a replacement to approximate Belady's and does not explore the practicality of deploying the learned policy in production, where the two primary concerns are the memory and latency overheads of the policy. To address these concerns, future work could investigate model-size reduction techniques, such as distillation (Hinton et al., 2015), pruning (Janowsky, 1989; Han et al., 2015; Sze et al., 2017), and quantization, as well as domains tolerating greater latency and memory use, such as software caches. Additionally, cache replacement decisions can be made at any time between misses to the same set, which provides a reasonably long latency window (e.g., on the order of seconds for software caches) for our replacement policy to make a decision. Furthermore, the overall goal of cache replacement is to minimize latency. While minimizing cache misses minimizes latency to a first approximation, cache misses incur variable amounts of latency (Qureshi et al., 2006), which could be addressed by fine-tuning learned policies to directly minimize latency via reinforcement learning.

Second, while Belady's algorithm provides an optimal replacement policy for a single-level cache, there is no known optimal policy for multiple levels of caches (as is common in CPUs and web services). This *hierarchical cache replacement* policy is a ripe area for deep learning and RL research, as is exploring the connection between cache replacement and prefetching, as they both involve selecting the optimal set of lines to be present in the cache. Cache replacement is backward looking (based on the accesses so far) while prefetching is forward looking (predicting future accesses directly (Hashemi et al., 2018; Shi et al., 2020)).

To facilitate further research in this area, we release a Gym environment for cache replacement, which easily extends to the hierarchical cache replacement setting, where RL is required as the optimal policy is unknown. We find cache replacement an attractive problem for the RL/IL communities, as it has significant real-world impact and data is highly available, in contrast to many current benchmarks that only have one of these two properties. In addition, cache replacement features several interesting challenges: rewards are highly delayed, as evicting a particular line may not lead to a cache hit/miss until thousands of timesteps later; the semantics of the action space dynamically changes, as the replacement policy chooses between differing cache lines at different states; the state space is large (e.g., 100,000s of unique addresses) and some programs require generalizing to new memory addresses at test time, not seen during training, similar to the rare words problem (Luong et al., 2014) in NLP; and as our ablations show, different programs exhibit wildly different cache access patterns, which can require different techniques to address. In general, we observe that computer systems exhibit many interesting machine learning (ML) problems, but have been relatively inaccessible to the ML community because they require sophisticated systems tools. We take steps to avoid this by releasing our cache replacement environment.

Reproducibility. Code for PARROT and our cache replacement Gym environment is available at `https://github.com/google-research/google-research/tree/master/cache_replacement`.

Acknowledgements

We thank Zhan Shi for insightful discussions and for providing the results for Glider, which we compare against. We also thank Chelsea Finn, Lisa Lee, and Amir Yazdanbakhsh for their comments on a draft of this paper. Finally, we thank the anonymous ICML reviewers for their useful feedback, which helped improve this paper.

This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. DGE-1656518.

References

Beckmann, N. and Sanchez, D. Maximizing cache performance under uncertainty. In *Proceedings of the International Symposium on High Performance Computer Architecture*, 2017.

Belady, L. A. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 1966.

Bengio, S., Vinyals, O., Jaitly, N., and Shazeer, N. Scheduled sampling for sequence prediction with recurrent neural networks. In *Advances in Neural Information Processing Systems*, pp. 1171–1179, 2015.

Bruening, D., Garnett, T., and Amarasinghe, S. An infrastructure for adaptive dynamic optimization. In *Proceedings of the International Symposium on Code Generation and Optimization*, 2003.

Choudhury, S., Kapoor, A., Ranade, G., Scherer, S., and Dey, D. Adaptive information gathering via imitation learning. *arXiv preprint arXiv:1705.07834*, 2017.

Cidon, A., Eisenman, A., Alizadeh, M., and Katti, S. Cliffhanger: Scaling performance cliffs in web memory caches. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, 2016.

Daumé, H., Langford, J., and Marcu, D. Search-based structured prediction. *Machine Learning*, 75(3):297–325, 2009.

Glorot, X. and Bengio, Y. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pp. 249–256, 2010.

Han, S., Mao, H., and Dally, W. J. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

Harty, K. and Cheriton, D. R. Application-controlled physical memory using external page-cache management. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 1992.

Hashemi, M., Swersky, K., Smith, J. A., Ayers, G., Litz, H., Chang, J., Kozyrakis, C., and Ranganathan, P. Learning memory access patterns. In *Proceedings of the International Conference on Machine Learning*, 2018.

Henning, J. L. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

Hester, T., Vecerik, M., Pietquin, O., Lanctot, M., Schaul, T., Piot, B., Horgan, D., Quan, J., Sendonaris, A., Osband, I., et al. Deep Q-learning from demonstrations. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2018.

Hinton, G., Vinyals, O., and Dean, J. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

Hosu, I.-A. and Rebedea, T. Playing Atari games with deep reinforcement learning and human checkpoint replay. *arXiv preprint arXiv:1607.05077*, 2016.

Jaderberg, M., Mnih, V., Czarnecki, W. M., Schaul, T., Leibo, J. Z., Silver, D., and Kavukcuoglu, K. Reinforcement learning with unsupervised auxiliary tasks. *arXiv preprint arXiv:1611.05397*, 2016.

Jain, A. and Lin, C. Back to the future: Leveraging Belady's algorithm for improved cache replacement. In *Proceedings of the International Symposium on Computer Architecture*, 2016.

Jaleel, A. Memory characterization of workloads using instrumentation-driven simulation. *Web Copy: http://www.glue.umd.edu/ajaleel/workload*, 2010.

Jaleel, A., Theobald, K. B., Steely, S. C., and Emer, J. High performance cache replacement using re-reference interval prediction (RRIP). In *Proceedings of the International Symposium on Computer Architecture*, 2010.

Janowsky, S. A. Pruning versus clipping in neural networks. *Physical Review A*, 39(12):6600, 1989.

Jouppi, N. P. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. *ACM SIGARCH Computer Architecture News*, 1990.

Khan, S. M., Tian, Y., and Jimenez, D. A. Sampling dead block prediction for last-level caches. In *Proceedings of the International Symposium on Microarchitecture*, 2010.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Kullback, S. and Leibler, R. A. On information and sufficiency. *The Annals of Mathematical Statistics*, 22(1):79–86, 1951.

Lample, G. and Chaplot, D. S. Playing FPS games with deep reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2017.

Luong, M.-T., Sutskever, I., Le, Q. V., Vinyals, O., and Zaremba, W. Addressing the rare word problem in neural machine translation. *arXiv preprint arXiv:1410.8206*, 2014.

Luong, M.-T., Pham, H., and Manning, C. D. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*, 2015.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., and Dean, J. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pp. 3111–3119, 2013.

Mirowski, P., Pascanu, R., Viola, F., Soyer, H., Ballard, A. J., Banino, A., Denil, M., Goroshin, R., Sifre, L., Kavukcuoglu, K., et al. Learning to navigate in complex environments. *arXiv preprint arXiv:1611.03673*, 2016.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, pp. 8024–8035, 2019.

Pomerleau, D. A. Alvinn: An autonomous land vehicle in a neural network. In *Advances in neural information processing systems*, pp. 305–313, 1989.

Qin, T., Liu, T.-Y., and Li, H. A general approximation framework for direct optimization of information retrieval measures. *Information Retrieval*, 13(4):375–397, 2010.

Qureshi, M. K., Lynch, D. N., Mutlu, O., and Patt, Y. N. A case for mlp-aware cache replacement. In *33rd International Symposium on Computer Architecture (ISCA'06)*, pp. 167–178. IEEE, 2006.

Qureshi, M. K., Jaleel, A., Patt, Y. N., Steely, S. C., and Emer, J. Adaptive insertion policies for high performance caching. In *Proceedings of the International Symposium on Computer Architecture*, 2007.

Ross, S. and Bagnell, J. A. Reinforcement and imitation learning via interactive no-regret learning. *arXiv preprint arXiv:1406.5979*, 2014.

Ross, S., Gordon, G., and Bagnell, D. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the International Conference on Artificial Intelligence and Statistics*, 2011.

Sabour, S., Chan, W., and Norouzi, M. Optimal completion distillation for sequence learning. *arXiv preprint arXiv:1810.01398*, 2018.

Seo, M., Kembhavi, A., Farhadi, A., and Hajishirzi, H. Bidirectional attention flow for machine comprehension. *arXiv preprint arXiv:1611.01603*, 2016.

Shi, Z., Huang, X., Jain, A., and Lin, C. Applying deep learning to the cache replacement problem. In *Proceedings of the International Symposium on Microarchitecture*, 2019.

Shi, Z., Swersky, K., Tarlow, D., Ranganathan, P., and Hashemi, M. Learning execution through neural code fusion. In *Proceedings of the International Conference on Learning Representations*, 2020.

Sun, W., Venkatraman, A., Gordon, G. J., Boots, B., and Bagnell, J. A. Deeply aggrevated: Differentiable imitation learning for sequential prediction. In *Proceedings of the International Conference on Machine Learning*, pp. 3309–3318, 2017.

Sze, V., Chen, Y.-H., Yang, T.-J., and Emer, J. S. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12):2295–2329, 2017.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. In *Advances in Neural Information Processing Systems*, 2017.

Vecerik, M., Hester, T., Scholz, J., Wang, F., Pietquin, O., Piot, B., Heess, N., Rothörl, T., Lampe, T., and Riedmiller, M. Leveraging demonstrations for deep reinforcement learning on robotics problems with sparse rewards. *arXiv preprint arXiv:1707.08817*, 2017.

Wang, H., He, H., Alizadeh, M., and Mao, H. Learning caching policies with subsampling. 2019.

Wu, C.-J., Jaleel, A., Hasenplaugh, W., Martonosi, M., Steely Jr, S. C., and Emer, J. Ship: Signature-based hit predictor for high performance caching. In *Proceedings of the International Symposium on Microarchitecture*, 2011.

Xu, Y., Frachtenberg, E., Jiang, S., and Paleczny, M. Characterizing Facebook's memcached workload. *IEEE Internet Computing*, 18(2):41–49, 2013.