Harvesting Randomness to Optimize Distributed Systems

Mathias Lecuyer[†], Joshua Lockerman[‡], Lamont Nelson[§], Siddhartha Sen[*], Amit Sharma[*], Aleksandrs Slivkins[*]
[*]Microsoft Research, [†]Columbia University, [§]New York University, [‡]Yale University

ABSTRACT

We view randomization through the lens of statistical machine learning: as a powerful resource for offline optimization. Cloud systems make randomized decisions all the time (*e.g.,* in load balancing), yet this randomness is rarely used for optimization after-the-fact. By casting system decisions in the framework of reinforcement learning, we show how to collect data from existing systems, *without modifying them*, to evaluate new policies, *without deploying them*. Our methodology, called *harvesting randomness*, has the potential to accurately estimate a policy's performance without the risk or cost of deploying it on live traffic. We quantify this optimization power and apply it to a real machine health scenario in Azure Compute. We also apply it to two prototyped scenarios, for load balancing (Nginx) and caching (Redis), with much less success, and use them to identify the systems and machine learning challenges to achieving our goal.

Our long-term agenda is to harvest the randomness in distributed systems to develop non-invasive and efficient techniques for optimizing them. Like CPU cycles and bandwidth, we view randomness as a valuable resource being wasted by the cloud, and we seek to remedy this.

1 INTRODUCTION

Cloud infrastructure systems make complex decisions everyday that choose among a set of actions based on some contextual information. For example, a datacenter controller chooses how long to wait for an unresponsive machine; an in-memory cache chooses which items to evict when space runs low; a load balancer chooses which backend server to route a request to. (See Table 1.) For each decision, a *policy* is used to choose an *action* given the *context* surrounding the decision, with the goal of optimizing some *reward* metric. Since the optimal policy is often unknown and may change, new policies are constantly being devised, tested, and deployed.

To mitigate the cost of deploying a bad policy, staged rollouts are used to expose the policy to increasing fractions of live traffic. If the infrastructure is available, an A/B test can be run to compare the new policy against the old one in a statistically sound manner [8, 37]. Indeed, it is now common practice to employ in-house (*e.g.,* Bing's EXP [14]) or commercial experimentation platforms (*e.g.,* Optimizely [30]) to amortize the cost of deployment, such as for Internet website optimization [15, 16]. Since the policy is exposed to live traffic, a nontrivial amount of development, test, and management effort is spent on each A/B test. But even if we completely ignore these costs, the fact remains that only 100% of traffic is available to share among all A/B tests. The more that are run concurrently, the longer each will take to achieve statistical significance, as shown in Fig. 1. Experiments also need to run long enough to rule out inherent daily or weekly variations; *e.g.,* in Bing a typical experiment lasts two weeks [14]. Thus even on the most advanced infrastructure, it is impractical to run more than a few hundred experiments at a time.

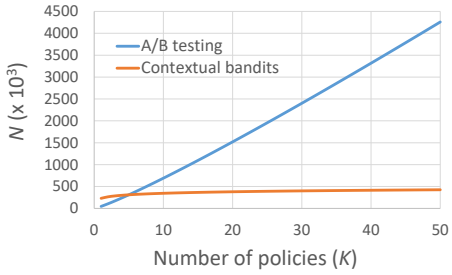

Figure 1: The amount of data (N) required to simultaneously evaluate K policies, using typical constants (see §4). Contextual bandits is exponentially more efficient than A/B testing, and can evaluate policies offline.

What if, instead, we could evaluate a policy *offline*, with the same guarantees *as if* we had run it in an online A/B test? Such a counterfactual methodology would be extremely powerful, because it would allow us to evaluate arbitrary policies without the cost of making them production-ready, or the risk of deploying them on live traffic. We could for example optimize over a large class of policies, *e.g.,* billions, to find the one with best performance. As it turns out, this problem is well-studied in reinforcement learning (RL) as the *off-policy evaluation* problem [38], or how to use data collected from a deployed policy to evaluate a different candidate policy. A necessary condition for off-policy evaluation is that the deployed policy makes *randomized decisions*: that is, given a context, the policy chooses each eligible action with some probability. In our experience with online content recommendation [1], adding randomization to an existing (non-randomized) product has been the main source of uncertainty, complications, and delay.

This is where the beauty of systems comes in: many systems *already make randomized decisions* in the form expected by RL, such as in load balancing, replica placement, cache eviction, etc.. Billions of these decisions are made every day

HotNets-XVI, November 30–December 1, 2017, Palo Alto, CA, USA

ACM ISBN 978-1-4503-5569-8/17/11…$15.00
https://doi.org/10.1145/3152434.3152435

	Machine health (Azure Compute)	Caching (Redis)	Load balancing (Nginx)
Decision	wait time before reboot	item to evict	server to route request to
Context	machine hardware/OS, failure history,...	per-item access history, size,...	request type, server load,...
Actions	time to wait	items currently in cache	backend servers
Reward [+/−]	[−]total downtime (scaled by # of VMs)	[+]hitrate	[−]99th percentile latency
Actions (CB)	minutes in {1,2,...,9}	subsample of items	clusters/service endpoints
Reward (CB)	(same)	[+]time to next access of evicted item	[−]request latency

Table 1: Example applications of RL in systems decisions. Rewards may be minimized [−] or maximized [+]. Actions and rewards can be reformulated to fit the framework of contextual bandits (CB), allowing efficient off-policy evaluation.

by our cloud infrastructure, and tucked away in system logs. We argue that this is a huge waste of a valuable resource: randomness. Our vision is to develop a *non-invasive* methodology for harvesting this randomness to enable *efficient* off-policy evaluation. By non-invasive, we mean that since systems already make randomized decisions, we should, in theory, be able to scavenge the data we need from logs they already collect. By efficient, we mean that our evaluation techniques should scale to a large number of policies, run quickly, and yield candidates that are deployable in practice.

Unfortunately, off-policy evaluation in RL is notoriously difficult and inefficient. Our line of attack is to start with a more tractable subset of RL called contextual bandits (CB) and show that many systems decisions can be recast in this framework (§2). CB supports efficient off-policy evaluation [7]: we quantify this efficiency and show that CB can evaluate exponentially more policies than A/B testing given the same amount of data, and does so *offline* (Fig. 1, §4). We apply our methodology successfully to machine health monitoring in Azure Compute, using data scavenged from their system logs. However, we find that the assumptions for CB are too strong for many systems settings, such as load balancing and caching, resulting in offline estimates that are inaccurate and misleading (§5). We plan to address these challenges by drawing on techniques from RL, or developing our own.

Our focus on off-policy evaluation is a departure from prior work applying RL to systems, which focuses almost exclusively on policy training. Being able to train a good policy does not imply the ability to evaluate it offline; often, the only way to accurately estimate its performance is to deploy it, or use complex, application-specific modeling [5, 34, 41]. Moreover, most proposed solutions are invasive: they interpose randomization and a continuous learning loop in the system in order to produce a good policy [1, 10, 21, 26].

We make the following contributions:

(1) We identify a natural framework for CB and off-policy evaluation in distributed systems.
(2) We develop a methodology for harvesting existing randomness without intervening in a live system, and quantify its optimization power.
(3) We successfully apply our methodology to machine health monitoring in Azure Compute. We use our failures in applying it to load balancing (Nginx) and caching (Redis) to identify systems and machine learning challenges.

We end with a vision for how we might achieve our goals given the challenges we face (§7).

2 BACKGROUND AND GOALS

This sections provides background on Reinforcement Learning (RL) and Contextual Bandits (CB), with examples of how we can cast systems decisions in this framework. We then give an overview of off-policy evaluation techniques.

Reinforcement learning. In *reinforcement learning* [38], an agent learns by interacting with the world as follows:

(1) The state of the world or *context* x is observed.
(2) An *action* $a \in A$ is taken (the set A may depend on x).
(3) The *reward* r for a is obtained.

A *policy* maps each context to an action (in step 2 above). The goal is to maximize the cumulative reward over a sequence of such interactions. Many system decisions match this setting; Table 1 shows real examples in machine health monitoring, load balancing, and caching. A distinct property of RL is that only *partial feedback* is observed for the action that was taken; nothing is learned for actions that were not taken. For instance, in the machine health example, we do not know what would have happened if we waited longer to reboot a machine. In contrast, supervised learning receives *full feedback*: given a context (*e.g.,* an image), the correct label (*e.g.,* dog) is always known.

To cope with partial feedback, RL algorithms balance *exploration*, or the use of randomization to experience new actions, with *exploitation* of knowledge gained so far. An RL policy makes *randomized decisions*: given a context, each eligible action is chosen with some nonzero probability. An interaction in RL thus generates a tuple $\langle x, a, r, p \rangle$, where p is the probability with which the policy chose a; we call these tuples *exploration data*. RL is most effective when each action gets adequate coverage, which favors small action spaces.

Table 1 shows examples of systems decisions that fall in the framework of RL. For example, in load balancing, the decision is which server to route a request to, based on context that includes server metrics (*e.g.,* load, CPU usage), to maximize the reward of (negative) 99th percentile latency.

Contextual bandits. *Contextual bandits* (CB) [2, 18] is a subset of RL that assumes interactions are independent of each other: one decision does not affect the context or reward observed by another decision. More formally, CB assumes:

A1. Contexts are independent and identically distributed (i.i.d).
A2. The reward given a (context, action) pair is i.i.d.

This is an important simplification. By assuming that current decisions do not impact future states, we know they do not impact future rewards: the reward for an action is simply

the one directly observed. This makes off-policy evaluation easier, because each action can be considered independently instead of as part of a sequence. Table 1 recasts our systems decisions in the CB framework by using short-term proxies for long-term rewards, *e.g.,* each request's latency instead of 99th percentile latency, and smaller action spaces.

Off-policy evaluation. *Off-policy evaluation* [38] uses exploration data collected from a deployed policy to evaluate a new candidate policy offline[1]. Such counterfactual reasoning is extremely powerful because it allows us to ask "what if" questions without the risk or cost of deploying a policy. In supervised learning, off-policy evaluation is trivial because we have full feedback on all action choices. In RL, we only obtain partial feedback, so the data must be randomized in order to avoid the biases of the deployed policy [4, 36]. Note that "randomized" here does not mean `rand()` has to be called for each decision: it is sufficient for the action choices to be independent of the context [17]. For example, a hash-based load balancing policy can be viewed as "random" if the context does not include the inputs to the hash.

There are three approaches to off-policy evaluation in RL: model-based approaches model the system workings and evaluate a policy against this model [23, 31]; function approximation methods directly approximate the long-term value of a policy [39]; importance sampling approaches use probabilistic weighting to correct the mismatch between the deployed policy and candidate policy's choices [33]. The first two approaches make assumptions about the real world and thus tend to be biased. The third approach is unbiased but tends to have high variance, especially if decisions impact future rewards over a long horizon [22]. Hybrid approaches exist [13].

The independence assumptions of CB address some of these issues and enable efficient off-policy evaluation [7, 19, 20]. We discuss an importance sampling method in §4.

3 HARVESTING RANDOMNESS

We describe a simple methodology for harvesting randomness systems to enable off-policy evaluation. The idea is to collect $\langle x, a, r, p\rangle$ exploration datapoints from a production system without intervening in it, as follows:

(1) **Scavenge** logs from an existing (live) system and extract the $\langle x, a, r\rangle$ information for each request.
(2) **Infer** the probability p of each decision using code inspection or analysis of the scavenged $\langle x, a, r\rangle$ data.
(3) **Evaluate/optimize** a policy offline using $\langle x, a, r, p\rangle$ data.

Our experience with production systems has shown that existing logging is adequate for recording the context surrounding a decision (x), the decision itself (a), and the reward (r). As with all machine learning, some amount of feature engineering is required to convert contextual information scavenged in step 1 into usable features. In our experience, p can often be inferred from code inspection, but a more robust approach is to do a regression on the $\langle x, a, r\rangle$ data to learn the probability distribution over actions. The feasibility of step 3 depends on the application setting; if it is CB, for example, then we can optimize a policy very data efficiently.

The above methodology may find a good policy without intervention, but deploying it, of course, does require intervention. Further, we may want to repeat steps 1-3 to continuously optimize the system. Frameworks like the Decision Service [1, 25] and NEXT [10] ease this deployment process, which is not the focus of this paper. Our goal is to apply this methodology to various systems to find good policies in the first place. Then, we can focus our deployment efforts on those systems for which step 3 predicts the highest gains.

We start with the applications in Table 1. For machine health, we obtain demonstrable gains which are detailed in §4. For caching and load balancing, off-policy evaluation brings more challenges, which we discuss in §5.

Machine health. We used real logs collected by Azure Compute to evaluate the machine health scenario. Azure Compute already logs detailed hardware/configuration information about each machine as well as context on past failures; neither is fast-changing. Per-machine downtimes (the reward) are also logged carefully as they directly impact customer SLAs. At the time of our data collection, Azure was using a safe default policy of waiting the maximal amount of time (10 min.) before rebooting, which actually gives us full feedback on what would have happened if we waited {1,2,...,9} min., similar to a supervised learning dataset! Thus, we can use this data to both optimize a CB policy—by simulating randomized data and applying off-policy evaluation—as well as obtain the ground truth performance (using supervised learning). Our results in §4 have convinced the Azure Compute team to deploy our CB policies in production [1].

Caching. We used the Redis key-value cache [35] to evaluate the caching scenario. Redis already samples items uniformly at random when making eviction decisions, and supports a variety of eviction policies (random, LRU, etc.). This makes it a good candidate for harvesting randomness. Redis maintains per-item contextual information (*e.g.,* last accessed time) but does not log it by default, so we added custom logging for this purpose. Determining the next time an evicted item is accessed (the reward) would require a more invasive change, since Redis does not maintain state for evicted items. Instead, we reconstruct this information during step 1 by looking ahead in the logs to when the item next appears. To obtain the ground truth performance of a policy, we deploy and measure it in our prototype. Caching violates some of the assumptions of CB, making off-policy evaluation challenging (§5).

Load balancing. We used Nginx [28] to evaluate the load balancing scenario. Nginx supports various load balancing policies (random, least loaded, etc.), several of which may be viewed as randomized (see §2), making it a good candidate for harvesting randomness. It can also be customized with modules, and many are provided by default. For example, we were able to use existing logging modules to log the context

[1] "Offline" does not mean "batch": off-policy evaluation may incrementally update; it just does not intervene in a live (online) system.

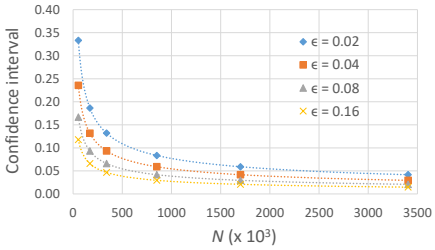

Figure 2: Theoretical accuracy of policy evaluation (Eq. 1) over a space of 10^6 policies (typical C, $\delta = 0.01$).

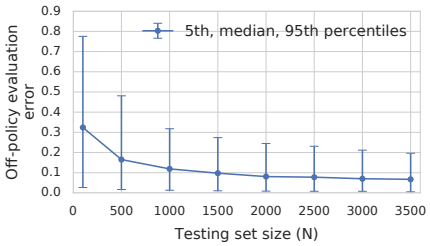

Figure 3: Off-policy evaluation error on a CB policy from machine health scenario, relative to full feedback.

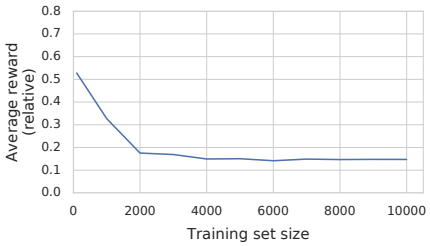

Figure 4: Convergence of CB training on the machine health data, relative to a full feedback model.

(*e.g.,* active connections per server) and reward (request latency) information; many other variables can be logged [29]. Similar to Redis, we obtain ground truth performance by deploying the policy in our prototype. Load balancing also violates some of the assumptions of CB (§5).

4 OFF-POLICY EVALUATION

We now describe basic off-policy evaluation in CB, and use it to apply our methodology to the machine health scenario.

Unlike A/B testing, which randomizes over policies, CB randomizes over *actions*. A single datapoint can then be used to evaluate any policy that would have chosen the same action. Specifically, given N exploration datapoints $\langle x_t, a_t, r_t, p_t \rangle$ collected from a deployed policy, we can evaluate any policy π by considering the datapoints where π's choice matches the logged action a_t. The simplest approach is to use *inverse propensity scoring* (`ips`) [9] to estimate π's average reward:

$$\texttt{ips}(\pi) = \frac{1}{N} \sum_{t=1}^{N} 1_{\{\pi(x_t)=a_t\}} \, r_t/p_t,$$

where $1_{\{\}}$ has value 1 when π's action matches the exploration data and 0 otherwise. By importance weighting each datapoint by the probability p_t, we obtain an *unbiased* estimate of π's performance, *i.e.,* it converges to the true average reward as $N \to \infty$. Intuitively, this weighting avoids penalizing (rewarding) π for bad (good) choices made by the deployed policy which π did not make. Note that the estimate is defined only if $p_t > 0$, or all actions are explored.

In `ips`, each interaction on which π matches the exploration data can be used to evaluate π; in contrast, A/B testing only uses data collected using π to evaluate π (so it must actually run π online). The ability to reuse data offline makes this approach exponentially more data-efficient than A/B testing, in the following sense. Suppose we wish to evaluate K different policies. Let ϵ be the minimum probability given to each action in the exploration data, and assume all rewards lie in $[0, 1]$. Then, with probability $1 - \delta$ the `ips` estimator yields a confidence interval of size:

$$\sqrt{\frac{C}{\epsilon N} \log \frac{K}{\delta}} \tag{1}$$

for all K policies simultaneously, where C is a small constant [1]. The error scales logarithmically in the number of policies. In contrast, with A/B testing the error could be as large as $C\sqrt{\frac{K}{N} \log \frac{K}{\delta}}$. Since the number of actions is much smaller than K, it follows that $\frac{1}{\epsilon} \ll K$, making A/B testing *exponentially worse*. Fig. 1 confirms this. The ability to evaluate any policy allows us to optimize over an entire class of policies Π to find the best one[2], with accuracy given by Eq. 1 (set $K = |\Pi|$). Typically Π is defined by a tunable template, such as decision trees, neural nets, or linear vectors.

By relating the number of decisions N made by a system to Eq. 1, we obtain a concrete measure of the wasted optimization potential in that system. Suppose we wish to find the best policy in a class of size $|\Pi| = 10^6$. Fig. 2 plots the theoretical accuracy of evaluating all candidates[2], for different values of ϵ and representative constants C, $\delta = 0.05$. For example, the $\epsilon = 0.04$ curve could represent an Azure edge proxy that load balances Bing maps requests over 25 clusters ($1/25 = 0.04$). Since rewards lie in $[0, 1]$, an error much smaller than 1 is desired, *e.g.,* < 0.05. A few insights are immediate:

- A minimum N points are required to overcome the competing parameters in Eq. 1. Beyond this point there are diminishing returns. For example, increasing N from 1.7 to 3.4 million improves accuracy by less than 0.01.
- A higher ϵ (more exploration) reduces the data required substantially. For example, doubling ϵ from 0.02 to 0.04 halves the data required in the ϵN term. This favors decisions over smaller action spaces.

In order to measure the practical performance of off-policy evaluation and optimization, we use real data collected from the machine health scenario in Azure Compute (Table 1). As mentioned earlier, this dataset has full feedback, allowing us to simulate exploration in a partial feedback setting—by only revealing the reward of a randomly chosen action, and hiding all others—while also providing ground truth performance.

Fig. 3 shows the error (relative to ground truth) of the `ips` estimator on a trained policy's performance, as measured on a testing dataset of growing size. The error bars show the 5th and 95th percentiles of the estimated value, computed from one thousand partial information simulations; the top of the error bar thus represents $\delta = 0.05$. The estimator's error follows the theoretical trend of Fig. 2. With only 3500 points, the error is below 20% with median error at 8%: this is already enough to conclude with high confidence that the learned policy outperforms the default used during data collection.

[2]This is done by an efficient search [7], not by evaluating every candidate.

Off-policy evaluation also enables us to optimize, or learn, a good policy, as shown in Fig. 4. Using a CB algorithm for policy optimization, and simulating 10,000 exploration datapoints from the dataset, we learn a policy that obtains an average reward (on a testing set) within 15% of a policy trained using supervised learning on the full feedback dataset. The CB algorithm converges very quickly, getting within 20% using only 2000 points. Although the full feedback model performs better, it is an idealized baseline that cannot be deployed long-term: as soon as we integrate it into the system, new interactions would only provide partial feedback.

5 TECHNICAL CHALLENGES

The machine health scenario fit well in the CB framework, enabling very data efficient off-policy evaluation and optimization. Other systems scenarios can raise significant challenges, however, as we discuss next.

Violations of independence. Recall from §2 that CB assumes that contexts (A1) and rewards (A2) are i.i.d.. A2 is violated, for example, when the workload or environment changes. Like prior work [1], we can address this by using incremental learning algorithms that continuously update the policy (*i.e.,* repeating steps 1-3 of our methodology).

A1 is more problematic. It requires that the distribution of contexts is not impacted by prior decisions, but this is routinely violated in many systems. For example in our load balancing scenario, the load of each endpoint is a useful metric to include in the context. However, prior routing decisions clearly influence these loads and thus change the context distribution. This completely breaks off-policy evaluation, as the following example shows. Consider a load balancer that routes requests randomly to two servers. Each server's latency is a linear function of the number of open connections, and server 2 is slower than server 1 by an additive constant, as shown on Fig. 5. We used Nginx to collect exploration data from such a system and used the `ips` estimator to evaluate different policies. (Recall that `ips` does not account for a policy's long-term impact on contexts.) Table 2 shows the off-policy estimates of the policies, compared to their true performance in an online deployment. Since in the collected data server 1 is always faster, evaluating a policy that always sends to server 1 yields good results! But if this policy is deployed, it will overload server 1 and perform abysmally.

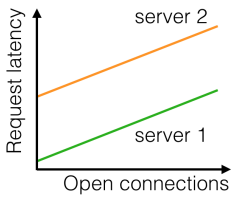

Figure 5: Setup.

To address this challenge, we plan to use off-policy estimators that account for long-term effects [40]. Intuitively, these estimators reweigh the data based on the probability of matching sequences of actions rather than single actions. Since the probability of matching long sequences is very low, these estimators suffer from high variance. We envision leveraging doubly robust techniques [13], which use modeling to predict rewards, to reduce this variance. The difficulty will be in devising models that meet our goals of being simple and flexible enough to work in a variety of systems settings, and being efficiently learnable from logged data.

Policy	Off-policy evaluation	Online evaluation
Random	0.44s	0.44s
Least loaded	0.36s	0.38s
Send to 1	0.31s	0.70s
CB policy	0.32s	0.35s

Table 2: Mean request latency of different load balancing policies (Nginx). Off-policy evaluation breaks for a policy that only sends to one server. CB optimization yields a policy that outperforms least loaded.

Policy	**Random**	**LRU**	**LFU**	**CB policy**	**Freq/size**
Hit rate	48.5%	48.2%	44.0%	48.7%	58.9%

Table 3: Hitrates of different cache eviction policies on a big/small item workload (Redis). The only policy that beats random eviction explicitly considers item size.

Finally, Table 2 shows that despite the ineffectiveness of policy evaluation in the load balancing scenario, CB is still able to optimize (learn) a good policy from the exploration data and outperform least loaded. This is because the CB algorithm learns a good estimator of each server's latency based on context, and greedily picking the lowest latency yields a good policy. The benefit of CB would increase with more request-specific context (*e.g.,* URI, arguments, cookies), as the algorithm would learn how different types of requests are processed by different servers, something least loaded cannot do. Overall, these results show that policy optimization can be much easier than policy evaluation in some settings.

Long-term rewards. Not all settings are amenable to training good policies, however. Another property of many systems decisions is that they have a long-term impact on future rewards. This is true in the caching scenario, for example. To demonstrate this, we collected exploration data from a Redis server configured with a random eviction policy, using a workload consisting of a few frequently-queried large items and many less-frequently-queried small items. The large items are queried twice as frequently but are four times as big: it is thus more efficient to cache the small items.

Table 3 shows the performance of different eviction policies, including one learned by a CB algorithm. Both the CB policy and LRU perform as poorly as random eviction, because they greedily keep the large items (expecting them to be queried again soon) without considering the opportunity cost of using more space. Indeed, a policy manually designed to take size into account (by optimizing the ratio of access frequency to size) has a hitrate 10 percentage points higher. This shows that failing to capture long-term effects can lead to bad optimization.

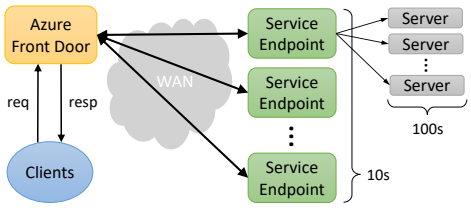

Figure 6: Hierarchical architecture of Azure Front Door.

One avenue to address this challenge is to build long-term effects into the optimization process. However, common techniques for doing this involve extensive modeling, system-specific simulators, or slow decision times. We plan to start with CB algorithms and minimally incorporate long-term techniques, to leverage them without giving up the efficiency of CB's off-policy evaluation.

Exploration coverage. Accounting for a policy's long-term impact seems necessary in many systems, but it also introduces an exploration challenge. Existing randomized heuristics often make independent decisions for each action, which may not provide coverage for longer-term effects. For instance, a uniform random load balancing policy will almost never choose the same server twenty times in a row. We will thus lack data to evaluate the long-term impact of a policy that always sends to one server. We plan to investigate two systems approaches to tackle this challenge.

First, we can adapt current heuristics slightly so that they yield richer exploration data. While this violates our goal of being non-invasive, it is certainly less invasive than deploying a new learning system. For example, instead of randomizing each request, a load balancer could randomize the share of traffic sent to each server during the next N requests. In Nginx, this is easily implemented by randomizing the weights assigned to each server.

Second, reliability testing in distributed systems can trigger uneven traffic and extreme conditions that lead to broader exploration. As an example, we could leverage Netflix's open-source Chaos Monkey [27], a system that triggers failures (*e.g.,* VM crash, high network latency) in production data centers. Such randomized failures, and the systems' responses, would generate valuable exploration data.

Hierarchy and large action spaces. Large action spaces are another impediment to good exploration. Choosing between many actions reduces the coverage of each action, increasing the amount of data needed for off-policy evaluation. Fortunately, hierarchical designs can help. For instance, Azure's edge proxy (Front Door) load balances over tens of service endpoints, while standard load balancers distribute requests within the local clusters (Fig. 6). This reduces the action space at each level, allowing us to apply our methodology to both levels if desired.

Data collection and distributed state. Reducing the action space also reduces the amount of context that needs to be logged. In caching for instance, it is impractical to log the context of every cached item during an eviction decision. We can reduce the action space and data collection by considering only a random subsample of the items. This is already how eviction works in Redis, for example.

Another data collection challenge is that state may be distributed or unavailable at the time of decision. For instance, Nginx and Azure Front Door may know the load of each endpoint because all requests are routed back through them, but they do not know the CPU or RAM usage of the endpoints. Collecting this data will inevitably result in stale or incomplete contexts. We suspect that CB algorithms can naturally tolerate staleness. If not, we might assist the learner by explicitly modeling staleness, or by using advanced networking solutions like RDMA to read remote contexts faster.

It seems unlikely that we can avoid modifying the logging of current systems altogether (step 1 of our methodology). However, the changes we have made in our example scenarios have been simple and minimal, and well worth the future optimization potential in our view. For instance, in Redis we required information about evicted items that was not retained, but most of it was discernable from the logs. In Nginx, existing logging modules already provided what we needed, and simply needed to be configured.

6 RELATED WORK

Our focus on off-policy evaluation is a significant departure from prior work applying RL to systems. Many such applications require complex, application-specific modeling or simulations [5, 34, 41], which are subject to bias if the model of the world is wrong. Other applications do not use a model, but rely on continuous interactions with the environment (*i.e.,* invasive deployments) to learn a good policy [3, 6, 10–12, 24, 26, 32]. The only prior work supporting off-policy evaluation is restricted to CB techniques, and we use it as a building block for our settings [1]. Moreover, many of these techniques leverage deep neural networks or search based policies, which are too slow for the kinds of systems decisions we are optimizing, such as caching and load balancing [3, 24, 26].

7 CLOSING: WASTED RANDOMNESS

We have laid out a methodology for harvesting randomness in systems infrastructure that allows us to evaluate policies without ever deploying them. In many cases, this can be done without intervention in the live system (*e.g.,* Azure Compute), or at most minor additional logging (*e.g.,* Redis, Nginx). Our experience with these applications suggests that opportunities for optimization may come in more forms than we anticipated. For example, the machine health scenario had no randomization, yet provided full information. The load balancing scenario provided perfect randomization, yet our off-policy evaluation techniques did not work.

Existing randomness in systems is being wasted at an alarming rate. We believe that harvesting it for systems optimization is an important endeavor that systems and machine learning researchers should jointly embark on.

REFERENCES

[1] Alekh Agarwal, Sarah Bird, Markus Cozowicz, Luong Hoang, John Langford, Stephen Lee, Jiaji Li, Dan Melamed, Gal Oshri, Oswaldo Ribas, Siddhartha Sen, and Alex Slivkins. 2017. The Power of Offline Evaluation in Online Decision Making. *CoRR* abs/1606.03966v2.

[2] Alekh Agarwal, Daniel Hsu, Satyen Kale, John Langford, Lihong Li, and Robert Schapire. 2014. Taming the Monster: A Fast and Simple Algorithm for Contextual Bandits. In *31st Intl. Conf. on Machine Learning (ICML)*.

[3] Enda Barrett, Enda Howley, and Jim Duggan. 2013. Applying reinforcement learning towards automating resource allocation and application scalability in the cloud. *Concurrency and Computation: Practice and Experience* (2013).

[4] Léon Bottou, Jonas Peters, Joaquin Quiñonero Candela, Denis Xavier Charles, Max Chickering, Elon Portugaly, Dipankar Ray, Patrice Y. Simard, and Ed Snelson. 2013. Counterfactual reasoning and learning systems: the example of computational advertising. *J. Mach. Learn. Res. (JMLR)* (2013).

[5] Xiangping Bu, Jia Rao, and Cheng-Zhong Xu. 2009. A reinforcement learning approach to online web systems auto-configuration. In *International Conference on Distributed Computing Systems (ICDCS)*.

[6] Xiangping Bu, Jia Rao, and Cheng-Zhong Xu. 2013. Coordinated Self-Configuration of Virtual Machines and Appliances Using a Model-Free Learning Approach. *IEEE Trans. Parallel Distrib. Syst.* (2013).

[7] Miroslav Dudík, John Langford, and Lihong Li. 2011. Doubly Robust Policy Evaluation and Learning. In *Intl. Conf. on Machine Learning (ICML)*.

[8] Alan S. Gerber and Donald P. Green. 2012. *Field Experiments: Design, Analysis, and Interpretation*. W.W. Norton&Co, Inc.

[9] D. G. Horvitz and D. J. Thompson. 1952. A Generalization of Sampling Without Replacement from a Finite Universe. *J. Amer. Statist. Assoc.*

[10] Kevin G Jamieson, Lalit Jain, Chris Fernandez, Nicholas J Glattard, and Rob Nowak. 2015. NEXT: A System for Real-World Development, Evaluation, and Application of Active Learning. In *Advances in Neural Information Processing Systems (NIPS)*.

[11] Junchen Jiang, Vyas Sekar, Ion Stoica, and Hui Zhang. 2010. Unleashing the Potential of Data-Driven Networking. In *International Conference on Communication Systems and Networks (COMSNETS)*.

[12] Junchen Jiang, Shijie Sun, Vyas Sekar, and Hui Zhang. 2017. Pytheas: Enabling Data-Driven Quality of Experience Optimization Using Group-Based Exploration-Exploitation. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[13] Nan Jiang and Lihong Li. 2016. Doubly Robust Off-policy Value Evaluation for Reinforcement Learning. In *Intl. Conf. on Machine Learning (ICML)*.

[14] Ron Kohavi, Alex Deng, Brian Frasca, Toby Walker, Ya Xu, and Nils Pohlmann. 2013. Online controlled experiments at large scale. In *ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining (KDD)*.

[15] Ron Kohavi and Roger Longbotham. 2015. Online Controlled Experiments and A/B Tests. In *Encyclopedia of Machine Learning and Data Mining*, Claude Sammut and Geoff Webb (Ed.). Springer. To appear.

[16] Ron Kohavi, Roger Longbotham, Dan Sommerfield, and Randal M. Henne. 2009. Controlled experiments on the web: survey and practical guide. *Data Min. Knowl. Discov.* (2009).

[17] John Langford, Alexander Strehl, and Jennifer Wortman. 2008. Exploration Scavenging. In *Intl. Conf. on Machine Learning (ICML)*.

[18] John Langford and Tong Zhang. 2007. The Epoch-Greedy Algorithm for Contextual Multi-armed Bandits. In *Advances in Neural Information Processing Systems (NIPS)*.

[19] Lihong Li, Wei Chu, John Langford, and Robert E. Schapire. 2010. A contextual-bandit approach to personalized news article recommendation. In *Intl. World Wide Web Conf. (WWW)*.

[20] Lihong Li, Wei Chu, John Langford, and Xuanhui Wang. 2011. Unbiased offline evaluation of contextual-bandit-based news article recommendation algorithms. In *ACM Intl. Conf. on Web Search and Data Mining (WSDM)*.

[21] Konstantinos Lolos, Ioannis Konstantinou, Verena Kantere, and Nectarios Koziris. 2017. Elastic Resource Management with Adaptive State Space Partitioning of Markov Decision Processes. *arXiv preprint arXiv:1702.02978* (2017).

[22] Travis Mandel, Yun-En Liu, Sergey Levine, Emma Brunskill, and Zoran Popovic. 2014. Offline Policy Evaluation Across Representations with Applications to Educational Games. In *International Conference on Autonomous Agents and Multi-agent Systems*.

[23] Shie Mannor, Duncan Simester, Peng Sun, and John N Tsitsiklis. 2007. Bias and variance approximation in value function estimates. *Management Science* (2007).

[24] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. 2016. Resource Management with Deep Reinforcement Learning. In *ACM Workshop on Hot Topics in Networks (HotNets)*.

[25] Microsoft. Accessed in 2017. Custom Decision Service. http://ds.microsoft.com. (Accessed in 2017).

[26] Azalia Mirhoseini, Hieu Pham, Quoc V Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. 2017. Device Placement Optimization with Reinforcement Learning. *arXiv preprint arXiv:1706.04972* (2017).

[27] Netflix. 2011. The Netflix Simian Army. https://medium.com/netflix-techblog/the-netflix-simian-army-16e57fbab116. (2011).

[28] Nginx. Accessed in 2017. https://www.nginx.com/. (Accessed in 2017).

[29] Nginx. Accessed in 2017. Nginx list or variables. https://nginx.org/en/docs/varindex.html. (Accessed in 2017).

[30] Optimizely. Accessed in 2017. Optimizely: A/B Testing & Personalization Platform. https://www.optimizely.com/. (Accessed in 2017).

[31] Cosmin Paduraru. 2012. *Off-policy evaluation in Markov decision processes*. Ph.D. Dissertation. McGill University.

[32] Barry Porter, Matthew Grieves, Roberto Rodrigues Filho, and David Leslie. 2016. REX: A Development Platform and Online Learning Approach for Runtime Emergent Software Systems. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[33] Doina Precup. 2000. Eligibility traces for off-policy policy evaluation. *Computer Science Department Faculty Publication Series* (2000).

[34] Jia Rao, Xiangping Bu, Cheng-Zhong Xu, Leyi Wang, and George Yin. 2009. VCONF: a reinforcement learning approach to virtual machines auto-configuration. In *International conference on Autonomic computing*.

[35] Redis. Accessed in 2017. Redis Key-Value Store. http://http://redis.io. (Accessed in 2017).

[36] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, and Michael Young. 2014. Machine Learning: The High-Interest Credit Card of Technical Debt. In *SE4ML: Software Engineering 4 Machine Learning*.

[37] Stephen M. Stigler. 1992. A Historical View of Statistical Concepts in Psychology and Educational Research. *American Journal of Education* (1992).

[38] Richard S Sutton and Andrew G Barto. 2017. *Reinforcement learning: An introduction*.

[39] Richard S. Sutton, Hamid Reza Maei, Doina Precup, Shalabh Bhatnagar, David Silver, Csaba Szepesvári, and Eric Wiewiora. 2009. Fast Gradient-descent Methods for Temporal-difference Learning with Linear Function Approximation. In *Intl. Conf. on Machine Learning (ICML)*.

[40] Philip S Thomas, Georgios Theocharous, and Mohammad Ghavamzadeh. 2015. High-Confidence Off-Policy Evaluation.. In *AAAI Conference on Artificial Intelligence*.

[41] Dimitrios Tsoumakos, Ioannis Konstantinou, Christina Boumpouka, Spyros Sioutas, and Nectarios Koziris. 2013. Automated, elastic resource provisioning for nosql clusters using tiramola. In *Cluster, Cloud and Grid Computing (CCGrid)*.