A Method for Evaluating Hyperparameter Sensitivity in Reinforcement Learning

Jacob Adkins
University of Alberta
Edmonton, AB T6G 2R3
jadkins@ualberta.ca

Michael Bowling
University of Alberta
Edmonton, AB T6G 2R3
mbowling@ualberta.ca

Adam White
University of Alberta
Edmonton, AB T6G 2R3
amw8@ualberta.ca

Abstract

The number of hyperparameters used in deep reinforcement learning algorithms has expanded rapidly. Hyperparameters often have complex nonlinear interactions, significantly impact performance, and are difficult to tune across sets of environments. This creates a challenge for practitioners who wish to apply reinforcement learning algorithms to new domains. Several methods have been proposed to study the relationship between algorithms and their hyperparameters, but the community lacks a widely accepted measure for characterizing hyperparameter sensitivity across sets of environments. We propose an empirical methodology for studying the relationship between an algorithm's hyperparameters and its performance over sets of environments. Our methodology enables practitioners to better understand the degree to which an algorithm's reported performance is attributable to per-environment hyperparameter tuning. We use our empirical methodology to assess how several commonly used normalization variants affect the hyperparameter sensitivity of PPO. The results suggest that the evaluated normalization variants, which improve performance, also increase hyperparameter sensitivity, indicating that several algorithmic performance improvements may be a result of an increased reliance on hyperparameter tuning.

1 Introduction

The performance of deep reinforcement learning (DRL) algorithms critically relies on the tuning of numerous hyperparameters, and with the introduction of each new algorithm, the number of these critical hyperparameters continues to grow. This increase can be observed in the progression of value-based DRL algorithms, starting from DQN (Mnih et al., 2015) which has 16 hyperparameters that the practitioner must choose, to Rainbow (Hessel et al., 2018) with 25 hyperparameters. [1] This proliferation is problematic because performance can vary drastically with respect to hyperparameters across environments. Often, small changes in a hyperparameter can lead to drastic changes in performance, and different environments require very different hyperparameter settings to achieve the reported good performances (Franke et al., 2021; Eimer et al., 2022; 2023; Patterson et al., 2024). Generally speaking, hyperparameter tuning requires a combinatorial search and thus many published results are based on a mix of default hyperparameter settings and informal hand-tuning

[1] Similar examples of algorithm development demonstrating a proliferation of hyperparameters can be observed in policy-gradient methods and model-based reinforcement learning (See Appendix A).

of key hyperparameters like the learning rate. Our standard evaluation methodologies do not reflect the sensitivity of performance to hyperparameter choices, and this is compounded by a lack of suitable measures to characterize said sensitivities.

There are many different ways one could characterize performance with respect to hyperparameter choices in reinforcement learning (RL), but the community lacks an agreed standard. Hyperparameter sensitivity curves, such as those found in the introductory textbook (Sutton & Barto, 2018), summarize performance with respect to several values of a key hyperparameter producing U-shaped curves. Sensitivity curves do not work well with many dimensions of hyperparameters, nor is there a well established way to use them to summarize performance in multiple environments. If computation is of no object, then performance percentiles can be used to compute the likelihood that an algorithm will perform well if its hyperparameters are randomly sampled from some distribution (Jordan et al., 2020). Although very general, this approach does not reflect how practitioners tune their algorithms. The Cross-environment Hyperparameter Benchmark (Patterson et al., 2024) compares algorithms by a mean normalized performance score across environments but ultimately focuses on the possibility of finding a single good setting of an algorithm's hyperparameters that performs well rather than characterizing sensitivity. What performance-only metrics lack is a measurement of what proportion of realized performance from per-environment hyperparameter tuning.

We propose an empirical methodology to better understand the interplay between hyperparameter tuning and an RL agent's performance. Our methodology consists of two metrics and graphical techniques for studying them. The first metric, called an algorithm's *hyperparameter sensitivity*, measures the degree to which an algorithm's peak reported performance relies upon per-environment hyperparameter tuning. This metric captures the degree to which per-environment tuning improves performance relative to the performance of the best-fixed hyperparameters setting across a distribution of environments. The second metric, named *effective hyperparameter dimensionality*, measures how many hyperparameters must be tuned to achieve near-peak performance. It is often unclear how important specific hyperparameters are and if they should be included in the tuning process. With these two metrics, we seek to gain insights into existing algorithms and drive research toward algorithmic improvements that reduce hyperparameter sensitivity.

We validate the utility of our methodology by studying several variants of PPO (Schulman et al., 2017) (described in Appendix C) that have been purported to reduce hyperparameter sensitivity and increase performance. We performed one of the first very large-scale hyperparameter sweeps over variants of the PPO algorithm consisting of over 4.3 million runs (13 trillion environment steps) in the Brax MuJoCo domains (Freeman et al., 2021). We investigate the relationship between performance and hyperparameter sensitivity with several commonly used normalization variants paired with PPO. We found that normalization variants, which increased PPO's tuned performance, also increased sensitivity. Other normalization variants had negligible effects on performance and marginal effects on hyperparameter sensitivity. This result contrasts the view that normalization makes DRL algorithms easier to tune and, as a consequence, results in improved performance.

2 Problem Setting and Notation

We formalize the agent-environment interaction as a continuing Markov Decision Process (MDP) with finite state space $\mathcal{S}$, finite action space A, bounded reward function $\mathcal{R}$: $\mathcal{S} \times A \times \mathcal{S} \to \mathcal{R} \subset \mathbb{R}$, transition function $\mathcal{P} : \mathcal{S} \times A \times \mathcal{S} \to [0, 1]$, and discount factor γ. At each timestep t, the agent observes the current state S_t, selects an action A_t, the environment outputs a scalar reward R_{t+1} and transitions to a new state S_{t+1}. The state-value function v_π: $\mathcal{S} \to \mathbb{R}$ is the state conditioned expected return $G_t \doteq R_{t+1} + \gamma R_{t+2} + ...$ following policy π defined as $v_\pi(s) \doteq \mathbb{E}_\pi[G_t|S_t = s]$. Similarly, the action-value function q_π: $\mathcal{S} \times A \to \mathbb{R}$ provides the state and action conditioned expected return following policy π defined by $q_\pi(s, a) \doteq \mathbb{E}_\pi[G_t|S_t = s, A_t = a]$. The agent's goal is to find a policy π which maximizes $v_\pi(s)$ for all $s \in \mathcal{S}$. The advantage function $A_\pi(s, a) \doteq q_\pi(s, a) - v_\pi(s)$ describes how much better taking an action a in state s is rather than sampling an action according to $\pi(\cdot|s)$, following policy π afterward. Given an estimate of the value function $\hat{v}$, an agent can update estimates of the value of states based on estimates of the values of successor states. An

n-step return is defined as $G_{t:t+n} \doteq R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n \hat{v}\left(S_{t+n}, \phi\right)$. A mixture of n-step returns, called a truncated-λ return can be created by weighting n-step returns by a factor $\lambda \in [0, 1]$, $G^{\lambda}_{t:t+n} \doteq (1-\lambda) \sum_{j=1}^{n-1} \lambda^{j-1} G_{t:t+j} + \lambda^{n-1} G_{t:t+n}$. Truncated-$\lambda$ returns are often used in DRL algorithms such as PPO (Schulman et al., 2017) to create estimates of state values and state-action advantages, which are used to approximate the value function and improve the policy.

Of particular interest to this work is the setting where an empiricist is evaluating a set of algorithms Ω across a set of environments $\mathcal{E}$. Each algorithm $\omega \in \Omega$ is presumed to have some number of hyperparameters $n(\omega)$. Each hyperparameter h_i, $1 \leq i \leq n(\omega)$ is chosen from some set of choices H^{ω}_i. The total hyperparameter space H^{ω} is defined as a Cartesian product of those choices $H^{\omega} \doteq H^{\omega}_1 \times H^{\omega}_2 \times ... H^{\omega}_{n(\omega)}$. Once an algorithm $\omega \in \Omega$ and a hyperparameter setting $h \in H^{\omega}$ are chosen, the tuple (ω, h) specifies an *agent*.

3 Hyperparameter Sensitivity

The role that hyperparameters play in affecting the performance of RL algorithms is under-studied. A *sensitive* algorithm is an algorithm that requires a great deal of per-environment hyperparameter tuning to obtain high performance. How much hyperparameter tuning goes into obtaining state-of-the-art (SOTA) results often goes unreported, and there is a danger of a cascading effect of algorithms becoming more sensitive and reliant upon unreported hyperparameter optimization (HPO) tuning procedures for performance. This section presents two contributions: a metric for assessing an algorithm's hyperparameter sensitivity, and a method of graphically analyzing the relationship between a hyperparameter sensitivity and performance along a 2-dimensional plane.

These tools may be used to develop a deeper understanding of existing algorithms and we hope will aid researchers in evaluating algorithms more holistically along dimensions other than just benchmark performance.

3.1 Sensitivity Metric

We want a performance metric that summarizes the learning process. Reinforcement learning agents learn online. As agents interact with their environment, they learn to maximize return. Not only are the returns realized by the final policy important but also the rate of learning. The chosen performance metric is the average return obtained during learning, also called the area under the curve (AUC) $p(\omega, e, h, \kappa)$ where $\omega \in \Omega$ is an algorithm, $e \in \mathcal{E}$ is an environment, $h \in H^{\omega}$ is a hyperparameter setting, and $\kappa \in \mathcal{K} \subset \mathbb{N}$ is the random number generator (RNG) seed. We choose area under the curve (AUC) as the performance metric $p(\omega, e, h)$ where $\omega \in \Omega$ is an algorithm, $e \in \mathcal{E}$ is an environment, $h \in H^{\omega}$ is a hyperparameter setting. Performance distributions are often skewed and multi-modal (Patterson et al., 2023). Many runs are needed to accurately estimate expected performance. We report performance after averaging over 200 runs and compute 95% bootstrap confidence intervals around computed statistics.

It is crucial to capture performance across sets of environments. Our choice of AUC as a performance metric does not allow for cross-environment performance comparisons directly as the magnitudes of returns vary between environments. Consider the distributions of performance presented in the left plot in Figure 4. The performance realized by good hyperparameter settings in Halfcheetah is orders of magnitude greater than the performance of good hyperparameter settings in Swimmer. Nevertheless, just because the absolute magnitude is lower or the range of observed performances is tighter, that does not mean the differences are any less significant. Thus, in order to consider how hyperparameters perform across sets of environments, we need to normalize performance to a standardized score.

In this work, use min-max scoring, although other normalization methods (Jordan et al., 2020), could be used. After conducting a large number of runs across different algorithms, environments, and hyperparameter settings, for each environment, find the maximum and minimum performance scores observed, $p_{\max}(e) \doteq \max_{\omega,e,h,\kappa} p(\omega, e, h, \kappa)$ and $p_{\min}(e) \doteq \min_{\omega,e,h,\kappa} p(\omega, e, h, \kappa)$. Then, for each algorithm, environment, and hyperparameter setting, the *normalized environment score* Γ :

$A \times E \times H^{\omega} \to [0, 1]$ is obtained by squashing performance

$$\Gamma(a, e, h) \doteq \frac{\hat{p}(a, e, h) - p_{\min}(e)}{p_{\max}(e) - p_{\min}(e)} \tag{3.1}$$

Note the right plot in Figure 4; the distributions of normalized scores for hyperparameter settings in Swimmer and Halfcheetah lie in a common range. Normalized scores allow practitioners to determine which fixed hyperparameter settings do well across multiple environments. A practitioner can find the hyperparameter setting that maximizes the mean normalized score across a set of environments. Consider the performance of the hyperparameter denoted by the blue stars in Figure 5. This setting performs in the top quartile of hyperparameter settings in both Swimmer and Halfcheetah. In contrast, consider the hyperparameter setting denoted by the red stars which was chosen to maximize Halfcheetah performance. While this hyperparameter setting sits at the top of the distribution for Halfcheetah, it performs rather poorly in Swimmer. This gap displays how much performance can be gained by tuning the hyperparameter for Halfcheetah. That is, how sensitive the algorithm is to per-environment hyperparameter tuning.

Given an algorithm $\omega \in \Omega$, we define its *hyperparameter sensitivity* Φ as follows:

$$\Phi(a) \doteq \frac{1}{|\mathcal{E}|} \sum_{e \in \mathcal{E}} \max_{h \in H^{\omega}} \Gamma(a, e, \omega) - \max_{h \in H^{\omega}} \frac{1}{|\mathcal{E}|} \sum_{e \in \mathcal{E}} \Gamma(a, e, \omega) \tag{3.2}$$

The hyperparameter sensitivity of an algorithm is the difference between its *per-environment tuned score*, the average normalized environment score with hyperparameters tuned per environment, and its *cross-environment tuned score*, the normalized environment score of the best-fixed hyperparameter setting across the distribution of environments. We can use this definition of hyperparameter sensitivity as a metric for gaining insights into algorithms. We seek to combine hyperparameter sensitivity with the existing performance-only paradigm of evaluation for a deeper understanding of algorithms.

3.2 Sensitivity Analysis

Modern RL algorithms are complex learning systems. Understanding them is a task that requires multiple dimensions of analysis. Benchmark performance has been the primary metric (and often the only one) used for evaluating algorithms. However, this is only one dimension along which algorithms can be measured. Hyperparameter sensitivity is an important dimension to consider in the evaluation space, especially as practitioners begin to apply RL algorithms to real-world applications. We propose the performance-sensitivity plane to aid in better understanding algorithms.

Consider the performance-sensitivity plane shown in Figure 1. To construct the plane, the center point is set to the hyperparameter sensitivity and per-environment tuned score of some reference point algorithm. We can consider how other algorithms relate to this reference point by considering which region of the plane they occupy. There are 5 regions of interest shaded by different colors and labeled numerically, which we will consider in turn.

An ideal algorithm would be both more performative and less sensitive. Therefore, algorithms that rest in **Region 1** (the top left quadrant) of the plane would be a strict improvement over the reference point algorithm. For some applications, perhaps additional sensitivity can be tolerated if the gains in performance are large enough. Algorithms that lie in **Region 2** are an example of this. The region represents algorithms whose increase in performance is greater than the corresponding increase in sensitivity. Conversely, for some applications sensitivity may matter a great deal and some performance loss can be endured. Algorithms that sit in **Region 3** are an example of those whose decrease in sensitivity outmatches their corresponding decrease in performance. Regions 1-3 represent algorithms that have notable redeeming qualities either in terms of performance, hyperparameter sensitivity, or both. However, perhaps a practitioner does not care about sensitivity. E.g., they want to maximize a score of a specific benchmark, and hyperparameter tuning is of no issue. Algorithms sitting in **Region 4** may be adequate as they are algorithms that demonstrate

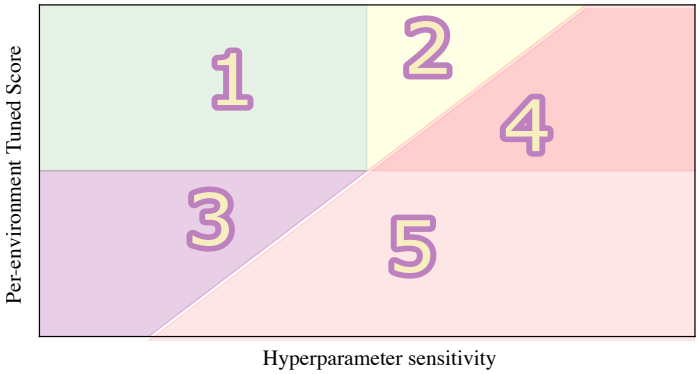

Figure 1: The performance-sensitivity plane for algorithmic evaluation. The center point indicates the hyperparameter sensitivity and performance of a reference point algorithm. The x-axis is the hyperparameter sensitivity metric as defined in equation 3.2. The y-axis is the per-environment tuned score (first term in equation 3.2). The diagonal line is the identity line shifted to intersect the reference point algorithm. The plane is then divided into 5 shaded regions that represent spaces of algorithms of varying qualities relative to the baseline.

performance improvements and an even higher reliance upon per-environment hyperparameter tuning. Finally, those unfortunate algorithms that live in **Region 5** are in a space with both lower performance and higher sensitivity, making them undesirable.

Often, new algorithms are created by modifying old algorithms; e.g. a target is normalized, a regularization term is added to the loss function, gradients are clipped, etc. A natural application of this diagram is to set the reference point to the hyperparameter sensitivity and performance of the original algorithm and study how the proposed modifications affect both sensitivity and performance. We illustrate an example of this using PPO as a reference point in Section 4.

4 Sensitivity Experiments

To illustrate the utility of the sensitivity analysis presented above, we performed an experiment to study the hyperparameter sensitivity and performance of several variants of the PPO algorithm, a widely used policy-gradient method in deep reinforcement learning (Schulman et al., 2017). We considered several normalization variants commonly used in PPO implementations (Andrychowicz et al., 2020; Huang et al., 2022) and some normalization variants, introduced in DreamerV3, that were purported to reduce hyperparameter sensitivity (Hafner et al., 2023; Sullivan et al., 2023).

4.1 Sensitivity Experiment with PPO variants

Consider the performance-sensitivity plane presented in Figure **??**. The reference point at the center is PPO's hyperparameter sensitivity and performance found without normalization. The error bars around performance and sensitivity are computed by from a 10000 sample 95% bootstrap confidence intervals. First, note that none of the normalization variants resulted in an improvement that both raised performance and lowered sensitivity. Observation zero-mean normalization enhanced performance with some increase in sensitivity (Region 2). The performance gain observed is consistent with previous findings Andrychowicz et al. (2020), which claim observation zero-mean normalization to be the most important PPO normalization variant for performance. However, this performance gain comes with a trade-off: an increase in hyperparameter sensitivity. Applying the symlog function to observations (as was done in DreamerV3) also raises performance but less so than the corresponding gain in sensitivity (placing it in region 4). Applying the symlog function to the value function target slightly lowered performance and increased sensitivity (region 5). Advantage zero-mean normalization had a similar effect, slightly lowering performance and increasing sensitivity (region 5). Both forms of advantage percentile scaling slightly lowered performance and also lowered sensitivity. Due to the width of the confidence intervals, it is unclear if they should be classified into regions 3 or 5.

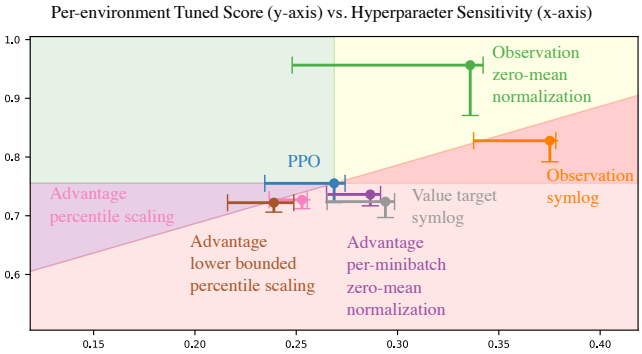

Figure 2: Performance-sensitivity plane with PPO as center reference point. Variants of PPO plotted. The x-axis indicates hyperparameter sensitivity as defined in equation 3.2. The y-axis represents the per-environment tuned score (first term in the sensitivity calculation of equation 3.2). Hyperparameter sensitivity and per-environment tuned score metrics were computed from a 200 run sweep of 625 hyperparameter settings across 5 Brax Mujoco environments (Ant, Halfcheetah, Hopper, Swimmer, and Walker2d). Error bars show the endpoints of 10000 sample 95% bootstrap confidence intervals around the performance and hyperparameter sensitivity statistics.

The most significant finding is that both forms of observation normalization appear to increase PPO's hyperparameter sensitivity. Performance gains observed by applying the symlog function to observations seem to be strictly due to an increased reliance on per-environment HPO. It is difficult to draw conclusions on the effects of the other normalization variants as there is significant overlap in the confidence intervals. The performance-sensitivity plane allows for insights into how algorithmic changes alter an algorithm's reliance on per-environment HPO tuning procedures with no additional computational expense other than what is already required from a grid search.

The performance-sensitivity plane allows for a richer understanding of algorithms than performance-only evaluation procedures, but it does not capture the full picture. Two algorithms can sit in the same location on the plane and yet have very different hyperparameter characteristics. Consider the case where there are two algorithms. The first algorithm is highly sensitive with respect to one hyperparameter, which needs to be carefully tuned per environment. The second algorithm has the same sensitivity but needs to be tuned per environment for dozens of hyperparameters with complex interactions. The hyperparameter sensitivity would not differentiate between these two algorithms. An additional metric is needed.

5 Effective Hyperparameter Dimensionality

There are many cases where a practitioner can tune some but not all of an algorithm's tunable hyperparameters. It may be the case that if a few key hyperparameters are tuned per environment, then a preponderance of an algorithm's potential performance can be gained. This motivates the definition of *effective hyperparameter dimensionality*, a metric that measures how many hyperparameters must be tuned in order to obtain near-peak performance.

For a given algorithm ω with hyperparameter space H^ω, number of tunable hyperparameters $n(\omega)$, and environment set $\mathcal{E}$, let $h^* \doteq \arg\max_{h \in H^\omega} \frac{1}{|\mathcal{E}|} \sum_{e \in \mathcal{E}} \Gamma(\omega, e, h)$ be the hyperparameter setting which maximizes the cross-environment tuned score. Define a similarity function $\rho : H^\omega \to [n(\omega)]$ that counts the number of hyperparameters in common with h^*, $\rho(h) = \sum_{i=1}^{n(\omega)} \mathbb{1}[h_i = h_i^*]$.

The *effective hyperparameter dimensionality* $d(\omega)$ is defined as

$$\begin{aligned} d(\omega) &= \max_{h \in H^\omega} \rho(h) \\ \text{s.t.} \quad & \frac{1}{|\mathcal{E}|} \sum_{e \in \mathcal{E}} \Gamma(\omega, e, h) \geq 0.95 \frac{1}{|\mathcal{E}|} \sum_{e \in \mathcal{E}} \max_{h' \in H^\omega} \Gamma(\omega, e, h') \end{aligned} \tag{5.3}$$

The hyperparameter dimensionality is the maximal number of hyperparameters that can be left to default (setting that maximizes cross-environment tuned performance) while retaining the majority

of the performance that can be realized by tuning per environment. The threshold of 95% of peak performance can be changed at a practitioner's discretion to whatever meets their performance requirements.

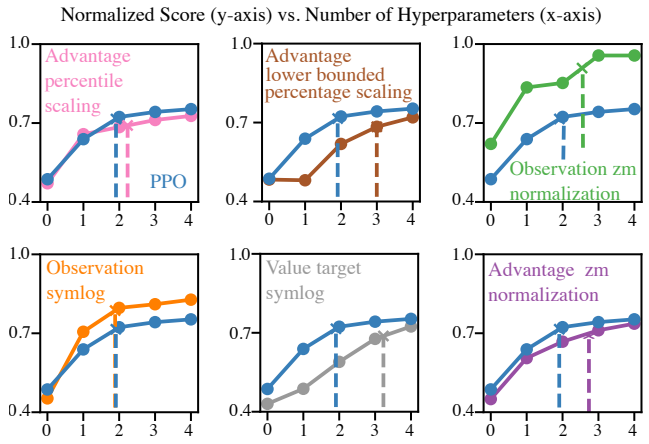

Figure 3: The performance as a function of the number of hyperparameters tuned per environment. The subplots compares PPO to the PPO variants studied. The x-axis indicates the size of the subset of hyperparameters being tuned. The zm is an abbreviation for zero-mean. The y-axis is the average normalized score across the environment set. Each dot indicates the normalized score obtained by tuning the most performant subset of hyperparameters of each size. The dashed line indicates the point at which the curve reaches 95% of peak performance.

For the same algorithmic variants of PPO studied above, Figure 3 displays normalized scores as a function of the number of hyperparameters tuned per environment (choosing the most performant subset to tune). The curve interpolates between the normalized scores. The vertical dashed line indicates the point on the curve that reaches 95% of the per-environment tuned score. In almost all cases, modifying PPO with normalization variants moves the point right, indicating these variants improve performance at the cost of increasing pressure on the number of hyperparameters necessary to tune. Note how the performance ranking shifts based on the number of hyperparameters tuned, such as with PPO and the observation symlog variant (bottom left plot). For some variants, such as PPO, performance flattens after tuning only two hyperparameters. Whereas for other variants, such as when the value function target is transformed by the symlog function, performance is almost linear in the number of hyperparameters tuned, suggesting sensitivity to all hyperparameters. On the performance-sensitivity plane in Figure ??, the value target symlog variant and advantage per-minibatch zero-mean normalization variants overlap. Yet, in Figure 3, we can see that the advantage per-minibatch zero-mean normalization variant can obtain higher levels of performance than the value target symlog variant when tuning on smaller subsets of hyperparameters. This observation that algorithms can have similar hyperparameter sensitivities and vastly different effective hyperparameter dimensionalities indicates the power of using both metrics for studying algorithms.

6 Conclusion

A next step is to apply the proposed sensitivity and dimensionality metrics to a larger set of algorithms and environments. Meta-learning methods have been proposed to tune hyperparameters online. But, these methods come with their own sets of hyperparameters that must be dealt with. A study comparing the sensitivities and dimensionalities of meta-learning methods to the sensitivities and dimensionalities of the base learning algorithms that they optimize would be prudent. As learning systems become more complicated, careful empirical practice is critical. Modern DRL algorithms contain an abundance of hyperparameters whose interactions and sensitivities are not well understood. Common practice, which is focused on achieving SOTA performance, risks overfitting to benchmark tasks and overly relying on HPO tuning processes. Most empirical work in DRL has focused only on evaluating algorithms based on benchmark performance, leaving the effects of hyperparameters under-studied. In this work, we propose a new evaluation methodology based on two metrics that allow practitioners to better understand how an algorithm's performance relates to its hyperparameters. We show how this methodology is useful in evaluating methods purported to mitigate sensitivity. We identify that the studied observation normalization methods, while improving performance, also increase hyperparameter sensitivity and that several normalization variants increase the number of sensitive hyperparameters.

References

Marcin Andrychowicz, Anton Raichuk, Piotr Stanczyk, Manu Orsini, Sertan Girgin, Raphaël Marinier, Léonard Hussenot, Matthieu Geist, Olivier Pietquin, Marcin Michalski, Sylvain Gelly, and Olivier Bachem. What Matters in On-Policy Reinforcement Learning? A Large-Scale Empirical Study. *CoRR*, abs/2006.05990, 2020.

Theresa Eimer, Carolin Benjamins, and Marius Lindauer. Hyperparameters in Contextual RL are Highly Situational. *CoRR*, abs/2212.10876, 2022.

Theresa Eimer, Marius Lindauer, and Roberta Raileanu. Hyperparameters in Reinforcement Learning and How To Tune Them. In *International Conference on Machine Learning (ICML)*, 2023.

Jörg KH Franke, Gregor Köhler, André Biedenkapp, and Frank Hutter. Sample-Efficient Automated Deep Reinforcement Learning. In *International Conference on Learning Representations (ICLR)*, 2021.

C. Daniel Freeman, Erik Frey, Anton Raichuk, Sertan Girgin, Igor Mordatch, and Olivier Bachem. Brax - A Differentiable Physics Engine for Large Scale Rigid Body Simulation. In *Neural Information Processing Systems Track on Datasets and Benchmarks (NeurIPS)*, 2021.

Danijar Hafner, Jurgis Pasukonis, Jimmy Ba, and Timothy P. Lillicrap. Mastering Diverse Domains through World Models. *CoRR*, abs/2301.04104, 2023.

Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Gheshlaghi Azar, and David Silver. Rainbow: Combining Improvements in Deep Reinforcement Learning. In *Conference on Artificial Intelligence (AAAI)*, 2018.

Shengyi Huang, Rousslan Fernand Julien Dossa, Antonin; Raffin, Anssi, Anssi Kanervisto, and Weixun Wang. The 37 Implementation Details of Proximal Policy Optimization. In *International Conference on Learning Representations Blog Track (ICLR)*, 2022.

Scott M. Jordan, Yash Chandak, Daniel Cohen, Mengxue Zhang, and Philip S. Thomas. Evaluating the Performance of Reinforcement Learning Algorithms. In *International Conference on Machine Learning (ICML)*, 2020.

Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. In *International Conference on Learning Representations (ICLR)*, 2015.

Chris Lu, Jakub Kuba, Alistair Letcher, Luke Metz, Christian Schroeder de Witt, and Jakob Foerster. Discovered Policy Optimisation. In *Neural Information Processing Systems (NeurIPS)*, 2022.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin A. Riedmiller, Andreas 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*, 518:529–533, 2015.

Andrew Patterson, Samuel Neumann, Martha White, and Adam White. Empirical Design in Reinforcement Learning. *CoRR*, abs/2304.01315, 2023.

Andrew Patterson, Samuel Neumann, Raksha Kumaraswamy, Martha White, and Adam M White. The Cross-Environment Hyperparameter Setting Benchmark for Reinforcement Learning. In *Reinforcement Learning Conference (RLC)*, 2024.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms. *CoRR*, abs/1707.06347, 2017.

Ryan Sullivan, Akarsh Kumar, Shengyi Huang, John P. Dickerson, and Joseph Suarez. Reward Scale Robustness for Proximal Policy Optimization via DreamerV3 Tricks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

Richard S Sutton and Andrew G Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2018.

Hado van Hasselt, Arthur Guez, Matteo Hessel, Volodymyr Mnih, and David Silver. Learning Values Across Many Orders of Magnitude. In *Neural Information Processing Systems (NeurIPS)*, 2016.

A Proliferation of Hyperparameters

Algorithm name	Year	Number of hyperparameters	Comments
DQN	2013	16	Hyperparameter table from paper.
PER	2016	20	(+16) inherited from DQN (+4) added for PER.
Rainbow	2017	25	(+20) inherited from PER (+5) (1 extra from changing optimizer from RMSProp to ADAM, n step return, 3 from distributional RL).
Agent57	2020	34	Hyperparameter table from paper.
A3C	2016	11	(+3) Uses RMSProp (+1) value function loss coefficient (+1) entropy coefficient (+1) number of actors (+1) batch size (+1) gradient clipping (+1) atari frame stacking (+1) target network update rate (+1) discount factor .
PPO-clip	2017	24	Hyperparameter table from paper. Additional hyperparameters discussed in ICLR blogpost. (+11) inherited from A3C (+1) from switching to ADAM (+1) λ returns (+1) number of epochs (+1) actor loss clipping (+1) value loss clipping (+2) Log stdev of action distribution LinearAnneal(-0.7,-1.6) (+1) ADAM LR annealing (+1) reward scaling (+2) reward clip (+2) post-normalization observation clip.
SAC	2018	22	Stable Baselines documentation. (+10) inherited from A3C (+1) ADAM (+1) number of epochs (+1) reward scaling (+2) reward clip + (+2) post-normalization observation clip + +1) entropy coefficient step-size (+1) target entropy (+2) action Gaussian noise (+1) polyak update.
DreamerV1	2020	26	Paper. (+1) number random seed episodes (+1) number of update steps (+1) buffer size (+1) batch size (+1) batch length (+1) imagination horizon (+1) λ (+1) actor step-size (+1) critic step-size (+3) ADAM (+1) RSSM units (+1) discount factor (+1) entropy coefficient (+1) target network update (+1) action model tanh scale factor (+1) step-size for world model (+1) gradient clipping (+1) KL regularizer clipping (+2) action Gaussian noise (+1) action repeat (+3) discrete action ϵ-greedy linear decay over first 200k steps.
DreamerV2	2022	28	Modification summary. (+1) number of discrete latents (+1) KL balancing (+1) actor gradient mixing (+1) weight decay (-2) removed action Gaussian noise.
DreamerV3	2023	42	Hyperparameter table from paper. Modification summary. (+1) World model loss clipping (+1) actor unimix random exploration (+4) advantage percentile scaling (2 percentiles, decay rate, lower bound) (+2) two additional β terms in world model loss (+1) Critic EMA decay (+1) Critic EMA regularizer coefficient (+2) Adaptive Gradient Clipping (+1) Critic replay loss scale (+1) Critic loss scale (+1) Actor loss scale (+1) latent unimix (-2) same step-size for all.

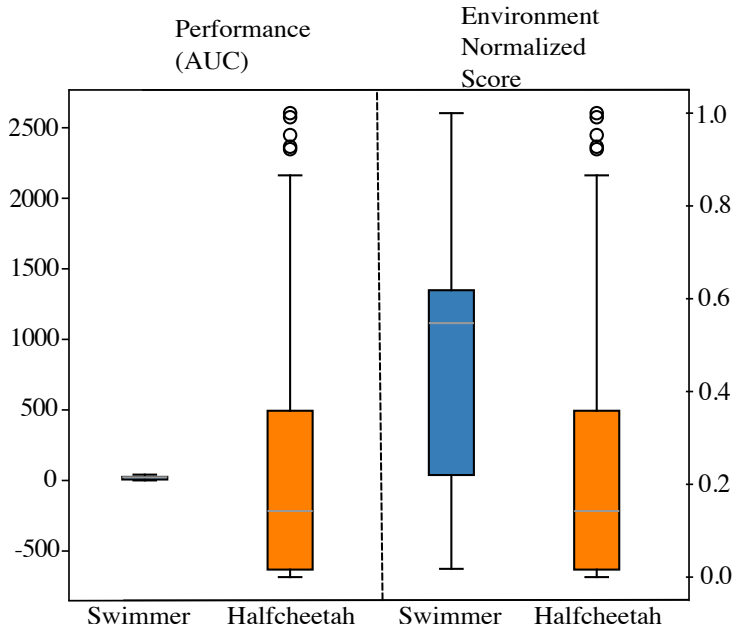

Figure 4: **Left**: The distributions of performance (AUC) over 625 hyperparameter settings for the PPO algorithm in Swimmer and Halfcheetah Brax environments (Freeman et al., 2021). **Right**: The same distributions after applying score normalization. Each data point is the mean AUC observed across 200 runs. Each run consisted of 3M steps of the agent-environment interaction.

B Score normalization boxplots

The boxplots shown in Figure 4 illustrate the need for score normalization to allow for cross-environment performance metrics. Figure 5 shows the distributions after score normalization and indicates the sensitivity gap that exists from tuning PPO for Halfcheetah.

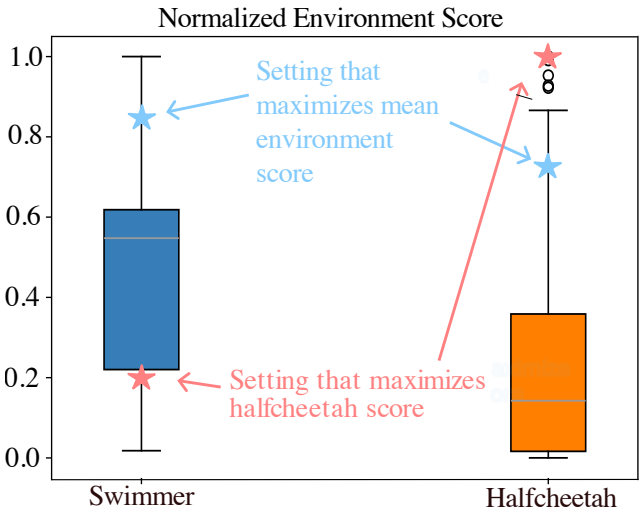

Figure 5: The distributions of environment normalized scores for 625 hyperparameter settings of the PPO algorithm in the Swimmer and Halfcheetah environments. The blue stars indicate the normalized score of the hyperparameter setting, which maximizes the mean normalized score across both environments. The red stars indicate the environment-normalized scores of the hyperparameter setting, which maximizes the Halfcheetah normalized score.

C Proximal Policy Optimization

PPO is an instance of an actor-critic method (Sutton & Barto, 2018) that maintains two neural networks: a policy network with parameters $\boldsymbol{\theta}$ and a value network with parameters $\mathbf{w}$. Given a policy $\pi_{\boldsymbol{\theta}_{old}}$, the agent performs a roll-out of T steps, $(S_1, A_1, R_1, S_2, A_2, R_2, ...S_T, A_T, R_T)$, this rollout is then split into m batches of length k. The critic is then updated via ADAM (Kingma & Ba, 2015) by MSE loss with the truncated-λ return as a target. The actor is updated via ADAM (Kingma & Ba, 2015) to optimize a clipped surrogate objective toward maximizing the expected return. An entropy regularizer is added to the actor loss to encourage exploration. We focus our attention on four key hyperparameters of PPO: the step-size for the critic $\alpha_{\mathbf{w}}$, the step-size for the actor $\alpha_{\boldsymbol{\theta}}$, the coefficient of the entropy regularizer τ, and the truncated-λ return mixing parameter λ.

C.1 Normalization variants

Several normalization variants have been used in PPO implementations. We focus on three categories of normalization: observation normalization, value function normalization, and advantage normalization. An intuition behind value function or advantage normalization for hyperparameter sensitivity is that the scale and sparsity of rewards vary greatly across environments and that value function or advantage normalization should make actor-critic updates invariant to these factors, (van Hasselt et al., 2016) leading to less need for tuning the step-size hyperparameters. Another claimed benefit of advantage normalization variants is that by normalizing the advantage term, it is easier to find an appropriate value for the entropy regularizer coefficient τ across the set of environments (Hafner et al., 2023). Observation normalization standardizes the network inputs. This can mitigate large gradients, which may stabilize the learning system for hyperparameter tuning (Hafner et al., 2023; Andrychowicz et al., 2020), especially step-sizes $\alpha_{\mathbf{w}}$ and $\alpha_{\boldsymbol{\theta}}$.

C.1.1 Advantage normalization

Per-minibatch zero-mean normalization: Per-minibatch zero-mean advantage normalization was identified as one of 37 implementation details that matter for PPO (Huang et al., 2022). When performing an update, the advantage estimates $\hat{A}_t$ used in the actor loss function are normalized by subtracting the mean $\mu_{\hat{A}_t}$ advantage estimates from trajectories in the sampled batch and dividing by the standard deviation $\sigma_{\hat{A}_t}$ of advantage estimates in the sampled batch $(\hat{A}_t - \mu_{\hat{A}_t})/\sigma_{\hat{A}_t}$.

Percentile scaling: Another form of advantage normalization was introduced in DreamerV3 (Hafner et al., 2023) ablations, which divides the approximate advantage term in the actor loss by a scaling factor. Exponential moving averages are maintained over the 95th percentile of advantage estimates $\text{perc}(\hat{A}_t, 95)$ and 5th percentile $\text{perc}(\hat{A}_t, 5)$ within sampled batches. In the actor loss, the advantage term is divided by the difference of the two percentile moving averages $\hat{A}_t / (\text{perc}(\hat{A}_t, 95) - \text{perc}(\hat{A}_t, 5))$.

Lower bounded percentile scaling: An alternate variant of percentile scaling is used in the DreamerV3 algorithm (Hafner et al., 2023). Lower bounded percentile scaling applies a max operation to the percentile scaling factor, preventing the estimated advantage term from blowing up if the percentile difference is small $\hat{A}_t / \max(1, \text{perc}(\hat{A}_t, 95) - \text{perc}(\hat{A}_t, 5))$.

C.1.2 Value target normalization

Symlog: DreamerV3 (Hafner et al., 2023) introduced a method of scaling down the magnitudes of target values by the symlog function. The symlog function and its inverse symexp are defined as:

$$\text{symlog}(x) \doteq \text{sign}(x) \ln(|x| + 1) \tag{C.4}$$

$$\text{symexp}(x) \doteq \text{sign}(x)(\exp(|x| - 1) \tag{C.5}$$

As in DreamerV3 (Hafner et al., 2023), Symlog is applied to the target in the critic mean squared error (MSE) loss, $\text{symlog}(G^{\lambda}_{t:t+n})$. And, symexp is applied to the output of the critic network. In a subsequent study that applied DreamerV3 tricks to PPO (Sullivan et al., 2023), it was reported that the symlog transformation of the value target was one of the most impactful tricks in environments without reward clipping when applied to PPO.

C.1.3 Observation normalization

Zero-mean normalization: A commonly used trick in PPO is to normalize observations by maintaining a running estimate of the mean μ_{o_t} and standard deviation σ_{o_t} of the observations $(o_t - \mu_{o_t})/\sigma_{o_t}$.

Symlog: An alternative form of observation normalization used by DreamerV3 is to compress observations by applying the symlog function, $\text{symlog}(o_t)$.

Previous work has reported the performance impact of the normalization variants commonly used with PPO: per-minibatch zero-mean advantage normalization and zero-mean observation normalization (Andrychowicz et al., 2020). Other work (Sullivan et al., 2023) has investigated how PPO performance is affected by the normalization variants introduced in DreamerV3 (lower bounded percentile scaling, value target symlog, and symlog observation); notably, this work did not perform any hyperparameter tuning for the modified algorithms. In our results, hyperparameter tuning demonstrated a significant effect on the relative performance of these algorithms.

Given these normalization variants, a natural question that arises is how do they affect the hyperparameter sensitivity of PPO? To the best of our knowledge, a careful study of the effect these normalization variants have on the hyperparameter sensitivity of PPO has yet to be done.

D Learning Curves

Figure 6 displays learning curves for each algorithm and environment in the study. For each algorithm, the hyperparameter setting was chosen to maximize the cross-environment tuned score. The hyperparameter settings chosen for each algorithm enable learning to occur in all environments. It is important to verify that learning occurs. If an algorithm does not learn in an environment, then the score normalization procedure will pick maximum and minimum values based on noise, which will lead to baseless sensitivity and dimensionality metrics.

E Broader Impact Statement

This work investigates an empirical methodology for evaluating the hyperparameter sensitivity of reinforcement learning agents. The immediate societal impact is minimal. However, our methodology may aid in the development of performative algorithms with low hyperparameter sensitivity. If this occurs, these algorithms will result in less need for hyperparameter tuning and, as a result, have a positive impact on lowering the carbon footprint of deep reinforcement learning experiments.

F Hyperparameter Sweep Details

The PPO implementation was heavily inspired by the PureJaxRL PPO implementation (Lu et al., 2022). The variants advantage per-minibatch zero-mean normalization and observation zero-mean normalization closely follow the implementation details of the original PPO repository (Schulman et al., 2017) with reference to the details described in (Huang et al., 2022). The variants: symlog observation, symlog value target, percentile scaling, and lower bounded percentile scaling closely follow the implementation of the DreamerV3 tricks applied to PPO shown in Sullivan et al. (2023) as well as referencing the original DreamerV3 repository Hafner et al. (2023).

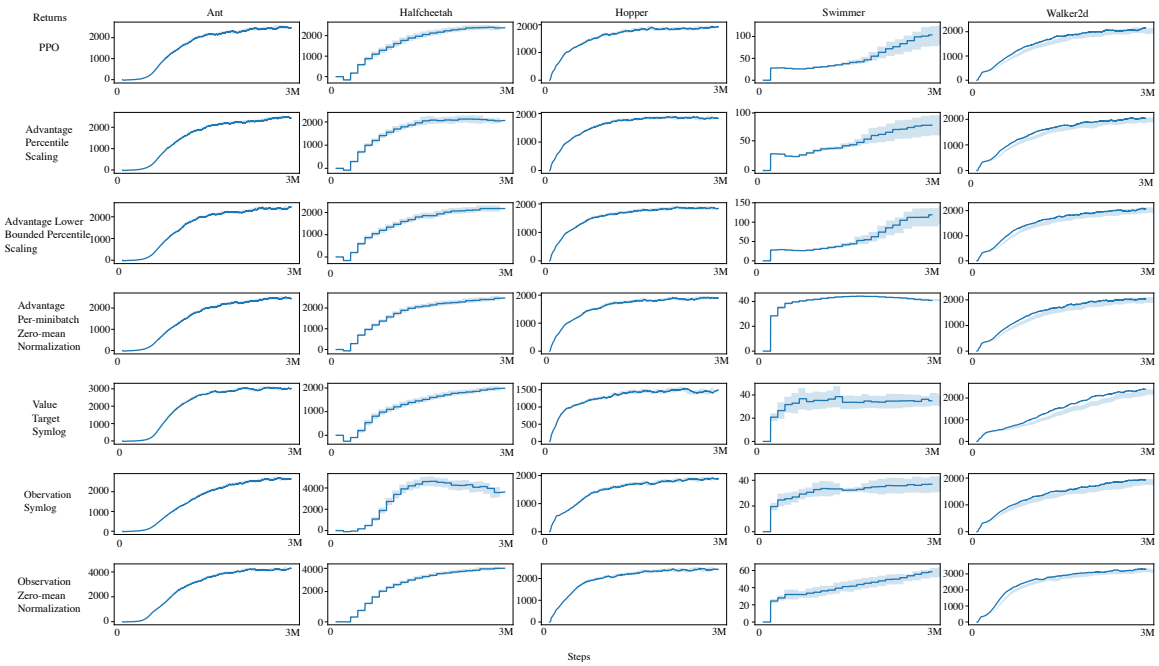

Figure 6: Learning curves for each algorithm and environment tested with the hyperparameter setting that maximized the cross-environment tuned score. The y-axis is the return. The x-axis is the environment steps. The shaded region is a 95% Student t-distribution confidence interval around the mean return over 200 runs.

The policy and critic networks were parametrized by fully connected MLP networks, each with two hidden layers of 256 units. The network used the tanh activation function. Separate ADAM optimizers (Kingma & Ba, 2015) were used for training the actor and critic networks. The environments used in the experiments were the Brax implementations of Ant, Halfcheetah, Hopper, Swimmer, and Walker2d.(Freeman et al., 2021). The hyperparameter sweeps were a grid search over eligibility trace $\lambda \in \{0.1, 0.3, 0.5, 0.7, 0.9\}$, entropy regularizer coefficient $\tau \in \{0.001, 0.01, 0.1, 1.0, 10.0\}$, the actor step-size $\alpha_{\boldsymbol{\theta}} \in \{0.00001, 0.0001, 0.001, 0.01, 0.1\}$, and the critic step-size $\alpha_{\mathbf{w}} \in \{0.00001, 0.0001, 0.001, 0.01, 0.1\}$. Each run lasted for 3M environment steps. 200 runs were performed for each of the algorithms, environments, and hyperparameter settings. There were 7 variants tested across 5 environments for each of the 625 different hyperparameter settings (4 hyperparameters. 5 values in each dimension of grid search. $5^4 = 625$) for a total of 200 runs (seeds) each. This resulted in a total number of 4375000 runs. Each run lasted for 3 million environment steps. As was done in PureJaxRL, the entire training loop was implemented to run on GPU. The experiment ran for approximately 7 GPU years on NVIDIA 32GB V100s.

`\usepackage[accepted]{rlc}`.