Evaluating the Performance of Reinforcement Learning Algorithms

Scott M. Jordan [1] **Yash Chandak** [1] **Daniel Cohen** [1] **Mengxue Zhang** [1] **Philip S. Thomas** [1]

Abstract

Performance evaluations are critical for quantifying algorithmic advances in reinforcement learning. Recent reproducibility analyses have shown that reported performance results are often inconsistent and difficult to replicate. In this work, we argue that the inconsistency of performance stems from the use of flawed evaluation metrics. Taking a step towards ensuring that reported results are consistent, we propose a new comprehensive evaluation methodology for reinforcement learning algorithms that produces reliable measurements of performance both on a single environment and when aggregated across environments. We demonstrate this method by evaluating a broad class of reinforcement learning algorithms on standard benchmark tasks.

1. Introduction

When applying reinforcement learning (RL), particularly to real-world applications, it is desirable to have algorithms that reliably achieve high levels of performance without requiring expert knowledge or significant human intervention. For researchers, having algorithms of this type would mean spending less time tuning algorithms to solve benchmark tasks and more time developing solutions to harder problems. Current evaluation practices do not properly account for the uncertainty in the results (Henderson et al., 2018) and neglect the difficulty of applying RL algorithms to a given problem. Consequently, existing RL algorithms are difficult to apply to real-world applications (Dulac-Arnold et al., 2019). To both make and track progress towards developing reliable and easy-to-use algorithms, we propose a principled evaluation procedure that quantifies the difficulty of using an algorithm.

For an evaluation procedure to be useful for measuring the usability of RL algorithms, we suggest that it should have four properties. First, to ensure accuracy and reliability, an evaluation procedure should be *scientific*, such that it provides information to answer a research question, tests a specific hypothesis, and quantifies any uncertainty in the results. Second, the performance metric captures the *usability* of the algorithm over a wide variety of environments. For a performance metric to capture the usability of an algorithm, it should include the time and effort spent tuning the algorithm's hyperparameters (e.g., step-size and policy structure). Third, the evaluation procedure should be *nonexploitative* (Balduzzi et al., 2018), meaning no algorithm should be favored by performing well on an over-represented subset of environments or by abusing a particular score normalization method. Fourth, an evaluation procedure should be *computationally tractable*, meaning that a typical researcher should be able to run the procedure and repeat experiments found in the literature.

As an evaluation procedure requires a question to answer, we pose the following to use throughout the paper: *which algorithm(s) perform well across a wide variety of environments with little or no environment-specific tuning*? Throughout this work, we refer to this question as the *general evaluation question*. This question is different from the one commonly asked in articles proposing a new algorithm, e.g., the *common question* is, can algorithm X outperform other algorithms on tasks A, B, and C? In contrast to the common question, the expected outcome for the general evaluation question is not to find methods that maximize performance with optimal hyperparameters but to identify algorithms that do not require extensive hyperparameter tuning and thus are easy to apply to new problems.

In this paper, we contend that the standard evaluation approaches do not satisfy the above properties, and are not able to answer the general evaluation question. Thus, we develop a new procedure for evaluating RL algorithms that overcomes these limitations and can accurately quantify the uncertainty of performance. The main ideas in our approach are as follows. We present an alternative view of an algorithm such that sampling its performance can be used to answer the general evaluation question. We define a new normalized performance measure, *performance percentiles*, which uses a relative measure of performance to compare algorithms across environments. We show how to use a

[1]College of Information and Computer Sciences, University of Massachusetts, MA, USA. Correspondence to: Scott Jordan <sjordan@cs.umass.edu>.

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

game-theoretic approach to construct an aggregate measure of performance that permits quantifying uncertainty. Lastly, we develop a technique, *performance bound propagation* (PBP), to quantify and account for uncertainty throughout the entire evaluation procedure. We provide source code so others may easily apply the methods we develop here.[1]

2. Notation and Preliminaries

In this section, we give notation used in this paper along with an overview of an evaluation procedure. In addition to this section, a list of symbols used in this paper is presented in Appendix C. We represent a performance metric of an algorithm, $i \in \mathcal{A}$, on an environment, $j \in \mathcal{M}$, as a random variable $X_{i,j}$. This representation captures the variability of results due to the choice of the random seed controlling the stochastic processes in the algorithm and the environment. The choice of the metric depends on the property being studied and is up to the experiment designer. The performance metric used in this paper is the average of the observed returns from one execution of the entire training procedure, which we refer to as the average return. The cumulative distribution function (CDF), $F_{X_{i,j}} : \mathbb{R} \to [0,1]$, describes the performance distribution of algorithm i on environment j such that $F_{X_{i,j}}(x) \coloneqq \Pr(X_{i,j} \leq x)$. The quantile function, $Q_{X_{i,j}}(\alpha) \coloneqq \inf_x\{x \in \mathbb{R} | F_{X_{i,j}}(x) \geq \alpha\}$, maps a cumulative probability, $\alpha \in (0,1)$, to a score such that α proportion of samples of $X_{i,j}$ are less than or equal to $Q_{X_{i,j}}(\alpha)$. A normalization function, $g \colon \mathbb{R} \times \mathcal{M} \to \mathbb{R}$, maps a score, x, an algorithm receives on an environment, j, to a normalized score, $g(x,j)$, which has a common scale for all environments. In this work, we seek an aggregate performance measure, $y_i \in \mathbb{R}$, for an algorithm, i, such that $y_i \coloneqq \sum_{j=1}^{|\mathcal{M}|} q_j \mathbf{E}[g(X_{i,j}, j)]$, where $q_j \geq 0$ for all $j \in \{1, \ldots, |\mathcal{M}|\}$ and $\sum_{j=1}^{|\mathcal{M}|} q_j = 1$. In Section 4, we discuss choices for the normalizing function g and weightings q that satisfy the properties specified in the introduction.

The primary quantities of interest in this paper are the aggregate performance measures for each algorithm and confidence intervals on that measure. Let $y \in \mathbb{R}^{|\mathcal{A}|}$ be a vector representing the aggregate performance for each algorithm. We desire confidence intervals, $Y^-, Y^+ \in \mathbb{R}^{|\mathcal{A}|} \times \mathbb{R}^{|\mathcal{A}|}$, such that, for a confidence level $\delta \in (0, 0.5]$,

$$\Pr\left(\forall i \in \{1, 2, \ldots, |\mathcal{A}|\}, y_i \in [Y_i^-, Y_i^+]\right) \geq 1 - \delta.$$

To compute an aggregate performance measure and its confidence intervals that meet the criteria laid out in the introduction, one must consider the entire evaluation procedure. We view an evaluation procedure to have three main components: data collection, data aggregation, and reporting of the results. During the data collection phase, samples are collected of the performance metric $X_{i,j}$ for each combination (i, j) of an algorithm $i \in \mathcal{A}$ and environment $j \in \mathcal{M}$. In the data aggregation phase, all samples of performance are normalized so the metric on each environment is on a similar scale, then they are aggregated to provide a summary of each algorithm's performance across all environments. Lastly, the uncertainty of the results is quantified and reported.

[1]Source code for this paper can be found at https://github.com/ScottJordan/EvaluationOfRLAlgs.

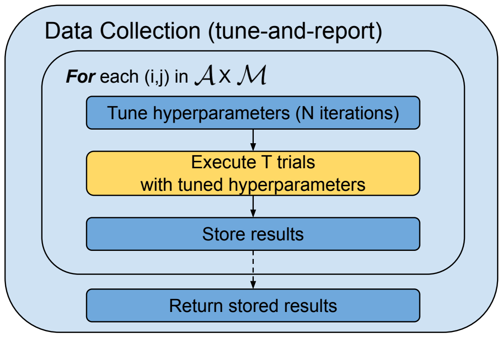

Data Collection (tune-and-report)
For each (i,j) in $\mathcal{A} \times \mathcal{M}$
Tune hyperparameters (N iterations)
Execute T trials with tuned hyperparameters
Store results
Return stored results

Figure 1. Data collection process of the tune-and-report method. The yellow box indicates trials using different random seeds.

3. Data Collection

In this section, we discuss how common data collection methods are unable to answer the general evaluation question and then present a new method that can. We first highlight the core difference in our approach to previous methods.

The main difference between data collection methods is in how the samples of performance are collected for each algorithm on each environment. Standard approaches rely on first tuning an algorithm's hyperparameters, i.e., any input to an algorithm that is not the environment, and then generating samples of performance. Our method instead relies on having a definition of an algorithm that can automatically select, sample, or adapt hyperparameters. This method can be used to answer the general evaluation question because its performance measure represents the knowledge required to use the algorithm. We discuss these approaches below.

3.1. Current Approaches

A typical evaluation procedure used in RL research is the *tune-and-report* method. As depicted in Figure 1, the tune-and-report method has two phases: a tuning phase and a testing phase. In the tuning phase, hyperparameters are optimized either manually or via a hyperparameter optimization algorithm. Then after tuning, only the best hyperparameters are selected and executed for T trials using different random seeds to provide an estimate of performance.

The tune-and-report data collection method does not satisfy the usability requirement or the scientific requirement. Recall that our objective is to capture the difficulty of using a particular algorithm. Because the tune-and-report method ignores the amount of data used to tune the hyperparameter, an algorithm that only works well after significant tuning could be favored over one that works well without environment-specific tuning, thus, violating the requirements.

Consider an extreme example of an RL algorithm that includes all policy parameters as hyperparameters. This algorithm would then likely be optimal after any iteration of hyperparameter tuning that finds the optimal policy. This effect is more subtle in standard algorithms, where hyperparameter tuning infers problem-specific information about how to search for the optimal policy, (e.g., how much exploration is needed, or how aggressive policy updates can be). Furthermore, this demotivates the creation of algorithms that are easier to use but do not improve performance after finding optimal hyperparameters.

The tune-and-report method violates the scientific property by not accurately capturing the uncertainty of performance. Multiple i.i.d. samples of performance are taken after hyperparameter tuning and used to compute a bound on the mean performance. However, these samples of performance do not account for the randomness due to hyperparameter tuning. As a result, any statistical claim would be inconsistent with repeated evaluations of this method. This has been observed in several studies where further hyperparameter tuning has shown no difference in performance relative to baseline methods (Lucic et al., 2018; Melis et al., 2018).

The evaluation procedure proposed by Dabney (2014) addresses issues with uncertainty due to hyperparameter tuning and performance not capturing the usability of algorithms. Dabney's evaluation procedure computes performance as a weighted average over all N iterations of hyperparameter tuning, and the entire tuning process repeats for T trials. Even though this evaluation procedure fixes the problems with the tune-and-report approach, it violates our computationally tractable property by requiring TN executions of the algorithm to produce just T samples of performance. In the case where $N = 1$ it is not clear how hyperparameters should be set. Furthermore, this style of evaluation does not cover the case where it is prohibitive to perform hyperparameter tuning, e.g., slow simulations, long agent lifetimes, lack of a simulator, and situations where it is dangerous or costly to deploy a bad policy. In these situations, it is desirable for algorithms to be insensitive to the choice of hyperparameters or able to adapt them during a single execution. It is in this setting that the general evaluation question can be answered.

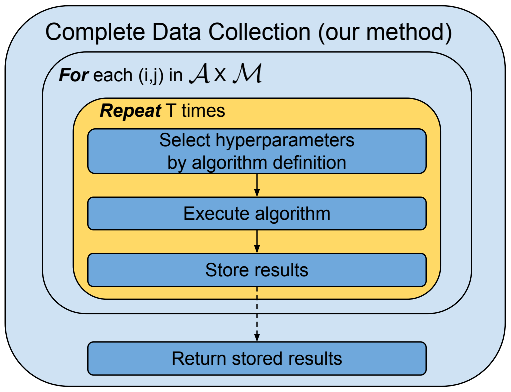

Complete Data Collection (our method)
For each (i,j) in $\mathcal{A} \times \mathcal{M}$
Repeat T times
Select hyperparameters by algorithm definition
Execute algorithm
Store results
Return stored results

Figure 2. Data collection process using complete algorithm definitions. The yellow box indicates using different random seeds.

3.2. Our Approach

In this section, we outline our method, *complete data collection*, that does not rely on hyperparameter tuning. If there were no hyperparameters to tune, evaluating algorithms would be simpler. Unfortunately, how to automatically set hyperparameters has been an understudied area. Thus, we introduce the notion of a complete algorithm definition.

Definition 1 (Algorithm Completeness)**.** *An algorithm is complete on an environment j, when defined such that the only required input to the algorithm is meta-information about environment j, e.g., the number of state features and actions.*

Algorithms with a complete definition can be used on an environment and without specifying any hyperparameters. Note that this does not say that an algorithm cannot receive forms of problem specific knowledge, only that it is not required. A well-defined algorithm will be able to infer effective combinations of hyperparameters or adapt them during learning. There are many ways to make an existing algorithm complete. In this work, algorithms are made complete by defining a distribution from which to randomly sample hyperparameters. Random sampling may produce poor or divergent behavior in the algorithm, but this only indicates that it is not yet known how to set the hyperparameters of the algorithm automatically. Thus, when faced with a new problem, finding decent hyperparameters will be challenging. One way to make an adaptive complete algorithm is to include a hyperparameter optimization method in the algorithm. However, all tuning must be done within the same fixed amount of time and cannot propagate information over trials used to obtain statistical significance.

Figure 2 shows the complete data collection method. For this method we limit the scope of algorithms to only include ones with complete definitions; thus, it does not violate any of the properties specified. This method satisfies the scientific requirement since it is designed to answer the general evaluation question, and the uncertainty of performance can be estimated using all of the trials. Again, this data collection method captures the difficulty of using an algorithm since the complete definition encodes the knowledge necessary for the algorithm to work effectively. The compute time of this method is tractable, since T executions of the algorithm produces T independent samples of performance.

The practical effects of using the complete data collection method are as follows. Researchers do not have to spend time tuning each algorithm to try and maximize performance. Fewer algorithm executions are required to obtain a statistically meaningful result. With this data collection method, improving upon algorithm definitions will become significant research contributions and lead to algorithms that are easy to apply to many problems.

4. Data Aggregation

Answering the general evaluation question requires a ranking of algorithms according to their performance on all environments $\mathcal{M}$. The aggregation step accomplishes this task by combining the performance data generated in the collection phase and summarizing it across all environments. However, data aggregation introduces several challenges. First, each environment has a different range of scores that need to be normalized to a common scale. Second, a uniform weighting of environments can introduce bias. For example, the set of environments might include many slight variants of one domain, giving that domain a larger weight than a single environment coming from a different domain.

4.1. Normalization

The goal in score normalization is to project scores from each environment onto the same scale while not being exploitable by the environment weighting. In this section, we first show how existing normalization techniques are exploitable or do not capture the properties of interest. Then we present our normalization technique: performance percentiles.

4.1.1. Current Approaches

We examine two normalization techniques: performance ratios and policy percentiles. We discuss other normalization methods in Appendix A. The performance ratio is commonly used with the Arcade Learning Environment to compare the performance of algorithms relative to human performance (Mnih et al., 2015; Machado et al., 2018). The performance ratio of two algorithms i and k on an environment j is $\mathbf{E}[X_{i,j}]/\mathbf{E}[X_{k,j}]$. This ratio is sensitive to the location and scale of the performance metric on each environment, such that an environment with scores in the range $[0, 1]$ will produce larger differences than those on the range $[1000, 1001]$. Furthermore, all changes in performance are assumed to be equally challenging, i.e., going from a score of 0.8 to 0.89 is the same difficulty as 0.9 to 0.99. This assumption of linearity of difficulty is not reflected on environments with nonlinear changes in the score as an agent improves, e.g., completing levels in Super Mario.

A critical flaw in the performance ratio is that it can produce an arbitrary ordering of algorithms when combined with the arithmetic mean, $\sum_j q_j \mathbf{E}[X_{i,j}]/\mathbf{E}[X_{k,j}]$ (Fleming & Wallace, 1986), meaning a different algorithm in the denominator could change the relative rankings. Using the geometric mean can address this weakness of performance ratios, but does not resolve the other issues.

Another normalization technique is *policy percentiles*, a method that projects the score of an algorithm through the performance CDF of random policy search (Dabney, 2014). The normalized score for an algorithm, i, is $F_{X_{\Pi,j}}(X_{i,j})$, where $F_{X_{\Pi,j}}$ is the performance CDF when a policy is sampled uniformly from a set of policies, Π, on an environment j, i.e, $\pi \sim U(\Pi)$. Policy percentiles have a unique advantage in that performance is scaled according to how difficult it is to achieve that level of performance relative to random policy search. Unfortunately, policy percentiles rely on specifying Π, which often has a large search space. As a result, most policies will perform poorly, making all scores approach 1.0. It is also infeasible to use when random policy search is unlikely to achieve high levels of performance. Despite these drawbacks, the scaling of scores according to a notion of difficulty is desirable, so we adapt this idea to use any algorithm's performance as a reference distribution.

4.1.2. Our Approach

An algorithm's performance distribution can have an interesting shape with large changes in performance that are due to divergence, lucky runs, or simply that small changes to a policy can result in large changes in performance (Jordan et al., 2018). These effects can be seen in Figure 3, where there is a quick rise in cumulative probability for a small increase in performance. Inspired by Dabney (2014)'s policy percentiles, we propose *performance percentiles*, a score normalization technique that can represent these intricacies.

The probability integral transform shows that projecting a random variable through its CDF transforms the variable to be uniform on $[0, 1]$ (Dodge & Commenges, 2006). Thus, normalizing an algorithm's performance by its CDF will equally distribute and represent a linear scaling of difficulty across $[0, 1]$. When normalizing performance against another algorithm's performance distribution, the normalized score distribution will shift towards zero when the algorithm is worse than the normalizing distribution and shift towards one when it is superior. As seen in Figure 3, the CDF can

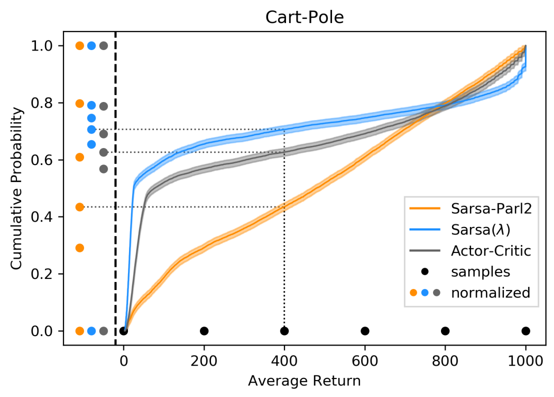


Figure 3. This plot shows the CDF of average returns for the Sarsa-Parl2, Sarsa(λ), and Actor-Critic algorithms on the Cart-Pole environment. Each line represents the empirical CDF using 10,000 trials and the shaded regions represent the 95% confidence intervals. To illustrate how the performance percentiles work, this plot shows how samples of performance (black dots) are normalized by each CDF, producing the normalized scores (colored dots). The correspondence between a single sample and its normalized score is shown by the dotted line.

be seen as encoding the relative difficulty of achieving a given level of performance, where large changes in an algorithm's CDF output indicate a high degree of difficulty for that algorithm to make an improvement and similarly small changes in output correspond to low change in difficulty. In this context difficulty refers to the amount of random chance (luck) needed to achieve a given level of performance.

To leverage these properties of the CDF, we define performance percentiles, that use a weighted average of each algorithm's CDF to normalize scores for each environment.

Definition 2 (Performance Percentile)**.** *In an evaluation of algorithms, $\mathcal{A}$, the performance percentile for a score x on an environment, j, is $F_{\bar{X}_j}(x, w_j)$, where $F_{\bar{X}_j}$ is the mixture of CDFs $F_{\bar{X}_j}(x, w_j) := \sum_{i=1}^{|\mathcal{A}|} w_{j,i} F_{X_{i,j}}(x)$, with weights $w_j \in \mathbb{R}^{|\mathcal{A}|}$, $\sum_{i=1}^{|\mathcal{A}|} w_{j,i} = 1$, and $\forall i \; w_{j,i} \geq 0$.*

So we can say that performance percentiles capture the performance characteristic of an environment relative to some averaged algorithm. We discuss how to set the weights w_j in the next section.

Performance percentiles are closely related to the concept of (probabilistic) performance profiles (Dolan & Moré, 2002; Barreto et al., 2010). The difference being that performance profiles report the cumulative distribution of normalized performance metrics over a set of tasks (environments), whereas performance percentiles are a technique for normalizing scores on each task (environment).

4.2. Summarization

A weighting over environments is needed to form an aggregate measure. We desire a weighting over environments such that no algorithm can exploit the weightings to increase its ranking. Additionally, for the performance percentiles, we need to determine the weighting of algorithms to use as the reference distribution. Inspired by the work of Balduzzi et al. (2018), we propose a weighting of algorithms and environments, using the equilibrium of a two-player game.

In this game, one player, p, will try to select an algorithm to maximize the aggregate performance, while a second player, q, chooses the environment and reference algorithm to minimize p's score. Player p's pure strategy space, $\mathcal{S}_1$, is the set of algorithms $\mathcal{A}$, i.e., p plays a strategy $s_1 = i$ corresponding to an algorithm i. Player q's pure strategy space, $\mathcal{S}_2$, is the cross product of a set of environments, $\mathcal{M}$, and algorithms, $\mathcal{A}$, i.e., player q plays a strategy $s_2 = (j, k)$ corresponding to a choice of environment j and normalization algorithm k. We denote the pure strategy space of the game by $\mathcal{S} := \mathcal{S}_1 \times \mathcal{S}_2$. A strategy, $s \in \mathcal{S}$, can be represented by a tuple $s = (s_1, s_2) = (i, (j, k))$.

The utility of strategy s is measured by a payoff function $u_p \colon \mathcal{S} \to \mathbb{R}$ and $u_q \colon \mathcal{S} \to \mathbb{R}$ for players p and q respectively. The game is defined to be zero sum, i.e., $u_q(s) = -u_p(s)$. We define the payoff function to be $u_p(s) := \mathbf{E}[F_{X_{k,j}}(X_{i,j})]$. Both players p and q sample strategies from probability distributions $p \in \Delta(\mathcal{S}_1)$ and $q \in \Delta(\mathcal{S}_2)$, where $\Delta(\mathcal{X})$ is the set of all probability distributions over $\mathcal{X}$.

The equilibrium solution of this game naturally balances the normalization and environment weightings to counter each algorithm's strengths without conferring an advantage to a particular algorithm. Thus, the aggregate measure will be useful in answering the general evaluation question.

After finding a solution (p^*, q^*), the aggregate performance measure y_i for an algorithm i defined as

$$y_i := \sum_{j=1}^{|\mathcal{M}|} \sum_{k=1}^{|\mathcal{A}|} q^*_{j,k} \mathbf{E}[F_{X_{k,j}}(X_{i,j})]. \tag{1}$$

To find a solution (p^*, q^*), we employ the α-Rank technique (Omidshafiei et al., 2019), which returns a stationary distribution over the pure strategy space $\mathcal{S}$. α-Rank allows for efficient computation of both the equilibrium and confidence intervals on the aggregate performance (Rowland et al., 2019). We detail this method and details of our implementation in Appendix B.

5. Reporting Results

As it is crucial to quantify the uncertainty of all claimed performance measures, we first discuss how to compute confidence intervals for both single environment and aggregate measures, then give details on displaying the results.

5.1. Quantifying Uncertainty

In keeping with our objective to have a scientific evaluation, we require our evaluation procedure to quantify any uncertainty in the results. When concerned with only a single environment, standard concentration inequalities can compute confidence intervals on the mean performance. Similarly, when displaying the distribution of performance, one can apply standard techniques for bounding the empirical distribution of performance. However, computing confidence intervals on the aggregate has additional challenges.

Notice that in (1) computing the aggregate performance requires two unknown values: q^* and the mean normalized performance, $\mathbf{E}[F_{X_{k,j}}(X_{i,j})]$. Since q^* depends on mean normalized performance, any uncertainty in the mean normalized performance results in uncertainty in q^*. To compute valid confidence intervals on the aggregate performance, the uncertainty through the entire process must be considered.

We introduce a process to compute the confidence intervals, which we refer to as *performance bound propagation* (PBP). We represent PBP as a function PBP: $\mathcal{D} \times \mathbb{R} \to \mathbb{R}^{|\mathcal{A}|} \times \mathbb{R}^{|\mathcal{A}|}$, which maps a dataset $D \in \mathcal{D}$ containing all samples of performance and a confidence level $\delta \in (0, 0.5]$, to vectors Y^- and Y^+ representing the lower and upper confidence intervals, i.e., $(Y^-, Y^+) = \text{PBP}(D, \delta)$.

The overall procedure for PBP is as follows, first compute confidence intervals for each $F_{X_{i,j}}$, then using these intervals compute confidence intervals on each mean normalized performance, next determine an uncertainty set $\mathcal{Q}$ for q^* that results from uncertainty in the mean normalized performance, finally for each algorithm find the minimum and maximum aggregate performance over the uncertainty in the mean normalized performances and $\mathcal{Q}$. We provide pseudocode in Appendix C and source code in the repository.

We prove that PBP produces valid confidence intervals for a confidence level $\delta \in (0, 0.5]$ and a dataset D containing $T_{i,j} > 1$ samples of performance for all algorithms $i \in \mathcal{A}$ and environments $j \in \mathcal{M}$.

Theorem 1. *If* $(Y^-, Y^+) = PBP(D, \delta)$*, then*

$$\Pr\left(\forall i \in 1, 2, \ldots, |\mathcal{A}|,\ y_i \in [Y_i^-, Y_i^+]\right) \geq 1 - \delta.$$

Proof. Although the creation of valid confidence intervals is critical to this contribution, due to space restrictions it is presented in Appendix C. □

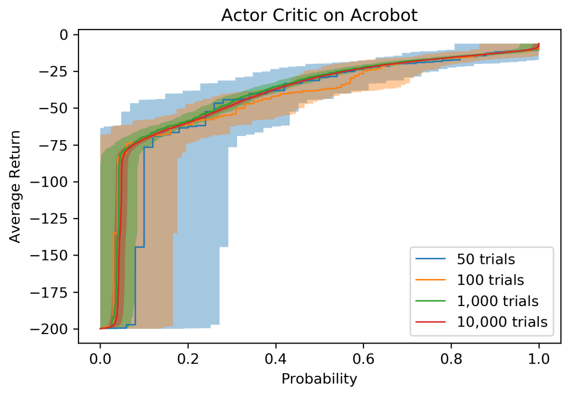


Figure 4. This plot shows the distribution of average returns for the Actor-Critic algorithm on the Acrobot environment. The x-axis represents a probability and the y-axis represents the average return such that the proportion of trials that have a value less than or equal to y is x, e.g., at $x = 0.5$, y is the median. Each line represents the empirical quantile function using a different number of trials and the shaded regions represent the 95% confidence intervals computed using concentration inequalities. In this plot, the larger the area under the curve, the better the performance. This plot highlights the large amount of uncertainty when using small sample sizes and how much it decreases with more samples.

5.2. Displaying Results

In this section, we describe our method for reporting the results. There are three parts to our method: answering the stated hypothesis, providing tables and plots showing the performance and ranking of algorithms for all environments, and the aggregate score, then for each performance measure, provide confidence intervals to convey uncertainty.

The learning curve plot is a standard in RL and displays a performance metric (often the return) over regular intervals during learning. While this type of plot might be informative for describing some aspects of the algorithm's performance, it does not directly show the performance metric used to compare algorithms, making visual comparisons less obvious. Therefore, to provide the most information to the reader, we suggest plotting the distribution of performance for each algorithm on each environment. Plotting the distribution of performance has been suggested in many fields as a means to convey more information, (Dolan & Moré, 2002; Farahmand et al., 2010; Reimers & Gurevych, 2017; Cohen et al., 2018). Often in RL, the object is to maximize a metric, so we suggest showing the quantile function over the CDF as it allows for a more natural interpretation of the performance, i.e., the higher the curve, the better the performance (Bellemare et al., 2013). Figure 4 show the performance distribution with 95% confidence intervals for different sample sizes. It is worth noting that when tuning hyperparameters the data needed to compute these distributions is already being collected, but only the results from the tuned runs are being reported. By only reporting only the tuned performance it shows what an algorithm can achieve not what it is likely to achieve.

6. Experimental Results

In this section, we describe and report the results of experiments to illustrate how this evaluation procedure can answer the general evaluation question and identify when a modification to an algorithm or its definition improves performance. We also investigate the reliability of different bounding techniques on the aggregate performance measure.

6.1. Experiment Description

To demonstrate the evaluation procedure we compare the algorithms: Actor-Critic with eligibility traces (AC) (Sutton & Barto, 2018), Q(λ), Sarsa(λ), (Sutton & Barto, 1998), NAC-TD (Morimura et al., 2005; Degris et al., 2012; Thomas, 2014), and proximal policy optimization (PPO) (Schulman et al., 2017). The learning rate is often the most sensitive hyperparameter in RL algorithms. So, we include three versions of Sarsa(λ), Q(λ), and AC: a base version, a version that scales the step-size with the number of parameters (e.g., Sarsa(λ)-s), and an adaptive step-size method, Parl2 (Dabney, 2014), that does not require specifying the step size. Since none of these algorithms have an existing complete definition, we create one by randomly sampling hyperparameters from fixed ranges. We consider all parameters necessary to construct each algorithm, e.g., step-size, function approximator, discount factor, eligibility trace decay. For the continuous state environments, each algorithm employs linear function approximation using the Fourier basis (Konidaris et al., 2011) with a randomly sampled order. See Appendix E for full details of each algorithm.

These algorithms are evaluated on 15 environments, eight discrete MDPs, half with stochastic transition dynamics, and seven continuous state environments: Cart-Pole (Florian, 2007), Mountain Car (Sutton & Barto, 1998), Acrobot (Sutton, 1995), and four variations of the pinball environment (Konidaris & Barto, 2009; Geramifard et al., 2015). For each independent trial, the environments have their dynamics randomly perturbed to help mitigate environment overfitting (Whiteson et al., 2011); see code for details. For further details about the experiment see Appendix F.

While these environments have simple features compared to the Arcade Learning Environment (Bellemare et al., 2013), they remain useful in evaluating RL algorithms for three reasons. First is that experiments finish quickly. Second, the environments provide interesting insights into an algorithm's behavior. Third, as our results will show, there is not yet a complete algorithm that can reliably solve each one.

Aggregate Performance		
Algorithms	**Score**	**Rank**
Sarsa-Parl2	$0.4623(\mathbf{0.3904}, 0.5537)$	1 (2,1)
Q-Parl2	$0.4366(\mathbf{0.3782}, 0.5632)$	2 (2,1)
AC-Parl2	$0.1578(0.0765, \mathbf{0.3129})$	3 (11,3)
Sarsa(λ)-s	$0.0930(0.0337, 0.2276)$	4 (11,3)
AC-s	$0.0851(0.0305, 0.2146)$	5 (11,3)
Sarsa(λ)	$0.0831(0.0290, 0.2019)$	6 (11,3)
AC	$0.0785(0.0275, 0.2033)$	7 (11,3)
Q(λ)-s	$0.0689(0.0237, 0.1973)$	8 (11,3)
Q(λ)	$0.0640(0.0214, 0.1780)$	9 (11,3)
NAC-TD	$0.0516(0.0180, 0.1636)$	10 (11,3)
PPO	$0.0508(0.0169, 0.1749)$	11 (11,3)

Table 1. Aggregate performance measures for each algorithm and their rank. The parentheses contain the intervals computed using PBP and together all hold with 95% confidence. The bolded numbers identify the best ranked statistically significant differences.

We execute each algorithm on each environment for 10,000 trials. While this number of trials may seem excessive, our goal is to detect a statistically meaningful result. Detecting such a result is challenging because the variance of RL algorithms performance is high; we are comparing $|\mathcal{A}| \times |\mathcal{M}| = 165$ random variables, and we do not assume the performances are normally distributed. Computationally, executing ten thousand trials is not burdensome if one uses an efficient programming language such as Julia (Bezanson et al., 2017) or C++, where we have noticed approximately two orders of magnitude faster execution than similar Python implementations. We investigate using smaller sample sizes at the end of this section.

6.2. Algorithm Comparison

The aggregate performance measures and confidence intervals are illustrated in Figure 5 and given in Table 1. Appendix I lists the performance tables and distribution plots for each environment. Examining the empirical performances in these figures, we notice two trends. The first is that our evaluation procedure can identify differences that are not noticeable in standard evaluations. For example, all algorithms perform near optimally when tuned properly (indicated by the high end of the performance distribution). The primary differences between algorithms are in the frequency of high performance and divergence (indicated by low end of the performance distribution). Parl2 methods rarely diverge, giving a large boost in performance relative to the standard methods.

The second trend is that our evaluation procedure can identify when theoretical properties do or do not make an al-

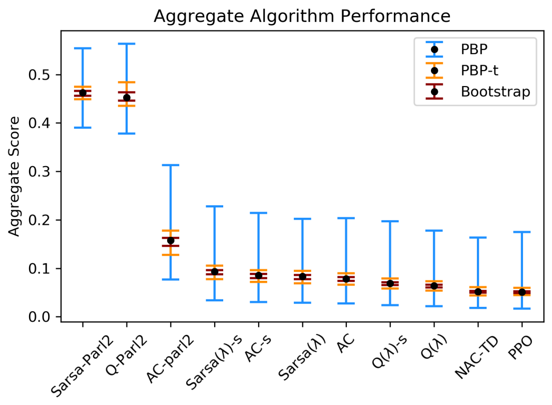


Figure 5. The aggregate performance for each algorithm with confidence intervals using PBP, PBP-t, and bootstrap. The width of each interval is scaled so all intervals hold with 95% confidence.

gorithm more usable. For example, Sarsa(λ) algorithms outperform their Q(λ) counterparts. This result *might* stem from the fact that Sarsa(λ) is known to converge with linear function approximation (Perkins & Precup, 2002) while Q(λ) is known to diverge (Baird, 1995; Wiering, 2004). Additionally, NAC-TD performs worse than AC despite that natural gradients are a superior ascent direction. This result is due in part because it is unknown how to set the three step-sizes in NAC-TD, making it more difficult to use than AC. Together these observations point out the deficiency in the way new algorithms have been evaluated. That is, tuning hyperparameters hides the lack of knowledge required to use the algorithm, introducing bias that favors the new algorithm. In contrast, our method forces this knowledge to be encoded into the algorithm, leading to a more fair and reliable comparison.

6.3. Experiment Uncertainty

While the trends discussed above might hold true in general, we must quantify our uncertainty. Based on the confidence intervals given using PBP, we claim with 95% confidence that on these environments and according to our algorithm definitions, Sarsa-Parl2 and Q-Parl2 have a higher aggregate performance of average returns than all other algorithms in the experiment. It is clear that 10,000 trials per algorithm per environment is not enough to detect a unique ranking of algorithms using the nonparametric confidence intervals in PBP. We now consider alternative methods, PBP-t, and the percentile bootstrap. PBP-t replaces the nonparameteric intervals in PBP with ones based on the Student's t-distribution. We detail these methods in Appendix G. From Figure 5, it is clear that both alternative bounds are tighter and thus useful in detecting differences. Since assumptions of these bounds are different and not typically satisfied, it is unclear if they are valid.

Confidence Interval Performance						
	PBP		PBP-t		Bootstrap	
Samples	FR	SIG	FR	SIG	FR	SIG
10	0.0	0.0	1.000	0.00	0.112	0.11
30	0.0	0.0	0.000	0.00	0.092	0.37
100	0.0	0.0	0.000	0.02	0.084	0.74
1,000	0.0	0.0	0.000	0.34	0.057	0.83
10,000	0.0	0.33	0.003	0.83	0.069	0.83

Table 2. Table showing the failure rate (FR) and proportion of significant pairwise comparison (SIG) identified for $\delta = 0.05$ using different bounding techniques and sample sizes. The first column represents the sample size. The second, third, and fourth columns represent the results for PBP, PBP-t, and bootstrap bound methods respectively. For each sample size, 1,000 experiments were conducted.

To test the different bounding techniques, we estimate the failure rate of each confidence interval technique at different sample sizes. For this experiment we execute 1,000 trials of the evaluation procedure using sample sizes (trials per algorithm per environment) of 10, 30, 100, 1,000, and 10,000. There are a total of 11.14 million samples per algorithm per environment. To reduce computation costs, we limit this experiment to only include Sarsa(λ)-Parl2, Q(λ)-Parl2, AC-Parl2, and Sarsa(λ)-s. Additionally, we reduce the environment set to be the discrete environments and Mountain Car. We compute the failure rate of the confidence intervals, where a valid confidence interval will have a failure rate less than or equal to δ, e.g., for $\delta = 0.05$ failure rate should be less than $\leq 5\%$. We report the failure rate and the proportion of statistically significant pairwise comparisons in Table 2. All methods use the same data, so the results are not independent.

The PBP method has zero failures indicating it is overly conservative. The failure rate of PBP-t is expected to converge to zero as the number of samples increase due to the central limit theorem. PBP-t begins to identify significant results at a sample size of 1,000, but it is only at 10,000 that it can identify all pairwise differences.[2] The bootstrap technique has the tightest intervals, but has a high failure rate.

These results are stochastic and will not necessarily hold with different numbers of algorithms and environments. So, one should use caution in making claims that rely on either PBP-t or bootstrap. Nevertheless, to detect statistically significant results, we recommend running between 1,000, and 10,000 samples, and using the PBP-t over bootstrap.

While this number of trials seems, high it is a necessity as comparison of multiple algorithms over many environments

[2]Sarsa-Parl2 and Q-Parl2 have similar performance on discrete environments so we consider detecting 83% of results optimal.

is a challenging statistical problem with many sources of uncertainty. Thus, one should be skeptical of results that use substantially fewer trials. Additionally, researchers are already conducting many trials that go unreported when tuning hyperparameters. Since our method requires no hyperparameter tuning, researchers can instead spend the same amount of time collecting more trials that can be used to quantify uncertainty.

There are a few ways that the number of trials needed can be reduced. The first is to think carefully about what question one should answer so that only a few algorithms and environments are required. The second is to use active sampling techniques to determine when to stop generating samples of performance for each algorithm environment pair (Rowland et al., 2019). It is important to caution the reader that this process can bias the results if the sequential tests are not accounted for (Howard et al., 2018).

Summarizing our experiments, we make the following observations. Our experiments with complete algorithms show that there is still more work required to make standard RL algorithms work reliably on even extremely simple benchmark problems. As a result of our evaluation procedure, we were able to identify performance differences in algorithms that are not noticeable under standard evaluation procedures. The tests of the confidence intervals suggest that both PBP and PBP-t provide reliable estimates of uncertainty. These outcomes suggest that this evaluation procedure will be useful in comparing the performance of RL algorithms.

7. Related Work

This paper is not the first to investigate and address issues in empirically evaluating algorithms. The evaluation of algorithms has become a signficant enough topic to spawn its own field of study, known as *experimental algorithmics* (Fleischer et al., 2002; McGeoch, 2012).

In RL, there have been significant efforts to discuss and improve the evaluation of algorithms (Whiteson & Littman, 2011). One common theme has been to produce shared benchmark environments, such as those in the annual reinforcement learning competitions (Whiteson et al., 2010; Dimitrakakis et al., 2014), the Arcade Learning Environment (Bellemare et al., 2013), and numerous others which are to long to list here. Recently, there has been a trend of explicit investigations into the reproducibility of reported results (Henderson et al., 2018; Islam et al., 2017; Khetarpal et al., 2018; Colas et al., 2018). These efforts are in part due to the inadequate experimental practices and reporting in RL and general machine learning (Pineau et al., 2020; Lipton & Steinhardt, 2018). Similar to these studies, this work has been motivated by the need for a more reliable evaluation procedure to compare algorithms. The primary difference in our work to these is that the knowledge required to use an algorithm gets included in the performance metric.

An important aspect of evaluation not discussed so far in this paper is competitive versus scientific testing (Hooker, 1995). Competitive testing is the practice of having algorithms compete for top performance on benchmark tasks. Scientific testing is the careful experimentation of algorithms to gain insight into how an algorithm works. The main difference in these two approaches is that competitive testing only says *which* algorithms worked well but not *why*, whereas scientific testing directly investigates the what, when, how, or why better performance can be achieved.

There are several examples of recent works using scientific testing to expand our understanding of commonly used methods. Lyle et al. (2019) compares distributional RL approaches using different function approximation schemes showing that distributional approaches are only effective when nonlinear function approximation is used. Tucker et al. (2018) explore the sources of variance reduction in action dependent control variates showing that improvement was small or due to additional bias. Witty et al. (2018) and Atrey et al. (2020) investigate learned behaviors of an agent playing Atari 2600 games using ToyBox (Foley et al., 2018), a tool designed explicitly to enable carefully controlled experimentation of RL agents. While, at first glance the techniques developed here seems to be only compatible with competitive testing, this is only because we specified question with a competitive answer. The techniques developed here, particularly complete algorithm definitions, can be used to accurately evaluate the impact of various algorithmic choices. This allows for the careful experimentation to determine what components are essential to an algorithm.

8. Conclusion

The evaluation framework that we propose provides a principled method for evaluating RL algorithms. This approach facilitates fair comparisons of algorithms by removing unintentional biases common in the research setting. By developing a method to establish high-confidence bounds over this approach, we provide the framework necessary for reliable comparisons. We hope that our provided implementations will allow other researchers to easily leverage this approach to report the performances of the algorithms they create.

Acknowledgements

The authors would like to thank Kaleigh Clary, Emma Tosch, and members of the Autonomous Learning Laboratory: Blossom Metevier, James Kostas, and Chris Nota, for discussion and feedback on various versions of this manuscript. Additionally, we would like to thank the reviewers and meta-reviewers for their comments, which helped improved this paper.

This work was performed in part using high performance computing equipment obtained under a grant from the Collaborative R&D Fund managed by the Massachusetts Technology Collaborative. This work was supported in part by a gift from Adobe. This work was supported in part by the Center for Intelligent Information Retrieval. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the sponsor. Research reported in this paper was sponsored in part by the CCDC Army Research Laboratory under Cooperative Agreement W911NF-17-2-0196 (ARL IoBT CRA). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

References

Anderson, T. W. Confidence limits for the value of an arbitrary bounded random variable with a continuous distribution function. *Bulletin of The International and Statistical Institute*, 43:249–251, 1969.

Atrey, A., Clary, K., and Jensen, D. D. Exploratory not explanatory: Counterfactual analysis of saliency maps for deep reinforcement learning. In *8th International Conference on Learning Representations, ICLR*. OpenReview.net, 2020.

Baird, L. C. Residual algorithms: Reinforcement learning with function approximation. In Prieditis, A. and Russell, S. J. (eds.), *Machine Learning, Proceedings of the Twelfth International Conference on Machine Learning*, pp. 30–37. Morgan Kaufmann, 1995.

Balduzzi, D., Tuyls, K., Pérolat, J., and Graepel, T. Re-evaluating evaluation. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems, NeurIPS.*, pp. 3272–3283, 2018.

Barreto, A. M. S., Bernardino, H. S., and Barbosa, H. J. C. Probabilistic performance profiles for the experimental evaluation of stochastic algorithms. In Pelikan, M. and Branke, J. (eds.), *Genetic and Evolutionary Computation Conference, GECCO*, pp. 751–758. ACM, 2010.

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

Bezanson, J., Edelman, A., Karpinski, S., and Shah, V. B. Julia: A fresh approach to numerical computing. *SIAM Review*, 59(1):65–98, 2017.

Cohen, D., Jordan, S. M., and Croft, W. B. Distributed evaluations: Ending neural point metrics. *CoRR*, abs/1806.03790, 2018.

Colas, C., Sigaud, O., and Oudeyer, P. How many random seeds? Statistical power analysis in deep reinforcement learning experiments. *CoRR*, abs/1806.08295, 2018.

Csáji, B. C., Jungers, R. M., and Blondel, V. D. Pagerank optimization by edge selection. *Discrete Applied Mathematics*, 169:73–87, 2014.

Dabney, W. C. *Adaptive step-sizes for reinforcement learning*. PhD thesis, University of Massachusetts Amherst, 2014.

de Kerchove, C., Ninove, L., and Dooren, P. V. Maximizing pagerank via outlinks. *CoRR*, abs/0711.2867, 2007.

Degris, T., Pilarski, P. M., and Sutton, R. S. Model-free reinforcement learning with continuous action in practice. In *American Control Conference, ACC*, pp. 2177–2182, 2012.

Dimitrakakis, C., Li, G., and Tziortziotis, N. The reinforcement learning competition 2014. *AI Magazine*, 35(3): 61–65, 2014.

Dodge, Y. and Commenges, D. *The Oxford dictionary of statistical terms*. Oxford University Press on Demand, 2006.

Dolan, E. D. and Moré, J. J. Benchmarking optimization software with performance profiles. *Math. Program.*, 91 (2):201–213, 2002.

Dulac-Arnold, G., Mankowitz, D. J., and Hester, T. Challenges of real-world reinforcement learning. *CoRR*, abs/1904.12901, 2019.

Dvoretzky, A., Kiefer, J., and Wolfowitz, J. Asymptotic minimax character of a sample distribution function and of the classical multinomial estimator. *Annals of Mathematical Statistics*, 27:642–669, 1956.

Farahmand, A. M., Ahmadabadi, M. N., Lucas, C., and Araabi, B. N. Interaction of culture-based learning and cooperative co-evolution and its application to automatic behavior-based system design. *IEEE Trans. Evolutionary Computation*, 14(1):23–57, 2010.

Fercoq, O., Akian, M., Bouhtou, M., and Gaubert, S. Ergodic control and polyhedral approaches to pagerank optimization. *IEEE Trans. Automat. Contr.*, 58(1):134–148, 2013.

Fleischer, R., Moret, B. M. E., and Schmidt, E. M. (eds.). *Experimental Algorithmics, From Algorithm Design to Robust and Efficient Software [Dagstuhl seminar, September 2000]*, volume 2547 of *Lecture Notes in Computer Science*, 2002. Springer.

Fleming, P. J. and Wallace, J. J. How not to lie with statistics: The correct way to summarize benchmark results. *Commun. ACM*, 29(3):218–221, 1986.

Florian, R. V. Correct equations for the dynamics of the cart-pole system. *Center for Cognitive and Neural Studies (Coneural), Romania*, 2007.

Foley, J., Tosch, E., Clary, K., and Jensen, D. Toybox: Better Atari Environments for Testing Reinforcement Learning Agents. In *NeurIPS 2018 Workshop on Systems for ML*, 2018.

Geramifard, A., Dann, C., Klein, R. H., Dabney, W., and How, J. P. RLPy: A value-function-based reinforcement learning framework for education and research. *Journal of Machine Learning Research*, 16:1573–1578, 2015.

Henderson, P., Islam, R., Bachman, P., Pineau, J., Precup, D., and Meger, D. Deep reinforcement learning that matters. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18)*, pp. 3207–3214, 2018.

Hooker, J. N. Testing heuristics: We have it all wrong. *Journal of Heuristics*, 1(1):33–42, 1995.

Howard, S. R., Ramdas, A., McAuliffe, J., and Sekhon, J. S. Uniform, nonparametric, non-asymptotic confidence sequences. *arXiv: Statistics Theory*, 2018.

Islam, R., Henderson, P., Gomrokchi, M., and Precup, D. Reproducibility of benchmarked deep reinforcement learning tasks for continuous control. *CoRR*, abs/1708.04133, 2017.

Jordan, S. M., Cohen, D., and Thomas, P. S. Using cumulative distribution based performance analysis to benchmark models. In *Critiquing and Correcting Trends in Machine Learning Workshop at Neural Information Processing Systems*, 2018.

Khetarpal, K., Ahmed, Z., Cianflone, A., Islam, R., and Pineau, J. Re-evaluate: Reproducibility in evaluating reinforcement learning algorithms. 2018.

Konidaris, G., Osentoski, S., and Thomas, P. S. Value function approximation in reinforcement learning using the fourier basis. In *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, AAAI*, 2011.

Konidaris, G. D. and Barto, A. G. Skill discovery in continuous reinforcement learning domains using skill chaining. In *Advances in Neural Information Processing Systems 22.*, pp. 1015–1023. Curran Associates, Inc., 2009.

Lipton, Z. C. and Steinhardt, J. Troubling trends in machine learning scholarship. *CoRR*, abs/1807.03341, 2018.

Lucic, M., Kurach, K., Michalski, M., Gelly, S., and Bousquet, O. Are gans created equal? A large-scale study. In *Advances in Neural Information Processing Systems 31.*, pp. 698–707, 2018.

Lyle, C., Bellemare, M. G., and Castro, P. S. A comparative analysis of expected and distributional reinforcement learning. In *The Thirty-Third AAAI Conference on Artificial Intelligence*, pp. 4504–4511. AAAI Press, 2019.

Machado, M. C., Bellemare, M. G., Talvitie, E., Veness, J., Hausknecht, M. J., and Bowling, M. Revisiting the arcade learning environment: Evaluation protocols and open problems for general agents. *J. Artif. Intell. Res.*, 61:523–562, 2018.

Massart, P. The tight constant in the Dvoretzky-Kiefer-Wolfowitz inequality. *The Annals of Probability*, 18(3): 1269–1283, 1990.

McGeoch, C. C. *A Guide to Experimental Algorithmics*. Cambridge University Press, 2012.

Melis, G., Dyer, C., and Blunsom, P. On the state of the art of evaluation in neural language models. In *6th International Conference on Learning Representations, ICLR*. OpenReview.net, 2018.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M. A., Fidjeland, A., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. Human-level control through deep reinforcement learning. *Nature*, 518(7540): 529–533, 2015.

Morimura, T., Uchibe, E. . i. e. j., and Doya, K. Utilizing the natural gradient in temporal difference reinforcement learning with eligibility traces. In *International Symposium on Information Geometry and Its Applications*, pp. 256–263, 2005.

Omidshafiei, S., Papadimitriou, C., Piliouras, G., Tuyls, K., Rowland, M., Lespiau, J.-B., Czarnecki, W. M., Lanctot, M., Perolat, J., and Munos, R. α-rank: Multi-agent evaluation by evolution. *Scientific reports*, 9(1):1–29, 2019.

Page, L., Brin, S., Motwani, R., and Winograd, T. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

Perkins, T. J. and Precup, D. A convergent form of approximate policy iteration. In *Advances in Neural Information Processing Systems 15*, pp. 1595–1602. MIT Press, 2002.

Pineau, J., Vincent-Lamarre, P., Sinha, K., Larivière, V., Beygelzimer, A., d'Alché-Buc, F., Fox, E. B., and Larochelle, H. Improving reproducibility in machine learning research (A report from the NeurIPS 2019 reproducibility program). *CoRR*, abs/2003.12206, 2020.

Puterman, M. L. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley Series in Probability and Statistics. Wiley, 1994.

Reimers, N. and Gurevych, I. Reporting score distributions makes a difference: Performance study of LSTM-networks for sequence tagging. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing, EMNLP*, pp. 338–348, 2017.

Rowland, M., Omidshafiei, S., Tuyls, K., Pérolat, J., Valko, M., Piliouras, G., and Munos, R. Multiagent evaluation under incomplete information. In *Advances in Neural Information Processing Systems 3, NeurIPS*, pp. 12270–12282, 2019.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

Sutton, R. S. Generalization in reinforcement learning: Successful examples using sparse coarse coding. In *Advances in Neural Information Processing Systems 8*, pp. 1038–1044, 1995.

Sutton, R. S. and Barto, A. G. *Reinforcement learning - an introduction*. Adaptive computation and machine learning. MIT Press, 1998.

Sutton, R. S. and Barto, A. G. *Reinforcement learning: An introduction*. MIT press, 2018.

Thomas, P. Bias in natural actor-critic algorithms. In *Proceedings of the 31th International Conference on Machine Learning, ICML*, pp. 441–448, 2014.

Tucker, G., Bhupatiraju, S., Gu, S., Turner, R. E., Ghahramani, Z., and Levine, S. The mirage of action-dependent baselines in reinforcement learning. In *Proceedings of the 35th International Conference on Machine Learning, ICML*, pp. 5022–5031, 2018.

Whiteson, S. and Littman, M. L. Introduction to the special issue on empirical evaluations in reinforcement learning. *Mach. Learn.*, 84(1-2):1–6, 2011.

Whiteson, S., Tanner, B., and White, A. M. Report on the 2008 reinforcement learning competition. *AI Magazine*, 31(2):81–94, 2010.

Whiteson, S., Tanner, B., Taylor, M. E., and Stone, P. Protecting against evaluation overfitting in empirical reinforcement learning. In *2011 IEEE Symposium on Adaptive Dynamic Programming And Reinforcement Learning, ADPRL*, pp. 120–127. IEEE, 2011.

Wiering, M. Convergence and divergence in standard and averaging reinforcement learning. In *Machine Learning: ECML 2004, 15th European Conference on Machine Learning*, volume 3201 of *Lecture Notes in Computer Science*, pp. 477–488. Springer, 2004.

Williams, R. J. and Baird, L. C. Tight performance bounds on greedy policies based on imperfect value functions. 1993.

Witty, S., Lee, J. K., Tosch, E., Atrey, A., Littman, M., and Jensen, D. Measuring and characterizing generalization in deep reinforcement learning. *arXiv preprint arXiv:1812.02868*, 2018.

Appendix

A. Other Normalization Methods

A simple normalization technique is to map scores on an environment j that are in the range $[a_j, b_j]$ to $[0, 1]$, i.e., $g(x, j) \coloneqq (x - a_j)/(b_j - a_j)$ (Bellemare et al., 2013). However, this can result in normalized performance measures that cluster in different regions of $[0, 1]$ for each environment. For example, consider one environment where a the minimum is -100, the maximum is 10 and a uniform random policy can score around 0. Similarly consider a second environment where the minimum score is 10, maximum is $1{,}000$, and random gets around 20. On the first environment, algorithms will tend to have a normalized performance near 1 and in the second case most algorithms will have a normalized performance near 0. So in the second environment algorithms will likely appear worse than algorithms in the first regardless of how close to optimal they are. This means the normalized performances are not really comparable.

A different version of this approach uses the minimum and maximum mean performance of each algorithm (Bellemare et al., 2013; Balduzzi et al., 2018). Let $\hat{\mu}_{i,j}$ be the sample mean of $X_{i,j}$. Then this normalization method uses the following function, $\bar{g}(i, j) \coloneqq (\hat{\mu}_{i,j} - \min_{i'} \hat{\mu}_{i',j})/(\max_{i'} \hat{\mu}_{i',j} - \min_{i'} \hat{\mu}_{i',j})$. This sets the best algorithm's performance on each environment to 1 and the worst to 0, spreading the range of values out over the whole interval $[0, 1]$. This normalization technique does not correct for nonlinear scaling of performance. As a result algorithms could be near 0 or 1 if there is an outlier algorithm that does very well or poorly. For example, one could introduce a terrible algorithm that just chooses one action the whole time. This makes the environment seem easier as all scores would be near 1 except for this bad algorithm. We would like the evaluation procedure to be robust to the addition of poor algorithms.

An alternative normalization technique proposed by Whiteson et al. (2011) uses the probability that one algorithm outperforms another on an environment, j, i.e., $\Pr(X_{i,j} \geq X_{k,j})$. This technique is intuitive and straight forward to estimate but neglects the difference in score magnitudes. For example, consider that algorithm i always scores a 1.0 and algorithm k always scores 0.99, the probability that i is better than k is 1, but the difference between them is small, and the normalized score of 1.0 neglects this difference.

B. α-Rank and our Implementation

The α-Rank procedure finds a solution to a game by computing the stationary distribution of strategy profiles when each player is allowed to change their strategy. This is done by constructing a directed graph where nodes are pure strategies and edges have weights corresponding to the probability that one of the players switches strategies. This graph can be represented by a Markov matrix, $C \in [0, 1]^{|\mathcal{S}| \times |\mathcal{S}|}$. The entry $C_{s,s'}$ corresponds to a probability of switching from a strategy s to s'. Only one player is allowed to change strategies at a time, so the possible transitions for a strategy $s = (i, (j, k))$, are any strategies $s' = (i', (j, k))$ or $s' = (i, (j', k'))$ for all $i', k' \in \mathcal{A}$ and $j' \in \mathcal{M}$.

The typical α-Rank procedure uses transition probabilities, $C_{s,s'}$, that are based on a logistic transformation of the payoff difference $u_l(s') - u_l(s)$. These differences are scaled by a parameter α and as α approaches ∞, the transition matrix approximates the Markov Conley chain (MCC), which is the motivation for using α-Rank as a solution concept for games. See the work of Omidshafiei et al. (2019) for more detailed information. The entries of the matrix for valid transitions are:

$$C_{s,s'} = \begin{cases} \eta \frac{1 - \exp\left(-\alpha(u_l(s') - u_l(s))\right)}{1 - \exp\left(-\alpha n(u_l(s') - u_l(s))\right)} & \text{if } u_l(s') = u_l(s) \\ \frac{\eta}{n} & \text{if } u_l(s') = u_l(s) \end{cases} \quad \text{and } C_{s,s} = 1 - \sum_{s' \neq s} C_{s,s'},$$

where $\eta = (\sum_k |\mathcal{S}^k| - 1)^{-1}$, l represents the player who switched from strategy s to s', n is the population constant (we set it to 50 following the prior work). The equilibrium over strategies is then given by the stationary distribution d of the Markov chain induced by C, i.e., d is a distribution such that $d = d\tilde{C}$. The equilibrium solution p^*, q^* are then the sum of probabilities in d for each strategy, i.e., $p^*_{s_1} = \sum_{s_2 \in \mathcal{S}_2} d_{s_1,s_2}$ and $q^*_{s_2} = \sum_{s_1 \in \mathcal{S}_1} d_{s_1,s_2}$. The aggregate performance can then be computed using q^* as in (1).

Theoretically, the hyperparameter α could be chosen arbitrarily high and the matrix C would still be irreducible, i.e., for all $s \in \mathcal{S}$, $d_s > 0$ and d is unique. However, due to numerical precision issues, a high value of α sets transition probabilities to zero for some dominated strategies, i.e., $u_l(s') < u_l(s)$, which can result in a matrix that is reducible. The suggested method to chose α is to tune it on a logarithmic scaled to find the highest value such that the transition matrix, C, is still irreducible (Omidshafiei et al., 2019).

This strategy works when the payoffs are known, but when they represent empirical samples of performance, then the value of α chosen will depend on the empirical payoff functions. Setting α based solely on the empirical payoffs could introduce bias to the matrix based on that sample. So we need a different solution without a data dependent hyperparameter.

In the MCC graph construction, all edges leading to strategies with strictly greater payoffs have the same positive weight. All edges that lead to strategies with the same payoff have he same weight but less than that of the strictly greater payoff. There are no transitions to strategies with worse payoffs. As $\alpha \to \infty$ the transitions probabilities quickly saturate to η if $u_l(s') > u_l(s)$ and 0 if $u_l(s') < u_l(s)$. So we construct the transition matrix, C, differently using the saturation values to set the transition probabilities C is close to the MCC construction. The entries for C that represent valid transitions in the graph are:

$$C_{s,s'} \coloneqq \begin{cases} \eta & \text{if } u_l(s') > u_l(s) \\ \frac{\eta}{m} & \text{if } u_l(s') = u_l(s) \\ 0 & \text{otherwise} \end{cases} \quad C_{s,s} \coloneqq 1 - \sum_{s' \neq s} C_{s,s'}. \tag{2}$$

However, this often makes the transition matrix reducible, i.e., the stationary distribution might have mass on only one strategy. To ensure C is irreducible we follow the damping approach used in PageRank (Page et al., 1999), i.e., $\tilde{C} = \gamma C + (1-\gamma)(1/|\mathcal{S}|)$, where $\gamma \in (0,1)$ is a hyperparameter and $1-\gamma$ represents the probability of randomly switching to any strategy in $\mathcal{S}$. During a Monte-Carlo simulation of transitions through $\tilde{C}$, states will transition from s to s' according to C, but with probability $1-\gamma$ the transition ignores C and switches to some strategy $s' \in \mathcal{S}$ chosen uniformly at random.

For $\gamma = 1$ the matrix is unchanged and represents the MCC solution, but is reducible. For γ near one, the stationary distribution will be similar to the solution given by the MCC solution with high weight placed on dominate strategies and small weight given to weak ones. As $\gamma \to 0$ the stationary distribution becomes more uniform as it is only considering shorter sequences of transitions before a random switch occurs. This method differs from the infinite-α approach presented by Rowland et al. (2019), but in the limit as $\gamma \to 1$ and $\alpha \to \infty$, the solutions have small differences. The approach using γ has a benefit in that there is no data dependent hyperparameter and it has a simple interpretation.

We chose to set $\gamma = (|\mathcal{S}|-1)/(|\mathcal{S}|)$ so that the expected number of transitions to occur before a random jump is $|\mathcal{S}|$. This allows for propagation of transition probabilities to cover every strategy combination. We could have chosen to set γ near one, e.g., $\gamma = 1 - 10^{-8}$, but this would make the computation of the confidence intervals take longer. This is because optimizing the C within confidence intervals $[C^-, C^+]$ (defined in the next section) is equivalent finding the optimal value function of a Markov decision process (MDP) with a discount parameter of γ. See the work of de Kerchove et al. (2007); Fercoq et al. (2013); Csáji et al. (2014) for more information on this connection. Solving and MDP with a discount γ near 1.0 causes the optimization process of value iteration and policy iteration to converge slower than if γ is small. So we chose γ such that it could still find solutions near the MCC solution, but remain computationally efficient.

Using this new definition of C we use the following alternative but equivalent method to compute the aggregate performance more efficiently (Fercoq et al., 2013):

$$y_i = \frac{1-\gamma}{|\mathcal{S}|} \sum_{s \in \mathcal{S}} v(s) \quad v = (I - \gamma C)^{-1} R_i, \tag{3}$$

where $R_i \in \mathbb{R}^{|\mathcal{S}|}$ is a vector with entries $R_i(s) \coloneqq \mathbf{E}[F_{X_{k,j}}(X_{i,j})]$ with $s = (s_1, s_2)$ and $s_2 = (j,k)$. Notice that s_1 is ignored because i is already specified by R_i.

C. Confidence Intervals on the Aggregate Performance

In this section, we detail the PBP procedure for computing confidence intervals Y^- and Y^+ on the aggregate performance y and prove that they hold with high probability. That is, we show that for any confidence level $\delta \in (0, 0.5]$;

$$\Pr(\forall i \in \mathcal{A},\ y_i \in [Y_i^-, Y_i^+]) \geq 1 - \delta.$$

We will first describe the PBP procedure to compute confidence intervals and then prove that they valid. A list of the symbols used in the construction of confidence intervals and their description are provided in Table 3 to refresh the reader. The steps to compute the confidence intervals are outlined in Algorithm 1.

Symbol List	
Symbol	**Description**
$\mathcal{A}$	set of algorithms in the evaluation
$\mathcal{M}$	set of environments in the evaluation
$X_{i,j}$	random variable representing performance of algorithm i on environment j
$T_{i,j}$	number of samples of performance for algorithm i on environment j
$x_{i,j,t}$	the t^{th}sample of performance of algorithm i on environment j and sorted such that $x_{i,j,t-1} \leq x_{i,j,t}$
D	data set containing all samples of performance for each algorithm on each environment
$y \in \mathbb{R}^{\|\mathcal{A}\|}$	y_i is the aggregate performance for each algorithm i
$Y^-, Y^+ \in \mathbb{R}^{\|\mathcal{A}\|}$	lower and upper confidence intervals on y computed using D
$\delta \in (0, 0.5]$	confidence level for the aggregate performance
$F_{X_{i,j}}$	cumulative distribution function (CDF) of$X_{i,j}$ and is also used for normalization
$\hat{F}_{X_{i,j}}$	empirical cumulative distribution function constructed using samples $x_{i,j,\cdot}$
$F^-_{X_{i,j}}, F^+_{X_{i,j}}$	lower and upper confidence intervals on $F_{X_{i,j}}$ computed using D
$z_{i,j,k}$	performance of algorithm i, i.e., $z_{i,j,k} = \mathbf{E}[F_{X_{k,j}}(X_{i,j})]$
$Z^-_{i,j,k}, Z^+_{i,j,k}$	lower and upper confidence intervals on $z_{i,j,k}$ computed using D.
$s_1 \in \mathcal{S}_1$	strategy for player p where $\mathcal{S}_1 = \mathcal{A}$ and s_1 is often denoted using i
$s_2 \in \mathcal{S}_2$	strategy for player q where $\mathcal{S}_2 = \mathcal{M} \times \mathcal{A}$ and s_2 is often denoted using (j, k)
$s \in \mathcal{S}$	joint strategy where $\mathcal{S} = \mathcal{S}_1 \times \mathcal{S}_2$, and $s = (s_1, s_2)$ is often denoted as $(i, (j, k))$
$p \in \Delta(\mathcal{S}_1)$	strategy for player p represented as a distribution over $\mathcal{S}_1$
$q \in \Delta(\mathcal{S}_2)$	strategy for player q represented as a distribution over $\mathcal{S}_2$
$u_p(s)$	payoff for player p when s is played, i.e.,$u_p(s) = \mathbf{E}[F_{X_{k,j}}(X_{i,j})]$
$u_q(s)$	payoff for player q when s is played, i.e., $u_q(s) = -u_p(s)$
$u^-_l(s), u^+_l(s)$	confidence intervals on $u_l(s)$ for player $l \in \{p, q\}$ computed using D

Table 3. List of symbols used to create confidence intervals on the aggregate performance.

Recall that the aggregate performance for an algorithm i is

$$y_i \coloneqq \sum_{j=1}^{|\mathcal{M}|} \sum_{k=1}^{|\mathcal{A}|} q^*_{j,k} \mathbf{E}[F_{X_{k,j}}(X_{i,j})],$$

where q^* is the equilibrium solution to the game specified in Section 4.2. To compute valid confidence intervals Y^-, Y^+ on y using a dataset D, the uncertainty of q^* and mean normalized performance $z_{i,j,k} = \mathbf{E}[F_{X_{k,j}}(X_{i,j})]$. PBP accomplishes this by three primary steps. The first step is to compute confidence intervals $Z^-_{i,j,k}, Z^+_{i,j,k}$ on $z_{i,j,k}$ such that

$$\Pr\left(\forall (i,j,k) \in \mathcal{A} \times \mathcal{M} \times \mathcal{A},\ z_{i,j,k} \in [Z^-_{i,j,k}, Z^+_{i,j,k}]\right) \geq 1 - \delta.$$

The second step is to compute the uncertainty set $\mathcal{Q}$ containing all possible q^* that are compatible with Z^- and Z^+. The last step is to compute the smallest and largest possible aggregate performances for each algorithm over these sets, i.e.,

$$Y^-_i = \min_{q \in \mathcal{Q}} \sum_{j=1}^{|\mathcal{M}|} \sum_{k=1}^{|\mathcal{A}|} q_{j,k} Z^-_{i,j,k} \text{ and } \quad Y^+_i = \max_{q \in \mathcal{Q}} \sum_{j=1}^{|\mathcal{M}|} \sum_{k=1}^{|\mathcal{A}|} q_{j,k} Z^+_{i,j,k}.$$

PBP follows this process, except in the last two steps $\mathcal{Q}$ is never explicitly constructed to improve computational efficiency. Intuitively, the procedure provides valid confidence intervals because all values to compute the aggregate performance depend on the normalized performance. So by guaranteeing with probability at least $1 - \delta$ that the true mean normalized performances will be between Z^- and Z^+, then so long as the the upper (lower) confidence interval computed is at least as large (small) as the maximum (minimum) of the aggregate score for any setting of $z \in [Z^-, Z^+]$, the confidence intervals will be valid.

Algorithm 1 Performance Bound Propagation (PBP)

```
1: Input: dataset $D$ containing samples of performance and a confidence level $\delta \in (0, 0.5])$
2: Output: $Y^-$, $Y^+$ confidence intervals on the aggregate performance
3: $\delta' \leftarrow \delta/(|\mathcal{A}||\mathcal{M}|)$;
4: $\texttt{sort\_ascending}(\{x_{i,j,t}\}_{t=1}^{T_{i,j}})$;
5: // Compute confidence intervals for the CDFs
6: for $i, j \in \mathcal{A} \times \mathcal{M}$ do
7:     $F^-_{X_{i,j}}, F^+_{X_{i,j}} \leftarrow \texttt{dkw\_bound}(\{x_{i,j,t}\}_{t=1}^{T_{i,j}}, \delta')$; // computation shown in (4)
8: end for
9: // Compute confidence intervals on the mean normalized performance
10: for $i, j, k \in \mathcal{A} \times \mathcal{M} \times \mathcal{A}$ do
11:     $Z^-_{i,j,k} \leftarrow F^-_{X_{k,j}}(x_{i,j,T}) - \sum_{t=0}^{T-1} \left[F^-_{X_{k,j}}(x_{i,j,t+1}) - F^-_{X_{k,j}}(x_{i,j,t})\right] F^+_{X_{i,j}}(x_{i,j,t})$;
12:     $Z^+_{i,j,k} \leftarrow F^+_{X_{k,j}}(x_{i,j,T+1}) - \sum_{t=1}^{T} \left[F^+_{X_{k,j}}(x_{i,j,t+1}) - F^+_{X_{k,j}}(x_{i,j,t})\right] F^-_{X_{i,j}}(x_{i,j,t})$;
13: end for
14: // Construct game quantities
15: $\mathcal{S} = \mathcal{A} \times (\mathcal{M} \times \mathcal{A})$;  strategy profile set
16: $\gamma \leftarrow \frac{|\mathcal{S}|-1}{|\mathcal{S}|}$
17: $C^-, C^+ \leftarrow \texttt{bound\_markov\_matrix}(Z^-, Z^+)$ as defined in (7).
18: // Optimize aggregate performance over all possible $C \in [C^-, C^+]$
19: for $i \in \mathcal{A}$ do
20:     $v \leftarrow \texttt{find\_optimal\_valuefunction}(C^-, C^+, R_i = -Z^-_{i,\cdot,\cdot})$; // solve (8)
21:     $Y^-_i \leftarrow \frac{(1-\gamma)}{|\mathcal{S}|} \sum_{s \in \mathcal{S}} |v(s)|$
22:     $v \leftarrow \texttt{find\_optimal\_valuefunction}(C^-, C^+, R_i = Z^+_{i,\cdot,\cdot})$; // solve (8)
23:     $Y^+_i \leftarrow \frac{(1-\gamma)}{|\mathcal{S}|} \sum_{s \in \mathcal{S}} |v(s)|$
24: end for
```

We break the rest of this section into two subsections. The first subsection discusses constructing the confidence intervals on the mean normalized performance and proving their validity. The second subsection describes how to construct the confidence intervals on the aggregate performance proves their validity.

C.1. Confidence intervals on the normalized performance

The normalized performance $z_{i,j,k} = \mathbf{E}[F_{X_{k,j}}(X_{i,j})]$ has two unknowns, $F_{X_{k,j}}$ and the distribution of $X_{i,j}$. To compute confidence intervals on $z_{i,j,k}$ for all i, j, k, confidence intervals are needed on the output on all distribution functions $F_{X_{i,j}}$. The confidence intervals on the distributions can then be combined to get confidence intervals on $z_{i,j,k}$.

To compute confidence intervals on $F_{X_{i,j}}$ we assume that $X_{i,j}$ is bounded on the interval $[a_j, b_j]$ for all $i \in \mathcal{A}$ and $j \in \mathcal{M}$. Let $\hat{F}_{X_{i,j}}$ be the empirical CDF with

$$\hat{F}_{X_{i,j}}(x) := \frac{1}{T_{i,j}} \sum_{t=1}^{T_{i,j}} \mathbf{1}_{x_{i,j,t} \leq x},$$

where $T_{i,j}$ is the number of samples of $X_{i,j}$, $x_{i,j,t}$ is the t^{th} sample of $X_{i,j}$, and $\mathbf{1}_A = 1$ if event A is true and 0 otherwise. Using the Dvoretzky–Kiefer–Wolfowitz (DKW) inequality (Dvoretzky et al., 1956) with tight constants (Massart, 1990), we define $F^-_{X_{i,j}}$ and $F^+_{X_{i,j}}$ to be the lower and upper confidence intervals on $F_{X_{i,j}}$, i.e.,

$$F^+_{X_{i,j}}(x) \coloneqq \begin{cases} 1 & \text{if } x \geq b \\ \min(1.0, \hat{F}_{X_{i,j}}(x) + \epsilon) & \text{if } a \leq x < b \\ 0 & \text{if } x < a \end{cases} \qquad F^-_{X_{i,j}}(x) \coloneqq \begin{cases} 1 & \text{if } x \geq b \\ \max(0.0, \hat{F}_{X_{i,j}}(x) - \epsilon) & \text{if } a \leq x < b \\ 0 & \text{if } x < a \end{cases} \quad \text{and } \epsilon = \sqrt{\frac{\ln \frac{2}{\delta'}}{2T_{i,j}}}, \tag{4}$$

where $\delta' \in (0, 0.5]$ is a confidence level and we use $\delta' = \delta/(|\mathcal{A}||\mathcal{M}|)$. By the DKW inequality with tight constants the following property is known:

Property 1 (DKW with tight constants confidence intervals)**.**

$$\Pr\left(\forall x \in \mathbb{R}, \;\; F_{X_{i,j}}(x) \in [F^-_{X_{i,j}}(x), F^+_{X_{i,j}}(x)]\right) \geq 1 - \delta'.$$

Proof. See the works of Dvoretzky et al. (1956) and Massart (1990). □

Further, by the union bound we have that

$$\Pr\left(\forall i \in \mathcal{A}, \forall j \in \mathcal{M}, \forall x \in \mathbb{R}, \;\; F_{X_{i,j}}(x) \in [F^-_{X_{i,j}}(x), F^+_{X_{i,j}}(x)]\right) \geq 1 - \delta. \tag{5}$$

To construct confidence intervals on the mean normalized performance, we will use Anderson's inequality (Anderson, 1969). Let X be a bounded random variable on $[a, b]$, with sorted samples $x_1 \leq x_2 \leq \cdots \leq x_T$, $x_0 = a$, and $x_{T+1} = b$. Let $g\colon \mathbb{R} \to \mathbb{R}$ be a monotonically increasing function. Anderson's inequality specifies for a confidence level $\delta \in (0, 0.5]$ the following high confidence bounds on $\mathbf{E}[g(X)]$:

$$\begin{aligned} \mathbf{E}[g(X)] &\geq g(x_T) && - \sum_{t=0}^{T-1} [g(x_{t+1}) - g(x_t)]\, F^+_X(x_t) \\ \mathbf{E}[g(X)] &\leq g(x_{T+1}) && - \sum_{t=1}^{T} [g(x_{t+1}) - g(x_t)]\, F^-_X(x_t), \end{aligned}$$

where $F^{+/-}_X$ uses the DKW inequality with tight constants and as defined in (4).

Anderson's inequality can be used to bound the mean normalized performance since $F_{X_{k,j}}$ is a monotonically increasing function and $\delta \in (0, 0.5]$. Since $F_{X_{k,j}}$ is unknown, we replace g in Anderson's inequality with $F^-_{X_{k,j}}$ for the lower bound and $F^+_{X_{k,j}}$ for the upper bound. This gives the following confidence intervals for $z_{i,j,k}$:

$$\begin{aligned} Z^-_{i,j,k} &= F^-_{X_{k,j}}(x_{i,j,T}) && - \sum_{t=0}^{T-1} \left[F^-_{X_{k,j}}(x_{i,j,t+1}) - F^-_{X_{k,j}}(x_{i,j,t})\right] F^+_{X_{i,j}}(x_{i,j,t}) \\ Z^+_{i,j,k} &= F^+_{X_{k,j}}(x_{i,j,T+1}) && - \sum_{t=1}^{T} \left[F^+_{X_{k,j}}(x_{i,j,t+1}) - F^+_{X_{k,j}}(x_{i,j,t})\right] F^-_{X_{i,j}}(x_{i,j,t}), \end{aligned} \tag{6}$$

where $T = T_{i,j}$, $x_{i,j,0} = a_j$, and $x_{i,j,T+1} = b_j$. We now prove the following lemma:

Lemma 1. *If Z^- and Z^+ are computed by (6), then:*

$$\Pr\left(\forall i, k \in \mathcal{A}, \forall j \in \mathcal{M}, \;\; z_{i,j,k} \in [Z^-_{i,j,k}, Z^+_{i,j,k}]\right) \geq 1 - \delta.$$

Proof. By Anderson's inequality we know that $Z^+_{i,j,k}$ is an high confidence upper bound on $\mathbf{E}[F^+_{X_{k,j}}(X_{i,j})]$ and similarly $Z^-_{i,j,k}$ is a high confidence lower bound on $\mathbf{E}[F^+_{X_{k,j}}(X_{i,j})]$, i.e.,

$$\Pr\left(\mathbf{E}[F^-_{X_{k,j}}(X_{i,j})] \geq Z^-_{i,j,k}\right) \geq 1 - \delta'/2$$
$$\Pr\left(\mathbf{E}[F^+_{X_{k,j}}(X_{i,j})] \leq Z^+_{i,j,k}\right) \geq 1 - \delta'/2.$$

By Property 1 we know that $\Pr\left(\forall x \in \mathbb{R},\ \ F_{X_{k,j}}(x) \in [F^-_{X_{k,j}}(x), F^+_{X_{k,j}}(x)]\right) \geq 1 - \delta'$, thus

$$\Pr\left(\forall i, k \in \mathcal{A}, \forall j \in \mathcal{M},\ \ z_{i,j,k} \in [Z^-_{i,j,k}, Z^+_{i,j,k}]\right) \geq 1 - 2\delta',$$

where $2\delta'$ comes from combining the failure rates of confidence intervals on the CDFs $F_{X_{i,j}}$ and $F_{X_{k,j}}$. The confidence intervals on the mean normalized performances can only fail if the confidence intervals on CDFs fail. As stated in (5), all confidence intervals on the CDFs contain the true CDFs with probability at least $1 - \delta$. Thus, all mean normalized performances hold with probability at least $1 - \delta$. □

The confidence intervals given by Z^- and Z^+ are guaranteed to hold for $T \geq 1$, and $\delta \in (0, 0.5]$, but are often conservative requiring a large number samples to identify a statistically meaningful result. So we empirically test alternatives that have either stricter assumptions or weaker theoretical justification.

C.2. Confidence intervals on the aggregate performance

In this section we will provide details on how to compute the confidence intervals on the aggregate performance using the confidence intervals on the mean normalized performance and then prove that they hold with high confidence. To construct confidence intervals and prove their validity we will make the following steps. First, we show that for a fixed weighting q that valid confidence intervals can be computed directly by using interval arithmetic. Second, we describe how to characterize the uncertainty of the game. Third, we make a connection between aggregate performance using the equilibrium solution q^* and the optimal average reward for a Markov decision process. Lastly we describe an optimization procedure for computing the optimal average reward, which corresponds to finding the lower and upper confidence intervals.

Before discussing how to bound the aggregate performance using the game theoretic solution, consider the case when weights q can be any probability distribution over algorithms and environments chosen before the experiment begins. Let weights $q \in [0,1]^{|\mathcal{A}| \times |\mathcal{M}|}$, such that $\sum_{j \in \mathcal{M}} \sum_{k \in \mathcal{A}} q_{j,k} = 1$. Let $\tilde{y}_i = \sum_{j \in \mathcal{M}} \sum_{k \in \mathcal{A}} q_{j,k} z_{i,j,k}$ be the aggregate performance for an algorithm $i \in \mathcal{A}$. The corresponding confidence intervals for $\tilde{y}_i$ are $\tilde{Y}_i^- = \sum_{j \in \mathcal{M}} \sum_{k \in \mathcal{A}} q_{j,k} Z^-_{i,j,k}$ and $\tilde{Y}_i^+ = \sum_{j \in \mathcal{M}} \sum_{k \in \mathcal{A}} q_{j,k} Z^+_{i,j,k}$.

Lemma 2. *If weights q are independent of the data D, then:*

$$\Pr\left(\forall i \in \mathcal{A},\ \tilde{y}_i \in [\tilde{Y}_i^-, \tilde{Y}_i^+]\right) \geq 1 - \delta$$

Proof. Applying the result of Lemma 1, all confidence intervals produced by $Z^-_{i,j,k}$ and $Z^+_{i,j,k}$ contain $z_{i,j,k}$ with probability $1 - \delta$. So interval arithmetic can be used to compute confidence intervals on the aggregate performance without changing the probability of failure, i.e., $\tilde{Y}_i^- = \sum_{j \in \mathcal{M}} \sum_{k \in \mathcal{A}} q_{j,k} Z^-_{i,j,k}$ and $\tilde{Y}_i^+ = \sum_{j \in \mathcal{M}} \sum_{k \in \mathcal{A}} q_{j,k} Z^+_{i,j,k}$ for each algorithm $i \in \mathcal{A}$. With these intervals then

$$\begin{aligned}\Pr\left(\forall i \in \mathcal{A},\ \tilde{y}_i \in [\tilde{Y}_i^-, \tilde{Y}_i^+]\right) &\geq 1 - \Pr\left(\bigcup_{i,k \in \mathcal{A}, j \in \mathcal{M}} z_{i,j,k} \notin [Z^-_{i,j,k}, Z^+_{i,j,k}]\right)\\ &\geq 1 - \delta.\end{aligned}$$

□

This method of aggregating performance highlights how the uncertainty of normalized scores propagates to the confidence intervals of the aggregate performance for a fixed weighting. Next we will show how to compute confidence intervals on y when using the dynamic weighting produce by the equilibrium solution to the two player game.

Instead of considering all possible equilibrium solutions q^*, recall that y_i depends on the Markov matrix C as shown in (3). To construct confidence intervals on y_i the uncertainty in creating the matrix C as defined in (2) needs to be considered (Rowland et al., 2019). The definition of C assumes certainty of payoffs of each strategy, but the empirical payoffs have uncertainty corresponding to Z^- and Z^+, i.e., for $s = (i, (j, k))$, $u_p^-(s) = Z^-_{i,j,k}$, $u_p^+(s) = Z^+_{i,j,k}$, $u_q^-(s) = -Z^+_{i,j,k}$, $u_q^+(s) = -Z^-i, j, k$. As a result, when the payoff confidence intervals overlap for two strategies s and s' this creates uncertainty in C. We define $C^-, C^+ \in [0, 1]^{|\mathcal{S}|\times|\mathcal{S}|}$ as the lower and upper confidence intervals on C with entries

$$C^-_{s,s'}, C^+_{s,s'} \coloneqq \begin{cases} (\eta, \eta) & \text{if } u_l^-(s') > u_l^+(s) \\ (0,0) & \text{if } u_l^-(s) > u_l^+(s') \\ (\frac{\eta}{n}, \frac{\eta}{n}) & \text{if } u_l^{+/-}(s) = u_l^{+/-}(s') \\ (0, \eta) & \text{otherwise} \end{cases} \quad \forall s' \in \mathcal{S} \setminus \{s\}$$

$$C^-_{s,s}, C^+_{s,s} \coloneqq \left(1 - \sum_{s'\neq s} C^+_{s,s'},\ 1 - \sum_{s'\neq s} C^-_{s,s'}\right) \tag{7}$$

To get the bounds on the aggregate performance over all possible C in these intervals the uncertainty of the stationary distribution on $\tilde{C} = \gamma C + (1-\gamma)(1/|\mathcal{S}|)$ has to be consider. This can be accomplished by first computing the minimum and maximum values of the stationary distribution for each strategy (Rowland et al., 2019) and then finding the minimum and maximum aggregate performances for all possible stationary distributions in this limits. However, individually computing these two quantities leads to looser bounds than directly estimating the minimum and maximum aggregate performance over all possible C because it ignores the correlations in the confidence intervals of the stationary distribution. To compute the minimum and maximum aggregate performance over all possible C, we need to make a connection between the average performance using the stationary distribution of $\tilde{C}$ and the average performance before termination on C.

Let d be a stationary distribution over strategy profiles $\mathcal{S}$ induce by the Markov matrix $\tilde{C} = \gamma C + (1-\gamma)(1/|\mathcal{S}|)$. Let q be the distribution of strategies for player q contained in d.

Lemma 3. *The following are equivalent*

$$y_i = \sum_{(j,k)\in\mathcal{S}_2} q_{j,k} z_{i,j,k}$$
$$y_i = \frac{1-\gamma}{|\mathcal{S}|}\sum_{s\in\mathcal{S}} v(s),$$

where $v = (I - \gamma C)^{-1} R_i$, $R_i \in \mathbb{R}^{|\mathcal{S}|}$ *such that* $R_i(s) = z_{i,j,k}$ *for* $s = (\cdot, (j, k))$.

Proof. We know that $q_{j,k} = \sum_{i\in\mathcal{S}_1} d(i, (j, k))$. This implies $y_i = \sum_{(j,k)\in\mathcal{S}_2} q_{j,k} z_{i,j,k} = \sum_{s\in\mathcal{S}} d(s) R_i(s)$. Applying 8.2.12 of Puterman (1994), we have the relation

$$v + y_i = R_i + \tilde{C}v$$
$$v + y_i = R_i + \gamma C v + \frac{(1-\gamma)}{|S|}\mathbf{1}^\top v,$$

where here $\mathbf{1} \in \mathbb{R}^{|\mathcal{S}|}$ is a vector of all ones. Then for $v = (I - \gamma C)^{-1} R_i$, $y_i = \frac{(1-\gamma)}{|S|}\mathbf{1}^\top v$. □

Finally, using techniques developed by Fercoq et al. (2013), the lower and upper bounds of the aggregate performance y_i can be bound by solving the following optimization problem:

$$\begin{aligned} \min_v \quad & \sum_{s\in\mathcal{S}} v(s) \\ \text{s.t.} \quad & v(s) \geq R_i(s) + \gamma \sum_{s'\in\mathcal{S}} C_{s,s'} v(s') \\ & C^- \leq C \leq C^+ \\ & \sum_{s'\in\mathcal{S}} C_{s,s'} = 1 \quad \forall s \in \mathcal{S}, \end{aligned} \tag{8}$$

where C is a free variable, $R_i(s) = -Z^-_{i,j,k}$, and $R_i(s) = Z^+_{i,j,k}$ are used to obtain the lower and upper bounds on y_i, respectively. Both bounds are computed as $Y_i^{+/-} = (1-\gamma)(1/|\mathcal{S}|)\sum_{s\in\mathcal{S}}|v(s)|$ using the respective solutions v. The absolution value of v is taken to account for the negativity of $R_i = -Z^-$. We compute the solution v in polynomial time using policy iteration based approach similar to Fercoq et al. (2013), but modify their algorithm to fit our matrix C. We detail our exact method in Appendix D.

Now to prove the main result we will use the previous lemmas in connection with PBP.

Theorem 1. *If* $(Y^-, Y^+) = PBP(D, \delta)$*, then*

$$\Pr\left(\forall i \in 1, 2, \dots, |\mathcal{A}|,\ y_i \in [Y_i^-, Y_i^+]\right) \geq 1 - \delta.$$

Proof. From Lemma 1 we know that Z^- and Z^+ are valid $1-\delta$ confidence intervals on z. Thus, applying Lemma 2 we know that a valid $1-\delta$ confidence intervals can be computed by a weighted sum of lower and upper bounds Z^- and Z^+, for any joint probability distribution q over environments and algorithms. Through Lemma 3 this is equivalent to assuming fixed Markov matrix C. The true matrix C is unknown, so the minimum and maximum intervals need to be found over a set $\mathcal{C}$ that contains all transition matrices that are compatible with Z^- and Z^+. Let $\mathcal{C} = \prod_{s\in\mathcal{S}}\mathcal{C}_s$, where $\mathcal{C}_s$ is the polytope of transition probabilities starting from strategy profile $s \in \mathcal{S}$, i.e.,

$$\mathcal{C}_s = \{C_{s,\cdot} : \forall s' \in \mathcal{S},\ C_{s,s'} \in [C^-_{s,s'}, C^+_{s,s'}], \sum_{s'\in\mathcal{S}} C_{s,s'} = 1\}.$$

The minimum and maximum aggregate values Y^- and Y^+ computed over $\mathcal{C}$ can be found in polynomial time using linear programming, value iteration, or policy iteration (Fercoq et al., 2013). PBP finds the minimum and maximum confidence intervals over all $C \in \mathcal{C}$, thus, they represent valid $1-\delta$ confidence intervals for each algorithm. □

D. Policy Iteration for Bounding the Aggregate Performance

This section we detail our approach for optimizing the aggregate performance over the uncertainty of the Markov matrix C to compute confidence intervals. Recall that the upper and lower high-confidence bounds on the aggregate performance for algorithm i can be found by solving the optimization problem in 8. Alternatively one can use a Dynamic Programming approach either using value iteration or policy iteration for more efficient optimization (Fercoq et al., 2013). Using value iteration has better scaling to the size of the state space in the optimization problem than policy iteration. However, we found that in small and moderate sized problems policy iteration was sufficient and used it in our experiments.

Our method is similar to that of Fercoq et al. (2013, Algorithm 1), except that we use policy iteration instead of value iteration and a modification to the dynamic programming operator since the transition probabilities in our problem do not depend on the number of out going edges. Algorithm 2 shows pseudocode of the policy iteration to find the the optimal value function v^* through the modification of the matrix $C \in \mathcal{C}$. The algorithm takes as input lower and upper bounds $C^-, C^+ \in \mathbb{R}^{|\mathcal{S}|\times|\mathcal{S}|}$ on C, a reward function $R \in \mathbb{R}^{|\mathcal{S}|}$, a discount factor $\gamma \in (0,1)$, a tolerance on the distance to optimal, $tol \in \mathbb{R}$, and the maximum number of iterations $max\_itrs \in \mathbb{N}^+$. We set $\gamma = (1 - |\mathcal{S}|)/|\mathcal{S}|$ and $tol = 1 \times 10^{-7}$, $max\_itrs = 400$ as default parameters. When finding the upper confidence interval, y_i^+, for algorithm i, R is a vector such that $R(s) = Z^+_{i,j,k}$ for all $s = (\cdot, (j,k)) \in \mathcal{S}$. Similarly, when finding y_i^-, R is such that $R(s) = -Z^-_{i,j,k}$.

Algorithm 2 has two main steps. First, greedily optimize C with respect to the current value function v, i.e., for all $s \in \mathcal{S}$

$$C_s = \operatorname*{arg\,max}_{C_s\in\mathcal{C}_s} R(s) + \gamma C_s v.$$

We provide pseudocode for this step in Algorithm 3. The second step is a value function update, which we compute exactly by solving the system of linear equations $v = R + \gamma C v$, by setting $v = (I - \gamma C)^{-1}R$. These steps repeat until C stops changing or a bound on the maximum absolute error in aggregate performance is below some threshold. The confidence interval on the aggregate performance is then returned as $(1-\gamma)/|\mathcal{S}|\sum_{s\in\mathcal{S}}|v(s)|$. Since C is sparse optimization to the code can be made to drastically speed up computation when $\mathcal{S}$ is large. We make some of these modifications in our implementation.

Due to small numerical errors this version of policy iteration may not keep the same policy, C, between successive iterations when at the optimal solution. To ensure that the procedure stops in a reasonable time and closely approximates the true

Algorithm 2 Policy Iteration for aggregate performance optimization

```
Input: $C^-, C^+ \in \mathbb{R}^{|\mathcal{S}|\times|\mathcal{S}|}$, $R \in \mathbb{R}^{|\mathcal{S}|}$, $\gamma \in (0,1) = (1-|\mathcal{S}|)/|\mathcal{S}|$, $tol \in \mathbb{R} = 1\times 10^{-7}$, $max\_itrs \in \mathbb{N}^+ = 400$
Initialize: $C \leftarrow C^-$, $v \leftarrow \mathbf{0} \in \mathbb{R}^{|\mathcal{S}|}$
$changed \leftarrow$ True
$iteration \leftarrow 0$
while changed do
  $changed \leftarrow$ False
  $iteration \leftarrow iteration + 1$
  if $iteration \geq max\_itrs$ then
    break
  end if
  for $s \in \mathcal{S}$ do
    $C'_s \leftarrow$ update_transition_row$(C^-_s, C^+_s, R(s), v, \gamma)$
    if $||C_s - C'_s||_\infty \geq 1\times 10^{-8}$ then
      $changed \leftarrow$ True
      $C_s \leftarrow C'_s$
    end if
  end for
  $v' \leftarrow (I-\gamma C)^{-1}R$
  $\epsilon \leftarrow ||v - v'||_\infty$ // max change in value function
  $\epsilon_{v^*} \leftarrow (2\epsilon\gamma)/(1-\gamma)$ // error bound on distance to $v^*$
  $\epsilon_{aggregate} \leftarrow (1-\gamma)\epsilon_{v^*}$ // bound on the maximum error to the aggregate performance
  if $\epsilon_{aggregate} < tol$ then
    $changed \leftarrow$ False
  end if
  $v \leftarrow v'$
end while
Return: $(1-\gamma)$mean$(|v|)$
```

solution we employ three techniques: an iteration limit, halting computation when C is ϵ-close to between iterations, and stopping when a bound on the distance to true solution is below a tolerance, tol. The first two approaches are straight forward an in the pseudocode. To bound the distance to the true solution we leverage prior work on bounding the distance of the current value function to the optimal value function. Consider the value function v and the subsequent value function v' obtained after one application of the Bellman operator. In our problem the Bellman operator is

$$v(s) = \max_{C_s \in \mathcal{C}_s} R(s) + \gamma \sum_{s' \in \mathcal{S}} C_{s,s'} v(s').$$

Let $\epsilon = \max_{s\in\mathcal{S}} |v(s) - v'(s)|$. Then the distance $\epsilon_{v^*} = \max_{s\in\mathcal{S}} |v'(s) - v^*(s)|$ of v' to the optimal value function v^*, is bounded above by ϵ (Williams & Baird, 1993), i.e,

$$\epsilon_{v^*} \leq (2\epsilon\gamma)/(1-\gamma).$$

We translate this this bound into a bound on the error to the confidence interval of y_i. Let y_i be the confidence interval computed using v' and y_i^* be the confidence interval computed using v^*. Then an upper bound on the error $\epsilon_{aggregate} = |y_i^* - y_i|$ is

Algorithm 3 `update_transition_row` procedure for updating C_s

```
**Input:** $C_s^-, C_s^+ \in \mathbb{R}^{|\mathcal{S}|}$, $r \in \mathbb{R}$, $v \in \mathbb{R}^{|\mathcal{S}|}$, $\gamma \in (0,1)$
Initialize: $C_s \leftarrow C_s^-$, $c \leftarrow \texttt{sum}(C_s^-)$
$w \leftarrow r + \gamma v$
$idxs \leftarrow \texttt{argsort}(w, \texttt{direction} = decreasing)$
**for** $i = 1$ **to** $|\mathcal{S}|$ **do**
  $idx \leftarrow idxs[i]$
  $\Delta c \leftarrow \min(C_s^+ - C_s^-, 1 - c)$
  $C_s[idx] \leftarrow C_s[idx] + \Delta c$
  $c \leftarrow c + \Delta c$
**end for**
**Return:** $C_s$
```

$$
\begin{aligned}
\epsilon_{aggregate} &= |y_i^* - y_i| \\
&= \left| \frac{1-\gamma}{|\mathcal{S}|} \sum_{s \in \mathcal{S}} |v^*(s)| - \frac{1-\gamma}{|\mathcal{S}|} \sum_{s \in \mathcal{S}} |v'(s)| \right| \\
&= \left| \frac{1-\gamma}{|\mathcal{S}|} (||v^*||_1 - ||v'||_1) \right| \\
&\leq \frac{1-\gamma}{|\mathcal{S}|} ||v^* - v'||_1 \\
&= \frac{1-\gamma}{|\mathcal{S}|} |\mathcal{S}| ||v^* - v'||_\infty \\
&= (1-\gamma) ||v^* - v'||_\infty \\
&\leq (1-\gamma) \max_{s \in \mathcal{S}} |v^*(s) - v'(s)| \\
&= (1-\gamma)\epsilon_{v^*} = 2\epsilon\gamma.
\end{aligned}
$$

E. Algorithm Definitions

This section provides the complete definition for each algorithm used in the experiments. Each algorithm is made complete by defining a distribution from, which hyperparameters are sampled. Table 4 shows distributions or values for any hyperparameter used in this work. For the continue state space environments, all of the algorithms use Fourier basis and linear function approximation (Konidaris et al., 2011). Note that $U(a, b)$ indicates a uniform random variable on $[a, b)$, and $U(\{\ldots\})$ indicates that a variable is sampled uniformly at random from a set of finite values.

Note that these ranges should not be considered optimal and could easily be improved for the environments in this experiment. The ranges were chosen to reflect a lack of knowledge about what optimal settings on an environment are and to be reflective of ranges one might expect optimal hyperparameters to fall in. The manual setting of these ranges leaks information based on our own experience with the algorithms and environments. However, since the definition is completely specified on these environments any favor to one algorithm could be easily identified and test for. Furthermore, any adaptive algorithm that can adjust these parameters during learning is likely to be superior than specifying better ranges through experience in this exact setup. Still, one should not tune ranges to fit any given set of environments.

F. Environments

This section describes the environments used in the experiments. All environments are listed in Table 5. Environments were recreated in the Julia language and many implementations follow closely to the that in the RLPy repository (Geramifard et al., 2015). Most environments are best described by either the paper publishing the environment or by examining the source code we provide. We also describe the discrete environments below.

There are eight discrete domains used in the work four Gridworld environments and four chain environments. The Gridworld

Algorithm	**Hyperparameter**	**Discrete**	**Continuous**
All	λ	$U(0,1)$	
All	γ	$\Gamma - e^{U(\ln 10^{-4}, \ln 0.05)}$	
All	Value function	Tabular	Linear with Fourier basis
All Sarsa(λ) and Q(λ)	ϵ	$U(0,1)$	
Sarsa(λ) and Q(λ)	α_q	$e^{U(\ln 10^{-3}, \ln 10^{-1})}$	$e^{U(\ln 10^{-6}, \ln 10^{-3})}$
Sarsa(λ)-s and Q(λ)-s	α_q	$e^{U(\ln 10^{-3}, \ln 10^{-1})}$	$e^{U(\ln 10^{-3}, \ln 10^{0})/\lvert\phi\rvert}$
All AC, PPO, and NACTD	Policy	Tabular Softmax	Linear Softmax with Fourier basis
AC, NACTD	α_v	$e^{U(\ln 10^{-3}, \ln 10^{-1})}$	$e^{U(\ln 10^{-6}, \ln 10^{-3})}$
AC, NACTD	α_p	$e^{U(\ln 10^{-3}, \ln 10^{-1})}$	$e^{U(\ln 10^{-6}, \ln 10^{-3})}$
AC-S	α_v	$e^{U(\ln 10^{-3}, \ln 10^{-1})}$	$e^{U(\ln 10^{-3}, \ln 10^{0})/\lvert\phi\rvert}$
AC-S, AC-Parl2	α_p	$e^{U(\ln 10^{-3}, \ln 10^{-1})}$	$e^{U(\ln 10^{-3}, \ln 10^{0})/(\lvert\phi\rvert \times \text{num\_actions})}$
NAC-TD	α_w	$e^{U(\ln 10^{-3}, \ln 10^{-1})}$	$e^{U(\ln 10^{-6}, \ln 10^{-3})}$
NAC-TD	normalize_gradient	True	
PPO	clip	$U(0.1, 0.3)$	
PPO	entropy_coef	$e^{U(\ln 10^{-8}, \ln 10^{-2})}$	
PPO	steps_per_batch	$2^{U(\log_2 64, \log_2 256)}$	
PPO	epochs	$U(\{1, \ldots, 10\})$	
PPO	batch_size	$2^{U(\log_2 16, \log_2 \min(64, \text{steps\_per\_batch}))}$	
PPO	Adam-ϵ	10^{-5}	
PPO	Adam-α	$e^{U(\ln 10^{-3}, \ln 10^{-1})}$	$e^{U(\ln 10^{-6}, \ln 10^{-3})}$
PPO	Adam-β_1	0.9	
PPO	Adam-β_2	0.999	
Fourier basis	dorder	N/A	$U(\{0, \ldots, 9\})$
Fourier basis	iorder	N/A	$U(\{1, \ldots, 9\})$
Fourier basis	trig	N/A	`cos`

Table 4. This table show the distributions from which each hyperparameter is sampled. The All algorithm means the hyperparameter and distribution were used for all algorithms. Steps sizes are labeled with various αs. The discount factor γ an algorithm uses is scaled down from Γ that is specified by the environment. For all environments used in this work $\Gamma = 1.0$. PPO uses the same learning rate for both the policy and value function. The max dependent order on the Fourier basis is limited such that no more than 10,000 features are generated as a result of dorder.

Environment	Num Episodes	State Space
Gridworld 5 Deterministic	100	Discrete
Gridworld 5 Stochastic	100	Discrete
Gridworld 10 Deterministic	100	Discrete
Gridworld 10 Stochastic	100	Discrete
Chain 10 Deterministic	100	Discrete
Chain 10 Stochastic	100	Discrete
Chain 50 Deterministic	100	Discrete
Chain 50 Stochastic	100	Discrete
Acrobot	500	Continuous
Cart-Pole	100	Continuous
MountainCar	100	Continuous
PinBall Empty	100	Continuous
PinBall Box	100	Continuous
Pinball Medium	100	Continuous
PinBall Single	200	Continuous

Table 5. This table list every used in this paper along with the number of episodes each algorithm was allowed to interact with the environment and its type of state space.

environments are an $N \times N$ grid with the agent starting every episode in the top left corner and goal state in the bottom right. The reward is -1 at every step until the goal state is reached, then the episode is then terminated. In every state there are four actions: up, down, left, and right. The transition dynamics are either deterministic, meaning an up action sends the agent up one state unless it is outside the map, or the dynamics are stochastic, meaning the agent might randomly move to one of the states perpendicular to the intended direction or stays in the current state. The chain environments are N chains where there are N states each with a connection only to the state directly to the left or right of it and end points only connect to the one state they are next to. The agent starts in state one (far left of the chain) and the goal state is state N (far right of the chain). The reward is -1 every step until the goal state is reached and then the episode terminates. Both gridworld and chain MDPs terminate episodes in a finite time based on the size of the problem. Gridworld problems terminate after $20N^2$ steps have been taken and chain environments terminate after $20N$ steps are taken.

G. Alternative Bounding Techniques

As pointed out in our description of PBP, we use the nonparametric concentration inequalities DKW and Anderson's inequalities. These inequalities are often conservative and lead to conservative confidence intervals. So we investigate two alternatives PBP-t a method that replaces DKW and Anderson's inequality with the one based on Students t-distribution and the percentile bootstrap.

In PBP-t everything is the same as PBP except we compute confidence intervals on the mean normalized performance as follows

$$Z^-_{i,j,k} = \mu_{i,j,k} - \frac{\hat{\sigma}}{\sqrt{T_{i,j}}} t_{1-\delta', T_{i,j}-1}$$
$$Z^+_{i,j,k} = \mu_{i,j,k} + \frac{\hat{\sigma}}{\sqrt{T_{i,j}}} t_{1-\delta', T_{i,j}-1}$$

where $z_{i,j,k,t} = \hat{F}_{X_{k,j}}(x_{i,j,t})$, $\mu_{i,j,k} = \frac{1}{T_{i,j}} \sum_{t=1}^{T_{i,j}} z_{i,j,k,t}$, $\hat{\sigma} = \sqrt{\sum_{t=1}^{T_{i,j}} (\mu_{i,j,k} - z_{i,j,k,t})^2 / (T_{i,j} - 1)}$, $t_{1-\delta',\nu}$ is the $100(1-\delta')$ percentile of Student's t-distribution with ν degrees of freedom, and we set $\delta' = \delta/(|\mathcal{A}||\mathcal{M}|)$. Notice that there are $|\mathcal{A}|^2|\mathcal{M}|$ comparisons being made, so $\delta' = \delta/(|\mathcal{A}|^2|\mathcal{M}|)$ should be used to account for more the multiple comparisons. However, it is likely that if one comparison with an algorithm i fails then that there will be failures with the other $|\mathcal{A}| - 1$ algorithms so we use the smaller δ' as a heuristic for tighter confidence intervals.

In the bootstrap procedure, we use the percentile bootstrap with a confidence level of $\delta' = \delta/(|\mathcal{A}||\mathcal{M}|)$ and 10,000 bootstrap samples of the aggregate performance. Each bootstrap is formed by re-sampling the performance of each algorithm on each environment for the collected data. Then the aggregate performance for each bootstrap is computed. The lower and upper confidence intervals are given by the $100(\delta'/2)$ and $100(1-\delta'/2)$ percentile from the bootstrap aggregate performance. Since $\delta'/2$ is often really small, 10,000 bootstrap samples are needed to get confidence intervals that more accurately reflect the true confidence intervals. This requires a substantial amount of compute time and can take over four hours for an original sample size of 10,000.

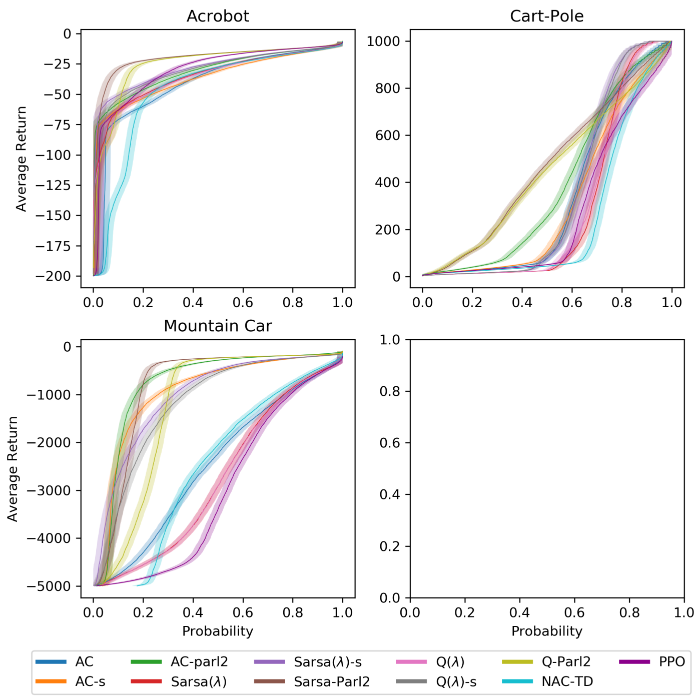
H. Confidence interval test experiment

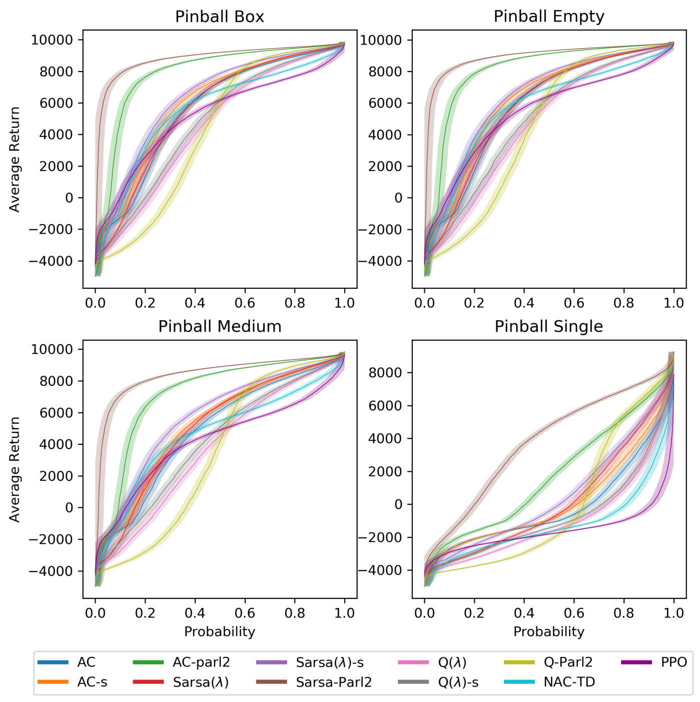
To test the different bounding techniques, we estimate the failure rate of each confidence interval technique at different sample sizes. For this experiment we execute 1,000 trials of the evaluation procedure using samples sizes (trials per algorithm per environment) of 10, 30, 100, 1,000, and 10,000. There are a total of 11.14 million samples per algorithm per environment. To reduce computation costs, we limit this experiment to only include the Sarsa-Parl2, Q-Parl2, AC-Parl2, and Sarsa(λ)-s. Additionally, we reduce the environment set to be the discrete environments and mountain car. Then we compute the proportion of violations for any confidence interval. All methods use the same data, so the results are not independent. We choose not to run independent samples of the bounds to limit our environmental impact. The method to compute of the proportion of violations and number of significant pairwise comparison can be found in the source code.

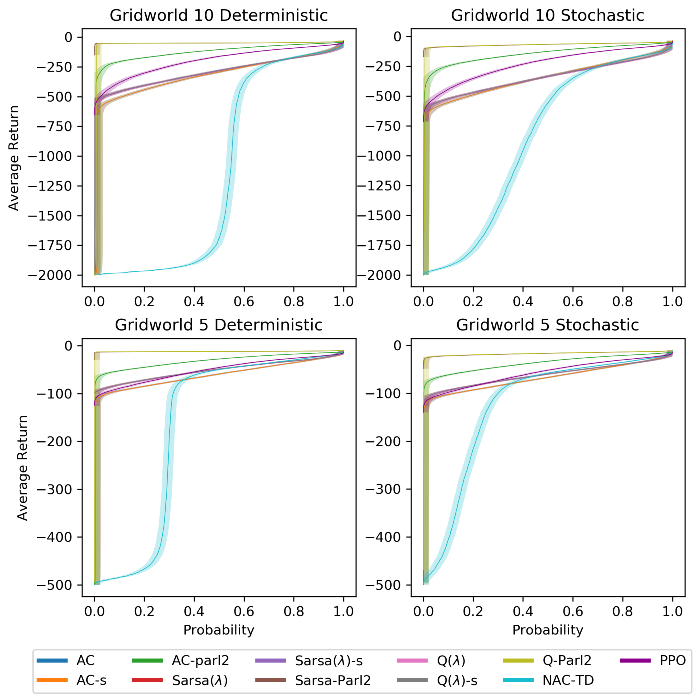
It it important to note that when this experiment was run, there was a bug in the code that made the step-size for the Parl2 methods smaller by a factor of 0.1. This does not invalidate the results, only that the algorithms run are not equivalent those used in the other experiments in this paper. The main impact of this difference was that Sarsa-Parl2 and Q-Parl2 did were not as effective on the discrete MDPs (though they diverged even less) and their scores were nearly the same. Since both of these algorithms had near identical scores on most of the environments, it became almost impossible to differentiate them, so detecting five out of six (83%) statistically significant comparisons is considered optimal for this experiment.

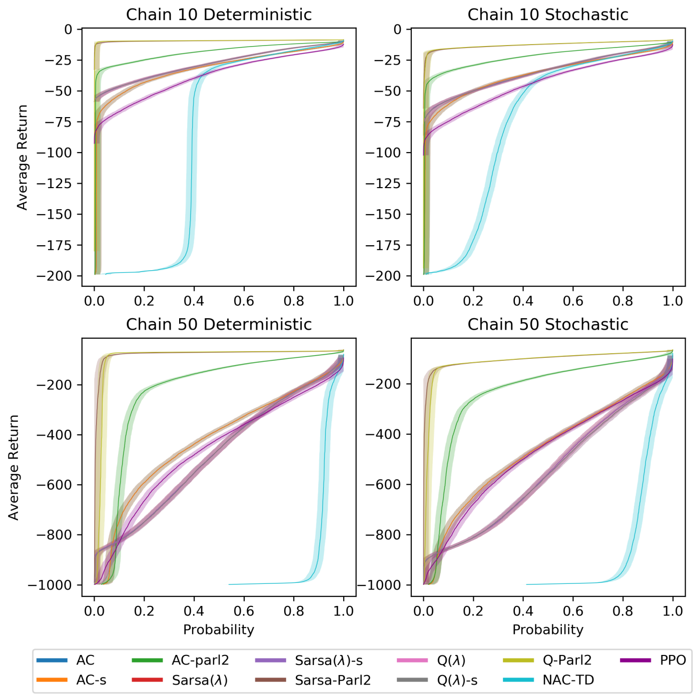
I. Performances

This section illustrates the distribution of performance of each algorithm. Tables of the average performance (rounded to the tenths place) along with the algorithm rank on that environment. In the Figures showing the performance distributions the shaded regions represent $100(1-0.05/|\mathcal{A}||\mathcal{M}|)\% \approx 99.9697\%$ confidence intervals computed via DKW. In the performance tables $100(1-0.05/|\mathcal{A}||\mathcal{M}|)\%$ confidence intervals are shown in parentheses.


Acrobot
Cart-Pole
Mountain Car
Average Return
Probability
AC
AC-s
AC-parl2
Sarsa(λ)
Sarsa(λ)-s
Sarsa-Parl2
Q(λ)
Q(λ)-s
Q-Parl2
NAC-TD
PPO


Pinball Box
Pinball Empty
Pinball Medium
Pinball Single
Average Return
Probability
AC
AC-s
AC-parl2
Sarsa(λ)
Sarsa(λ)-s
Sarsa-Parl2
Q(λ)
Q(λ)-s
Q-Parl2
NAC-TD
PPO


Gridworld 10 Deterministic
Gridworld 10 Stochastic
Gridworld 5 Deterministic
Gridworld 5 Stochastic
Average Return
Probability
0
−250
−500
−750
−1000
−1250
−1500
−1750
−2000
−100
−200
−300
−400
−500
0.0
0.2
0.4
0.6
0.8
1.0
AC
AC-s
AC-parl2
Sarsa(λ)
Sarsa(λ)-s
Sarsa-Parl2
Q(λ)
Q(λ)-s
Q-Parl2
NAC-TD
PPO


Chain 10 Deterministic
Chain 10 Stochastic
Chain 50 Deterministic
Chain 50 Stochastic
Average Return
Probability
0
−25
−50
−75
−100
−125
−150
−175
−200
−400
−600
−800
−1000
0.0
0.2
0.4
0.6
0.8
1.0
AC
AC-s
AC-parl2
Sarsa(λ)
Sarsa(λ)-s
Sarsa-Parl2
Q(λ)
Q(λ)-s
Q-Parl2
NAC-TD
PPO

Acrobot		
Algorithm	Mean	Rank
Sarsa-Parl2	-20.6 (-24.6, -18.1)	1 (2, 1)
Q-Parl2	-25.7 (-29.8, -22.8)	2 (6, 1)
Sarsa(λ)-s	-28.3 (-32.3, -26.3)	3 (6, 2)
Q(λ)-s	-30.3 (-34.3, -27.9)	4 (9, 2)
PPO	-30.5 (-34.5, -26.7)	5 (9, 2)
AC-parl2	-30.6 (-34.6, -28.8)	6 (9, 2)
Sarsa(λ)	-34.7 (-38.7, -32.4)	7 (10, 4)
Q(λ)	-35.7 (-39.7, -33.3)	8 (10, 4)
AC-s	-36.0 (-40.0, -33.5)	9 (10, 4)
AC	-41.4 (-45.4, -37.3)	10 (11, 7)
NAC-TD	-47.1 (-51.1, -43.0)	11 (11, 10)

Cart-Pole		
Algorithm	Mean	Rank
Sarsa-Parl2	469.2 (448.3, 490.0)	1 (2, 1)
Q-Parl2	450.3 (429.6, 471.1)	2 (2, 1)
AC-parl2	390.4 (369.7, 411.2)	3 (3, 3)
Sarsa(λ)-s	347.5 (326.5, 368.4)	4 (7, 4)
Q(λ)-s	345.5 (324.5, 366.3)	5 (7, 4)
AC	338.7 (317.8, 359.6)	6 (7, 4)
AC-s	320.6 (300.0, 341.5)	7 (9, 4)
Q(λ)	291.0 (270.1, 311.9)	8 (10, 7)
Sarsa(λ)	287.2 (265.9, 308.6)	9 (10, 7)
PPO	276.3 (256.1, 297.1)	10 (11, 8)
NAC-TD	244.7 (224.0, 265.5)	11 (11, 10)

Mountain Car		
Algorithm	Mean	Rank
AC-parl2	-791.6 (-894.0, -688.7)	1 (3, 1)
Sarsa-Parl2	-843.2 (-945.0, -740.5)	2 (4, 1)
AC-s	-966.0 (-1068.0, -863.1)	3 (4, 1)
Sarsa(λ)-s	-1024.2 (-1125.8, -923.5)	4 (5, 2)
Q(λ)-s	-1197.3 (-1298.9, -1094.9)	5 (6, 4)
Q-Parl2	-1269.3 (-1371.2, -1166.4)	6 (6, 5)
AC	-2477.1 (-2576.0, -2374.3)	7 (8, 7)
NAC-TD	-2489.3 (-2589.1, -2386.4)	8 (8, 7)
Sarsa(λ)	-2769.1 (-2867.8, -2666.2)	9 (10, 9)
Q(λ)	-2779.3 (-2877.9, -2676.4)	10 (10, 9)
PPO	-3026.3 (-3124.9, -2923.5)	11 (11, 11)

Chain 10 Deterministic		
Algorithm	Mean	Rank
Q-Parl2	-9.2 (-13.2, -9.1)	1 (2, 1)
Sarsa-Parl2	-9.3 (-13.3, -9.2)	2 (2, 1)
AC-parl2	-19.0 (-23.0, -17.7)	3 (3, 3)
Sarsa(λ)	-27.9 (-31.8, -26.9)	4 (9, 4)
Sarsa(λ)-s	-27.9 (-31.8, -26.9)	4 (9, 4)
Q(λ)	-27.9 (-31.9, -26.9)	6 (9, 4)
Q(λ)-s	-27.9 (-31.9, -26.9)	6 (9, 4)
AC	-32.2 (-36.2, -29.9)	8 (9, 4)
AC-s	-32.2 (-36.2, -29.9)	8 (9, 4)
PPO	-38.0 (-41.9, -36.5)	10 (10, 10)
NAC-TD	-89.7 (-93.7, -85.7)	11 (11, 11)

Chain 10 Stochastic		
Algorithm	Mean	Rank
Q-Parl2	-12.7 (-16.7, -12.2)	1 (2, 1)
Sarsa-Parl2	-12.8 (-16.8, -12.4)	2 (2, 1)
AC-parl2	-22.6 (-26.6, -21.2)	3 (3, 3)
Q(λ)	-34.9 (-38.9, -33.6)	4 (9, 4)
Q(λ)-s	-34.9 (-38.9, -33.6)	4 (9, 4)
Sarsa(λ)	-35.1 (-39.0, -33.7)	6 (9, 4)
Sarsa(λ)-s	-35.1 (-39.0, -33.7)	6 (9, 4)
AC	-36.9 (-40.8, -34.7)	8 (9, 4)
AC-s	-36.9 (-40.8, -34.7)	8 (9, 4)
PPO	-43.8 (-47.6, -42.1)	10 (10, 10)
NAC-TD	-75.2 (-79.1, -71.2)	11 (11, 11)

Chain 50 Deterministic			Chain 50 Stochastic		
Algorithm	Mean	Rank	Algorithm	Mean	Rank
Sarsa-Parl2	-78.3 (-97.9, -71.5)	1 (2, 1)	Sarsa-Parl2	-101.6 (-121.1, -96.4)	1 (2, 1)
Q-Parl2	-92.7 (-112.3, -74.6)	2 (2, 1)	Q-Parl2	-111.1 (-130.6, -97.5)	2 (2, 1)
AC-parl2	-231.0 (-250.5, -211.3)	3 (3, 3)	AC-parl2	-239.3 (-258.7, -219.7)	3 (3, 3)
AC	-425.5 (-444.1, -405.8)	4 (6, 4)	AC	-463.2 (-481.5, -443.6)	4 (6, 4)
AC-s	-425.5 (-444.1, -405.8)	4 (6, 4)	AC-s	-463.2 (-481.5, -443.6)	4 (6, 4)
PPO	-462.2 (-480.4, -442.6)	6 (10, 4)	PPO	-474.4 (-492.2, -455.3)	6 (6, 4)
Sarsa(λ)	-479.5 (-498.3, -462.4)	7 (10, 6)	Sarsa(λ)	-546.9 (-565.2, -529.2)	7 (10, 7)
Sarsa(λ)-s	-479.5 (-498.3, -462.4)	7 (10, 6)	Sarsa(λ)-s	-546.9 (-565.2, -529.2)	7 (10, 7)
Q(λ)	-481.6 (-500.4, -464.0)	9 (10, 6)	Q(λ)	-550.3 (-568.7, -532.5)	9 (10, 7)
Q(λ)-s	-481.6 (-500.4, -464.0)	9 (10, 6)	Q(λ)-s	-550.3 (-568.7, -532.5)	9 (10, 7)
NAC-TD	-928.4 (-946.8, -908.7)	11 (11, 11)	NAC-TD	-897.4 (-915.3, -877.8)	11 (11, 11)

Gridworld 10 Deterministic			Gridworld 10 Stochastic		
Algorithm	Mean	Rank	Algorithm	Mean	Rank
Sarsa-Parl2	-49.5 (-90.7, -48.9)	1 (2, 1)	Sarsa-Parl2	-67.5 (-108.6, -66.1)	1 (2, 1)
Q-Parl2	-60.8 (-102.0, -48.5)	2 (2, 1)	Q-Parl2	-75.4 (-116.4, -64.8)	2 (2, 1)
AC-parl2	-130.6 (-171.6, -115.1)	3 (3, 3)	AC-parl2	-146.4 (-187.3, -135.8)	3 (3, 3)
PPO	-199.6 (-240.4, -188.8)	4 (4, 4)	PPO	-229.5 (-270.1, -217.6)	4 (4, 4)
Q(λ)	-290.6 (-331.0, -278.9)	5 (10, 5)	Q(λ)	-339.2 (-379.2, -326.9)	5 (10, 5)
Q(λ)-s	-290.6 (-331.0, -278.9)	5 (10, 5)	Q(λ)-s	-339.2 (-379.2, -326.9)	5 (10, 5)
Sarsa(λ)	-291.8 (-332.3, -281.3)	7 (10, 5)	Sarsa(λ)	-342.3 (-382.4, -330.0)	7 (10, 5)
Sarsa(λ)-s	-291.8 (-332.3, -281.3)	7 (10, 5)	Sarsa(λ)-s	-342.3 (-382.4, -330.0)	7 (10, 5)
AC	-324.7 (-365.2, -295.3)	9 (10, 5)	AC	-347.2 (-387.4, -329.7)	9 (10, 5)
AC-s	-324.7 (-365.2, -295.3)	9 (10, 5)	AC-s	-347.2 (-387.4, -329.7)	9 (10, 5)
NAC-TD	-1141.3 (-1181.6, -1099.9)	11 (11, 11)	NAC-TD	-864.1 (-904.2, -823.3)	11 (11, 11)

Gridworld 5 Deterministic			Gridworld 5 Stochastic		
Algorithm	Mean	Rank	Algorithm	Mean	Rank
Sarsa-Parl2	-12.2 (-22.5, -12.1)	1 (2, 1)	Sarsa-Parl2	-17.0 (-27.2, -16.6)	1 (2, 1)
Q-Parl2	-12.5 (-22.7, -12.0)	2 (2, 1)	Q-Parl2	-17.1 (-27.4, -16.3)	2 (2, 1)
AC-parl2	-32.2 (-42.4, -30.1)	3 (3, 3)	AC-parl2	-37.2 (-47.3, -35.4)	3 (3, 3)
PPO	-51.1 (-61.2, -49.0)	4 (10, 4)	PPO	-57.2 (-67.3, -54.9)	4 (10, 4)
Q(λ)	-51.4 (-61.5, -49.6)	5 (10, 4)	Q(λ)	-61.5 (-71.6, -59.4)	5 (10, 4)
Q(λ)-s	-51.4 (-61.5, -49.6)	5 (10, 4)	Q(λ)-s	-61.5 (-71.6, -59.4)	5 (10, 4)
Sarsa(λ)	-51.9 (-62.0, -50.1)	7 (10, 4)	Sarsa(λ)	-61.5 (-71.6, -59.4)	7 (10, 4)
Sarsa(λ)-s	-51.9 (-62.0, -50.1)	7 (10, 4)	Sarsa(λ)-s	-61.5 (-71.6, -59.4)	7 (10, 4)
AC	-62.0 (-72.1, -58.2)	9 (10, 4)	AC	-67.6 (-77.6, -64.8)	9 (10, 4)
AC-s	-62.0 (-72.1, -58.2)	9 (10, 4)	AC-s	-67.6 (-77.6, -64.8)	9 (10, 4)
NAC-TD	-168.1 (-178.3, -157.9)	11 (11, 11)	NAC-TD	-125.4 (-135.5, -115.4)	11 (11, 11)

Pinball Box			Pinball Empty		
Algorithm	Mean	Rank	Algorithm	Mean	Rank
Sarsa-Parl2	8823.2 (8513.4, 8998.4)	1 (1, 1)	Sarsa-Parl2	8942.1 (8631.8, 9115.3)	1 (1, 1)
AC-parl2	7875.4 (7565.9, 8170.2)	2 (2, 2)	AC-parl2	8041.0 (7730.8, 8333.4)	2 (2, 2)
Sarsa(λ)-s	6288.3 (5980.0, 6569.4)	3 (4, 3)	Sarsa(λ)-s	6382.4 (6073.4, 6664.1)	3 (5, 3)
AC-s	5961.2 (5653.1, 6251.1)	4 (7, 3)	AC-s	6155.8 (5846.5, 6444.9)	4 (7, 3)
Sarsa(λ)	5603.0 (5295.1, 5889.7)	5 (8, 4)	AC	5814.2 (5505.4, 6097.3)	5 (8, 3)
AC	5602.5 (5295.1, 5886.8)	6 (8, 4)	Sarsa(λ)	5778.6 (5469.8, 6063.7)	6 (8, 4)
NAC-TD	5546.8 (5243.2, 5838.0)	7 (8, 4)	NAC-TD	5710.1 (5405.2, 5998.0)	7 (8, 4)
PPO	5184.6 (4882.6, 5447.4)	8 (9, 5)	PPO	5401.3 (5097.3, 5666.6)	8 (9, 5)
Q(λ)-s	4728.6 (4421.8, 5020.1)	9 (11, 8)	Q(λ)-s	4891.8 (4584.1, 5182.5)	9 (11, 8)
Q(λ)	4449.9 (4143.3, 4740.8)	10 (11, 9)	Q(λ)	4602.3 (4294.8, 4892.1)	10 (11, 9)
Q-Parl2	4246.5 (3937.2, 4539.9)	11 (11, 9)	Q-Parl2	4487.9 (4178.3, 4781.5)	11 (11, 9)

Pinball Medium			Pinball Single		
Algorithm	Mean	Rank	Algorithm	Mean	Rank
Sarsa-Parl2	8402.1 (8094.2, 8625.5)	1 (1, 1)	Sarsa-Parl2	3754.6 (3470.3, 4023.3)	1 (1, 1)
AC-parl2	7128.1 (6820.6, 7429.2)	2 (2, 2)	AC-parl2	1696.7 (1418.4, 1989.3)	2 (2, 2)
Sarsa(λ)-s	5667.4 (5361.4, 5949.4)	3 (4, 3)	Sarsa(λ)-s	514.1 (240.8, 793.2)	3 (5, 3)
AC-s	5195.5 (4890.3, 5487.2)	4 (7, 3)	Sarsa(λ)	119.2 (-151.7, 408.1)	4 (7, 3)
Sarsa(λ)	5050.5 (4745.1, 5336.5)	5 (7, 4)	AC-s	103.8 (-158.5, 389.6)	5 (7, 3)
AC	4835.7 (4531.8, 5123.8)	6 (7, 4)	Q-Parl2	-73.4 (-352.7, 214.7)	6 (7, 4)
NAC-TD	4652.5 (4353.9, 4940.2)	7 (9, 4)	AC	-197.1 (-451.4, 95.1)	7 (7, 4)
PPO	4227.1 (3932.0, 4493.3)	8 (10, 7)	Q(λ)-s	-746.6 (-1005.9, -457.2)	8 (10, 8)
Q(λ)-s	4084.7 (3780.6, 4375.1)	9 (10, 7)	Q(λ)	-986.1 (-1244.4, -693.8)	9 (10, 8)
Q(λ)	3689.6 (3386.2, 3981.9)	10 (11, 8)	NAC-TD	-1229.4 (-1463.7, -934.7)	10 (11, 8)
Q-Parl2	3404.5 (3097.9, 3699.2)	11 (11, 10)	PPO	-1602.2 (-1789.1, -1327.7)	11 (11, 10)