MEASURING THE RELIABILITY OF REINFORCEMENT LEARNING ALGORITHMS

Anonymous authors
Paper under double-blind review

ABSTRACT

Inadequate reliability is a well-known issue for reinforcement learning (RL) algorithms. This problem has gained increasing attention in recent years, and efforts to improve it have grown substantially. To aid RL researchers and production users with the evaluation and improvement of reliability, we propose a novel set of metrics that quantitatively measure different aspects of reliability. In this work, we address variability and risk, both during training and after learning (on a fixed policy). We designed these metrics to be general-purpose, and we also designed complementary statistical tests to enable rigorous comparisons on these metrics. In this paper, we first describe the desired properties of the metrics and their design, the aspects of reliability that they measure, and their applicability to different scenarios. We then describe the statistical tests and make additional practical recommendations for reporting results. Finally, we apply our metrics to a set of common RL algorithms and environments, compare them, and analyze the results.

1 INTRODUCTION

Reinforcement learning (RL) algorithms, especially Deep RL algorithms, tend to be highly variable in performance and considerably sensitive to a range of different factors, including implementation details, hyper-parameters, choice of environments, and even random seeds (Henderson et al., 2017). This variability hinders reproducible research, and can be costly or even dangerous for real-world applications. Furthermore, it impedes scientific progress in the field when practitioners cannot reliably evaluate or predict the performance of any particular algorithm, compare different algorithms, or even compare different implementations of the same algorithm.

Recently, Henderson et al. (2017) has performed a detailed analysis of reliability for several policy gradient algorithms, while Duan et al. (2016) has benchmarked average performance of different continuous-control algorithms.

In this work, we aim to devise a set of metrics that measure reliability of RL algorithms. Our analysis distinguishes between several typical modes to evaluate RL performance: "evaluation during training", which is computed over the course of training, vs. "evaluation after learning", which is computed after training to evaluate a fixed policy. These metrics are also designed to measure different aspects of reliability, e.g. reproducibility (variability *across* training runs and variability *across* rollouts of a fixed policy) or stability (variability *within* training runs). Additionally, the metrics capture multiple aspects of variability – dispersion (the width of a distribution), and risk (the heaviness and extremity of the tails of a distribution).

Standardized measures of reliability can benefit the field of RL by allowing RL practitioners to compare algorithms in a rigorous and consistent way. This in turn allows the field to measure progress, and also informs the selection of algorithms for both research and production environments. By measuring various aspects of reliability, we can also identify particular strengths and weaknesses of algorithms, allowing users to pinpoint specific areas of improvement.

In this paper, in addition to describing these reliability metrics, we also present practical recommendations for statistical tests to compare metric results and how to report the results more generally. As examples, we apply these metrics to a set of algorithms and environments (discrete and continuous,

		Dispersion (D)	Risk (R)
DURING TRAINING	**Across Time (T) (within training runs)**	IQR* within windows, after detrending	**Short-term:** CVaR† on first-order differences **Long-term**: CVaR† on Drawdown
	Across Runs (R)	IQR* across training runs, after low-pass filtering.	CVaR† across runs
AFTER LEARNING	**Across rollouts on a Fixed Policy (F)**	IQR* across rollouts for a fixed policy	CVaR† across rollouts for a fixed policy

Table 1: Summary of our proposed metrics for evaluation. For evaluation DURING TRAINING, which measures reliability over the course of training an algorithm, the inputs to the metrics are the performance curves of an algorithm, evaluated at regular intervals during a single training run (or on a set of training runs). For evaluation AFTER LEARNING, which measures reliability of an already-trained policy, the inputs to the metrics are the performance scores of a set of rollouts of that fixed policy. *IQR: inter-quartile range. †CVaR: conditional value at risk.

off-policy and on-policy). We will release the code used in this paper as an open-source Python package to ease the adoption of these metrics and their complementary statistics.[1]

2 RELIABILITY METRICS

We target three different axes of variability, and two different measures of variability along each axis. We denote each of these by a letter, and each metric as a combination of an axis + a measure, e.g. "DR" for "Dispersion Across Runs". See Table 1 for a summary. Please see Appendix A.1 for more detailed definitions of the terms used here.

2.1 AXES OF VARIABILITY

Our metrics target the following three axes of variability. The first two capture reliability "during training", while the last captures reliability of a fixed policy "after learning". We borrow this terminology from Machado et al. (2018).

During training: Across Time (T) In the setting of evaluation during training, one desirable property for an RL algorithm is to be stable within each training run. In general, smooth monotonic improvement is preferable to noisy fluctuations around a positive trend, or unpredictable swings in performance.

This type of stability is important for several reasons. During learning, especially when deployed for real applications, it can be costly or even dangerous for an algorithm to have unpredictable levels of performance. Even in cases where bouts of poor performance do not directly cause harm, e.g. if training in simulation, high instability implies that algorithms have to be check-pointed and evaluated more frequently in order to catch the peak performance of the algorithm. Furthermore, while training, it can be a waste of computational resources to train an unstable algorithm that tends to forget previously learned behaviors.

During training: Across Runs (R) During training, RL algorithms should have easily and consistently reproducible performances across multiple training instances. Depending on the components that we allow to vary across training runs, this variability can encapsulate the algorithm's sensitivity to a variety of factors, such as: random seed and initializations of the optimization, random seed and

[1]This package will be released before the ICLR conference.

initializations of the environment, implementation details, and hyper-parameter settings. Depending on the goals of the analysis, these factors can be held constant or allowed to vary, in order to disentangle the contribution of each factor to variability in training performance.

After learning: Across rollouts of a fixed policy (F) When evaluating a fixed policy, a natural concern is the variability in performance across multiple rollouts of that fixed policy. Each rollout may be specified e.g. in terms of a number of actions, environment steps, or episodes. This metric combines stochasticity from the environment with stochasticity from the training procedure (the optimization), and practitioners may sometimes wish to keep one or the other constant if it is important to disentangle the two factors (e.g. holding constant the random seed of the environment while allowing the random seed controlling optimization to vary across rollouts).

2.2 MEASURES OF VARIABILITY

For each axis of variability, we have two kinds of measures: dispersion and risk.

Dispersion Dispersion is the width of the distribution. To measure dispersion, we use "robust statistics" such as the ***Inter-quartile range (IQR)*** (i.e. the difference between the 75th and 25th percentiles) and the ***Median absolute deviation from the median (MAD)***, rather than statistics more specifically suited to normal distributions, like variance or standard deviation. [2] We prefer to use IQR over MAD, because it is more appropriate for asymmetric distributions (Peter J. Rousseeuw & Christophe Croux, 1993).

Risk In many cases, we are concerned about the worst-case scenarios. Therefore, we define risk as the heaviness and extent of the tails of the distribution. This is complementary to measures of dispersion like IQR, which cuts off the tails of the distribution. To measure risk, we use the ***Conditional Value at Risk (CVaR)***, also known as "expected shortfall". CVaR measures the expected loss in the worst-case scenarios, defined by some quantile α. It is computed as the expected value in the left-most tail of a distribution (Acerbi & Tasche, 2002). Originally developed in finance, it has also seen recent adoption in Safe RL as an additional component of the objective function, e.g. Bäuerle & Ott (2011); Chow & Ghavamzadeh (2014); Tamar et al. (2015). We use CVaR of a random variable X for a given quantile α to be defined as:

$$\text{CVaR}_\alpha(X) = \mathbb{E}\left[X | X \leq VaR_\alpha(X)\right] \tag{1}$$

where $\alpha \in (0, 1)$ and the VaR_α (Value at Risk) is just the α-quantile of the distribution of X.

2.3 DESIDERATA

We required that our metrics and statistical tests fulfill the following criteria:

- A minimal number of configuration parameters – to facilitate standardization as well as to minimize "researcher degrees of freedom" (where flexibility may allow users to tune settings to produce more favorable results, leading to an inflated rate of false positives) (Simmons et al., 2011).
- Robust statistics, where possible. Robust statistics are less sensitive to outliers and have more reliable performance for a wider range of distributions. Robust statistics are especially important when applied to training performance, which tends to be highly non-Gaussian, making metrics such as variance and standard deviation inappropriate. For example, training performance can often be be bi-modal, with a concentration of points near the starting level and another concentration at the level of asymptotic performance.
- Invariance to sampling frequency – results should not be biased by the frequency at which an algorithm was evaluated during training. See Section 2.5 for further discussion.

[2] Note that our aim here is to measure the variability of the distribution, rather than to characterize the uncertainty in estimating a statistical parameter of that distribution. Therefore, confidence intervals and other similar methods are not suitable for the aim of measuring dispersion.

- Enable meaningful statistical comparisons on the metrics, while making minimal assumptions about the distribution of the results. We thus designed statistical procedures that are non-parametric (Section 4).

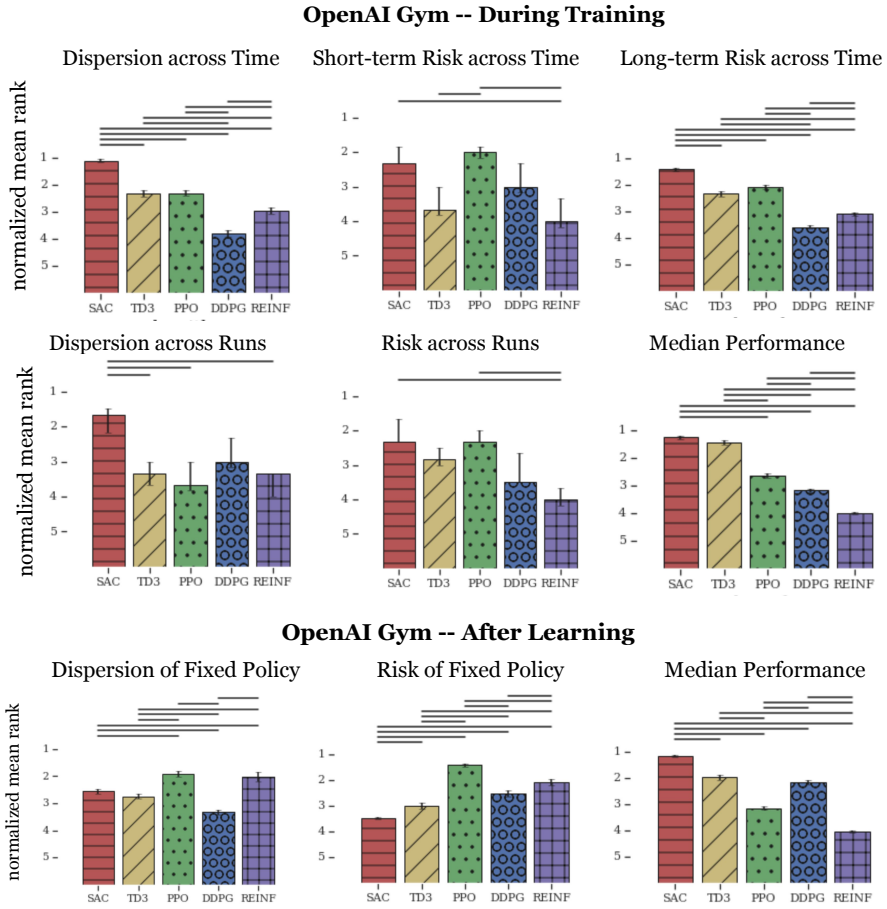

Figure 1: Reliability metrics and median performance for continuous control RL algorithms (DDPG, TD3, SAC, REINFORCE, and PPO) tested on OpenAI Gym environments. Rank 1 always indicates "best" reliability, e.g. lowest IQR across runs. Error bars are 95% bootstrap confidence intervals (# bootstraps = 1,000). Significant pairwise differences in ranking between pairs of algorithms are indicated by black horizontal lines above the colored bars. ($\alpha = 0.05$ with Benjamini-Yekutieli correction, permutation test with # permutations = 1,000). Note that the best algorithms by median performance are not always the best algorithms on reliability.

2.4 Metric Definitions

Dispersion across Time (DT): IQR across Time To measure dispersion across time (DT), we wished to isolate higher-frequency variability, rather than capturing longer-term trends. Not least, we do not want our metrics to be influenced by positive trends of improvement during training, which are in fact desirable sources of variation in the training performance. Therefore, we apply detrending before computing dispersion metrics: Inter-quartile range (IQR) within a sliding window along the training curve (after detrending). For detrending, we used differencing (i.e. $y_t\prime = y_t - y_{t-1}$).[3]

Short-term Risk across Time (SRT): CVaR on Differences For this measure, we wish to measure the most extreme short-term losses over time. To do this, we apply CVaR to the changes in

[3]Please see Appendix A.2 for a more detailed discussion of different types of detrending, and the rationale for choosing differencing here.

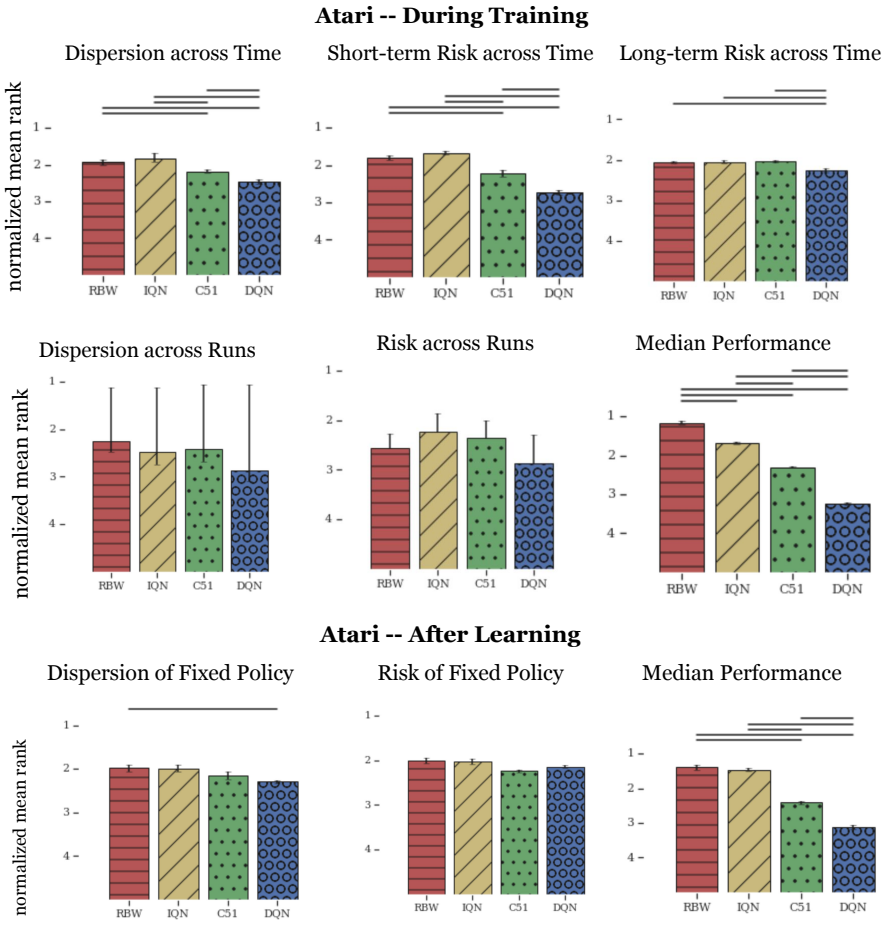

Figure 2: Reliability metrics and median performance for four DQN-variants (C51, DQN: Deep Q-network, IQ: Implicit Quantiles, and RBW: Rainbow) tested on 60 Atari games. Rank 1 always indicates "best" reliability, e.g. lowest IQR across runs. Significant pairwise differences in ranking between pairs of algorithms are indicated by black lines above the colored circles. ($\alpha = 0.05$ with Benjamini-Yekutieli correction, permutation test with # permutations = 1,000). Note that the best algorithms by median performance are not always the best algorithms on reliability. Error bars are 95% bootstrap confidence intervals (# bootstraps = 1,000).

performance from one evaluation point to the next. I.e., in Eq. 1, X represents the differences from one evaluation time-point to the next. We first compute the time-point to time-point differences on each training run. These differences are normalized by the distance between time-points, to ensure invariance to evaluation frequency (see Section 2.5). Then, we obtain the distribution of these differences, and find the α-quantile. Finally, we compute the expected value of the distribution below the α-quantile (for the upper tail, we use the $1 - \alpha$-quantile). This gives us the worst-case (or best-case) expected drop in performance during training, from one point of evaluation to the next.

Long-term Risk across Time (LRT): CVaR on Drawdown For this measure, we would also like to be able to capture whether an algorithm has the potential to lose a lot of performance relative to its peak, even if on a longer timescale, e.g. over an accumulation of small drops. For this measure, we apply CVaR to the ***Drawdown***. The Drawdown at time T is the drop in performance relative to the highest peak so far, and is another measure borrowed from economics (Chekhlov et al., 2005). I.e. $\text{Drawdown}_T = R_T - \max_{t<=T} R_t$. Like the SRT metric, the LRT can capture unusually large short-term drops in performance, but can also capture unusually large drops that occur over longer timescales.

Dispersion across Runs (DR): IQR across Runs Unlike the rest of the metrics described here, the dispersion across training runs has previously been used to characterize performance (e.g. Duan et al. (2016); Islam et al. (2017); Bellemare et al. (2017); Fortunato et al. (2017); Nagarajan et al. (2018)). This is usually measured by taking the variance or standard deviation across training runs at a set of evaluation points. We build on the existing practice by recommending first performing low-pass filtering of the training data, to filter out high-frequency variability within runs (this is instead measured using Dispersion across Time, DT). We also replace variance or standard deviation with robust statistics like IQR.

Risk across Runs (RR): CVaR across Runs In order to measure Risk across Runs (RR), we apply CVaR to the final performance of all the runs. This gives a measure of the expected performance of the worst runs.

Dispersion across Fixed-Policy Rollouts (DF): IQR across Rollouts When evaluating a fixed policy, we are interested in variability in performance when the same policy is rolled out multiple times. To compute this metric, we simply compute the IQR on the performance of the rollouts.

Risk across Fixed-Policy Rollouts (RF): CVaR across Rollouts This metric is similar to DF, except that we apply CVaR on the rollout performances.

2.5 Invariance to frequency of evaluation

Different experiments and different datasets may obtain evaluations at different frequencies during training. Therefore, the reliability metrics should be unbiased by the choice of evaluation frequency. As long as there are no cyclical patterns in performance, the frequency of evaluation will not bias any of the metrics except Long-Term Risk across Time (LRT). For all other metrics, changes in the frequency of evaluation will simply lead to more or less noisy estimates of these metrics. For LRT, comparisons should only be made if the frequency of evaluation is held constant across experiments.

3 Recommendations for reporting metrics and parameters

Whether evaluating an algorithm for practical use or for research, we recommend evaluating all of the reliability metrics described above. Each metric measures a different aspect of reliability, and can help pinpoint specific strengths and weaknesses of the algorithm. Evaluating the metrics is easy with the open-source Python package that will be released soon.

Reporting parameters. Even given our purposeful efforts to minimize the number of parameters in the reliability metrics, a few remain to be specified by the user that can affect the results, namely: window size (for Dispersion across Time), frequency threshold for low-pass and high-pass filtering (Dispersion across Time, Dispersion across Runs), evaluation frequency (only for Long-term Risk across Time), and length of training runs. Therefore, when reporting these metrics, these parameters need to be clearly specified, and must also be held constant across experiments for meaningful comparisons. The same is true for any other parameters that affect evaluation, e.g., the number of roll-outs per evaluation, the parameters of the environment, whether on-line or off-line evaluation is used, and the random seeds chosen.

Collapsing across evaluation points. Some of the in-training reliability metrics (Dispersion across Runs, Risk across Runs, and Dispersion across Time) need to be evaluated at multiple evaluation points along the training runs. Since it is useful to obtain a small number of values to summarize each metric, we recommend dividing the training run into "timeframes" (e.g. beginning, middle, and end), and collapsing across all evaluation points within each timeframe.

Normalization by performance. Different algorithms can have vastly different ranges of performance even on the same task, and variability in performance tends to scale with actual performance. Thus, we normalize our metrics in post-processing by a measure of the range of performance for each algorithm. For "during training" reliability, we recommend normalizing by the median range of performance, which we operationalize as the $p_{P_95} - p_{t=0}$, where p_{P_95} is the 95th percentile and

$p_{t=0}$ is the starting performance. For "after learning" reliability, the range of performance may not be available, in which case we use the median performance directly.

Ranking the algorithms. Because different environments have different ranges and distributions of reward, we must be careful when aggregating across environments or comparing between environments. Thus, if the analysis involves more than one environment, the per-environment median results for the algorithms should first be converted to ranks, by ranking all algorithms within each task. To summarize the performance of a single algorithm across multiple tasks, we may compute the mean ranking across tasks.

4 Confidence Intervals and Statistical Significance Tests for Comparison

4.1 Confidence intervals

We assume that the metric values have been converted to mean rankings, as explained in section 3. To obtain confidence intervals on the mean rankings for each algorithm, we apply bootstrap sampling on the runs, by resampling runs with replacement (Efron & Tibshirani, 1986).

For metrics that are evaluated per-run (e.g. Dispersion across Time), we can simply resample the metric values directly, and then recompute the mean rankings on each resampling to obtain a distribution over the rankings, allowing us to compute confidence intervals. For metrics that are evaluated across-runs, we need to resample the runs themselves, then evaluate the metrics on each resampling, before recomputing the mean rankings to obtain a distribution on the mean rankings.

4.2 Significance Tests for Comparing algorithms

Commonly, we would like to compare algorithms evaluated on a fixed set of environments. To determine whether any two algorithms have statistically significant differences in their metric rankings, one may perform an exact permutation test on each pair of algorithms. Such tests allow us to compute a p-value for the null hypothesis (probability that the methods are in fact indistinguishable on the reliability metric).

We designed our permutation tests based on the null hypothesis that runs are exchangeable across the two algorithms being compared. In brief, let A and B be sets of performance measurements for algorithms a and b. Let $Metric(X)$ be a reliability metric, e.g. the inter-quartile range across runs, computed on a set of measurements X. $MetricRanking(X)$ is the mean ranking across tasks on X, compared to the other algorithms being considered. We compute test statistic

$$s_{MetricRanking}(A, B) = MetricRanking(A) - MetricRanking(B).$$

Next we compute the distribution for $s_{MetricRanking}$ under the null hypothesis that the methods are equivalent, i.e. that performance measurements should have the same distribution for a and b. We do this by computing random partitions A', B' of $\{A \cup B\}$, and computing the test statistic $s_{MetricRanking}(A', B')$ on each partition. This yields a distribution for $s_{MetricRanking}$ (for sufficiently many samples), and the p-value can be computed from the percentile value of $s_{MetricRanking}(A, B)$ in this distribution. As with the confidence intervals, a different procedure is required for per-run vs across-run metrics. Please see Appendix A.3 for diagrams illustrating the permutation test procedures.

When performing pairwise comparisons between algorithms, it is critical to include corrections for multiple comparisons. This is because the probability of incorrect inferences increases with a greater number of simultaneous comparisons. We recommend using the Benjamini-Yekutieli method, which controls the false discovery rate (FDR), i.e., the proportion of rejected null hypotheses that are false.[4]

[4]For situations in which a user wishes instead to control the family-wise error rate (FWER; the probability of incorrectly rejecting at least one true null hypothesis), we recommend using the Holm-Bonferroni method.

4.3 REPORTING ON STATISTICAL TESTS

Itt is important to report the details of any statistical tests performed, e.g. which test was used, the significance threshold, and the type of multiple-comparisons correction used.

5 ANALYSIS OF RELIABILITY FOR COMMON ALGORITHMS AND ENVIRONMENTS

In this section, we provide examples of applying the reliability metrics to a number of RL algorithms and environments, following the recommendations described above.

5.1 CONTINUOUS CONTROL ALGORITHMS ON OPENAI GYM

We applied the reliability metrics to algorithms tested on seven continuous control environments from the Open-AI Gym (Greg Brockman et al., 2016) run on the MuJoCo physics simulator (Todorov et al., 2012). We tested REINFORCE (Sutton et al., 2000), DDPG (Lillicrap et al., 2015), PPO (Schulman et al., 2017), TD3 (Fujimoto et al., 2018), and SAC (Haarnoja et al., 2018) on the following Gym environments: Ant-v2, HalfCheetah-v2, Humanoid-v2, Reacher-v2, Swimmer-v2, and Walker2d-v2. We used the implementations of DDPG, TD3, and SAC from the TF-Agents library (Guadarrama et al., 2018). Each algorithm was run on each environment for 30 independent training runs.

Each algorithm was evaluated using a shared set of hyper-parameters for all environments. We used a blackbox optimizer (Golovin et al., 2017) to tune the hyperparameters on a per-task basis. During training, we evaluated the algorithms at a frequency of 1000 training steps. Each algorithm was run for a total of two million environment steps. For the "online" evaluations we used to generate training curves, we averaged returns over recent training episodes collected using the exploration policy as the policy evolves. For evaluations after learning on a fixed policy, we took the last checkpoint from each training run as the fixed policy for evaluation. Each of these policies was then evaluated for 30 rollouts, where each rollout was defined as 1000 environment steps.

5.2 DISCRETE CONTROL: DQN VARIANTS ON ATARI

We also applied the reliability metrics to the training data released as part of the Dopamine package (Castro et al., 2018). The data comprise the training runs of four RL algorithms, each applied to 60 Atari games. The RL algorithms are: DQN (Mnih et al., 2015), Implicit Quantile (Dabney et al., 2018), C51 (Bellemare et al., 2017), and a variant of Rainbow implementing the three most important components (Hessel & Modayil, 2018). The algorithms were trained on each game for 5 training runs. Hyper-parameters follow the original papers, but modified as necessary to follow Rainbow (Hessel & Modayil, 2018), to ensure apples-to-apples comparison.

During training, the algorithms were evaluated in an "online" fashion every 1 million frames, averaging across the training episodes as recommended for evaluations on the ALE (Machado et al., 2018). Each training run consisted of approximately 200 million Atari frames (rounding to the nearest episode boundary every 1 million frames). The raw training curves are shown in Appendix A.4. For evaluations after learning on a fixed policy ("after learning"), we took the last checkpoint from each training run as the fixed policies for evaluation. We then evaluated each of these policies for 125,000 environment steps.

5.3 PARAMETERS FOR RELIABILITY METRICS, CONFIDENCE INTERVALS, AND STATISTICAL TESTS

For the OpenAI Gym environments, we applied a sliding window of 100000 training steps for Dispersion across Time. For the Atari experiments, used a sliding window size of 25 on top of the evaluations for the Dispersion across Time. For metrics with multiple evaluation points, we divided each training run into 3 timeframes and averaged the metric rankings within each timeframe. Because the results were extremely similar for all three timeframes, we here report just the final timeframes.

Statistical tests for comparing algorithms were performed according to the recommendations in section 4. We used pairwise permutation tests using 10,000 permutations per test, with a significance threshold of 0.05 and Benjamini-Yekutieli multiple-comparisons correction.

5.4 MEDIAN PERFORMANCE

The median performance of an algorithm is not a reliability metric, but it is interesting to see side-by-side with the reliability metrics. For analyzing median performance for the DQN variants, we used the normalization scheme of (Mnih et al., 2015), where an algorithm's performance is normalized against a lower baseline (e.g. the performance of a random policy) and an upper baseline (e.g. the performance of a human): $P_{\text{normalized}} = \frac{P - B_{\text{lower}}}{B_{\text{upper}} - B_{\text{lower}}}$. Median performance was not normalized for the continuous control algorithms.

5.5 RESULTS

The reliability metric rankings are shown in Fig. 1 for the OpenAI Gym results. We see that, according to Median Performance during training, SAC and TD3 have the best performance and perform similarly well, while REINFORCE performs the worst. However, SAC outperforms TD3 on all reliability metrics during training. Furthermore, both SAC and TD3 perform relatively poorly on all reliability metrics after learning, despite performing best on median performance.

The reliability metric rankings are shown in Fig. 2 for the Atari results. Here we see a similar result that, even though Rainbow performs significantly better than IQN in Median Performance, IQN performs numerically or significantly better than Rainbow on many of the reliability metrics.

The differing patterns in these metrics demonstrates that reliability is a separate dimension that needs to be inspected separately from mean or median performance – two algorithms may have similar median performance but may nonetheless significantly different reliability, as with SAC and TD3 above. Additionally, these results demonstrate that reliability along one axis does not necessarily correlate with reliability on other axes, demonstrating the value of evaluating these different dimensions so that algorithms can be compared and selected based on the requirements of the problem at hand.

6 CONCLUSION

We have presented a number of novel metrics, designed to measure different aspects of reliability of RL algorithms. We motivated the design goals and choices made in constructing these metrics, and also presented practical recommendations for the measurement of reliability for RL. We also presented examples of applying these metrics to common RL algorithms and environments, and showed that these metrics can reveal strengths and weaknesses of an algorithm that are obscured when we only inspect mean or median performance.

REFERENCES

Carlo Acerbi and Dirk Tasche. Expected Shortfall: A Natural Coherent Alternative to Value at Risk. *Economic Notes*, 31(2):379–388, July 2002. ISSN 0391-5026, 1468-0300. doi: 10.1111/1468-0300.00091. URL http://doi.wiley.com/10.1111/1468-0300.00091.

Marc G. Bellemare, Will Dabney, and Rémi Munos. A Distributional Perspective on Reinforcement Learning. *arXiv:1707.06887 [cs, stat]*, July 2017. URL http://arxiv.org/abs/1707.06887. arXiv: 1707.06887.

Nicole Bäuerle and Jonathan Ott. Markov Decision Processes with Average-Value-at-Risk criteria. *Mathematical Methods of Operations Research*, 74(3):361–379, December 2011. ISSN 1432-2994, 1432-5217. doi: 10.1007/s00186-011-0367-0. URL http://link.springer.com/10.1007/s00186-011-0367-0.

Pablo Samuel Castro, Subhodeep Moitra, Carles Gelada, Saurabh Kumar, and Marc G. Bellemare. Dopamine: A research framework for deep reinforcement learning. *CoRR*, abs/1812.06110, 2018. URL http://arxiv.org/abs/1812.06110.

Alexei Chekhlov, Stanislav Uryasev, and Michael Zabarankin. Drawdown measure in portfolio optimization. *International Journal of Theoretical and Applied Finance*, 8(1):46, 2005.

Yinlam Chow and Mohammad Ghavamzadeh. Algorithms for CVaR Optimization in MDPs. *Advances in Neural Information Processing Systems*, pp. 9, 2014.

Will Dabney, Georg Ostrovski, David Silver, and Rémi Munos. Implicit Quantile Networks for Distributional Reinforcement Learning. *Thirty-fifth International Conference on Machine Learning*, pp. 10, 2018.

Yan Duan, Xi Chen, Rein Houthooft, John Schulman, and Pieter Abbeel. Benchmarking Deep Reinforcement Learning for Continuous Control. In *International Conference on Machine Learning*, pp. 1329–1338, June 2016. URL http://proceedings.mlr.press/v48/duan16.html.

B. Efron and R. Tibshirani. Bootstrap Methods for Standard Errors, Confidence Intervals, and Other Measures of Statistical Accuracy. *Statistical Science*, 1(1):54–75, February 1986. ISSN 0883-4237, 2168-8745. doi: 10.1214/ss/1177013815. URL http://projecteuclid.org/euclid.ss/1177013815.

Meire Fortunato, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, Ian Osband, Alex Graves, Vlad Mnih, Remi Munos, Demis Hassabis, Olivier Pietquin, Charles Blundell, and Shane Legg. Noisy Networks for Exploration. *arXiv:1706.10295 [cs, stat]*, June 2017. URL http://arxiv.org/abs/1706.10295. arXiv: 1706.10295.

Scott Fujimoto, Herke van Hoof, and David Meger. Addressing Function Approximation Error in Actor-Critic Methods. *arXiv:1802.09477 [cs, stat]*, February 2018. URL http://arxiv.org/abs/1802.09477. arXiv: 1802.09477.

Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D. Sculley. Google Vizier: A Service for Black-Box Optimization. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '17*, pp. 1487–1495, Halifax, NS, Canada, 2017. ACM Press. ISBN 978-1-4503-4887-4. doi: 10.1145/3097983.3098043. URL http://dl.acm.org/citation.cfm?doid=3097983.3098043.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym, 2016.

Sergio Guadarrama, Anoop Korattikara, Pablo Castro Oscar Ramirez, Ethan Holly, Sam Fishman, Ke Wang, Chris Harris Ekaterina Gonina, Vincent Vanhoucke, and Eugene Brevdo. TF-Agents: A library for reinforcement learning in tensorflow. https://github.com/tensorflow/agents, 2018. URL https://github.com/tensorflow/agents. [Online; accessed 30-November-2018].

Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor. *arXiv:1801.01290 [cs, stat]*, January 2018. URL http://arxiv.org/abs/1801.01290. arXiv: 1801.01290.

James D. Hamilton. *Time Series Analysis*. Princeton University Press, 1994.

Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep Reinforcement Learning that Matters. *arXiv:1709.06560 [cs, stat]*, September 2017. URL http://arxiv.org/abs/1709.06560. arXiv: 1709.06560.

Matteo Hessel and Joseph Modayil. Rainbow: Combining Improvements in Deep Reinforcement Learning. *AAAI*, pp. 8, 2018.

Riashat Islam, Peter Henderson, Maziar Gomrokchi, and Doina Precup. Reproducibility of Benchmarked Deep Reinforcement Learning Tasks for Continuous Control. *arXiv:1708.04133 [cs]*, August 2017. URL http://arxiv.org/abs/1708.04133. arXiv: 1708.04133.

Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv:1509.02971 [cs, stat]*, September 2015. URL http://arxiv.org/abs/1509.02971. arXiv: 1509.02971.

Marlos C. Machado, Marc G. Bellemare, Erik Talvitie, Joel Veness, Matthew Hausknecht, and Michael Bowling. Revisiting the Arcade Learning Environment: Evaluation Protocols and Open Problems for General Agents. *Journal of Artificial Intelligence Research*, 61:523–562, March 2018. ISSN 1076-9757. doi: 10.1613/jair.5699. URL https://www.jair.org/index.php/jair/article/view/11182.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, February 2015. ISSN 1476-4687. doi: 10.1038/nature14236. URL https://www.nature.com/articles/nature14236/.

Prabhat Nagarajan, Garrett Warnell, and Peter Stone. Deterministic Implementations for Reproducibility in Deep Reinforcement Learning. *arXiv:1809.05676 [cs]*, September 2018. URL http://arxiv.org/abs/1809.05676. arXiv: 1809.05676.

Charles R Nelson and Charles I Plosser. Trends and random walks in macroeconomic time series. *Journal of Monetary Economics*, 10:139–162, 1982.

Peter J. Rousseeuw and Christophe Croux. Alternatives to the MedianAbsolute Deviation. *Journal of the American Statistical Association*, 1993.

Said E. Said and David A. Dickey. Testing for unit roots in autoregressive-moving average models of unknown order. *Biometrika*, 71(3):599–607, 1984.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms. *arXiv:1707.06347 [cs]*, July 2017. URL http://arxiv.org/abs/1707.06347. arXiv: 1707.06347.

Joseph P. Simmons, Leif D. Nelson, and Uri Simonsohn. False-Positive Psychology: Undisclosed Flexibility in Data Collection and Analysis Allows Presenting Anything as Significant. *Psychological Science*, 22(11):1359–1366, November 2011. ISSN 0956-7976, 1467-9280. doi: 10.1177/0956797611417632. URL http://journals.sagepub.com/doi/10.1177/0956797611417632.

Richard S Sutton, David A McAllester, Satinder P Singh, and Yishay Mansour. Policy Gradient Methods for Reinforcement Learning with Function Approximation. In *NIPS'99 Proceedings of the 12th International Conference on Neural Information Processing Systems*, pp. 7, 2000.

Aviv Tamar, Yonatan Glassner, and Shie Mannor. Optimizing the CVaR via Sampling. *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, pp. 7, 2015.

Emanuel Todorov, Tom Erez, and Yuval Tassa. MuJoCo: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 5026–5033, Vilamoura-Algarve, Portugal, October 2012. IEEE. ISBN 978-1-4673-1736-8 978-1-4673-1737-5 978-1-4673-1735-1. doi: 10.1109/IROS.2012.6386109. URL http://ieeexplore.ieee.org/document/6386109/.

A Appendix

A.1 Assumptions and Definitions

Reinforcement Learning algorithms vary widely in design, and our metrics are based on certain notions that should span the gamut of RL algorithms.

Policy A policy $\pi_\Theta(a_i|s_i)$ is a distribution over actions a_i given a current (input) state s_i. We assume policies are parameterized by a parameter Θ.

Agent An agent is defined as a distribution over policies (or equivalently a distribution over parameters Θ). In many cases, an agent will be a single policy but for population-based RL methods, the agent is a discrete set of policies.

Window A window is a collection of states over which the agent is assumed to have small variation. A window could be a sequence of consecutive time steps for a sequential RL algorithm, or a collection of states at the same training step of a distributed RL algorithm with a parameter server (all agents share Θ).

Performance The performance of an agent is the mean or median per-epoch reward from running that agent. If the agent is a single policy, then the performance $p(\pi_\Theta)$ is the mean or median per-epoch reward for that agent. If the agent is a distribution $D(\Theta)$ of policies, then the performance is the median of $p(\pi_\Theta)$ with $\Theta \sim D$.

Training Run A training run is a sequence of updates to the agent $D(\Theta)$ from running a reinforcement learning algorithm. It leads to a trained agent $D_{final}(\Theta)$. Multiple training runs share no information with each other.

We cannot directly measure performance since it is a statistic across an infinite sample of evaluation runs of an agent. Instead we use windows to compute sample medians to approximate performance.

A.2 Detrending by differencing

Typically, de-trending can be performed in two main ways (Nelson & Plosser, 1982; Hamilton, 1994). Differencing (i.e. $y_t\prime = y_t - y_{t-1}$) is more appropriate for difference-stationary (DS) processes (e.g. a random walk: $y_t = y_{t-1} + b + \epsilon_t$), where the shocks ϵ_t accumulate over time. For trend-stationary (TS) processes, which are characterized by stationary fluctuations around a deterministic trend, e.g. $y_t = a + b * t + \epsilon_t$, it is more appropriate to fit and subtract that trend.

We performed an analysis of real training runs and verified that the data are indeed approximately DS, and that differencing does indeed remove the majority of time-dependent structure. For this analysis we used the training runs on Atari as described in 5.2. Before differencing, the Augmented Dickey-Fuller test (ADF test, also known as a difference-stationarity test; Said E. Said & David A. Dickey (1984)) rejects the null hypothesis of a unit root on only 72% of the runs; after differencing, the ADF test rejects the null hypothesis on 92% of the runs (p-value threshold 0.05). For the ADF test, the rejection of a unit root (of the autoregressive lag polynomial) implies the alternate hypothesis, which is that the time series is trend-stationary.

Therefore, our training curves are better characterized as an accumulation of shocks, i.e. as DS processes, rather than as mean-reverting TS processes. They are not actually purely DS because the shocks ϵ_t are not stationary over time, but because we compute standard deviation within sliding windows, we can capture the non-stationarity and change in variability over time. Thus, we chose to detrend using differencing.

As a further note in favor of detrending by differencing, it is useful to observe that the standard deviation is defined relative to a mean, as $\sigma = \sqrt{E[(X-\mu)^2]}$. On the raw data (without differencing), the standard deviation would be defined relative to μ as average performance, so that any improvements in performance are included in that computation of variance. On the other hand, if we compute variance on the 1st-order differences, we are using a μ that represents the average *change* in performance, which is a more reasonable baseline to compute variance against, when we are concerned with the variability of those changes. A final benefit of differencing is that it is parameter-free.

A.3 Illustrations of Permutation Test Procedures

In Figures 3 and 4, we show diagrams illustrating the procedure for computing permutation tests to compare pairs of algorithms, on a given metric.

A.4 Raw training curves for OpenAI Mujoco tasks

In Figure 5, we show the raw training curves for the TF-Agents implementations of continuous-control algorithms, applied to the OpenAI Mujoco tasks. These are compared against baselines from the literature, where available (DDPG and TD3: Fujimoto et al. (2018), PPO: Schulman et al. (2017), SAC: Haarnoja et al. (2018))

Raw training curves for Dopamine implementations of the DQN-variants, applied to the Atari tasks, are available here: https://google.github.io/dopamine/baselines/plots.html.

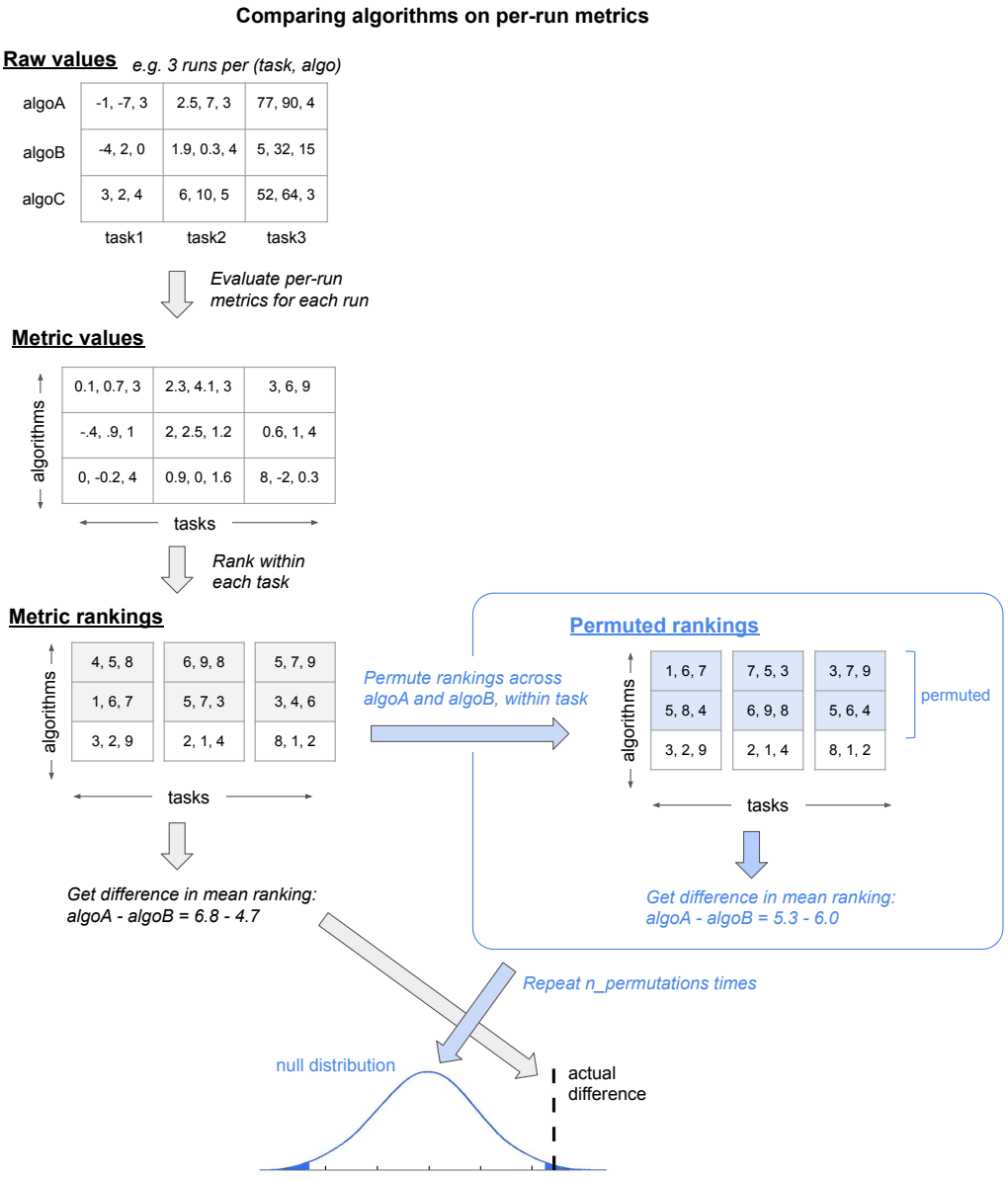

Figure 3: Diagram illustrating the computation of the permutation tests for **per-run metrics** (Dispersion across Time, Short-term Risk across Time, Long-term Risk across Time). In this example, we are comparing Algorithm A and Algorithm B, and there are only 3 algorithms, 3 tasks, and 3 runs per (task, algo) pair. To compute the difference in average rankings for two algorithms, follow the gray arrows. To compute a null distribution of difference in average rankings (by permuting the runs), follow the blue arrows a number of times (e.g. 1,000 times). Once the null distribution has been computed, the actual value of the difference can be compared with the null distribution to obtain a p-value.

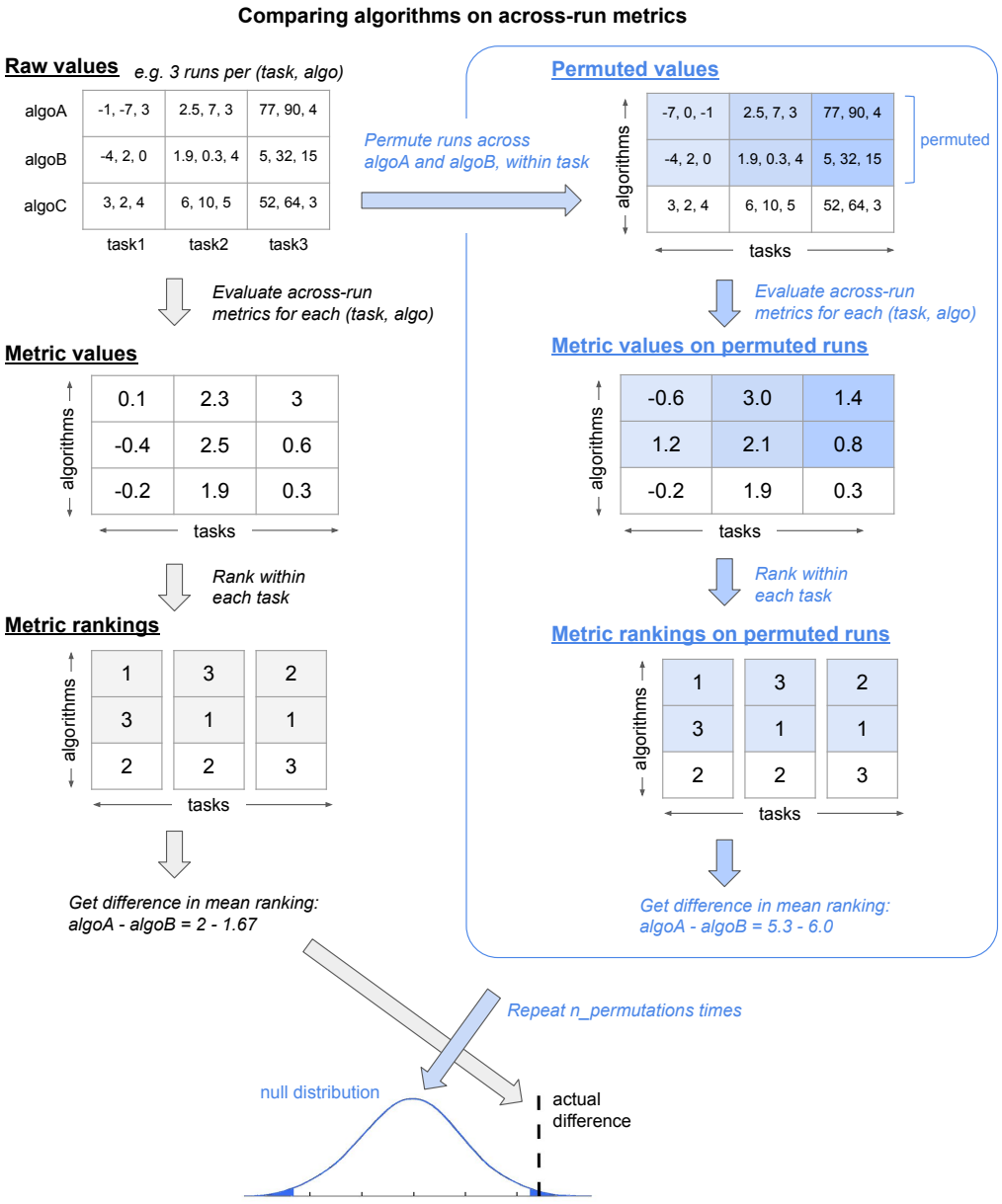

Figure 4: Diagram illustrating the computation of the permutation tests for **across-run or across-rollout metrics** (Dispersion across Runs, Risk Across Runs, Dispersion across Fixed-policy rollouts, Risk across Fixed-Policy rollouts). In this example, we are comparing Algorithm A and Algorithm B, and there are only 3 algorithms, 3 tasks, and 3 runs per (task, algo) pair. To compute the difference in average rankings for two algorithms, follow the gray arrows. To compute a null distribution of difference in average rankings (by permuting the runs), follow the blue arrows a number of times (e.g. 1,000 times). Once the null distribution has been computed, the actual value of the difference can be compared with the null distribution to obtain a p-value.

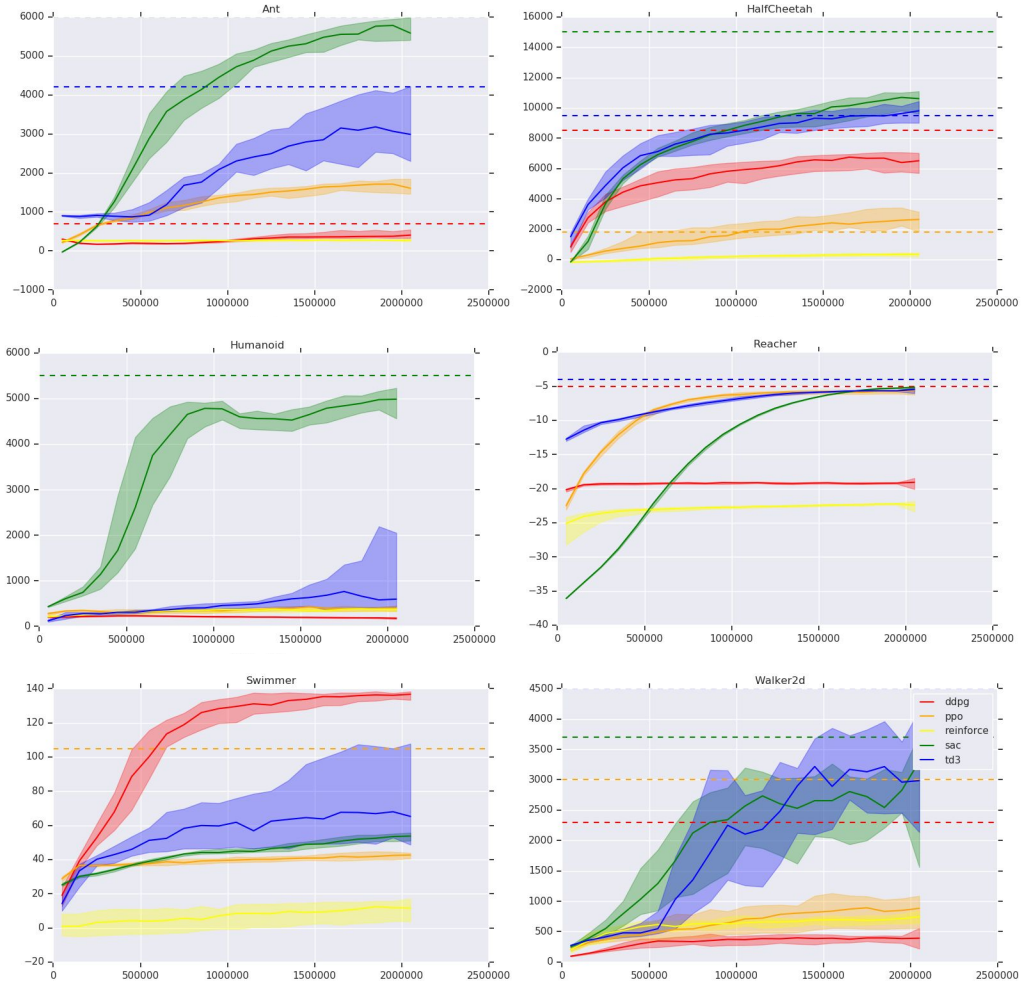

Figure 5: Raw training curves for OpenAI Mujoco tasks. Dotted lines indicate baseline performance from the literature, where available.