Offline Evaluation of Online Reinforcement Learning Algorithms

Travis Mandel[1], **Yun-En Liu**[2], **Emma Brunskill**[3], **and Zoran Popović**[1,2]
[1]Center for Game Science, Computer Science & Engineering, University of Washington, Seattle, WA
[2]Enlearn™, Seattle, WA
[3]School of Computer Science, Carnegie Mellon University, Pittsburgh, PA
{tmandel, zoran}@cs.washington.edu, yunliu@enlearn.org, ebrun@cs.cmu.edu

Abstract

In many real-world reinforcement learning problems, we have access to an existing dataset and would like to use it to evaluate various learning approaches. Typically, one would prefer not to deploy a fixed policy, but rather an algorithm that learns to improve its behavior as it gains more experience. Therefore, we seek to evaluate how a proposed algorithm *learns* in our environment, meaning we need to evaluate how an algorithm would have gathered experience if it were run online. In this work, we develop three new evaluation approaches which guarantee that, given some history, algorithms are fed samples from the distribution that they would have encountered if they were run online. Additionally, we are the first to propose an approach that is provably unbiased given finite data, eliminating bias due to the length of the evaluation. Finally, we compare the sample-efficiency of these approaches on multiple datasets, including one from a real-world deployment of an educational game.

1 Introduction

There is a growing interest in deploying reinforcement learning (RL) agents in real-world environments, such as healthcare or education. In these high-risk situations one cannot deploy an arbitrary algorithm and hope it works well. Instead one needs confidence in an algorithm before risking deployment. Additionally, we often have a large number of algorithms (and associated hyperparameter settings), and it is unclear which will work best in our setting. We would like a way to compare these algorithms without needing to collect new data, which could be risky or expensive.

An important related problem is developing testbeds on which we can evaluate new reinforcement learning algorithms. Historically, these algorithms have been evaluated on simple hand-designed problems from the literature, often with a small number of states or state variables. Recently, work has considered using a diverse suite of Atari games as a testbed for evaluating reinforcement learning algorithms (Bellemare et al. 2013). However, it is not clear that these artificial problems accurately reflect the complex structure present in real-world environments. An attractive alternative is to use precollected real-world datasets to evaluate new RL algorithms on real problems of interest in domains such as healthcare, education, or e-commerce.

These problems have ignited a recent renewal of interest in offline policy evaluation in the RL community (Mandel et al. 2014; Thomas, Theocharous, and Ghavamzadeh 2015), where one uses a precollected dataset to achieve high-quality estimates of the performance of a proposed policy. However, this prior work focuses only on evaluating a *fixed policy* learned from historical data. In many real-world problems, we would instead prefer to deploy a *learning algorithm* that continues to learn over time, as we expect that it will improve over time and thus (eventually) outperform such a fixed policy. Further, we wish to develop testbeds for RL algorithms which evaluate how they learn over time, not just the final policy they produce.

However, evaluating a learning algorithm is very different from evaluating a fixed policy. We cannot evaluate an algorithm's ability to learn by, for example, feeding it 70% of the precollected dataset as training data and evaluating the produced policy on the remaining 30%. Online, it would have collected a different training dataset based on how it trades off exploration and exploitation. In order to evaluate the performance of an algorithm as it learns, we need to simulate running the algorithm by allowing it to interact with the evaluator as it would with the real (stationary) environment, and record the resulting performance estimates (e.g. cumulative reward). See figure 1.[1]

A typical approach to creating such an evaluator is to build a model using the historical data, particularly if the environment is known to be a discrete MDP. However, this approach can result in error that accumulates at least quadratically with the evaluation length (Ross, Gordon, and Bagnell 2011). Equally important, in practice it can result in very poor estimates, as we demonstrate in our experiments section. Worse, in complex real-world domains, it is often unclear how to build accurate models. An alternate approach is to try to adapt importance sampling techniques to this problem, but the variance of this approach is unusably high if we wish to evaluate an algorithm for hundreds of timesteps (Dudík et al. 2014).

[1]In the bandit community, this problem setup is called *nonstationary policy evaluation*, but we avoid use of this term to prevent confusion, as these terms are used in many different RL contexts.

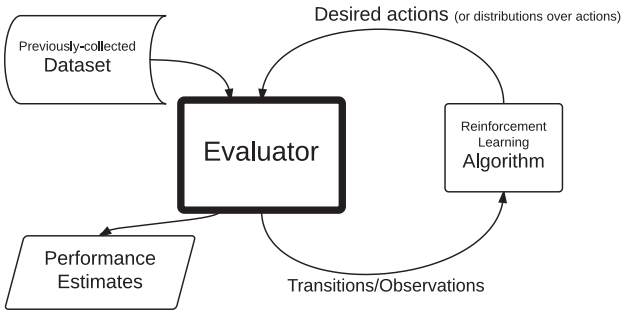


Figure 1: Evaluation process: We are interested in developing evaluators that use a previously-collected dataset to interact with an arbitrary reinforcement learning algorithm as it would interact with the true environment. As it interacts, the evaluator produces performance estimates (e.g. cumulative reward).

In this paper, we present, to our knowledge, the first methods for using historical data to evaluate how an RL algorithm would perform online, which possess both meaningful guarantees on the quality of the resulting performance estimates and good empirical performance. Building upon state-of-the-art work in offline bandit algorithm evaluation (Li et al. 2011; Mandel et al. 2015), we develop three evaluation approaches for reinforcement learning algorithms: queue-based (Queue), per-state rejection sampling (PSRS), and per-episode rejection sampling (PERS). We prove that given the current history, and that the algorithm receives a next observation and reward, that observation and reward is drawn from a distribution identical to the distribution the algorithm would have encountered if it were run in the real world. We show how to modify PERS to achieve stronger guarantees, namely per-timestep unbiasedness given a finite dataset, a property that has not previously been shown even for bandit evaluation methods. Our experiments, including those that use data from a real educational domain, show these methods have different tradeoffs. For example, some are more useful for short-horizon representation-agnostic settings, while others are better suited for long-horizon known-state-space settings. For an overview of further tradeoffs see Table 1. We believe this work will be useful for practitioners who wish to evaluate RL algorithms in a reliable manner given access to historical data.

2 Background and Setting

A discrete Markov Decision Process (MDP) is specified by a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{T}, s_I)$ where $\mathcal{S}$ is a discrete state space, $\mathcal{A}$ is a discrete action space, $\mathcal{R}$ is a mapping from state, action, next state tuples to a distribution over real valued rewards, $\mathcal{T}$ is a transition model that maps state, action, next state tuples to a probability between 0 and 1, and s_I denotes the starting state[2]. We consider an episode to end (and a new episode to begin) when the system transitions back to initial state s_I.

[2]Our techniques could still apply given multiple starting states, but for simplicity we assume a single starting state s_I

We assume, unless otherwise specified, that the domain consists of an episodic MDP $\mathcal{M}$ with a given state space $\mathcal{S}$ and action space $\mathcal{A}$, but unknown reward model $\mathcal{R}$ and transition model $\mathcal{T}$. As input we assume a set D of N transitions, (s, a, r, s') drawn from a fixed sampling policy π_e.

Our objective is to use this data to evaluate the performance of a RL algorithm $\mathbb{A}$. Specifically, without loss of generality[3] we will discuss estimating the discounted sum of rewards obtained by the algorithm $\mathbb{A}$ for a sequence of episodes, e.g. $R^{\mathbb{A}}(i) = \sum_{j=0}^{L(i)-1} \gamma^j r_j$ where $L(i)$ denotes the number of interactions in the i^{th} episode.

At each timestep $t = 1 \ldots \infty$, the algorithm $\mathbb{A}$ outputs a (possibility stochastic) policy π_b from which the next action should be drawn, potentially sampling a set of random numbers as part of this process. For concreteness, we refer to this (possibly random length) vector of random samples used by $\mathbb{A}$ on a given timestep with the variable χ. Let H_T be the history of (s, a, r, s') and χ consumed by $\mathbb{A}$ up to time T. Then, we can say that the behavior of $\mathbb{A}$ at time T depends only on H_T.

Our goal is to create evaluators (sometimes called replayers) that enable us to simulate running the algorithm $\mathbb{A}$ in the true MDP $\mathcal{M}$ using the input dataset D. One key aspect of our proposed evaluators is that they terminate at some timestep. To that end, let g_t denote the event that we do not terminate before outputting an estimate at timestep t (so g_t implies $g_1, \ldots, g_{t-1}$). In order to compare the evaluator to reality, let $P_{\mathcal{R}}(x)$ denote the probability (or pdf if x contains continuous rewards[4]) of generating x under the evaluator, and $P_{\mathcal{E}}(x)$ denote the probability of generating x in the true environment (the MDP $\mathcal{M}$). Similarly, $\mathbb{E}_{\mathcal{R}}[x]$ is the expected value of a random variable x under the evaluator, and $\mathbb{E}_{\mathcal{E}}[x]$ is the expected value of x under the true environment ($\mathcal{M}$).

We will shortly introduce several evaluators. Due to space limitations, we provide only proof sketches: full proofs are in the appendix (available at http://grail.cs.washington.edu/projects/nonstationaryeval).

What guarantees do we desire on the estimates our evaluators produce? Unbiasedness of the reward estimate on episode i is a natural choice, but it is unclear what this means if we do not always output an estimate of episode i due to termination caused by the limited size/coverage of our dataset. Therefore, we show a guarantee that is in some sense weaker, but applies given a finite dataset: Given some history, the evaluator either terminates or updates the algorithm as it would if run online. Given this guarantee, the empirical question is how early termination occurs, which we address experimentally. We now highlight some of the properties we would like an evaluator to possess, which are summarized in Table 1.

1. *Given some history, the (s, a, r, s') tuples provided to $\mathbb{A}$ have the same distribution as those the agent would*

[3]It is easy to modify the evaluators to compute other statistics of the interaction of the algorithm $\mathbb{A}$ with the evaluator, such as the cumulative reward, or the variance of rewards.

[4]In this case, sums should be considered to be integrals

	Samples true	Unbiased estimate of i-th episode performance	Allows unknown sampling distribution	Does not assume Markov	Computationally efficient
Queue	✓	×	✓	×	✓
PSRS	✓	×	×	×	✓
PERS	✓	×(Variants:✓)	×	✓	Not always

Table 1: Desired properties of evaluation approaches, and a comparison of the three evaluators introduced in this paper. We did not include the sample efficiency, because although it is a key metric it is typically domain-dependent.

receive in the true MDP $\mathcal{M}$. Specifically, we desire $P_{\mathcal{R}}(s, a, r, s', \chi | H_T, g_T) = P_{\mathcal{E}}(s, a, r, s', \chi | H_T)$ so that $P_{\mathcal{R}}(H_{T+1} | H_T, g_T) = P_{\mathcal{E}}(H_{T+1} | H_T)$. As mentioned above, this guarantee allows us to ensure that the algorithm is fed on-policy samples, guaranteeing the algorithm behaves similarly to how it would online.

2. *High sample efficiency*. Since all of our approaches only provide estimates for a finite number of episodes before terminating due to lack of data in D, we want to make efficient use of data to evaluate $\mathbb{A}$ for as long as possible.
3. *Given an input i, outputs an unbiased estimate of $R^{\mathbb{A}}(i)$*. Specifically, $\mathbb{E}_{\mathcal{R}}[R^{\mathbb{A}}(i)] = \mathbb{E}_{\mathcal{E}}[R^{\mathbb{A}}(i)]$. Note that this is non-trivial to ensure, since the evaluation may halt before the i-th episode is reached.
4. *Can leverage data D collected using an unknown sampling distribution π_e*. In some situations it may be difficult to log or access the sampling policy π_e, for example in the case where human doctors choose treatments for patients.
5. *Does not assume the environment is a discrete MDP with a known state space $\mathcal{S}$*. In many real world problems, the state space is unknown, partially observed, or continuous, so we cannot always rely on Markov assumptions.
6. *Computationally efficient*.

3 Related work

Work in reinforcement learning has typically focused on evaluating fixed policies using importance sampling techniques (Precup 2000). Importance sampling is widely-used in off-policy learning, as an objective function when using policy gradient methods (Levine and Koltun 2013; Peshkin and Shelton 2002) or as a way to re-weight samples in off-policy TD-learning methods (Mahmood, van Hasselt, and Sutton 2014; Sutton, Mahmood, and White 2015; Maei and Sutton 2010). Additionally, this approach has recently enabled practitioners to evaluate learned policies on complex real-world settings (Thomas, Theocharous, and Ghavamzadeh 2015; Mandel et al. 2014). However, this work focuses on evaluating fixed policies, we are not aware of work specifically focusing on the problem of evaluating how an RL algorithm would learn online, which involves feeding the algorithm new training samples as well as evaluating its current performance. It is worth noting that any of the above-mentioned off-policy learning algorithms could be evaluated using our methods.

Our methods do bear a relationship to off-policy learning work which has evaluated policies by synthesizing artificial trajectories (Fonteneau et al. 2010; 2013). Unlike our work, this approach focuses only on evaluating fixed policies. It also assumes a degree of Lipschitz continuity in some continuous space, which introduces bias. There are some connections: our queue-based estimator could be viewed as related to their work, but focused on evaluating learning algorithms in the discrete MDP policy case.

One area of related work is in the area of (possibly contextual) multi-armed bandits, in which the corresponding problem is termed "nonstationary policy evaluation". Past work has showed evaluation methods that are guaranteed to be unbiased (Li et al. 2011), or have low bias (Dudík et al. 2012; 2014), but only assuming an infinite data stream. Other work has focused on evaluators that perform well empirically but lack this unbiasedness (Mary, Preux, and Nicol 2014). Work by Mandel et al. 2015 in the non-contextual bandit setting show guarantees similar to ours, that issued feedback comes from the true distribution even with finite data. However, in addition to focusing on the more general setting of reinforcement learning, we also show stronger guarantees of unbiasedness even given a finite dataset.

Algorithm 1 Queue-based Evaluator

Input: Dataset D, RL Algorithm $\mathbb{A}$, Starting state s_I
Output: $R^{\mathbb{A}}$ s.t. $R^{\mathbb{A}}(i)$ is sum of rewards in ep. i
$Q[s, a] = Queue(RandomOrder((s_i, a_i, r, s') \in D,$ s.t. $s_i = s, a_i = a)), \forall s \in \mathcal{S}, a \in \mathcal{A}$
for $i = 1$ to ∞ **do**
    $s = s_I, t = 0, r_i = 0$
    Let π_b be $\mathbb{A}$'s initial policy
    while $\neg(t > 0$ and $s == s_I)$ **do**
        $a_b \sim \pi_b(s)$
        if $Q[s, a_b]$ is empty **then return** $R^{\mathbb{A}}$
        $(r, s') = Q[s, a_b].pop()$
        Update $\mathbb{A}$ with (s, a, r, s'), yields new policy π_b
        $r_i = r_i + \gamma^t r, s = s', t = t + 1$
    $R^{\mathbb{A}}(i) = r_i$

4 Queue-based Evaluator

We first propose a *queue-based* evaluator for evaluating algorithms for episodic MDPs with a provided state $\mathcal{S}$ and action $\mathcal{A}$ space (Algorithm 1). This technique is inspired by the queue-based approach to evaluation in non-contextual bandits (Mandel et al. 2015). The key idea is to place feedback (next states and rewards) in queues, and remove elements

based on the current state and chosen action, terminating evaluation when we hit an empty queue. Specifically, first we partition the input dataset D into queues, one queue per (s, a) pair, and fill each queue $Q(s, a)$ with a (random) ordering of all tuples $(r, s') \in D$ s.t. $(s_i = s, a_i = a, r_i = r, s'_i = s')$ To simulate algorithm $\mathbb{A}$ starting from a known state s_k, the algorithm $\mathbb{A}$ outputs a policy π_b, and selects an action a sampled from $\pi_b(s_k)$.[5]

The evaluator then removes a tuple (r, s') from queue $Q[s_k, a]$, which is used to update the algorithm $\mathbb{A}$ and its policy π_b, and simulate a transition to the next state s'. By the Markov assumption, tuples (r, s') are i.i.d. given the prior state and selected action, and therefore an element drawn without replacement from the queue has the same distribution as that in the true environment. The evaluator terminates and outputs the reward vector $R^{\mathbb{A}}$, when it seeks to draw a sample from an empty queue.[6]

Unlike many offline evaluation approaches (such as importance sampling for policy evaluation), our queue evaluator does not require knowledge of the sampling distribution π_e used to generate D. It can even use data gathered from a deterministic sampling distribution. Both properties are useful for many domains (for example, it may be hard to know the stochastic policy used by a doctor to make a decision).

Theorem 4.1. *Assuming the environment is an MDP with state space $\mathcal{S}$ and the randomness involved in drawing from π_b is treated as internal to $\mathbb{A}$, given any history of interactions H_T, if the queue-based evaluator produces a (s, a, r, s') tuple, the distribution of this tuple and subsequent internal randomness χ under the queue-based evaluator is identical to the true distribution the agent would have encountered if it was run online. That is, $P_{\mathcal{R}}(s, a, r, s', \chi | H_T, g_T) = P_{\mathcal{E}}(s, a, r, s', \chi | H_T)$, which gives us that $P_{\mathcal{R}}(H_{T+1} | H_T, g_T) = P_{\mathcal{E}}(H_{T+1} | H_T)$.*

Proof Sketch. The proof follows fairly directly from the fact that placing an (r, s') tuple drawn from $\mathcal{M}$ in $Q[s, a]$ and sampling from Q without replacement results in a sample from the true distribution. See the appendix (available at http://grail.cs.washington.edu/projects/nonstationaryeval).

Note that theorem 4.1 requires us to condition on the fact that $\mathbb{A}$ reveals no randomness, that is, we consider the randomness involved in drawing from π_b on line 8 to be considered as internal, that is (included in χ). This means the guarantee is slightly weaker than the approaches we will present in sections 5 and 6, which condition on general π_b.

5 Per-State Rejection Sampling Evaluator

Ideally, we would like an evaluator that can recognize when the algorithm chooses actions similarly to the sampling distribution, in order use more of the data. For example, in the extreme case where we know the algorithm we are evaluating always outputs the sampling policy, we should be able to make use of all data, or close to it. However, the queue method only uses the sampled action, and thus cannot determine directly whether or not the distribution over actions at each step (π_b) is similar to the sampling policy (π_e). This can make a major difference in practice: If π_b and π_e are both uniform, and the action space is large relative to the amount of data, we will be likely to hit an empty queue if we sample a fresh action from π_b. But, if we know the distributions are the same we can simply take the first sampled action from π_e. Being able to take advantage of stochastic distributions in this way is sometimes referred to as leveraging *revealed randomness* in the candidate algorithm (Dudík et al. 2012).

To better leverage this similarity, we introduce the Per-State Rejection Sampling (PSRS) evaluator (see Algorithm 2), inspired by approaches used in contextual bandits (Li et al. 2011; Dudík et al. 2012). PSRS divides data into streams for each state s, consisting of a (randomized) list of the subsequent (a, r, s') tuples that were encountered from s in the input data. Specifically, given the current state s, our goal is to sample a tuple (a, r, s') such that a is sampled from algorithm $\mathbb{A}$'s current policy $\pi_b(s)$, and r and s' are sampled from the true environment. We already know that given the Markov property, once we select an action a that r and s' in a tuple (s, a, r, s') represent true samples from the underlying Markov environment. The challenge then becomes to sample an action a from $\pi_b(s)$ using the actions sampled by the sampling distribution $\pi_e(s)$ for the current state s. To do this, a rejection sampling algorithm[7] samples a uniform number u between 0 and 1, and accepts a sample (s, a, r, s') from D if $u < \frac{\pi_b(a|s)}{M \pi_e(a|s)}$, where $\pi_b(a|s)$ is the probability under the candidate distribution of sampling action a for state s, $\pi_e(a|s)$ is the corresponding quantity for the sampling distribution, and M is an upper bound on their ratio, $M \geq \max_a \frac{\pi_b(a|s)}{\pi_e(a|s)}$. M is computed by iterating over actions[8] (line 8). It is well known that samples rejection sampling accepts represent true samples from the desired distribution, here π_b (Gelman et al. 2014).

Slightly surprisingly, even if $\mathbb{A}$ always outputs the sampling policy π_e, we do not always consume all samples (in other words PSRS is not *self-idempotent*), unless the original ordering of the streams is preserved (see appendix, available at http://grail.cs.washington.edu/projects/nonstationaryeval). Still, in the face of stochasticity PSRS can be significantly more data-efficient than the Queue-based evaluator.

Theorem 5.1. *Assume the environment is an MDP with state space $\mathcal{S}$, π_e is known, and for all a, $\pi_e(a) > 0$ if $\pi_b(a) > 0$. Then if the evaluator produces a (s, a, r, s') tuple, the distribution of (s, a, r, s') tuple returned by PSRS*

[5] Note that since we only use π_b to draw the next action, this does not prevent $\mathbb{A}$ from internally using a policy that depends on more than s (for example, s and t in finite horizon settings).

[6] For details about why this is necessary, see the appendix (available at http://grail.cs.washington.edu/projects/nonstationaryeval).

[7] One might wonder if we could reduce variance by using an importance sampling instead of rejection sampling approach here. Although in theory possible, one has to keep track of all the different states of the algorithm with and without each datapoint accepted, which is computationally intractable.

[8] This approach is efficient in the since that it takes time linear in $|\mathcal{A}|$, however in very large action spaces this might be too expensive. In certain situations it may be possible to analytically derive a bound on the ratio to avoid this computation.

Algorithm 2 Per-State Rejection Sampling Evaluator

Input: Dataset D, RL Algorithm $\mathbb{A}$, Start state s_I, π_e
Output: Output: $R^{\mathbb{A}}$ s.t. $R^{\mathbb{A}}(i)$ is sum of rewards in ep. i
$Q[s] = Queue(RandomOrder((s_i, a_i, r, s') \in D \text{ s.t. } s_i = s)), \forall s \in \mathcal{S}$
for $i = 1$ to ∞ **do**
    $s = s_I$,$t = 0, r_i = 0$
    Let π_b be $\mathbb{A}$'s initial policy
    while $\neg(t > 0$ and $s_t == s_I)$ **do**
        $M = \max_a \frac{\pi_b(a|s)}{\pi_e(a|s)}$
        (a, r, s') = Q[s].pop()
        if Q[s] is empty **then return** $R^{\mathbb{A}}$
        Sample $u \sim Uniform(0, 1)$
        if $u > \frac{\pi_b(a|s)}{M\pi_e(a|s)}$ **then**
            Reject sample, go to line 9
        Update $\mathbb{A}$ with (s, a, r, s'), yields new policy π_b
        $r_i = r_i + \gamma^t r, s = s', t = t + 1$
    $R^{\mathbb{A}}(i) = r_i$

(and subsequent internal randomness χ) given any history of interactions H_T is identical to the true distribution the agent would have encountered if was run online. Precisely, $P_{\mathcal{R}}(s, a, r, s', \chi|H_T, g_T) = P_{\mathcal{E}}(s, a, r, s', \chi|H_T)$, which gives us that $P_{\mathcal{R}}(H_{T+1}|H_T, g_T) = P_{\mathcal{E}}(H_{T+1}|H_T)$.

Proof Sketch. The proof follows fairly directly from the fact that given finite dataset, rejection sampling returns samples from the correct distribution (Lemma 1 in the appendix, available at http://grail.cs.washington.edu/projects/nonstationaryeval).

Algorithm 3 Per-Episode Rejection Sampling Evaluator

Input: Dataset of episodes D, RL Algorithm $\mathbb{A}$, π_e
Output: Output: $R^{\mathbb{A}}$ s.t. $R^{\mathbb{A}}(i)$ is sum of rewards in ep. i
Randomly shuffle D
Store present state $\mathcal{A}$ of algorithm $\mathbb{A}$
$M = calculateEpisodeM(\mathcal{A}, \pi_e)$ (see the appendix)
$i = 1$, Let π_b be $\mathbb{A}$'s initial policy
for $e \in D$ **do**
    $p = 1.0, h = [], t = 0, r_i = 0$
    for $(o, a, r) \in e$ **do**
        $h \rightarrow (h, o)$
        $p = p\frac{\pi_b(a|h)}{\pi_e(a|h)}$
        Update $\mathbb{A}$ with (o, a, r), output new policy π_b
        $h \rightarrow (h, a, r)$, $r_i = r_i + \gamma^t r$
    Sample $u \sim Uniform(0, 1)$
    if $u > \frac{p}{M}$ **then**
        Roll back algorithm: $\mathbb{A} = \mathcal{A}$
    else
        Store present state $\mathcal{A}$ of algorithm $\mathbb{A}$
        $M = calculateEpisodeM(\mathcal{A}, \pi_e)$
        $R^{\mathbb{A}}(i) = r_i, i = i + 1$
return $R^{\mathbb{A}}$

6 Per-Episode Rejection Sampling

The previous methods assumed the environment is a MDP with a known state space. We now consider the more general setting where the environment consists of a (possibly high dimensional, continuous) observation space $\mathcal{O}$, and a discrete action space $\mathcal{A}$. The dynamics of the environment can depend on the full history of prior observations, actions, and rewards, $h_t = o_0, \ldots, o_t, a_0, \ldots, a_{t-1}, r_0, \ldots, r_{t-1}$. Multiple existing models, such as POMDPs and PSRs, can be represented in this setting. We would like to build an evaluator that is *representation-agnostic*, i.e. does not require Markov assumptions, and whose sample-efficiency does not depend on the size of the observation space.

We introduce the Per-Episode Rejection Sampler (PERS) evaluator (Algorithm 3) that evaluates RL algorithms in these more generic environments. In this setting we assume that the dataset D consists of a stream of episodes, where each episode e represents an ordered trajectory of actions, rewards and observations, $(o_0, a_0, r_0, o_1, a_1, r_1, \ldots, r_{l(e)})$ obtained by executing the sampling distribution π_e for $l(e)-1$ time steps in the environment. We assume that π_e may also be a function of the full history h_t in this episode up to the current time point. For simplicity of notation, instead of keeping track of multiple policies π_b, we simply write π_b (which could implicitly depend on χ).

PERS operates similarly to PSRS, but performs rejection sampling at the episode level. This involves computing the ratio of $\frac{\Pi_{t=0}^{l(e)}\pi_b(a_t|h_t)}{M\Pi_{t=0}^{l(e)}\pi_e(a_t|h_t)}$, and accepting or rejecting the episode according to whether a random variable sampled from the uniform distribution is lower than the computed ratio. As M is a constant that represents the maximum possible ratio between the candidate and sampling *episode* probabilities, it can be computationally involved to compute M exactly. Due to space limitations, we present approaches for computing M in the appendix (available at http://grail.cs.washington.edu/projects/nonstationaryeval). Note that since the probability of accepting an episode is based only on a ratio of action probabilities, one major benefit to PERS is that its sample-efficiency does not depend on the size of the observation space. However, it does depend strongly on the episode length, as we will see in our experiments.

Although PERS works on an episode-level, to handle algorithms that update after every timestep, it updates $\mathbb{A}$ throughput the episode and "rolls back" the state of the algorithm if the episode is rejected (see Algorithm 3).

Unlike PSRS, PERS is self-idempotent, meaning if $\mathbb{A}$ always outputs π_e we accept all data. This follows since if $\pi_e(a_t|h_t) = \pi_b(a_t|h_t)$, $M = 1$ and $\frac{\Pi_{t=0}^{l(e)}\pi_b(a_t|h_t)}{M\Pi_{t=0}^{l(e)}\pi_e(a_t|h_t)} = 1$.

Theorem 6.1. *Assuming π_e is known, and $\pi_b(e) > 0 \rightarrow \pi_e(e) > 0$ for all possible episodes e and all π_b, and PERS outputs an episode e, then the distribution of e (and subsequent internal randomness χ) given any history of episodic interactions H_T using PERS is identical to the true distribution the agent would have encountered if it was run online. That is, $P_{\mathcal{E}}(e, \chi|H_T) = P_{\mathcal{R}}(e, \chi|H_T, g_T)$, which gives us*

that $P_{\mathcal{R}}(H_{T+1}|H_T, g_T) = P_{\mathcal{E}}(H_{T+1}|H_T)$.

Proof Sketch. The proof follows fairly directly from the fact that given finite dataset, rejection sampling returns samples from the correct distribution (Lemma 1 in the appendix, available at http://grail.cs.washington.edu/projects/nonstationaryeval).

7 Unbiasedness Guarantees in the Per-Episode case

Our previous guarantees stated that if we return a sample, it is from the true distribution given the history. Although this is fairly strong, it does not ensure $R^A(i)$ is an unbiased estimate of the reward obtained by A in episode i. The difficulty is that across multiple runs of evaluation, the evaluator may terminate after different numbers of episodes. The probability of termination depends on a host of factors (how random the policy is, which state we are in, etc.). This can result in a bias, as certain situations may be more likely to reach a given length than others.

For example, consider running the queue-based approach on a 3-state MDP: s_I is the initial state, if we take action a_0 we transition to state s_1, if we take action a_1 we transition to s_2. The episode always ends after timestep 2. Imagine the sampling policy chose a_1 99% of the time, but our algorithm chose a_1 50% of the time. If we run the queue approach many times in this setting, runs where the algorithm chose a_1 will be much more likely to reach timestep 2 than those where it chose a_0, since s_2 is likely to have many more samples than s_1. This can result in a bias: if the agent receives a higher reward for ending in s_2 compared to s_1, the average reward it receives at timestep 2 will be overestimated.

One approach proposed by past work (Mandel et al. 2015; Dudík et al. 2014) is to assume T (the maximum timestep/episode count for which we report estimates) is small enough such that over multiple runs of evaluation we usually terminate after T; however it can be difficult to fully bound the remaining bias. Eliminating this bias for the state-based methods is difficult, since the the agent is much more likely to terminate if it transitions to a sparsely-visited state, and so the probability of terminating is hard to compute as it depends on the unknown transition probabilities.

However, modifying PERS to use a fixed M throughout its operation allows us to show that if PERS outputs an estimate, that estimate is unbiased (Theorem 7.1). In practice one will likely have to overestimate this M, for example by bounding $p(x)$ by 1 (or $(1-\epsilon)$ for epsilon-greedy) and calculating the minimum $q(x)$.

Theorem 7.1. *If M is held fixed throughout the operation of PERS,* π_e *is known, and* $\pi_b(e) > 0 \rightarrow \pi_e(e) > 0$ *for all possible episodes* e *and all* π_b*, then if PERS outputs an estimate of some function* $f(H_T)$ *at episode T, that estimate is an unbiased estimator of* $f(H_T)$ *at episode T, in other words,* $\mathbb{E}_{\mathcal{R}}[f(H_T)|g_T, \ldots, g_1] = \sum_{H_T} f(H_T) P_{\mathcal{E}}(H_T) = \mathbb{E}_{\mathcal{E}}[f(H_T)]$*. For example, if* $f(H_T) = R^{\mathbb{A}}(T)$*, the estimate is an unbiased estimator of* $R^{\mathbb{A}}(T)$ *given* $g_T, \ldots, g_1$.

Proof Sketch. We first show that if M is fixed, the probability that each episode is accepted is constant $(1/M)$. This allows us to show that whether we continue or not (g_T) is conditionally independent of H_{T-1}. This lets us remove the conditioning on H_{T-1} in Theorem 6.1 to give us that $P_{\mathcal{R}}(H_T|g_T, \ldots, g_1) = P_{\mathcal{E}}(H_T)$, meaning the distribution over histories after T accepted episodes is correct, from which conditional unbiasedness is easily shown.

Although useful, this guarantee has the downside that the estimate is still conditional on the fact that our approach does not terminate. Theorem 7.2 shows that it is possible to use a further modification of Fixed-M PERS based on importance weighting to always issue unbiased estimates for N total episodes. For a discussion of the empirical downsides to this approach, see the appendix (available at http://grail.cs.washington.edu/projects/nonstationaryeval).

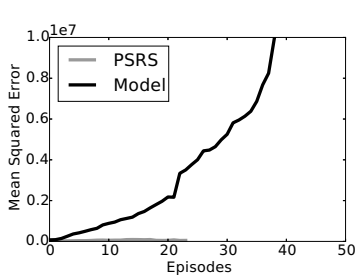
Theorem 7.2. *Assuming for each* T, $R^{\mathbb{A}}(T)$ *is divided by by* $\phi = 1 - Binomial(N, 1/M).cdf(k-1)$*, and after terminating at timestep k we output 0 as estimates of reward for episodes* $k+1, \ldots, N$*, and M is held fixed throughout the operation of PERS, and* π_e *is known, and* $\pi_b(e) > 0 \rightarrow \pi_e(e) > 0$ *for all possible episodes* e *and all* π_b*, then the estimate of reward output at each episode* $T = 1 \ldots N$ *is an unbiased estimator of* $R^{\mathbb{A}}(T)$.

Proof Sketch. Outputting an estimate of the reward at an episode T by either dividing the observed reward by the probability of reaching T (aka $P(g_T, ..., g_1)$), for a run of the evaluator that reaches at least T episodes, or else outputting a 0 if the evaluation has terminated, is an importance weighting technique that ensures the expectation is correct.

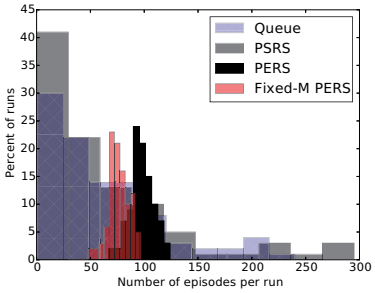
8 Experiments

Any RL algorithm could potentially be run with these evaluators. Here, we show results evaluating Posterior Sampling Reinforcement Learning (PSRL) (Osband et al. 2013, Strens 2000), which has shown good empirical and theoretical performance in the finite horizon case. The standard version of PSRL creates one deterministic policy each episode based on a single posterior sample; however, we can sample the posterior multiple times to create multiple policies and randomly choose between them at each step, which allows us to test our evaluators with more or less revealed randomness.

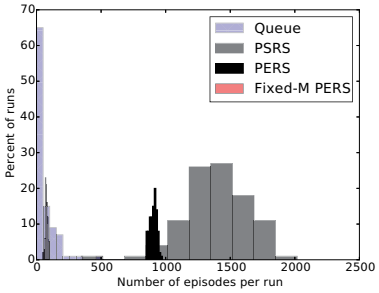
Comparison to a model-based approach We first compare PSRS to a model-based approach on SixArms (Strehl and Littman 2004), a small MDP environment. Our goal is to evaluate the cumulative reward of PSRL run with 10 posterior samples, given a dataset of 100 samples collected using a uniform sampling policy. The model-based approach uses the dataset to build an MLE MDP model. Mean squared error was computed against the average of 1000 runs against the true environment. For details see the appendix (available at http://grail.cs.washington.edu/projects/nonstationaryeval).
In Figure 2a we see that the model-based approach starts fairly accurate but quickly begins returning very poor estimates. In this setting, the estimates it returned indicated that PSRL was learning much more quickly than it would in reality. In contrast, our PSRS approach returns much more accurate estimates and ceases evaluation instead of issuing poor estimates.



(a) PSRS tends to be much more accurate than a model-based approach.



(b) Comparing on Treefrog Treasure with 3 timesteps and 1 PSRL posterior sample.



(c) Comparing on Treefrog with 3 timesteps and 10 PSRL posterior samples.

Figure 2: Experimental results.

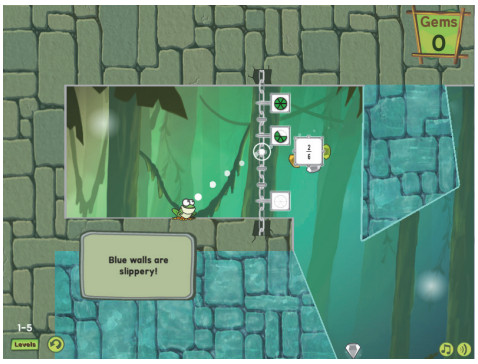


Figure 3: Treefrog Treasure: players guide a frog through a dynamic world, solving number line problems.

Length Results All three of our estimators produce samples from the correct distribution at every step. However, they may provide different length trajectories before termination. To understand the data-efficiency of each evaluator, we tested them on a real-world educational game dataset, as well as a small but well-known MDP example.

Treefrog Treasure is an educational fractions game (Figure 3). The player controls a frog to navigate levels and jump through numberlines. We have 11 actions which control parameters of the numberlines. Our reward is based on whether students learn (based on pretest-to-postest improvement) and whether they remain engaged (measured by whether the student quit before the posttest). We used a state space consisting of the history of actions and whether or not the student took more than 4 tries to pass a numberline (note that this space grows exponentially with the horizon). We varied the considered horizon between 3 and 4 in our experiments. We collected a dataset of 11,550 players collected from a child-focused educational website, collected using a semi-uniform sampling policy. More complete descriptions of the game, experimental methodology, method of calculating M, and details of PSRL can be found in the appendix (available at http://grail.cs.washington.edu/projects/nonstationaryeval).

Figure 2 shows results on Treefrog Treasure, with histograms over 100 complete runs of each evaluator. The graphs show how many episodes the estimator could evaluate the RL algorithm for, with more being better. PERS does slightly better in a short-horizon deterministic setting (Figure 2b). Increasing the posterior samples greatly improves performance of rejection sampling methods (Figure 2c).

We also examined an increased horizon of 4 (graphs provided in appendix, available at http://grail.cs.washington.edu/projects/nonstationaryeval). Given deterministic policies on this larger state space, all three methods are more or less indistinguishable; however, revealing more randomness causes PERS to overtake PSRS (mean 260.54 vs. 173.52). As an extreme case, we also tried a random policy: this large amount of revealed randomness benefits the rejection sampling methods, especially PERS, which evaluates for much longer than the other approaches. PERS outperforms PSRS here because there are small differences between the random candidate policy and the semi-random sampling policy, and thus if PSRS enters a state with little data it is likely to terminate.

The fixed-M PERS method does much worse than the standard version, typically barely accepting any episodes, with notable exceptions when the horizon is short (Figure 2b). Since it does not adjust M it cannot take advantage of revealed randomness (Figure 2c). However, we still feel that this approach can be useful when one desires truly unbiased estimates, and when the horizon is short. Finally, we also note that PERS tends to have the lowest variance, which makes it an attractive approach since to reduce bias one needs to have a high percentage of runs terminating after the desired length.

The state space used in Treefrog Treasure grows exponentially with the horizon. To examine a contrasting case with a small state space (6 states), but a long horizon (20), we also test our approaches in Riverswim (Osband, Russo, and Van Roy 2013), a standard toy MDP environment. The results can be found in the appendix (available at http://grail.cs.washington.edu/projects/nonstationaryeval), but in general we found that PERS and its variants suffer greatly from the long horizon, while Queue and PSRS

do much better, with PSRS doing particularly well if randomness is revealed.

Our conclusion is that the PERS does quite well, especially if randomness is revealed and the horizon is short. It appears there is little reason to choose Queue over PSRS, except if the sampling distribution is unknown. This is surprising because it conflicts with the results of Mandel et al. 2015. They found a queue-based approach to be more efficient than rejection sampling in a non-contextual bandit setting, since data remained in the queues for future use instead of being rejected. The key difference is that in bandits there is only one state, so we do not encounter the problem that we happen to land on an undersampled state, hit an empty queue by chance, and have to terminate the whole evaluation procedure. If the candidate policy behaves randomly at unvisited states, as is the case with 10-sample PSRL, PSRS can mitigate this problem by recognizing the similarity between sampling and candidate distributions to accept the samples at that state, therefore being much less likely to terminate evaluation when a sparsely-visited state is encountered.

9 Conclusion

We have developed three novel approaches for evaluating how RL algorithms perform online: the most important differences are summarized in Table 1. All methods have guarantees that, given some history, if a sample is output it comes from the true distribution. Further, we developed a variant of PERS with even stronger guarantees of unbiasedness. Empirically, there are a variety of tradeoffs to navigate between the methods, based on horizon, revealed randomness in the candidate algorithm, and state space size. We anticipate these approaches will find wide use when one wishes to compare different reinforcement learning algorithms on a real-world problem before deployment. Further, we are excited at the possibility of using these approaches to create real-world testbeds for reinforcement learning problems, perhaps even leading to RL competitions similar to those which related contextual bandit evaluation work (Li et al. 2011) enabled in that setting (Chou and Lin 2012). Future theoretical work includes analyzing the sample complexity of our approaches and deriving tight deviation bounds on the returned estimates. Another interesting direction is developing more accurate estimators, e.g. by using doubly-robust estimation techniques (Dudík et al. 2012).

Acknowledgments This work was supported by the NSF BIGDATA grant No. DGE-1546510, the Office of Naval Research grant N00014-12-C-0158, the Bill and Melinda Gates Foundation grant OPP1031488, the Hewlett Foundation grant 2012-8161, Adobe, Google, and Microsoft.

References

Bellemare, M. G.; Naddaf, Y.; Veness, J.; and Bowling, M. 2013. The arcade learning environment: an evaluation platform for general agents. *Journal of Artificial Intelligence Research* 47(1):253–279.

Chou, K.-C., and Lin, H.-T. 2012. Balancing between estimated reward and uncertainty during news article recommendation for ICML 2012 exploration and exploitation challenge. In *ICML 2012 Workshop: Exploration and Exploitation*, volume 3.

Dudík, M.; Erhan, D.; Langford, J.; and Li, L. 2012. Sample-efficient nonstationary policy evaluation for contextual bandits. *UAI*.

Dudík, M.; Erhan, D.; Langford, J.; Li, L.; et al. 2014. Doubly robust policy evaluation and optimization. *Statistical Science* 29(4):485–511.

Fonteneau, R.; Murphy, S.; Wehenkel, L.; and Ernst, D. 2010. Model-free monte carlo-like policy evaluation. In *Thirteenth International Conference on Artificial Intelligence and Statistics (AISTATS 2010)*.

Fonteneau, R.; Murphy, S. A.; Wehenkel, L.; and Ernst, D. 2013. Batch mode reinforcement learning based on the synthesis of artificial trajectories. *Annals of operations research* 208(1):383–416.

Gelman, A.; Carlin, J. B.; Stern, H. S.; and Rubin, D. B. 2014. *Bayesian data analysis*, volume 2. Taylor & Francis.

Levine, S., and Koltun, V. 2013. Guided policy search. In *Proceedings of The 30th International Conference on Machine Learning*, 1–9.

Li, L.; Chu, W.; Langford, J.; and Wang, X. 2011. Unbiased offline evaluation of contextual-bandit-based news article recommendation algorithms. In *WSDM*, 297–306. ACM.

Maei, H. R., and Sutton, R. S. 2010. GQ (λ): A general gradient algorithm for temporal-difference prediction learning with eligibility traces. In *Proceedings of the Third Conference on Artificial General Intelligence*, volume 1, 91–96.

Mahmood, A. R.; van Hasselt, H. P.; and Sutton, R. S. 2014. Weighted importance sampling for off-policy learning with linear function approximation. In *Advances in Neural Information Processing Systems*, 3014–3022.

Mandel, T.; Liu, Y.-E.; Levine, S.; Brunskill, E.; and Popović, Z. 2014. Offline policy evaluation across representations with applications to educational games. In *AAMAS*, 1077–1084. IFAAMAS.

Mandel, T.; Liu, Y.-E.; Brunskill, E.; and Popović, Z. 2015. The queue method: Handling delay, heuristics, prior data, and evaluation in bandits. *AAAI*.

Mary, J.; Preux, P.; and Nicol, O. 2014. Improving offline evaluation of contextual bandit algorithms via bootstrapping techniques. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, 172–180.

Osband, I.; Russo, D.; and Van Roy, B. 2013. (More) efficient reinforcement learning via posterior sampling. In *Advances in Neural Information Processing Systems*, 3003–3011.

Peshkin, L., and Shelton, C. R. 2002. Learning from scarce experience. *ICML*.

Precup, D. 2000. Eligibility traces for off-policy policy evaluation. *Computer Science Department Faculty Publication Series* 80.

Ross, S.; Gordon, G. J.; and Bagnell, J. A. 2011. A reduction of imitation learning and structured prediction to no-regret online learning. *AISTATS*.

Strehl, A. L., and Littman, M. L. 2004. An empirical evaluation of interval estimation for Markov decision processes. In *Tools with Artificial Intelligence, 2004. ICTAI 2004. 16th IEEE International Conference on*, 128–135. IEEE.

Strens, M. 2000. A Bayesian framework for reinforcement learning. In *ICML*, 943–950.

Sutton, R. S.; Mahmood, A. R.; and White, M. 2015. An emphatic approach to the problem of off-policy temporal-difference learning. *arXiv preprint arXiv:1503.04269*.

Thomas, P. S.; Theocharous, G.; and Ghavamzadeh, M. 2015. High confidence off-policy evaluation. *AAAI*.