ARLBench: Flexible and Efficient Benchmarking for Hyperparameter Optimization in Reinforcement Learning

Jannis Becktepe*[1]
becktepe@stud.uni-hannover.de

Julian Dierkes*[2]
dierkes@aim.rwth-aachen.de

Carolin Benjamins[1] **Aditya Mohan**[1] **David Salinas**[3] **Raghu Rajan**[3]

Frank Hutter[4,3] **Holger H. Hoos**[2] **Marius Lindauer**[1,5] **Theresa Eimer**[1]

[1] Leibniz University Hannover, [2] RWTH Aachen University, [3] University of Freiburg,
[4] ELLIS Institute Tübingen, [5] L3S Research Center
*Both authors contributed equally to this work

Abstract

Hyperparameters are a critical factor in reliably training well-performing reinforcement learning (RL) agents. Unfortunately, developing and evaluating automated approaches for tuning such hyperparameters is both costly and time-consuming. As a result, such approaches are often only evaluated on a single domain or algorithm, making comparisons difficult and limiting insights into their generalizability. We propose ARLBench, a benchmark for hyperparameter optimization (HPO) in RL that allows comparisons of diverse HPO approaches while being highly efficient in evaluation. To enable research into HPO in RL, even in settings with low compute resources, we select a representative subset of HPO tasks spanning a variety of algorithm and environment combinations. This selection allows for generating a performance profile of an automated RL (AutoRL) method using only a fraction of the compute previously necessary, enabling a broader range of researchers to work on HPO in RL. With the extensive and large-scale dataset on hyperparameter landscapes that our selection is based on, ARLBench is an efficient, flexible, and future-oriented foundation for research on AutoRL. Both the benchmark and the dataset are available at `https://github.com/automl/arlbench`.

1 Introduction

Deep Reinforcement Learning (RL) algorithms require careful configuration of many different design decisions and hyperparameters to reliably work in practice [Farsang and Szegletes, 2021, Pislar et al., 2022], such as learning rates [Gulde et al., 2020] or batch sizes [Obando-Ceron et al., 2023]. Automated reinforcement learning (AutoRL) [Parker-Holder et al., 2022], a sub-field of automated machine learning (AutoML), makes these design decisions in a data-driven manner. In fact, recent work has shown that such a data-driven approach offers the best way of navigating hyperparameters in RL [Zhang et al., 2021, Eimer et al., 2023], due to the complex and changing hyperparameter optimization landscapes encountered [Mohan et al., 2023].

Research on hyperparameter optimization (HPO) for RL has been gaining traction in recent years [Jaderberg et al., 2017, Parker-Holder et al., 2020, Franke et al., 2021, Wan et al., 2022].

17th European Workshop on Reinforcement Learning (EWRL 2024).

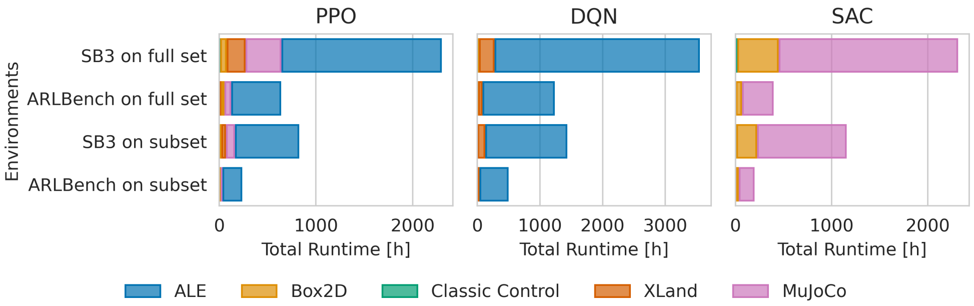


Figure 1: Running time comparison for an HPO method of 32 RL runs using 10 seeds each on the full environment set and our subsets between ARLBench and StableBaselines3 (SB3) [Raffin et al., 2021]. This results in speedup factors due to JAX of 3.59 for PPO, 2.87 for DQN, and 5.78 for SAC of ARLBench, compared to SB3 on the full set. The subset selection further decreases the running time by a factor of 2.67 for PPO, 2.49 for DQN, and 2.0 for SAC. Comparing ARLBench on the subset to SB3 on the full set, the total speedups are 9.6 for PPO, 7.14 for DQN, and 11.61 for SAC. Running time comparisons for each environment category can be found in Appendix E. Note the bars for some domains, especially on ARLBench, may be very small due to low running time.

While such approaches promise to streamline the application of RL by providing users with well-performing hyperparameter configurations for their RL tasks, it is hard to discern their actual quality; each HPO method is usually evaluated on a limited number of environments, combined with a different HPO configuration space (see, e.g., the differences between Parker-Holder et al. [2020] and Shala et al. [2024]). This inability to compare HPO approaches and AutoRL approaches more broadly leads to a lack of clarity and, ultimately, a lack of adoption of an approach that shows great promise in making RL overall more efficient and easier to apply.

One reason for the inconsistent evaluations in the current HPO literature is the wealth of RL algorithms and environments, each with its own challenges. While some environments require the processing of image observations [Bellemare et al., 2013, Cobbe et al., 2020], others focus more on finding the optimal solutions in settings with sparse reward signals [Nikulin et al., 2023]. It is fundamentally unclear which environment and algorithm combinations should be considered representative tasks for the current scope of RL research and thus useful as evaluation settings for AutoRL approaches.

We focus on the following question: *Which environments should we evaluate a given RL algorithm on to obtain a reliable performance estimate of an AutoRL method?* To answer it, we first implement highly efficient and configurable versions of three popular RL algorithms: DQN [Mnih et al., 2015], PPO [Schulman et al., 2017], and SAC [Haarnoja et al., 2018]. We subsequently conduct a large-scale study across different environment domains (ALE games [Bellemare et al., 2017], Classic Control and Box2D simulations [Brockman et al., 2016, Towers et al., 2023], Brax robot walkers [Freeman et al., 2021], and grid-based exploration [Nikulin et al., 2023]) to generate hyperparameter landscapes for these algorithms. This study, which we publish as a meta-dataset, allows us to assess the performance of given hyperparameter configurations for each algorithm and environment.

Based on the scores, we follow the method proposed by Aitchison et al. [2023] to find the subset of environments with the highest capability for predicting the average performance across all environments in order to model the RL task space. This subset thus matches the tasks the RL community cares about better than previous work on HPO for RL, while reducing computational demands for evaluation. This provides the research community with an empirically sound benchmark for HPO, which we dub **ARLBench**. It is highly efficient, taking only 937 GPU hours to evaluate an HPO budget of 32 full RL trainings for 10 runs each on all three algorithm subsets. StableBaselines3 [Raffin et al., 2021] (SB3) on the full set of environments would take 8 163 GPU hours, resulting in average speedup factors of 9.6 for PPO, 7.14 for DQN, and 11.61 for SAC as shown in Figure 1.

ARLBench is designed with current AutoRL and AutoML methods in mind; partial execution as used in many contemporary HPO methods [Li et al., 2017, Awad et al., 2021, Lindauer et al., 2022] is built into the benchmark structure just like dynamic optimization in arbitrary intervals, as, e.g., in population-based training [Jaderberg et al., 2017] variations. Moreover, various training data from ARLBench, including performance measures such as evaluation rewards and gradient history, can be used in adaptive HPO methods. ARLBench additionally supports large configuration spaces, making most low-level design decisions and architectures configurable for each algorithm. This flexibility and running time efficiency spawns a range of new insights into approaches to AutoRL.

In short, our key contributions are: (i) A highly efficient benchmark for HPO in RL, which natively supports diverse categories of HPO approaches; (ii) an environment subset selection for standardized comparisons that covers the RL task space, both (i) and (ii) together improving computational feasibility by an order of magnitude; (iii) a set of performance data on our benchmark with over 100 000 total runs spanning various RL algorithms, environments, seeds, and configurations (equivalent to 32 588 GPU hours).

2 Related Work: Benchmarking HPO for RL

Several works study the impact that such hyperparameter settings have on RL algorithms [Henderson et al., 2018, Andrychowicz et al., 2021, Obando-Ceron and Castro, 2021, Obando-Ceron et al., 2023] and show that they mostly do not transfer across environments [Ceron et al., 2024]. Automated configuration of these algorithms, on the other hand, is not as common, especially compared to the body of work in AutoML [Hutter et al., 2019]. Previous work has shown, however, that HPO approaches can find high-performing hyperparameter configurations quite efficiently [Xu et al., 2018, Parker-Holder et al., 2020, Zhang et al., 2021, Franke et al., 2021, Flennerhag et al., 2022, Wan et al., 2022]. These approaches range from standard HPO, including multi-fidelity optimization [Falkner et al., 2018, Awad et al., 2021], and algorithm configuration tools from AutoML [Schede et al., 2022, Dierkes et al., 2024] to novel strategies aiming to adapt to the dynamic nature of RL algorithms. Most popular is the population-based training (PBT) line of work [Jaderberg et al., 2017, Wan et al., 2022], which evolves hyperparameter schedules via a population of agents, resulting in a dynamic configuration strategy. For adaptive dynamic HPO, second-order optimization can be used to learn hyperparameter schedules online [Xu et al., 2018, Flennerhag et al., 2022]. Most of these, however, are not directly comparable due to different algorithms, environments, and configuration spaces in their experiments, making it difficult to find clear state-of-the-art and thus promising directions for future work [Eimer et al., 2023].

Besides this lack of comparisons in HPO for RL, the cost of training and evaluation is a significant factor hindering progress in the field. Tabular benchmarks [Ying et al., 2019, Klein and Hutter, 2019] offer a low-cost option when benchmarking HPO. Such benchmarks are essentially databases, from which the results of running a given algorithm are looked up rather than performing actual runs. Currently, the only benchmark library for HPO in RL is a tabular benchmark: HPO-RL-Bench [Shala et al., 2024]. It contains results for five RL algorithms on 22 different environments with three random seeds each. HPO-RL-Bench offers significantly reduced configuration spaces of only up to three hyperparameters, narrowed down from typically larger spaces of 10 to 13 hyperparameters, e.g., in ARLBench and SB3 [Raffin et al., 2021], and is based solely on a pre-computed dataset. Its dynamic option is further reduced to only two hyperparameters, each with three possible values at two switching points. We believe, therefore, that HPO-RL-Bench and ARLBench will fulfill different roles: HPO-RL-Bench can provide zero-cost evaluations of expensive domains, while for ARLBench, we prioritized flexibility in what and when to configure while still allowing fast evaluations.

Benchmarks are essential in the broader AutoML domain; Benchmarks such as HPOBench [Eggensperger et al., 2021], HPO-B [Pineda et al., 2021] and YAHPO-Gym [Pfisterer et al., 2022] have been contributing to research progress in HPO. In contrast, ARLBench focuses exclusively on RL, a domain that has only been included with a single toy scenario in HPOBench so far. Given that Mohan et al. [2023] have shown that the RL HPO landscapes do not seem as benign as Pushak and Hoos [2018] describe the HPO landscapes for supervised learning overall (see Appendix H.1), it is necessary to offer a dedicated RL benchmark with a diverse task set. The NAS-Bench benchmarks for neural architecture search (NAS) are examples of benchmarking supporting efficient research: NAS-Bench-101 [Ying et al., 2019] is a tabular benchmark, which NAS-Bench-201 [Dong and Yang, 2020] extends to a larger configuration space, and NAS-Bench-301 [Zela et al., 2022] uses this data to propose surrogate models [Eggensperger et al., 2014, Klein et al., 2019] that can predict performance even for unseen architectures. Building on these, several dozens of specialized NAS benchmarks have been developed [Mehta et al., 2022]. We expect that benchmarking HPO in RL will similarly become a focal point within the community towards advancing the configuration of RL algorithms.

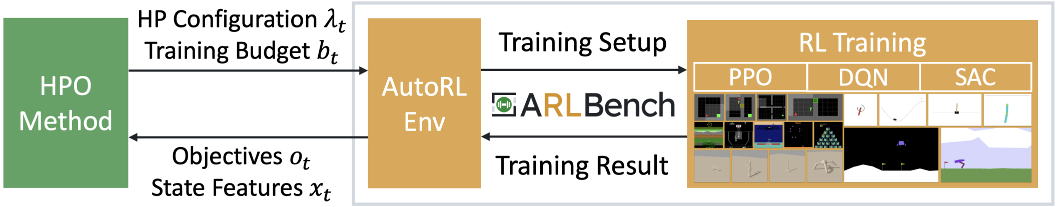


Figure 2: Overview of the ARLBench framework. The AutoRL environment, providing a *Gymnasium*-like interface [Towers et al., 2023], is the interaction point for HPO methods. At optimization step t, the optimizer selects a hyperparameter configuration λ_t and a training budget (number of steps) b_t. Then, the RL algorithm is trained using the given configuration and budget. As a result, the AutoRL environment returns the training result in the form of optimization objectives o_t, e.g., the evaluation return and runtime, and state features x_t, e.g., gradients during training.

3 Implementing ARLBench

In this section, we discuss the implementation of the ARLBench framework. Notably, we elaborate on essential considerations for the benchmark and its two main components: the AutoRL Environment HPO interface and the RL algorithm implementations.

3.1 Benchmark Desiderata for ARLBench

Given the limitations of HPO-RL-Bench [Shala et al., 2024] compared to the kinds of methods we see in HPO for RL, our three main priorities in constructing ARLBench are (i) enabling the large configuration spaces required for RL, (ii) prioritizing fast execution times, and (iii) supporting dynamic and reactive hyperparameter schedules.

Configuration Space Size. Eimer et al. [2023] have shown that most hyperparameters contribute to the training success of RL algorithms. Furthermore, our knowledge of how hyperparameters act on RL algorithms continues to expand, most recently, e.g., by showing the importance of batch sizes in certain RL settings [Obando-Ceron et al., 2023]. Thus, limiting the configurability of a benchmark will lead to the insights we gather outpacing the benchmarking capabilities of the community. Therefore, we enable large and flexible configuration spaces for all algorithms. To achieve this, however, we cannot simply extend the tabular HPO-RL-Bench, as the computational expense required for larger configuration spaces would grow exponentially in the number of hyperparameters. A long-term solution would be to train surrogate models to predict performance. However, as the data requirements for reliable and dynamic surrogates in RL are presently unclear, we focus on building a good online benchmark first and use it to generate preliminary landscapes. We hope this approach allows the building of better RL-specific surrogate models in future work.

Running Time. An alternative to using surrogate models is building an efficient way of evaluating hyperparameter configurations in RL. JAX [Bradbury et al., 2018] enables significant efficiency gains, leading to RL agents training on many domains in mere minutes or seconds [Lu, 2022, Toledo, 2024]. We exploit this while providing RL algorithms that are easy to configure for commonly used HPO methods, including multi-objective and multi-fidelity optimization.

Dynamic Configuration. Finally, we aim to enable dynamic configurations that allow hyperparameter settings to be adjusted during a single RL training session, recognizing that the optimal hyperparameters can evolve as training progresses [Mohan et al., 2023]. One way of doing this is by providing checkpoint capabilities that support the seamless continuation of RL training. Most population-based methods, for example, find schedules with 10–20 hyperparameter changes during a single training run [Jaderberg et al., 2017, Parker-Holder et al., 2022, Wan et al., 2022], while other methods, such as hyperparameter adaptation via meta-gradients, can configure much more often and even require information about the current algorithm state.

3.2 The HPO Interface: The AutoRL Environment

As shown in Figure 2, the AutoRL Environment is the main building block of ARLBench and connects all the critical parts for HPO in RL. It provides a powerful, flexible, and dynamic interface to support various HPO methods in an interface that, for ease of use, functions similarly to *Gymnasium* [Towers et al., 2023]. During the optimization, the HPO method selects a hyperparameter configuration λ_t and training budget b_t for the current optimization step t. Given these, the AutoRL Environment sets up the algorithm and RL environment and performs the actual RL training. In addition to an evaluation reward, data such as gradients and losses are collected during training. Depending on the user's preferences, the AutoRL Environment then extracts optimization objectives, such as the average evaluation reward, training running time, or carbon emissions [Courty et al., 2024], as well as optional information on the RL algorithm's internal state, e.g., the variance of the gradients.

The AutoRL Environment supports static and dynamic HPO methods. While static methods start the inner RL training from scratch for each configuration, dynamic approaches can keep the training state, which includes the neural network parameters, optimizer state, and replay buffer. To support the latter, we integrate an easy-to-use yet powerful checkpointing mechanism. This enables HPO methods to restore, duplicate, or checkpoint the training state at any point during the dynamic optimization.

3.3 RL Training

To address the computational efficiency of RL algorithms, we implement the entire training pipeline using JAX [Bradbury et al., 2018]. We re-implement DQN [Mnih et al., 2015], PPO [Schulman et al., 2017], and SAC [Haarnoja et al., 2018] in order to make them highly configurable, enable dynamic execution, and ensure compatibility with different target environments. Wherever we use code from external sources (Freeman et al. [2021], Lu [2022], Toledo et al. [2023]; licensed under Apache-2.0), it is referenced in the code. We compare our implementation to SB3 [Raffin et al., 2021] in Appendix E and find very similar learning curves for the speedups in Figure 1. We support a range of environment frameworks, particularly *Brax* [Freeman et al., 2021], *Gymnax* [Lange, 2022], *Gymnasium* [Towers et al., 2023], *Envpool* [Weng et al., 2022], and *XLand* [Nikulin et al., 2023]. This results in a broad coverage of RL domains, including robotic simulations, grid worlds, and video games, such as the ALE [Bellemare et al., 2013]. We ensure compatibility with these different environments and their APIs with our own ARLBench *Environment* class, allowing for future updates and continued support of changing interfaces in RL.

4 Finding Representative Benchmarking Settings

Highly efficient implementations are crucial for efficient benchmarking of HPO methods for RL. However, they represent just a fraction of the overall picture: prior work has focused primarily on a single-task domain, due to a lack of insight regarding which RL domains to target. To tackle this issue, we aim to find a subset of RL environments representative of the broader RL field. First, we study the hyperparameter landscapes for a large set of environments using random sampling of configurations. To ensure the feasibility of our experiments in terms of computational resources, we select a representative subset of environments from each domain. In particular, we select a total of 21 environments: five ALE games (Atari-5), three Box2D environments, four Brax walkers, five classic control environments, and four XLand environments (see Appendix D). Then, we use the Atari-5 [Aitchison et al., 2023] method to find a set of environments for testing HPO approaches in RL. Ultimately, we validate that this subset is representative of the HPO landscape of all RL tasks we consider. In total, we spent 10 105 h on CPUs and 32 588 h on GPUs (see Appendix I).

4.1 Data Collection

For each combination of algorithm and environment, we aim to estimate the hyperparameter landscape, i.e., the relationship between a certain hyperparameter configuration and its performance. Therefore, we run an RL algorithm on 256 Sobol-sampled configurations [Sobol, 1967]. With configuration spaces ranging from 10 to 13 hyperparameters, this is roughly equivalent to the search space covering initial design recommendations of Jones et al. [1998]. We run each configuration for 10 random seeds. The performance is measured by evaluating the final policy induced by the configuration on a dedicated evaluation environment with a different random seed. We collect 128 episodes and calculate

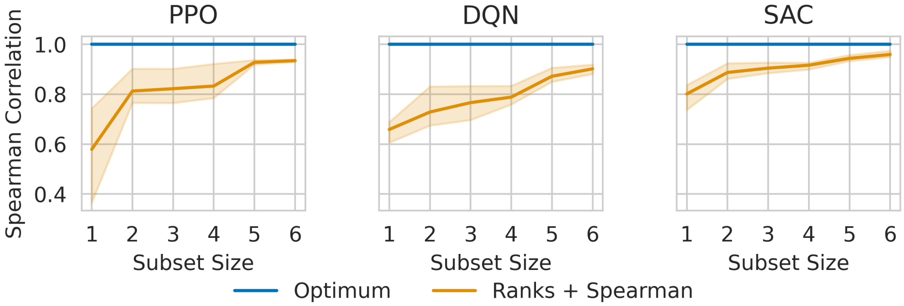


Figure 3: Comparison of the Spearman correlation for different subset sizes with confidence intervals from 5-fold cross-validation on the configurations.

the average cumulative reward. This dataset can be found on GitHub as well as on Huggingface: `https://huggingface.co/datasets/autorl-org/arlbench`.

4.2 Subset Selection

Based on the collected evaluation rewards, we aim to find a subset of environments on which to evaluate an AutoRL method. Due to discrete and continuous action spaces, the algorithms differ in the environments they are compatible with. Therefore, we perform this selection for each algorithm individually. The set of environments for PPO contains all 21 evaluated environments, while DQN is limited to discrete action spaces (13 environments), and SAC only supports continuous action spaces (8 environments). The full sets of environments per algorithm are listed in Appendix D. Additional details on the subset selection are stated in Appendix G.1.

Finding an optimal subset. For selecting an optimal subset, we use the method proposed by Aitchison et al. [2023]. Let Λ be the set of hyperparameter configurations for an algorithm and $\mathcal{E}$ the corresponding set of environments. For each evaluated hyperparameter configuration $\lambda \in \Lambda$ and environment $e \in \mathcal{E}$, we are given a performance score p_λ^e. We define $\overline{p}_\lambda^{\mathcal{E}} := \frac{1}{|\mathcal{E}|} \cdot \sum_{e \in \mathcal{E}} p_\lambda^e$ as the average score of a configuration λ across all environments. Given a subset of environments $\mathcal{I} \subset \mathcal{E}$ of size $C \in \mathbb{N}$, we use a linear regression model f to predict $\overline{p}_\lambda^{\mathcal{E}}$ from the scores p_λ^e for all $e \in \mathcal{I}$, i.e., $\hat{p}_\lambda^{\mathcal{E}} := f(p_\lambda^{e_1}, \cdots, p_\lambda^{e_C})$. An optimal subset $\mathcal{I}^*$ of size C is defined as

$$\mathcal{I}^* \in \underset{\mathcal{I}=\{e_1,\cdots,e_C\}\subset\mathcal{E}}{\arg\min}\; d(\hat{p}^{\mathcal{E}}, \overline{p}^{\mathcal{E}}) \text{ with } \hat{p}^{\mathcal{E}} = (\hat{p}_\lambda^{\mathcal{E}})_{\lambda\in\Lambda} \text{ and } \hat{p}_\lambda^{\mathcal{E}} = f(p_\lambda^{e_1}, \cdots, p_\lambda^{e_C}), \tag{1}$$

where d is a distance metric between the predicted and target hyperparameter landscapes, i.e., the vector of predicted scores $\hat{p}^{\mathcal{E}} = (\hat{p}_\lambda^{\mathcal{E}})_{\lambda\in\Lambda}$ and the vector of target scores $\overline{p}^{\mathcal{E}} = (\overline{p}_\lambda^{\mathcal{E}})_{\lambda\in\Lambda}$ spanning across the configurations $\lambda \in \Lambda$. The performance attained on the subset then provides the best approximation of the performance across all environments for subsets of size C. Note that the environment selection does not take HPO behavior into account. We could perform the subselection to approximate HPO results directly. However, the performance discrepancies we see for HPO methods in the literature [Eimer et al., 2023, Shala et al., 2024] suggest that we do not yet know how to best apply HPO methods to RL. Therefore, we currently lack reliable methodologies to obtain the necessary data allowing us to infer direct relationships between environments and performance of HPO methods.

Selection Strategy. Although reward scales vary drastically across environments, we lack the human expert scores [Aitchison et al., 2023] to normalize rewards per environment. Instead, we apply a rank-based normalization method to obtain the performances p_λ^e. For an environment e, we train policies using each hyperparameter configuration λ across 10 different random seeds and evaluate each resulting policy to obtain its mean return. The performance p_λ^e of a configuration λ is then determined by the average rank with respect to the mean return over its 10 random seeds when compared to all other configurations within the same environment e. Now, we can fit a linear model to predict the average ranks across the full set, given the ranks on the subset. We use the Spearman correlation coefficient ρ_p as a similarity metric, leading to $d(\hat{p}_\lambda^{\mathcal{E}}, \overline{p}_\lambda^{\mathcal{E}}) := 1 - \rho_p(\hat{p}_\lambda^{\mathcal{E}}, \overline{p}_\lambda^{\mathcal{E}})$ in Equation 1. Our choice of ρ_p is motivated by our interest in capturing relationships between two return distributions robustly

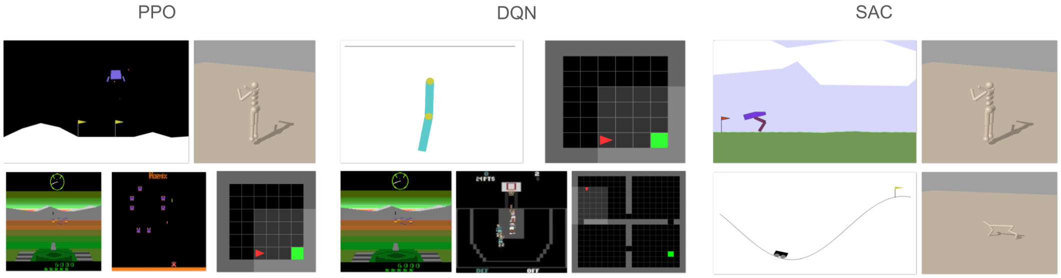


Figure 4: Selected set of representative environments per algorithm. For PPO, the discrete variant of LunarLander was selected.

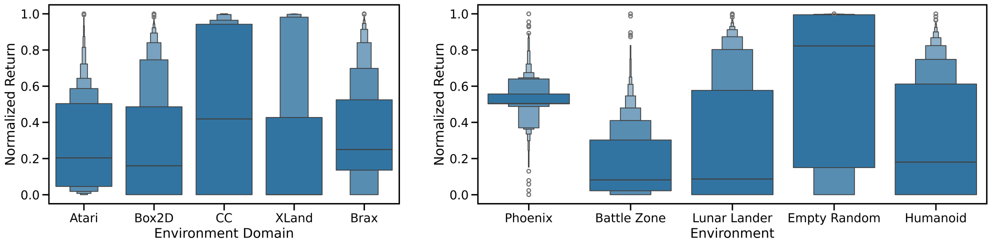


Figure 5: Comparison of the return distributions over hyperparameter configurations of PPO on all 21 environments (left) and the selected subset of 5 environments (right). For the same comparisons for DQN and SAC, see Appendix H.2.

by focusing on relative rankings rather than exact values. Figure 3 shows the Spearman correlation coefficient for the top three subsets of different sizes with confidence intervals computed using 5-fold cross-validation on the configurations. The results are fairly consistent for all algorithms except for very small subsets for PPO. Furthermore, they exhibit a high correlation to the full environment set, even when considering only a few environments. Based on these correlations, we select five environments for PPO and DQN from their respective full sets of 21 and 13 environments. The selected PPO subset shows a correlation of 0.95, while the DQN subset has a correlation of 0.92. For SAC, we select a subset of four environments, achieving a correlation of 0.94 with the full set of eight environments. Further details can be found in Figure 4 and Appendix G.3. A single training on all environments in all three subsets takes around 2.93 GPU hours, compared to 7.12 GPU hours for the full set of environments, where the ALE environments are limited to Atari-5 in the full set.

4.3 Validating ARLBench

Having selected a subset per algorithm, we still need to ensure this subset is representative of the full environment set from an HPO perspective. To investigate this, we examine (i) the HPO landscape, in particular the return distributions, budget correlations, and hyperparameter importance, and (ii) the performance of different HPO optimizers on the subset and full environment set. For most of the following analysis, we use DeepCAVE [Sass et al., 2022], as a monitoring package for HPO. We use 95% confidence intervals in our reported results as suggested by Agarwal et al. [2021].

Comparing HPO Landscapes. We first analyze the differences in HPO landscapes between the full environment set and our subset. This is especially important since we do not use HPO performance data for the selection but still want to ensure that HPO approaches will encounter the same overall landscape characteristics on the subsets as on all benchmarks. We argue that this yields the first insights into the consistency of HPO performance on the subset and full set of environments. To see if the overall RL algorithm performance changes, Figure 5 shows the distribution of returns in our random samples of PPO, normalized per domain by the performance scores seen in our pre-study. For the Box2D and Brax environments, we set fixed minimum scores of -200 and -2000, respectively, to mitigate artificially low performance caused by numerical instabilities. We see that the subset includes a diverse selection of return distributions: from a large bias of configurations towards the

	PPO	Subset PPO	DQN	Subset DQN	SAC	Subset SAC
#HPs $\geq 5\%$	2.2	1.2	2.54	2.75	1.86	1.25
#Interactions $\geq 5\%$	1.62	2.2	1.3	1.2	1.0	1.0

Table 1: Number of hyperparameters and hyperparameter interactions with over 5% importance on the full set and subset for each algorithm.

lower end of the performance spectrum (in BattleZone, LunarLander, and Humanoid), an even spread biased towards higher performance (in EmptyRandom) to a dense concentration of performances towards the middle (in Phoenix). Most environment domains show a similar trend to BattleZone, LunarLander, and EmptyRandom: there is a wide spread of configurations, with a bias towards low performances. Our subset thus captures this dominant trend as well as the tendency of XLand and Classic Control for a more even performance distribution. The Phoenix environment reflects the opposite behavior, ensuring that similar environments outside of the typical performance distribution are included in our subset. These different patterns in performance with regard to hyperparameter settings suggest that the selected subset is likely to test these variations in HPO behavior.

Additionally, for the environment domains, the performance at different points in training is similarly correlated with the final performance, see Appendix H.4 for the full analysis. This shows that a large proportion of the RL algorithms' behavior regarding their hyperparameters is preserved in the subset selection (see Appendix H.2 for full results). In our fANOVA analysis [Hutter et al., 2014] (see Appendix H.3 for full results), we verify that the number of important hyperparameters stays consistent. For most algorithm-environment combinations, only two to four hyperparameters have an importance of at least 5%, though the specific important ones differ, similar to common observations in HPO [Bergstra and Bengio, 2012]. Table 1 shows that the number of important hyperparameters and their interactions remain consistent in the subset, with the highest deviation being between 2.2 hyperparameters above 5% importance on average for PPO on the whole environment set and 1.2 on the subset. Our results, along with the observed similarities in return distributions, suggest that the main properties of the HPO landscapes are preserved in our subselection.

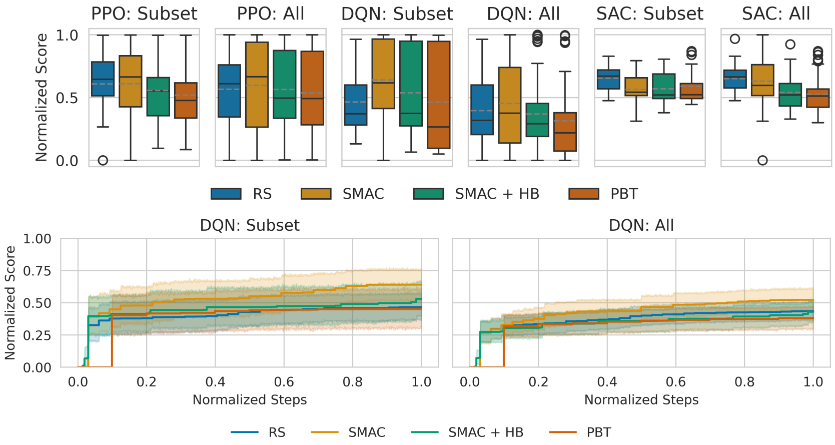


Figure 6: Comparison of HPO methods' scores on the subset and full environment set (higher is better). **Top**: Performance distributions over optimizer runs and environments. Medians and means are visualized using black and dotted gray lines, respectively. **Bottom**: HPO anytime performance with 95% confidence intervals. See Appendices G.3 for PPO and SAC , and J for details. We note that we do not consider inter-quartile means to prevent disregarding environments (top), especially since we are using only three optimizer runs (bottom).

Comparing HPO Optimizers. To further validate the subset selection, we run four HPO optimizers with a budget of 32 full training runs each for all algorithms and environments. We use five runs, i.e., random seeds for each HPO optimizer and each configuration is evaluated on three random seeds, following recommendations by Eimer et al. [2023]. We believe five seeds are a good compromise to obtain valid insights while accounting for the associated high computational demand. While statistically significant insights might require many more seeds, we believe five seeds are sufficient for obtaining preliminary insights into the compatibility of the full set of environments and our chosen subsets. To reflect the current range of HPO tools for RL, we select random search (RS) [Bergstra and Bengio, 2012], PBT [Jaderberg et al., 2017], the Bayesian optimization tool SMAC [Lindauer et al., 2022] as well as SMAC in combination with the Hyperband scheduler [Li et al., 2017] (SMAC+HB). We compare the results on the subsets and the full set of environments.

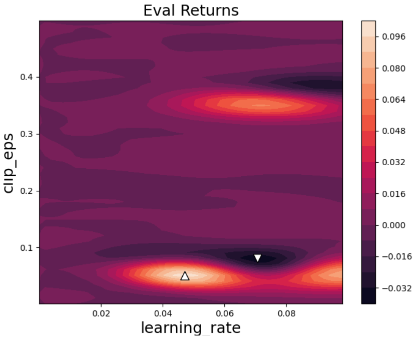


Figure 7: The hyperparameter landscape of PPO on Ant. Lighter is better, (mean performance over 10 seeds). Similar configurations perform very differently: *high returns occur next to almost failure modes*.

Figure 6 shows the HPO optimizer scores, normalized per domain by the performances seen in our pre-study, for each algorithm on the subset and all environments. The overall performance of each HPO optimizer is represented by the mean performance across all environments in the respective set of environments. We observe that the scores are distributed similarly between the full set and the subset on each algorithm for the final scores. Median and mean scores for all algorithms closely align with the respective scores of the subsets in terms of ranking.

For HPO anytime performance, the relative order remains consistent across both sets for RS, PBT, and SMAC+HB, with the only major difference that RS scores higher on the complete set earlier on; This, however, is due to merely a slight difference in scores, which is still within the confidence interval of RS. Our analysis shows that overall, the best mean HPO optimizer performance is achieved by SMAC (without HB) and RS. Although the confidence intervals overlap, the overall trends we observe in the subsets and full sets of environments stay consistent. In previous work [Eimer et al., 2023, Shala et al., 2024], multi-fidelity optimizers were shown to perform quite well, and PBT performed worst overall. Here, multi-fidelity optimization still outperforms PBT for DQN, but black-box optimization with SMAC without HB and RS overperform in comparison. This is likely due to the wider variety of environment domains we consider, some of which might not be suitable for the partial evaluations of multi-fidelity approaches (e.g. XLand). Another important factor is the inclusion of SAC where RS is especially strong. Even for PPO and DQN, however, it is striking how closely SMAC as a state-of-the-art HPO optimizer compares to RS, which typically performs worse than SMAC and other state-of-the-art HPO methods in standard supervised ML settings [Turner et al., 2021, Lindauer et al., 2022].

Looking at a partial hyperparameter landscape in Figure 7, we see a possible reason: this is far from the benign HPO landscapes Pushak and Hoos [2018] found for supervised learning. This shows that simply applying common HPO packages will not be sufficient to solve HPO for all RL tasks; a dedicated, specific effort is needed. We present further landscape plots in Appendix H.1, showing the contrast between benign and adverse landscapes we found during our experiments.

5 Limitations and Future Work

Due to the dimensions of complexity involved in this topic, including the computational expense and wealth of RL algorithms and environments, ARLBench has some limitations. First, we manually selected the underlying set of algorithms and environments from those used in the RL community at large. This gave rise to a focus on model-free learning in combination with base versions of PPO, DQN, and SAC. In the future, we will cover extensions to these algorithms, such as advanced types of replay strategies [Kapturowski et al., 2019], multi-step or exploration strategies [Amin et al., 2021, Pislar et al., 2022]. Additionally, we would like to enable ARLBench to evaluate policy generalization, ensuring that optimized policies perform well in previously unseen environments [Kirk et al., 2023, Benjamins et al., 2023, Mohan et al., 2024, Benjamins et al., 2024]. Further research on hyperparameter landscapes in RL [Mohan et al., 2023] can inform useful future additions.

Using a selected subset reduces computational costs but may increase variance due to the smaller sample size, requiring careful experimental design to ensure statistically significant results. Additionally, the selection of subsets could also consider training time, aiming for the most informative and the least costly subsets. Currently, we prioritize higher validation accuracy over reduced running time, even if one environment is slightly less important but significantly faster.

The computational cost itself remains a limitation of the benchmark. While our setting is much cheaper to evaluate and enables many more research groups to do thorough research on AutoRL, it is still by no means as cheap as surrogate or table lookups would be. Our highest priority is the flexibility of large configuration spaces and dynamic configuration, representing real-world HPO applications of RL; we do not see purely tabular benchmarks as an alternative in this exploratory phase of the field. Instead, we believe surrogate models [Eggensperger et al., 2015, 2018, Zela et al., 2022] will be crucial for more efficient HPO in RL, though modeling the dynamic nature of HPO in surrogates remains an open challenge. Our published meta-dataset, the largest one for AutoRL to date, enables the first steps towards such dynamic surrogates.

Furthermore, there are additional elements of AutoRL research our benchmark does not yet fully support. We designed it to be future-oriented, with a benchmark structure that can, in principle, support second-order optimization methods, learning based on internal algorithmic aspects, such as losses and activation functions, or architecture search. However, we believe integrating these aspects into ARLBench first requires research into how state-based HPO in RL and NAS for RL should be approached. The same holds for concepts such as discovering RL algorithms [Co-Reyes et al., 2021, Jackson et al., 2024], where no standard interface exists for evaluating a learned algorithm. Nonetheless, our environment subsets can aid in the evaluation of these approaches. Integrating AutoRL for environment components, such as environment design [Jiang et al., 2021, Parker-Holder et al., 2022], into ARLBench poses a challenge because most RL environments do not inherently support these approaches. Improving the compatibility of the environment frameworks in ARLBench will facilitate the integration of these methods into the benchmark.

6 Conclusion

We propose a benchmark for HPO in RL that supports this emerging field of research by (i) providing a general, easily integrable and extensible way of evaluating various paradigms for HPO in RL; (ii) reducing computational costs with highly efficient implementations, while expanding the evaluation coverage of HPO methods by selecting informative environment subsets, achieving over 16 times the efficiency compared to standard frameworks; (iii) publishing a large set of performance data for future use in AutoRL research. Such a concerted effort is necessary to help the community work in a common direction and democratize AutoRL as a research field. While its set of algorithms and environments will evolve within the coming years, ARLBench is built to allow for easy extension, e.g., to AutoML paradigms such as NAS, which are currently underrepresented in RL. Therefore, ARLBench will catalyze the development of increasingly efficient HPO methods for RL that perform well across algorithms and environments.

References

R. Agarwal, M. Schwarzer, P. Samuel Castro, A. C. Courville, and M. G. Bellemare. Deep reinforcement learning at the edge of the statistical precipice. In *Proc. of NeurIPS'21*, 2021.

M. Aitchison, P. Sweetser, and M. Hutter. Atari-5: Distilling the arcade learning environment down to five games. In *Proc. of ICML'23*. PMLR, 2023.

S. Amin, M. Gomrokchi, H. Satija, H. van Hoof, and D. Precup. A survey of exploration methods in reinforcement learning. *arXiv preprint arXiv:2109.00157*, 2021.

M. Andrychowicz, A. Raichuk, P. Stańczyk, M. Orsini, S. Girgin, Raphaël Marinier, L. Hussenot, M. Geist, O. Pietquin, M. Michalski, S. Gelly, and O. Bachem. What matters for on-policy deep actor-critic methods? A large-scale study. In *Proc. of ICLR'21*, 2021.

N. Awad, N. Mallik, and F. Hutter. DEHB: Evolutionary hyberband for scalable, robust and efficient hyperparameter optimization. In *Proc. of IJCAI'21*, 2021.

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

M. G. Bellemare, W. Dabney, and Rémi R. Munos. A distributional perspective on reinforcement learning. In *Proc. of ICML'17*, 2017.

C. Benjamins, T. Eimer, F. Schubert, A. Mohan, S. Döhler, A. Biedenkapp, B. Rosenhan, F. Hutter, and M. Lindauer. Contextualize me – the case for context in reinforcement learning. *Transactions on Machine Learning Research*, 2023.

C. Benjamins, G. Cenikj, A. Nikolikj, A. Mohan, T. Eftimov, and M. Lindauer. Instance selection for dynamic algorithm configuration with reinforcement learning: Improving generalization. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, 2024.

J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13:281–305, 2012.

J. Bradbury, R. Frostig, P. Hawkins, M. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL `http://github.com/google/jax`.

G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. OpenAI gym. *arxiv preprint*, arXiv:1606.01540, 2016.

J. S. O. Ceron, J. G. M. Araújo, A. Courville, and P. S. Castro. On the consistency of hyper-parameter selection in value-based deep reinforcement learning. *Reinforcement Learning Journal*, 1, 2024.

J. Co-Reyes, Y. Miao, D. Peng, E. Real, Q. Le, S. Levine, H. Lee, and A. Faust. Evolving reinforcement learning algorithms. In *Proc. of ICLR'21*, 2021.

K. Cobbe, C. Hesse, J. Hilton, and J. Schulman. Leveraging procedural generation to benchmark reinforcement learning. In *Proc. of ICML'20*, 2020.

B. Courty, V. Schmidt, S. Luccioni, Goyal-Kamal, M. Coutarel, B. Feld, J. Lecourt, L. Connell, A. Saboni, Inimaz, supatomic, M. Léval, L. Blanche, A. Cruveiller, ouminasara, F. Zhao, A. Joshi, A. Bogroff, H. de Lavoreille, N. Laskaris, E. Abati, D. Blank, Z. Wang, A. Catovic, M. Alencon, M. Stęchły, C. Bauer, Lucas-Otavio, JPW, and MinervaBooks. mlco2/codecarbon: v2.4.1, May 2024.

J. Dierkes, E. Cramer, H. Hoos, and S. Trimpe. Combining automated optimisation of hyperparameters and reward shape. *Reinforcement Learning Journal*, 1, 2024.

X. Dong and Y. Yang. NAS-Bench-201: Extending the scope of reproducible neural architecture search. In *Proc. of ICLR'20*, 2020.

K. Eggensperger, F. Hutter, H. Hoos, and K. Leyton-Brown. Surrogate benchmarks for hyperparameter optimization. In *MetaSel'14*, 2014.

K. Eggensperger, F. Hutter, H. Hoos, and K. Leyton-Brown. Efficient benchmarking of hyperparameter optimizers via surrogates. In *Proc. of AAAI'15*, 2015.

K. Eggensperger, M. Lindauer, H. Hoos, F. Hutter, and K. Leyton-Brown. Efficient benchmarking of algorithm configurators via model-based surrogates. *Machine Learning*, 107(1):15–41, 2018.

K. Eggensperger, P. Müller, N. Mallik, M. Feurer, R. Sass, A. Klein, N. Awad, M. Lindauer, and F. Hutter. HPOBench: A collection of reproducible multi-fidelity benchmark problems for HPO. In *Proc. of NeurIPS'21 Datasets and Benchmarks Track*, 2021.

T. Eimer, M. Lindauer, and R. Raileanu. Hyperparameters in reinforcement learning and how to tune them. In *Proc. of ICML'23*, 2023.

S. Falkner, A. Klein, and F. Hutter. BOHB: Robust and efficient Hyperparameter Optimization at scale. In *Proc. of ICML'18*, pages 1437–1446, 2018.

M. Farsang and L. Szegletes. Decaying clipping range in proximal policy optimization. In *15th IEEE International Symposium on Applied Computational Intelligence and Informatics, SACI 2021*. IEEE, 2021.

S. Flennerhag, Y. Schroecker, T. Zahavy, H. van Hasselt, D. Silver, and S. Singh. Bootstrapped meta-learning. In *Proc. of ICLR'22*, 2022.

J. Franke, G. Köhler, A. Biedenkapp, and F. Hutter. Sample-efficient automated deep reinforcement learning. In *Proc. of ICLR'21*, 2021.

C. Freeman, E. Frey, A. Raichuk, S. Girgin, I. Mordatch, and O. Bachem. Brax - A differentiable physics engine for large scale rigid body simulation. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021*, 2021.

R. Gulde, M. Tuscher, A. Csiszar, O. Riedel, and A. Verl. Deep reinforcement learning using cyclical learning rates. In *Third International Conference on Artificial Intelligence for Industries, AI4I*, 2020.

T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *Proc. of ICML'18*, 2018.

P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger. Deep reinforcement learning that matters. In *Proc. of AAAI'18*, 2018.

F. Hutter, H. Hoos, and K. Leyton-Brown. An efficient approach for assessing hyperparameter importance. In *Proc. of ICML'14*, 2014.

F. Hutter, L. Kotthoff, and J. Vanschoren, editors. *Automated Machine Learning: Methods, Systems, Challenges*. Springer, 2019. Available for free at `http://automl.org/book`.

M. Jackson, C. Lu, L. Kirsch, R. Lange, S. Whiteson, and J. Foerster. Discovering temporally-aware reinforcement learning algorithms. In *Proc. of ICLR'24*, 2024.

M. Jaderberg, V. Dalibard, S. Osindero, W. Czarnecki, J. Donahue, A. Razavi, O. Vinyals, T. Green, I. Dunning, K. Simonyan, C. Fernando, and K. Kavukcuoglu. Population based training of neural networks. *arXiv:1711.09846 [cs.LG]*, 2017.

M. Jiang, E. Grefenstette, and T. Rocktäschel. Prioritized level replay. In *Proc. of ICML'21*, 2021.

D. Jones, M. Schonlau, and W. Welch. Efficient global optimization of expensive black box functions. *Journal of Global Optimization*, 13:455–492, 1998.

S. Kapturowski, G. Ostrovski, J. Quan, R. Munos, and W. Dabney. Recurrent experience replay in distributed reinforcement learning. In *Proc. of ICLR'19*, 2019.

R. Kirk, A. Zhang, E. Grefenstette, and T. Rocktäschel. A survey of zero-shot generalisation in deep reinforcement learning. *Journal of Artificial Intelligence Research*, 76:201–264, 2023.

A. Klein and F. Hutter. Tabular benchmarks for joint architecture and hyperparameter optimization. *arXiv:1905.04970[cs.LG]*, 2019.

A. Klein, Z. Dai, F. Hutter, N. Lawrence, and J. Gonzalez. Meta-surrogate benchmarking for hyperparameter optimization. In *Proc. of NeurIPS'19*, 2019.

R. Lange. gymnax: A JAX-based reinforcement learning environment library, 2022. URL `http://github.com/RobertTLange/gymnax`.

L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar. Hyperband: Bandit-based configuration evaluation for hyperparameter optimization. In *Proc. of ICLR'17*, 2017.

M. Lindauer, K. Eggensperger, M. Feurer, A. Biedenkapp, D. Deng, C. Benjamins, T. Ruhkopf, R. Sass, and F. Hutter. SMAC3: A versatile bayesian optimization package for hyperparameter optimization. *Journal of Machine Learning Research*, 23(54):1–9, 2022.

C. Lu. PureJaxRL (end-to-end RL training in pure JAX), 2022. URL `https://github.com/luchris429/purejaxrl`.

Y. Mehta, C. White, A. Zela, A. Krishnakumar, G. Zabergja, S. Moradian, M. Safari, K. Yu, and F. Hutter. NAS-Bench-Suite: NAS evaluation is (now) surprisingly easy. In *Proc. of ICLR'22*, 2022.

V. Mnih, K. Kavukcuoglu, D. Silver, A. Rusu, J. Veness, M. Bellemare, A. Graves, M. Riedmiller, A. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 02 2015.

A. Mohan, C. Benjamins, K. Wienecke, A. Dockhorn, and M. Lindauer. AutoRL hyperparameter landscapes. In *Proc. of AutoML Conf'23*. PMLR, 2023.

A. Mohan, A. Zhang, and M. Lindauer. Structure in deep reinforcement learning: A survey and open problems. *Journal of Artificial Intelligence Research*, 79:1167–1236, 2024.

A. Nikulin, V. Kurenkov, I. Zisman, V. Sinii, A. Agarkov, and S. Kolesnikov. XLand-minigrid: Scalable meta-reinforcement learning environments in JAX. In *NeurIPS'23 Workshop on Intrinsically-Motivated and Open-Ended Learning*, 2023.

J. Obando-Ceron and P. Castro. Revisiting rainbow: Promoting more insightful and inclusive deep reinforcement learning research. In *Proc. of ICML'21*, 2021.

J. Obando-Ceron, M. Bellemare, and P. Castro. Small batch deep reinforcement learning. In *Proc. of NeurIPS'23*, 2023.

J. Parker-Holder, V. Nguyen, and S. J. Roberts. Provably efficient online hyperparameter optimization with population-based bandits. In *Proc. of NeurIPS'20*, 2020.

J. Parker-Holder, M. Jiang, M. Dennis, M. Samvelyan, J. Foerster, E. Grefenstette, and T. Rocktäschel. Evolving curricula with regret-based environment design. In *Proc. of ICML'22*, 2022.

J. Parker-Holder, R. Rajan, X. Song, A. Biedenkapp, Y. Miao, T. Eimer, B. Zhang, V. Nguyen, R. Calandra, A. Faust, F. Hutter, and M. Lindauer. Automated reinforcement learning (AutoRL): A survey and open problems. *Journal of Artificial Intelligence Research*, 74:517–568, 2022.

F. Pfisterer, L. Schneider, J. Moosbauer, M. Binder, and B. Bischl. YAHPO Gym – an efficient multi-objective multi-fidelity benchmark for hyperparameter optimization. In *Proc. of AutoML Conf'22*. PMLR, 2022.

S. Pineda, H. Jomaa, M. Wistuba, and J. Grabocka. HPO-B: A large-scale reproducible benchmark for black-box HPO based on OpenML. In *Proc. of NeurIPS'21 Datasets and Benchmarks Track*, 2021.

M. Pislar, D. Szepesvari, G. Ostrovski, D. Borsa, and T. Schaul. When should agents explore? In *Proc. of ICLR'22*, 2022.

Y. Pushak and H. Hoos. Algorithm configuration landscapes: - more benign than expected? In *Proc. of PPSN'18*, 2018.

A. Raffin. RL baselines3 zoo. `https://github.com/DLR-RM/rl-baselines3-zoo`, 2020.

A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22: 268:1–268:8, 2021.

R. Sass, E. Bergman, A. Biedenkapp, F. Hutter, and M. Lindauer. Deepcave: An interactive analysis tool for automated machine learning. In *ICML ReALML Workshop*, 2022.

E. Schede, J. Brandt, A. Tornede, M. Wever, V. Bengs, E. Hüllermeier, and K. Tierney. A survey of methods for automated algorithm configuration. *Journal of Artificial Intelligence Research*, 75: 425–487, 2022.

J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arXiv:1707.06347 [cs.LG]*, 2017.

G. Shala, S. P. Arango, A. Biedenkapp, F. Hutter, and J. Grabocka. HPO-RL-Bench: A zero-cost benchmark for HPO in reinforcement learning. In *Proc. of AutoML Conf'24*. PMLR, 2024.

I. Sobol. On the distribution of points in a cube and the approximate evaluation of integrals. *USSR Computational Mathematics and Mathematical Physics*, 7(4):86–112, 1967.

E. Toledo. Stoix: Distributed single-agent reinforcement learning end-to-end in JAX, April 2024. URL `https://github.com/EdanToledo/Stoix`.

E. Toledo, L. Midgley, D. Byrne, C. R. Tilbury, M. Macfarlane, C. Courtot, and A. Laterre. Flashbax: Streamlining experience replay buffers for reinforcement learning with JAX, 2023. URL `https://github.com/instadeepai/flashbax/`.

M. Towers, J. Terry, A. Kwiatkowski, J. Balis, G. de Cola, T. Deleu, M. Goulão, A. Kallinteris, A. KG, M. Krimmel, R. Perez-Vicente, A. Pierré, S. Schulhoff, J. Tai, A. Shen, and O. Younis. Gymnasium, 2023. URL `https://zenodo.org/record/8127025`.

R. Turner, D. Eriksson, M. McCourt, J. Kiili, E. Laaksonen, Z. Xu, and I. Guyon. Bayesian optimization is superior to random search for machine learning hyperparameter tuning: Analysis of the Black-Box Optimization Challenge 2020. In *Proc. of NeurIPS'20 Competition and Demonstration*, 2021.

C. Voelcker, M. Hussing, and E. Eaton. Can we hop in general? a discussion of benchmark selection and design using the hopper environment. In *Finding the Frame: An RLC Workshop for Examining Conceptual Frameworks*, 2024.

X. Wan, C. Lu, J. Parker-Holder, P. Ball, V. Nguyen, B. Ru, and M. Osborne. Bayesian generational population-based training. In *Proc. of AutoML Conf'22*. PMLR, 2022.

J. Weng, M. Lin, S. Huang, B. Liu, D. Makoviichuk, V. Makoviychuk, Z. Liu, Y. Song, T. Luo, Y. Jiang, Z. Xu, and S. Yan. EnvPool: A highly parallel reinforcement learning environment execution engine. In *Proc. of NeurIPS'22*, 2022.

Z. Xu, H. van Hasselt, and D. Silver. Meta-gradient reinforcement learning. In *Proc. of NeurIPS'18*, 2018.

C. Ying, A. Klein, E. Christiansen, E. Real, K. Murphy, and F. Hutter. NAS-Bench-101: Towards reproducible neural architecture search. In *Proc. of ICML'19*, 2019.

A. Zela, J. Siems, L. Zimmer, J. Lukasik, M. Keuper, and F. Hutter. Surrogate NAS benchmarks: Going beyond the limited search spaces of tabular NAS benchmarks. In *Proc. of ICLR'22*, 2022.

B. Zhang, R. Rajan, L. Pineda, N. Lambert, A. Biedenkapp, K. Chua, F. Hutter, and R. Calandra. On the importance of hyperparameter optimization for model-based reinforcement learning. In *Proc. of AISTATS'21*, 2021.

Checklist

1. For all authors...
    (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
    (b) Did you describe the limitations of your work? [Yes]
    (c) Did you discuss any potential negative societal impacts of your work? [NA]
    (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
    (a) Did you state the full set of assumptions of all theoretical results? [NA]
    (b) Did you include complete proofs of all theoretical results? [NA]
3. If you ran experiments (e.g. for benchmarks)...
    (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes]
    (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes]
    (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes]
    (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes]
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
    (a) If your work uses existing assets, did you cite the creators? [Yes]
    (b) Did you mention the license of the assets? [Yes]
    (c) Did you include any new assets either in the supplemental material or as a URL? [Yes]
    (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [NA]
    (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [NA]
5. If you used crowdsourcing or conducted research with human subjects...
    (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [NA]
    (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [NA]
    (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [NA]

A Dataset Description

Our dataset is hosted on HuggingFace for easy and continued access: `https://huggingface.co/datasets/autorl-org/arlbench`. See the page there for in-depth information on data value and distributions. The croissant meta-data can also be found here: `https://github.com/automl/arlbench/blob/experiments/croissant_metadata.json`. Everything needed to reproduce the data can be found in the 'experiments' branch of our GitHub repository: `https://github.com/automl/arlbench/tree/experiments` It is intended to be used in continued research on AutoRL, e.g., by using it in warm-starting HPO optimizers, proposing novel analysis methods or meta-learning on it. The dataset is in CSV format, making it easily readable. We license it under the BSD-3 license.

B Reproducing Our Results

Below we describe our hardware setup and steps for reproducing our experiments.

B.1 Execution Environment

To conduct the experiments detailed in this paper, we pooled various computing resources. Below, we describe the different hardware setups used for CPU and GPU-based training.

CPU Jobs. Compute nodes with CPUs of type AMD Milan 7763, 2.45 GHz, each 2x 64 cores, 128GB main memory

GPU Jobs.

V100 Cluster: Compute nodes with CPUs of type Intel Xeon Platinum 8160, 2.1 GHz, each 2x 24 cores, 180GB main memory. Each node comes with 16 GPUs of type NVIDIA V100-SXM2 with NVLink and 32 GB HBM2, 5120 CUDA cores, 640 Tensor cores, 128 GB main memory

A100 Cluster: Compute nodes with CPUs of type AMD Milan 7763, 2.45 GHz, each 2x 64 cores, 126GB main memory. Each node comes with 1 GPU of type NVIDIA A100 with NVLink and 40 GB HBM2, 6,912 CUDA cores, 432 Tensor cores, 16 GB main memory

H100 Cluster: Compute nodes with CPUs of type Intel Xeon 8468 Sapphire, 2.1 GHz, each 2x 48 cores, 512GB main memory. Each node comes with 4 GPUs of type NVIDIA H100 with NVLink and 96 GB HBM2e, 16,896 CUDA cores, 528 Tensor cores, 512 GB main memory

B.2 Experiment Code

We provide code and runscripts for all of our dependencies in the 'experiments' branch of our repository: `https://github.com/automl/arlbench/tree/experiments`.

All scripts relating to the dataset creation and HPO optimizer runs are in 'runscripts'. For the performance over time plots, see 'runtime_comparison'. 'rs_data_analysis' contains the analysis of the HPO landscapes. For the subset selection, see 'subset_selection'. The subset validation and performance over time plots for the HPO optimizers can be found in 'subset_validation'. Additionally, we provide all of our raw data in 'results_finished' with 'results_combined' containing dataset aggregates. Instructions for the usage of all of these can be found in the ReadMe file of that branch.

C Maintenance Plan

Following [Eggensperger et al., 2021] and [Pfisterer et al., 2022], we provide a maintenance plan for the future of ARLBench. For our feature roadmap, see: `https://github.com/orgs/automl/projects/17`

Who Maintains. ARLBench is being developed and maintained as a cooperation between the Institute of AI at the Leibniz University of Hannover and the chair for AI Methodology at the RWTH Aachen University.

Contact. Improvement requests, issues and questions can be asked via issue in our GitHub repository: `https://github.com/automl/arlbench`. The contact e-mails we provide can be used for the same purpose.

Errata. There are no errata.

Library Updates. We plan on updating the library with new features, specifically more extensive state features, more algorithms and added environment frameworks. We also welcome updates via external pull requests which we will test and integrate into ARLBench. Changes will be communicated via the changelog of our GitHub and PyPI releases.

Support for Older Versions. Older versions of ARLBench will continue to be available on PyPI and GitHub, but we will only provide limited support.

Contributions. Contributions to ARLBench from external parties are welcome in any form, be extensions to other environment frameworks, added algorithms or extensions of the core interface. We describe the contribution process in our documentation: `https://automl.github.io/arlbench/main/CONTRIBUTING.html`. These contributions are managed via GitHub pull requests.

Dependencies. All of our dependencies are listed here in the GitHub repository: `https://github.com/automl/arlbench/blob/main/pyproject.toml`.

D Overview of all Environments

Tables 2, 3 and 4 provide an overview of all environments we executed for a given RL algorithm, including the underlying framework used and the number of environment steps for training.

Category	Framework	Name	#timesteps
ALE	Envpool	BattleZone-v5	10^7
ALE	Envpool	DoubleDunk-v5	10^7
ALE	Envpool	Phoenix-v5	10^7
ALE	Envpool	Qbert-v5	10^7
ALE	Envpool	NameThisGame-v5	10^7
Box2D	Envpool	LunarLander-v2	10^6
Box2D	Envpool	LunarLanderContinuous-v2	10^6
Box2D	Envpool	BipedalWalker-v3	10^6
Walker	Brax	Ant	$5 \cdot 10^7$
Walker	Brax	HalfCheetah	$5 \cdot 10^7$
Walker	Brax	Hopper	$5 \cdot 10^7$
Walker	Brax	Humanoid	$5 \cdot 10^7$
Classic Control	Gymnax	Acrobot-v1	10^6
Classic Control	Gymnax	CartPole-v1	10^5
Classic Control	Gymnax	MountainCarContinuous-v0	$2 \cdot 10^4$
Classic Control	Gymnax	MountainCar-v0	10^6
Classic Control	Gymnax	Pendulum-v1	10^5
xland	XLand-xland	xland-DoorKey-5x5	10^6
xland	XLand-xland	xland-EmptyRandom-5x5	10^5
xland	XLand-xland	xland-FourRooms	10^6
xland	XLand-xland	xland-Unlock	10^6

Table 2: Environments for PPO.

Category	Framework	Name	#timesteps
ALE	Envpool	BattleZone-v5	10^7
ALE	Envpool	DoubleDunk-v5	10^7
ALE	Envpool	Phoenix-v5	10^7
ALE	Envpool	Qbert-v5	10^7
ALE	Envpool	NameThisGame-v5	10^7
Box2D	Envpool	LunarLander-v2	10^6
Classic Control	Gymnax	Acrobot-v1	10^5
Classic Control	Gymnax	CartPole-v1	$5 \cdot 10^4$
Classic Control	Gymnax	MountainCar-v0	$12 \cdot 10^4$
xland	XLand-xland	xland-DoorKey-5x5	10^6
xland	XLand-xland	xland-EmptyRandom-5x5	10^5
xland	XLand-xland	xland-FourRooms	10^6
xland	XLand-xland	xland-Unlock	10^6

Table 3: Environments for DQN.

Category	Framework	Name	#timesteps
Box2D	Envpool	LunarLanderContinuous-v2	$5 \cdot 10^5$
Box2D	Envpool	BipedalWalker-v2	$5 \cdot 10^5$
Walker	Brax	Ant	$5 \cdot 10^6$
Walker	Brax	HalfCheetah	$5 \cdot 10^6$
Walker	Brax	Hopper	$5 \cdot 10^6$
Walker	Brax	Humanoid	$5 \cdot 10^6$
Classic Control	Gymnax	MountainCarContinuous-v0	$5 \cdot 10^4$
Classic Control	Gymnax	Pendulum-v1	$2 \cdot 10^4$

Table 4: Environments for SAC.

E Performance Comparisons with Other RL Frameworks

To validate the correctness of our implementations beyond unit testing, we compare their performance on a range of environments to established RL frameworks in terms of reward achieved and running time. Additionally, running time comparisons for each environment category are shown in Figures 8, 9, 10, 11, and12.

Figure 13 compares the resulting learning curves between ARLBench, SB3 and the Brax default agent. We use the Brax agent instead of SB3 since SB3 performed significantly worse than we expected. In most of our tests we observed very similar behavior with the other framworks and ARLBench outperforming the other two times and showing comparable learning curves for all other experiments. In the case of DQN, where SB3 performed better on CartPole and worse on Pong, SB3's results look noisy, possibly causing this discrepancy in both directions. SB3 also outperforms ARLbench on Pendulum, though this difference is fairly slight. For PPO on Ant, ARLBench performs quite a bit better than the Brax default agent, though their performances of SAC are the same. Inconsistencies in learning curves can be due to differences in the implementations of algorithms[1] and environments [Voelcker et al., 2024]. Overall, this shows that our algorithms perform on par with other commonly used implementations.

[1] `https://iclr-blog-track.github.io/2022/03/25/ppo-implementation-details/`

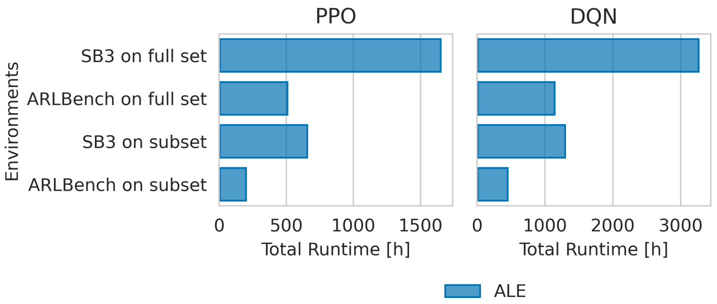


Figure 8: Running time comparison for an HPO method of 32 RL runs, using 10 seeds each on the full environment set and our subsets between ARLBench and SB3 for ALE. JAX-related speedup factors are 3.21 for PPO and 2.83 for DQN. Total speedup factors of the ARLBench subset compared to the full set of environments in SB3 are 8.03 for PPO and 7.08 for DQN. Note: As ALE environments have discrete action spaces, SAC is left out in this figure.

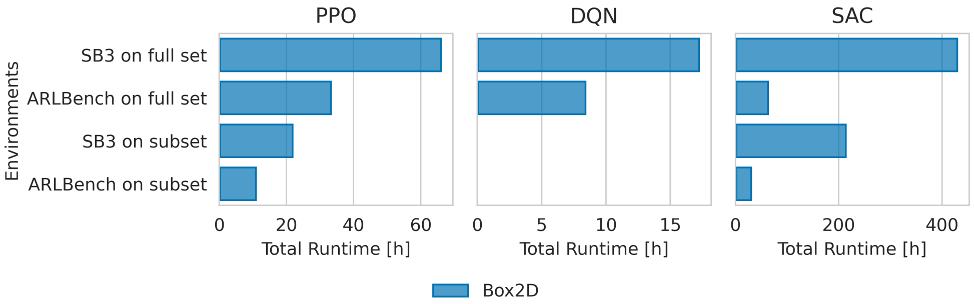


Figure 9: Running time comparison for an HPO method of 32 RL runs, using 10 seeds each on the full environment set and our subsets between ARLBench and SB3 for Box2D. JAX-related speedup factors are 1.97 for PPO, 2.04 for DQN, and 6.64 for SAC. Total speedup factors of the ARLBench subset compared to the full set of environments in SB3 are 5.92 for PPO and 13.27 for SAC. As no Box2D environment is part of the DQN subset, there is no total speedup factor for DQN.

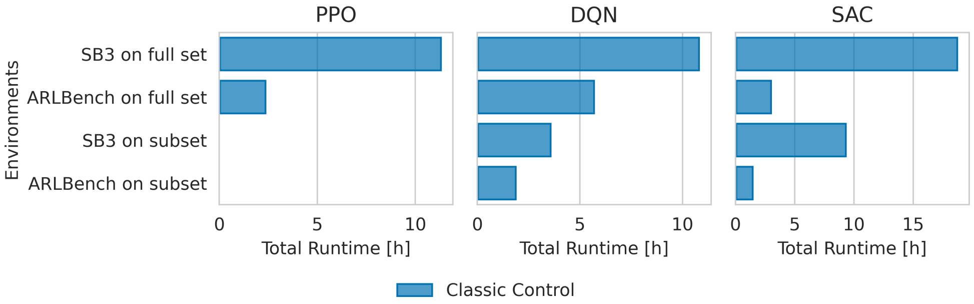


Figure 10: Running time comparison for an HPO method of 32 RL runs, using 10 seeds each on the full environment set and our subsets between ARLBench and SB3 for Classic Control. JAX-related speedup factors are 4.72 for PPO, 1.89 for DQN, and 6.10 for SAC. Total speedup factors of the ARLBench subset compared to the full set of environments in SB3 are 5.68 for DQN and 12.2 for SAC. As no Classic Control environment are part of the PPO subset, there is no total speedup factor for PPO.

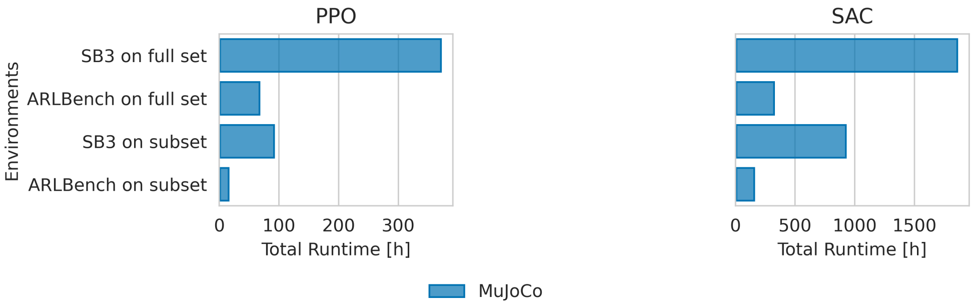


Figure 11: Running time comparison for an HPO method of 32 RL runs, using 10 seeds each on the full environment set and our subsets between ARLBench and SB3 for Brax/MuJoCo. JAX-related speedup factors are 5.4 for PPO and 5.64 for SAC. Total speedup factors of the ARLBench subset compared to the full set of environments in SB3 are 21.62 for PPO, and 11.28 for SAC. Note: As MuJoCo environments have continuious action spaces DQN is left out in this figure.

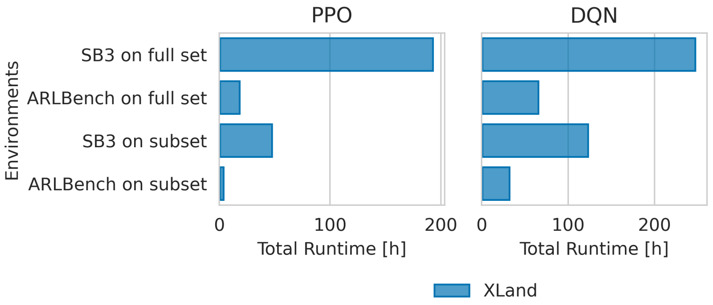


Figure 12: Running time comparison for an HPO method of 32 RL runs, using 10 seeds each on the full environment set and our subsets between ARLBench and SB3 for XLand. JAX-related speedup factors are 10.02 for PPO and 3.72 for DQN. Total speedup factors of the ARLBench subset compared to the full set of environments in SB3 are 40.07 for PPO and 7.42 for DQN. Note: As ALE environments have discrete action spaces SAC is left out in this figure.

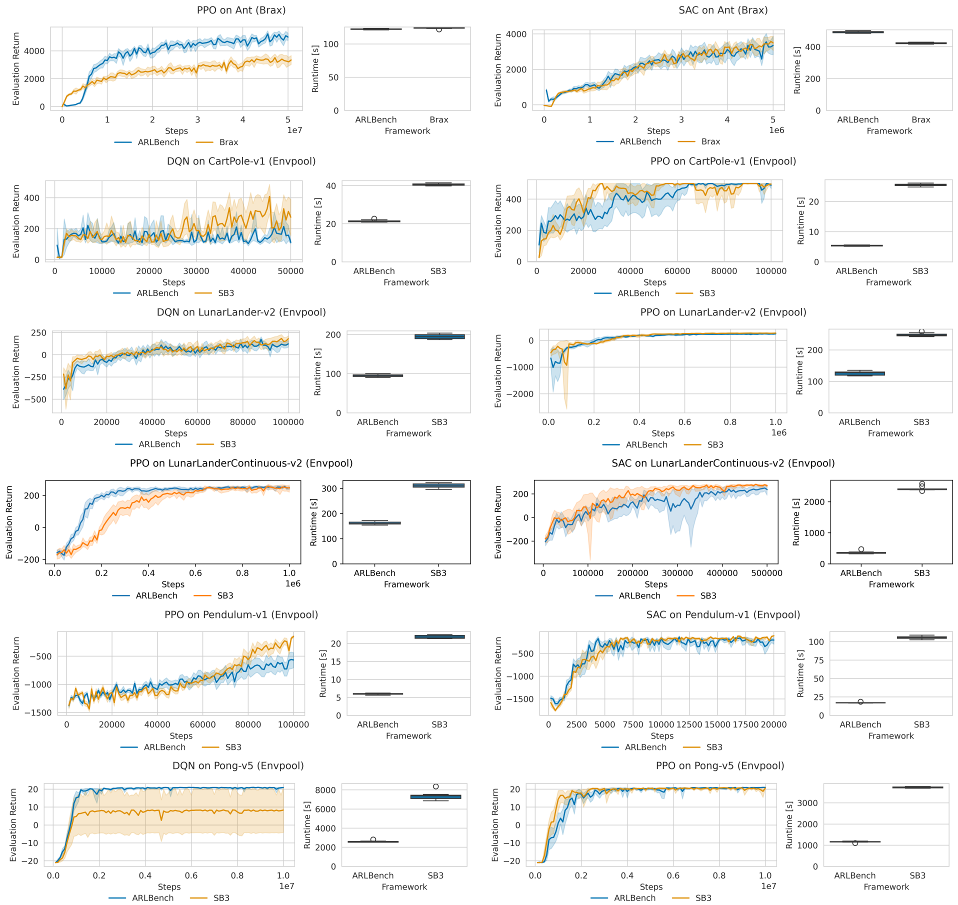


Figure 13: Performance comparisons of ARLBench and the Brax default agent, StableBaselines3 [Raffin et al., 2021]

Table 5 show the speedups we achieve in terms of running time over StableBaselines3 (SB3) [Raffin et al., 2021] on all subsets while Tables 6, 7 and 8 list the same for each environment individually. As already discussed, we see a consistently large speedup, most pronounced for the Brax walkers with a factor of 8.57 for PPO and 10.67 for SAC. The lowest speedups we observe are still close to a factor of 2: 1.89 for DQN CartPole as well as 1.91 and 1.97 respectively for PPO LunarLander and LunarLanderContinuous.

Algorithm	Set	ARLBench	SB3	Speedup
PPO	All	2h	7.18h	3.59
PPO	Subset	0.74h	2.58h	3.48
DQN	All	3.87h	11.10h	2.87
DQN	Subset	1.55h	4.49h	2.89
SAC	All	1.25h	7.23h	5.78
SAC	Subset	0.62h	3.61h	5.82
Sum	All	7.12h	25.51h	3.58
Sum	Subset	2.91h	10.68h	3.67

Table 5: Running time comparisons for a single RL training between ARLBench and StableBaselines3 (SB3) [Raffin et al., 2021] on the set of all environments and the selected subset. The numbers are based on the results in Tables 6, 7, and 8. For each environment category, we use the running times from the experiments to estimate the overall running time for this category.

Category	Framework	Name	ARLBench	SB3	Speedup
Classic Control	Envpool	CartPole-v1	5.42s	25.54s	4.72
Classic Control	Envpool	Pendulum-v1	5.98s	21.87s	3.66
Box2D	Envpool	LunarLander-v2	125.87s	248.54s	1.97
Box2D	Envpool	LunarLanderContinuous-v2	162.77s	311.25s	1.91
Minigrid	XLand	Minigrid-DoorKey-5x5	54.38s	544.71s	10.01
ALE	Envpool	Pong-v5	1161.11s	3728.58s	3.21
Walker	Envpool	Ant	194.09s	1048.84s	
Walker	Brax	Ant	122.28s	1048.84s*	8.57
				Average	4.86

Table 6: Speedup of ARLBench PPO compared to StableBaselines3 (SB3) [Raffin et al., 2021] on different envrionments. *Note: Since SB3 is not compatible with Brax without manual interface adapation, we compare the results of MuJoCo + SB3 and Brax + ARLBench.

Category	Framework	Name	ARLBench	SB3	Speedup
Classic Control	Envpool	CartPole-v1	21.5s	40.68s	1.89
Box2D	Envpool	LunarLander-v2	95.27s	194.61s	2.04
Minigrid	XLand	Minigrid-DoorKey-5x5	187.73s	697.64s	3.71
ALE	Envpool	Pong-v5	2602.69s	7373.40s	2.83
				Average	2.15

Table 7: Speedup of ARLBench DQN compared to StableBaselines3 (SB3) [Raffin et al., 2021] on different envrionments.

Category	Framework	Name	ARLBench	SB3	Speedup
Classic Control	Envpool	Pendulum-v1	17.32s	105.67s	6.10
Box2D	Envpool	LunarLanderContinuous-v2	365.45s	2425.04s	6.64
Walker	Envpool	Ant	930.06s	5245.17s	
Walker	Brax	Ant	491.70s	5245.17s*	10.67
				Average	7.80

Table 8: Speedup of ARLBench SAC compared to StableBaselines3 (SB3) [Raffin et al., 2021] on different envrionments. *Note: Since SB3 is not compatible with Brax without manual interface adapation, we compare the results of MuJoCo + SB3 and Brax + ARLBench.

F Algorithm Search Spaces

For all algorithms, we used extensive search spaces covering almost all hyperparameters that are commonly optimized. The search spaces for PPO, DQN and SAC are presented in Table 9, 10 and 11 respectively. We choose not to optimize some hyperparameters to keep the computational resources constant for each training. The default values for these hyperparameters for each environment domain have been inferred from stable-baselines3 zoo Raffin [2020] and Google Brax's hyperparameter sweeps Freeman et al. [2021] and are shown in Tables 9, 10 and 11 accordingly. The search space for the batch sizes was set to one power of two below and above its baseline value.

Hyperparameter	Box2D	XLand	ALE	CC	Brax
batch size	$\{32, 64, 128\}$		$\{128, 256, 512\}$		$\{512, 1024, 2048\}$
number of environments	16	8			2048
number of steps	1024	32	128	32	512
update epochs	4	10	4		
learning rate	$\log([10^{-6}, 10^{-1}])$				
entropy coefficient	$[0.0, 0.5]$				
gae lambda	$[0.8, 0.9999]$				
policy clipping	$[0.0, 0.5]$				
value clipping	$[0.0, 0.5]$				
normalize advantages	$\{\text{Yes}, \text{No}\}$				
value function coefficient	$[0.0, 1.0]$				
max gradient norm	$[0.0, 1.0]$				

Table 9: The hyperparameter search space for PPO. To keep the computational costs feasible we choose not to optimize the number of steps per epoch and update epochs.

Hyperparameter	ALE	Box2D	CC	XLand
batch size	$\{16, 32, 64\}$	$\{64, 128, 256\}$		$\{32, 64, 128\}$
number of environments	8	4	1	4
buffer priority sampling	$\{\text{Yes}, \text{No}\}$			
buffer α	$[0.01, 1.0]$			
buffer β	$[0.01, 1.0]$			
buffer ϵ	$\log([10^{-7}, 10^{-3}])$			
buffer size	$[1024, 10^6]$			
initial epsilon	$[0.5, 1.0]$			
target epsilon	$[0.001, 0.2]$			
learning rate	$\log([10^{-6}, 10^{-1}])$			
learning starts	$[1, 2048]$			
use target network	$\{\text{Yes}, \text{No}\}$			
target update interval	$[1, 2000]$			

Table 10: The hyperparameter search space for DQN. The target update interval is a conditional hyperparameter that is only optimized when a target network is used. Similarly, buffer α, β and ϵ are only optimized when priority sampling is used. If the number of training steps is smaller than the upper limit of the buffer size, the buffer size limit is reduced accordingly.

Hyperparameter	Box2D	CC	Brax
batch size	$\{128, 256, 512\}$	$\{256, 512, 1024\}$	$\{512, 1024, 2048\}$
number of environments	1	1	64
buffer priority sampling	$\{\text{Yes}, \text{No}\}$		
buffer α	$[0.01, 1.0]$		
buffer β	$[0.01, 1.0]$		
buffer ϵ	$\log([10^{-7}, 10^{-3}])$		
buffer size	$[1024, 10^6]$		
learning rate	$\log([10^{-6}, 10^{-1}])$		
learning starts	$[1, 2048]$		
use target network	$\{\text{Yes}, \text{No}\}$		
tau	$[0.01, 1.0]$		
reward scale	$\log([0.1, 10])$		

Table 11: The hyperparameter search space for SAC. The hyperparameter tau is a conditional parameter that is only optimized when a target network is used. Similarly, buffer α, β and ϵ are only optimized when priority sampling is used. If the number of training steps is smaller than the upper limit of the buffer size, the buffer size limit is reduced accordingly.

G Subset Selection

We provide additional information on the subset selection in the form of explanations, alternative selection methods, and a more detailed look into the results, including environment weights.

G.1 Additional Explanation

We select the subset based on the hyperparameter landscapes obtained through Sobol sampling. For each randomly sampled hyperparameter configuration, the RL algorithm is trained and evaluated on a separate evaluation environment. As evaluation metric, we collect the cumulative episode rewards, i.e., return of 128 episodes and calculate the mean. The mean return for environment $e \in \mathcal{E}$ and hyperparameter configuration $\lambda \in \Lambda$ is denoted as r^e_λ and calculated as

$$r^e_\lambda = \mathbb{E}_{s \sim \mathcal{S}} \left[\frac{1}{128} \cdot \sum_{i=1}^{128} \sum_{t=1}^{T} R_t^{(i,s)} \right] \tag{2}$$

where $\mathcal{S}$ is the set of 10 random seeds, T is the number of steps in the i-th evaluation episode, and R_t corresponds to the reward at time step t in the i-th episode for seed s. As reward ranges differ across environments, we have to apply normalization to compare the corresponding results. However, normalization based on human expert scores is not possible as done by Aitchison et al. [2023] for the selection of Atari-5. We apply rank-based normalization to compare the rewards of different environments. By ranking the rewards r^e_λ of all configurations $\lambda \in \Lambda$ for a given environment e, with higher rewards corresponding to higher ranks, and normalizing these ranks to the interval $[0, 1]$, we obtain the performance scores p^e_λ. The performance score p^e_λ for each configuration λ in environment e is given by:

$$p^e_\lambda = \frac{\text{rank}(r^e_\lambda) - \min_{\lambda' \in \Lambda} \text{rank}(r^e_{\lambda'})}{\max_{\lambda' \in \Lambda} \text{rank}(r^e_{\lambda'}) - \min_{\lambda' \in \Lambda} \text{rank}(r^e_{\lambda'})}, \tag{3}$$

where $\text{rank}(r^e_\lambda)$ denotes the rank of the return r^e_λ among all returns in environment e.

For the regression model, we use the *LinearRegression* class from the *scikit-learn*[2] package. This relies on the ordinary least squares method for fitting, which is invariant to permutation of features, i.e., environments.

[2] `https://scikit-learn.org`

G.2 Alternative Methods

In addition to our chosen method of rank-based normalization in combination with the Spearman correlation as a distance metric, we compare alternative normalization methods as well as MSE for the distance. Figure 14 shows the validation error of different combinations while Figure 15 shows the resulting Spearman correlation to the full environment set. While MSE might produce a good validation error, the resulting correlation is significantly worse than using the Spearman correlation for the distance. Min-max normalization performs slightly worse than rank normalization for the validation error. Therefore we chose rank-based normalization with Spearman correlation for our subset selection.

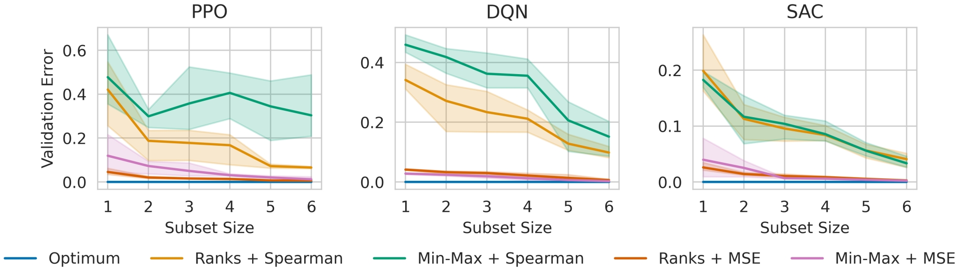


Figure 14: Comparison of the validation error of different ranking methods and error functions based on the subset size.

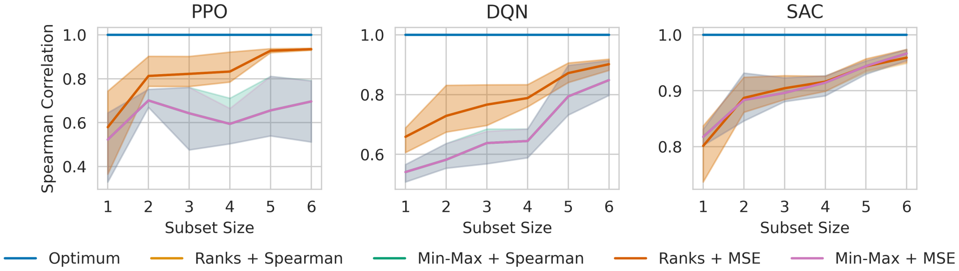


Figure 15: Comparison of the validation error of different ranking methods and error functions based on the subset size. Please note, that not all lines in this plot are visible due to overlaps. The reason is that the approaches Ranks + Spearman and Ranks + MSE as well as Min-Max + MSE and Min-Max Spearman each results in the exact same Spearman correlation and thus are not distinguishable in the plot.

G.3 Extended Subset Results

In addition to the environments in the subsets, we also provide the exact weights for each environment in the subsets in Table 12. Furthermore, Figures 16 and 17 show the optimization-over-time results for DQN and SAC.

Algorithm	Environments (with predicted weights)	ρ_s
PPO	0.21× LunarLander, 0.21× Humanoid, 0.18× BattleZone, 0.12× Phoenix, 0.23× XLand-EmptyRandom	0.96
DQN	0.33× Acrobot, 0.11× NameThisGame, 0.22× DoubleDunk 0.12× XLand-FourRooms, 0.18× XLand-EmptyRandom	0.96
SAC	0.32× BipedalWalker, 0.31× HalfCheetah, 0.15× Hopper, 0.19× MountainCarContinuous	0.97

Table 12: The environment subsets selected for each algorithm with their Spearman correlation to the full environment set.

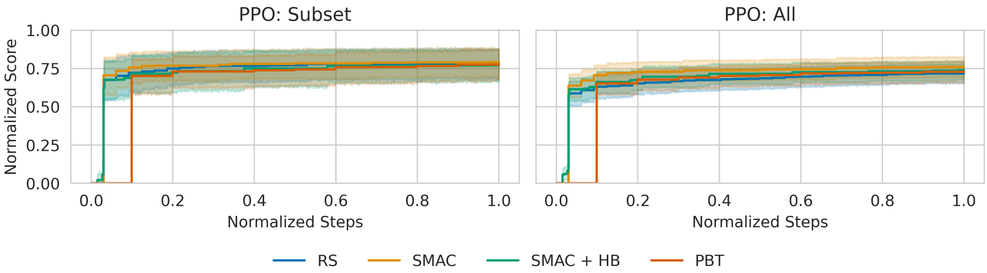


Figure 16: Anytime performance of the HPO methods for PPO.

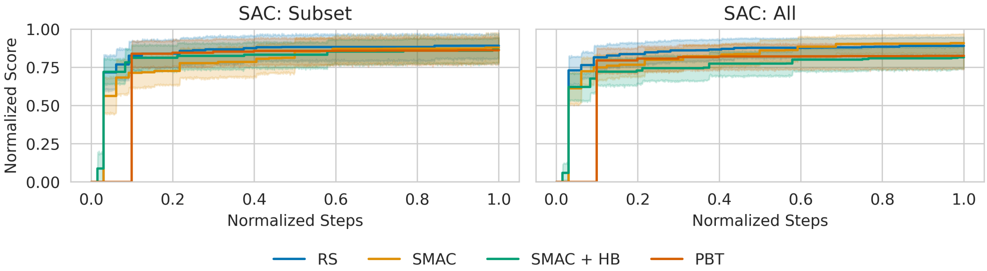


Figure 17: Anytime performance of the HPO methods for SAC.

H Landscape Analysis

We use DeepCave [Sass et al., 2022] to analyze our performance dataset with regards to performance distribution, hyperparameter importance and budget correlation over time. Please note that in some cases, results can be missing, due to consistent numerical errors in the analysis, e.g., in the case of SAC on Halfcheetah.

H.1 Landscape Behaviour

Algorithm configuration landscapes are often found to show relatively benign structure, characterized by unimodal responses and compensatory or negligible interactions [Pushak and Hoos, 2018]. However, in our experiments, we observe that some partial hyperparameter landscapes deviate from these traits, displaying challenging structure instead. Figure 18 highlights the contrast between benign and adverse landscapes in our experiments, providing further insight into their differing characteristics.

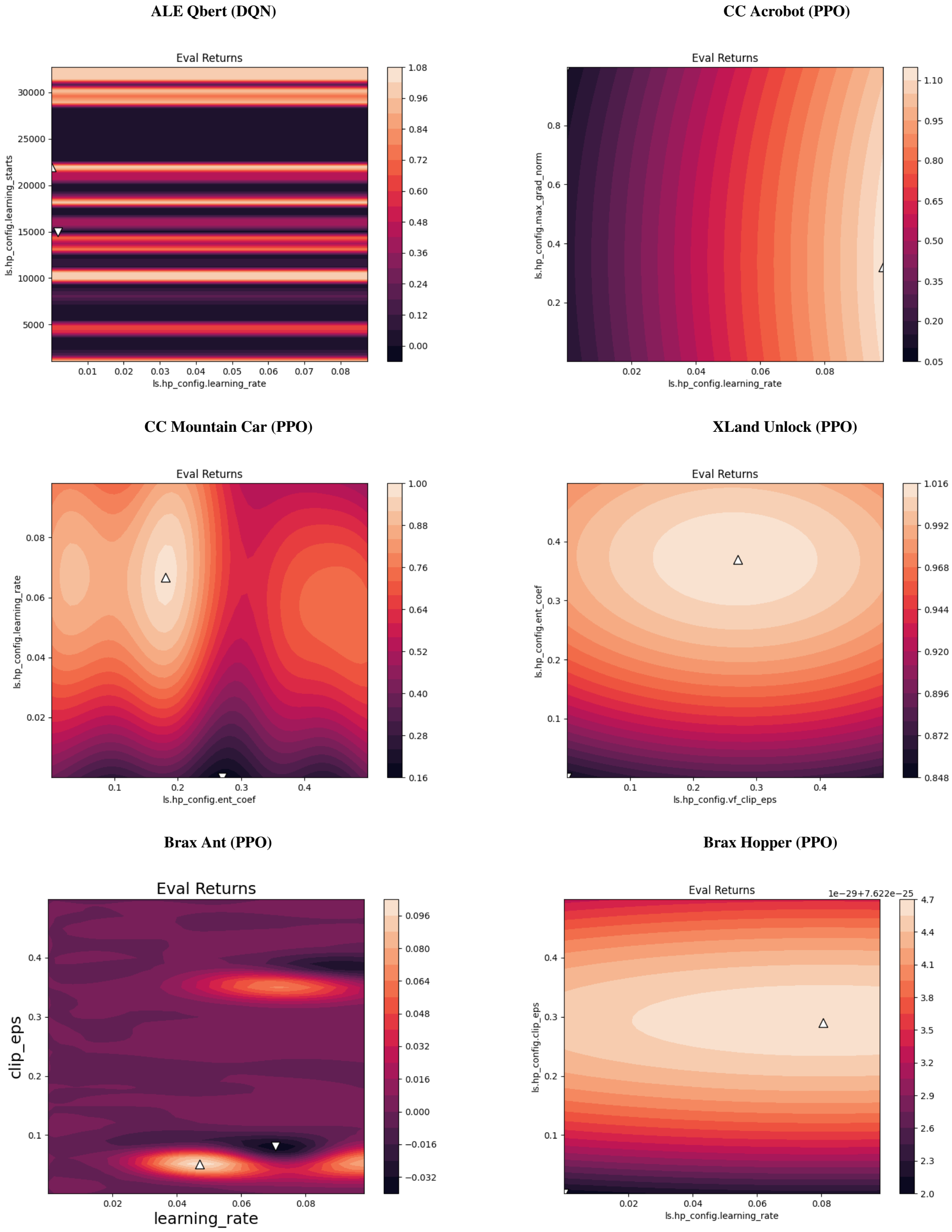


Figure 18: Comparison of adverse landscapes on the left with typical benign landscapes on the right. Lighter is better, mean performance over 10 seeds. Adverse landscapes exhibit multi-modality, whereas benign landscapes are uni-modal and display minimal to no hyperparameter interaction.

H.2 Performance Distributions

Completing the results from the performance distributions comparison in Section 4.3, Figures 19 and 20 show the distribution of scores for the domains and subsets of DQN and SAC, respectively. Just like for PPO, there are fairly direct correspondences between selected environments and the score distributions of the full domains. The only seeming exception is Box2D for DQN which has a lot of low scores that are not directly represented by one selected environment. Acrobot in that subset, however, covers a lot of such bad configurations even though it has higher performances overall.

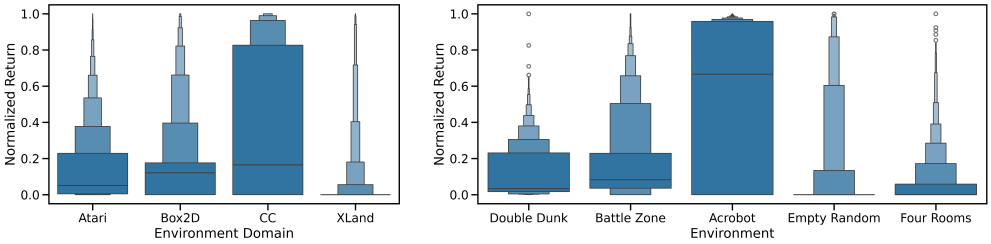


Figure 19: Score distribution across environment domains and the selected subset of DQN.

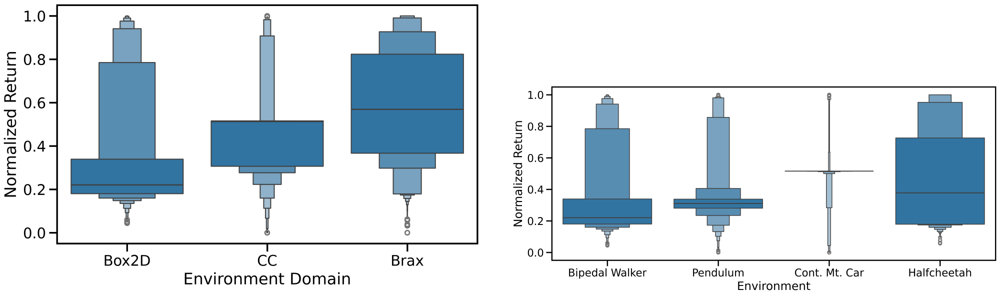


Figure 20: Score distribution across environment domains and the selected subset of SAC.

H.3 Hyperparameter Importances

Tables 13, 14 and 15 show extended information on the number of important hyperparameters for each environment domain as well as the subset and full environment set. The set of top 3 interactions are shown in Table 16 for PPO, Table 17 for DQN and Table 18 for SAC. We also include the full set of importance plots for each environment in Figures 21, 22, 23 and 24 for PPO, Figures 25, 26 and 27 for DQN and Figure 28 for SAC.

	ALE	Box2D	CC	Xland	Brax	All	Subset
#HPs with over 10% importance	1.6	1.5	1.0	1.75	0.75	1.3	1.0
#HPs with over 5% importance	1.6	1.5	3.4	2.25	1.75	2.2	1.2
#HPs with over 3% importance	2.0	3.0	4.0	2.5	3.0	2.9	2.0

Table 13: Fraction of hyperparameters with over 10%, 5%, and 3% importance on the full set and subset for PPO.

	ALE	Box2D	CC	Xland	All	Subset
#HPs with over 10% importance	2.0	1.0	1.0	2.25	1.77	2.0
#HPs with over 5% importance	2.8	1.0	1.33	3.5	2.54	2.75
#HPs with over 3% importance	3.8	2.0	2.33	4.0	3.38	3.5

Table 14: Fraction of hyperparameters with over 10%, 5%, and 3% importance on the full set and subset for DQN.

	Box2D	CC	Brax	All	Subset
#HPs with over 10% importance	1.0	1.5	1.0	1.14	0.75
#HPs with over 5% importance	1.0	3.0	1.5	1.86	1.25
#HPs with over 3% importance	1.0	3.5	2.75	2.71	2.5

Table 15: Fraction of hyperparameters with over 10%, 5%, and 3% importance on the full set and subset for SAC.

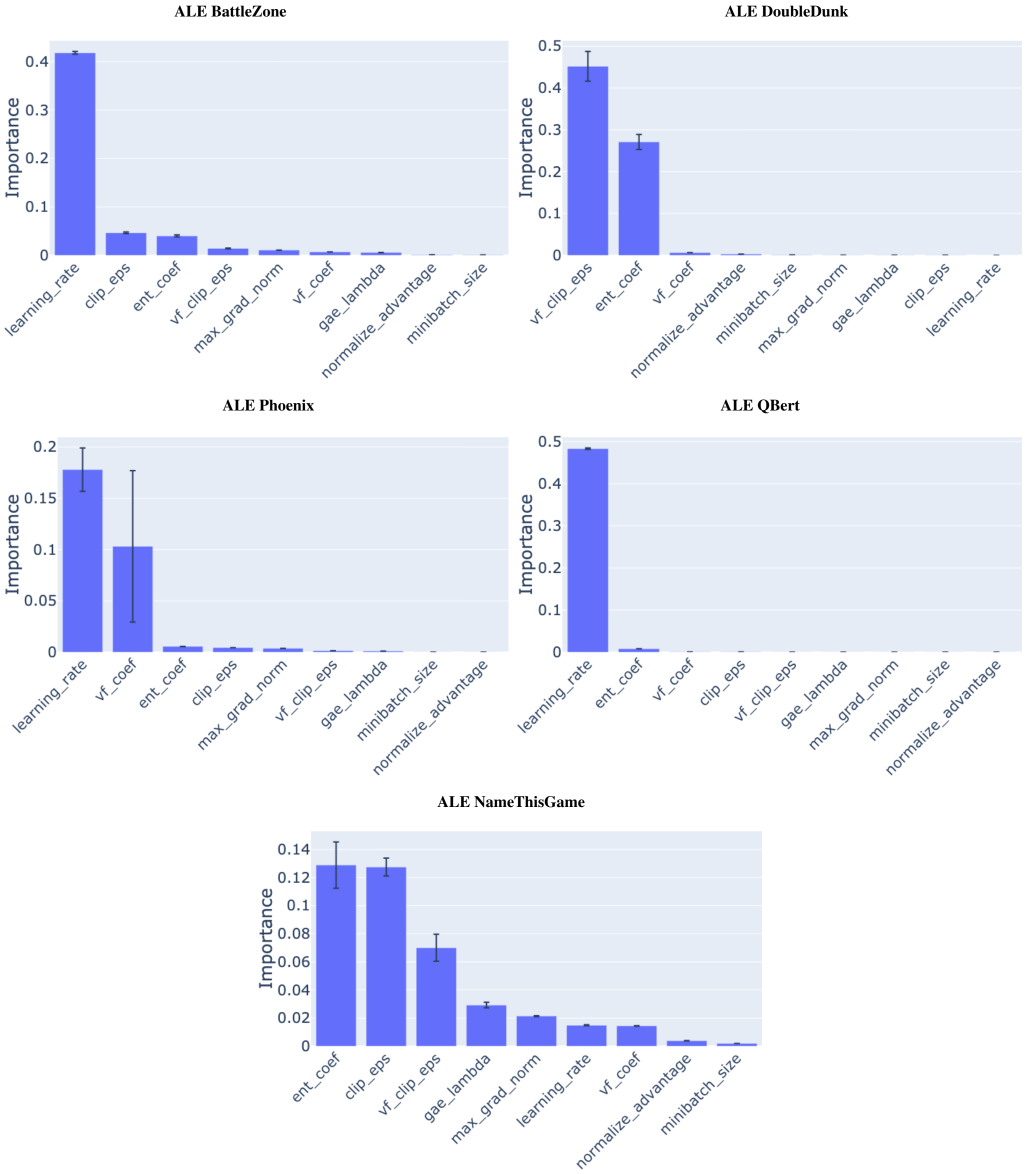


Figure 21: Hyperparameter importances for PPO: ALE.

Environment	Highest Interaction	2nd Highest Interaction	3rd Highest Interaction
ALE QBert	(ent coef, learning rate) : 0.38	(learning rate, vf coef) : 0.02	(learning rate, max grad norm) : 0.01
ALE DoubleDunk	(ent coef, vf coef) : 0.09	(ent coef, vf clip eps) : 0.07	(ent coef, normalize advantage) : 0.04
ALE Phoenix	(learning rate, vf clip eps) : 0.11	(ent coef, learning rate) : 0.08	(learning rate, vf coef) : 0.08
ALE NameThisGame	(clip eps, vf clip eps) : 0.06	(ent coef, vf coef) : 0.05	(clip eps, ent coef) : 0.05
ALE BattleZone	(learning rate, normalize advantage) : 0.05	(learning rate, vf coef) : 0.05	(clip eps, learning rate) : 0.04
Box2D LunarLander	(ent coef, learning rate) : 0.02	(clip eps, learning rate) : 0.02	(learning rate, vf coef) : 0.02
Box2D LunarLanderContinuous	(clip eps, normalize advantage) : 0.09	(clip eps, learning rate) : 0.07	(clip eps, max grad norm) : 0.04
Box2D BipedalWalker	(learning rate, normalize advantage) : 0.03	(ent coef, normalize advantage) : 0.02	(ent coef, vf clip eps) : 0.02
CC Acrobot	(learning rate, minibatch size) : 0.07	(max grad norm, minibatch size) : 0.05	(learning rate, max grad norm) : 0.02
CC CartPole	(ent coef, learning rate) : 0.06	(gae lambda, learning rate) : 0.04	(learning rate, vf coef) : 0.03
CC MountainCar	(clip eps, ent coef) : 0.07	(max grad norm, vf coef) : 0.04	(ent coef, vf coef) : 0.04
CC Pendulum	(learning rate, max grad norm) : 0.05	(ent coef, learning rate) : 0.04	(ent coef, normalize advantage) : 0.03
CC ContinuousMountainCar	(clip eps, ent coef) : 0.04	(ent coef, normalize advantage) : 0.03	(normalize advantage, vf coef) : 0.02
XLand DoorKey	(learning rate, normalize advantage) : 0.19	(max grad norm, minibatch size) : 0.13	(ent coef, normalize advantage) : 0.03
XLand EmptyRandom	(learning rate, normalize advantage) : 0.08	(learning rate, vf coef) : 0.03	(ent coef, learning rate) : 0.02
XLand FourRooms	(learning rate, vf clip eps) : 0.04	(ent coef, learning rate) : 0.04	(clip eps, vf clip eps) : 0.03
XLand Unlock	(clip eps, ent coef) : 0.05	(ent coef, max grad norm) : 0.04	(ent coef, vf clip eps) : 0.03
Brax Ant	(clip eps, learning rate) : 0.09	(learning rate, max grad norm) : 0.06	(learning rate, minibatch size) : 0.06
Brax Hopper	(learning rate, vf clip eps) : 0.08	(learning rate, max grad norm) : 0.08	(learning rate, vf coef) : 0.07
Brax Hopper	(clip eps, learning rate) : nan	(clip eps, vf coef) : nan	(learning rate, vf coef) : nan
Brax Humanoid	(vf clip eps, vf coef) : 0.22	(gae lambda, vf coef) : 0.08	(clip eps, vf coef) : 0.07

Table 16: Interactions effects for PPO via fANOVA.

Environment	Highest Interaction	2nd Highest Interaction	3rd Highest Interaction
ALE Qbert	(buffer batch size, learning rate) : 0.08	(learning rate, target epsilon) : 0.06	(learning rate, target update interval) : 0.02
ALE DoubleDunk	(buffer alpha, learning rate) : 0.05	(buffer alpha, target epsilon) : 0.04	(buffer alpha, target update interval) : 0.03
ALE Phoenix	(buffer batch size, target epsilon) : 0.05	(buffer alpha, target epsilon) : 0.04	(buffer alpha, learning rate) : 0.04
ALE NameThisGame	(learning rate, learning starts) : 0.05	(learning rate, target epsilon) : 0.04	(buffer alpha, learning rate) : 0.04
ALE BattleZone	(buffer size, initial epsilon) : 0.07	(buffer alpha, learning rate) : 0.03	(buffer alpha, initial epsilon) : 0.03
Box2D LunarLander	(buffer size, learning rate) : 0.04	(buffer alpha, learning rate) : 0.03	(learning rate, learning starts) : 0.03
CC Acrobot	(learning rate, target update interval) : 0.05	(buffer alpha, learning rate) : 0.01	(initial epsilon, learning rate) : 0.01
CC CartPole	(buffer epsilon, learning rate) : 0.03	(learning rate, target epsilon) : 0.02	(learning rate, learning starts) : 0.02
CC MountainCar	(learning rate, target update interval) : 0.11	(buffer alpha, learning rate) : 0.04	(buffer alpha, target update interval) : 0.02
XLand DoorKey	(buffer alpha, learning rate) : 0.06	(buffer size, target epsilon) : 0.06	(learning rate, learning starts) : 0.04
XLand EmptyRandom	(buffer alpha, learning rate) : 0.09	(buffer alpha, initial epsilon) : 0.06	(buffer alpha, target epsilon) : 0.05
XLand FourRooms	(buffer alpha, learning rate) : 0.04	(buffer size, target epsilon) : 0.04	(buffer epsilon, learning starts) : 0.02
XLand Unlock	(buffer alpha, learning starts) : 0.06	(buffer alpha, buffer size) : 0.06	(buffer alpha, learning rate) : 0.04

Table 17: Interaction effects for DQN hyperparameters via fANOVA.

Environment	Highest Interaction	2nd Highest Interaction	3rd Highest Interaction
Box2D LunarLanderContinuous	(buffer batch size, learning rate) : 0.04	(buffer size, learning rate) : 0.03	(buffer batch size, buffer size) : 0.02
Box2D BipedalWalker	(learning rate, reward scale) : 0.05	(buffer alpha, learning rate) : 0.04	(buffer beta, learning rate) : 0.02
CC Pendulum	(alpha, alpha auto) : nan	(alpha, buffer alpha) : nan	(alpha, buffer batch size) : nan
CC MountainCarContinuous	(buffer size, tau) : 0.1	(buffer alpha, buffer beta) : 0.05	(buffer alpha, learning starts) : 0.04
Brax Ant	(learning rate, tau) : 0.1	(learning rate, reward scale) : 0.07	(buffer size, learning rate) : 0.02
Brax HalfCheetah	(buffer beta, buffer size) : 0.03	(buffer beta, tau) : 0.03	(buffer beta, learning starts) : 0.03
Brax Hopper	(learning rate, tau) : 0.15	(learning rate, reward scale) : 0.09	(learning rate, use target network) : 0.03
Brax Humanoid	(buffer alpha, tau) : 0.05	(buffer beta, learning starts) : 0.03	(buffer batch size, learning starts) : 0.02

Table 18: Interaction effects for SAC hyperparameters via fANOVA.

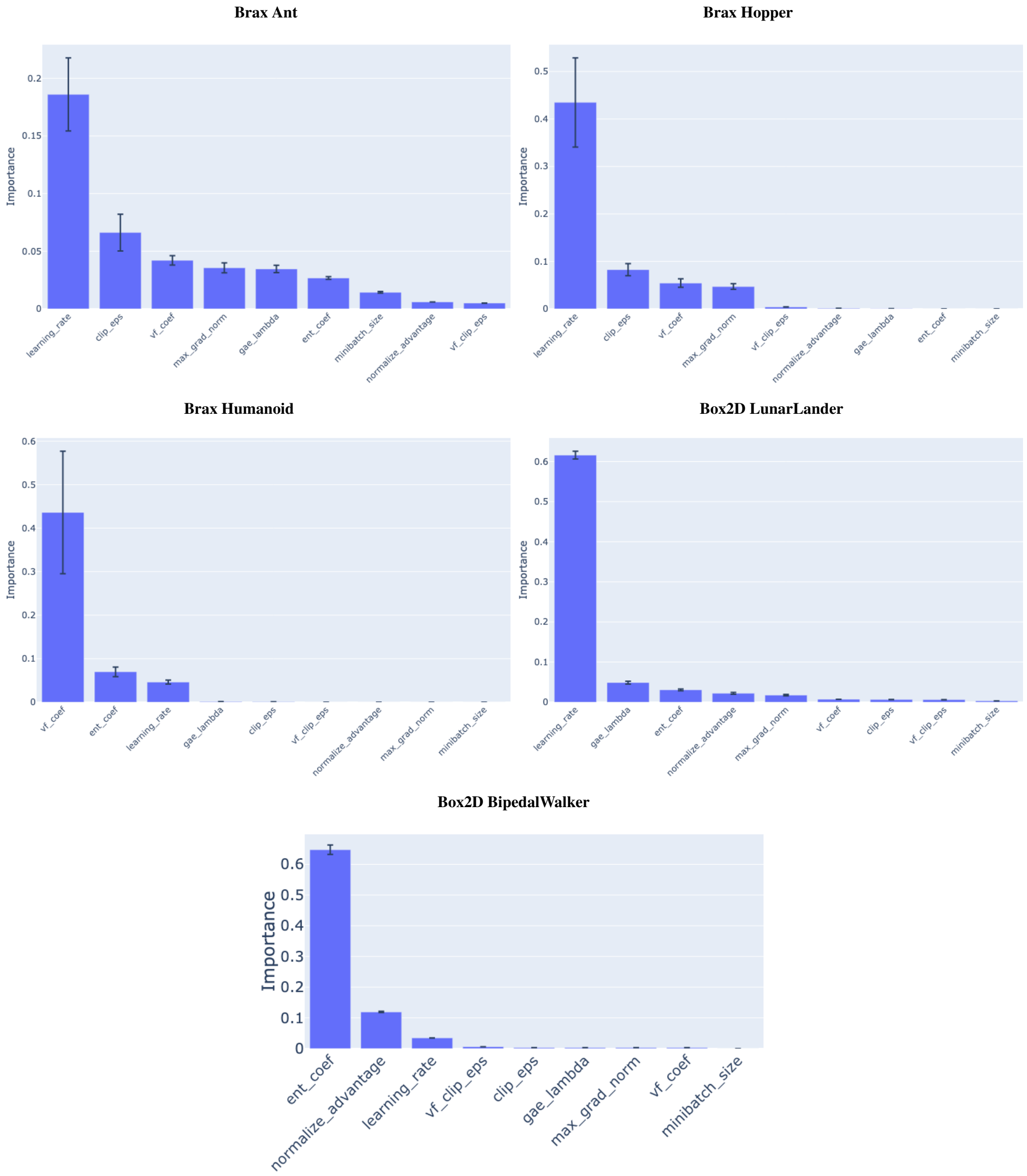
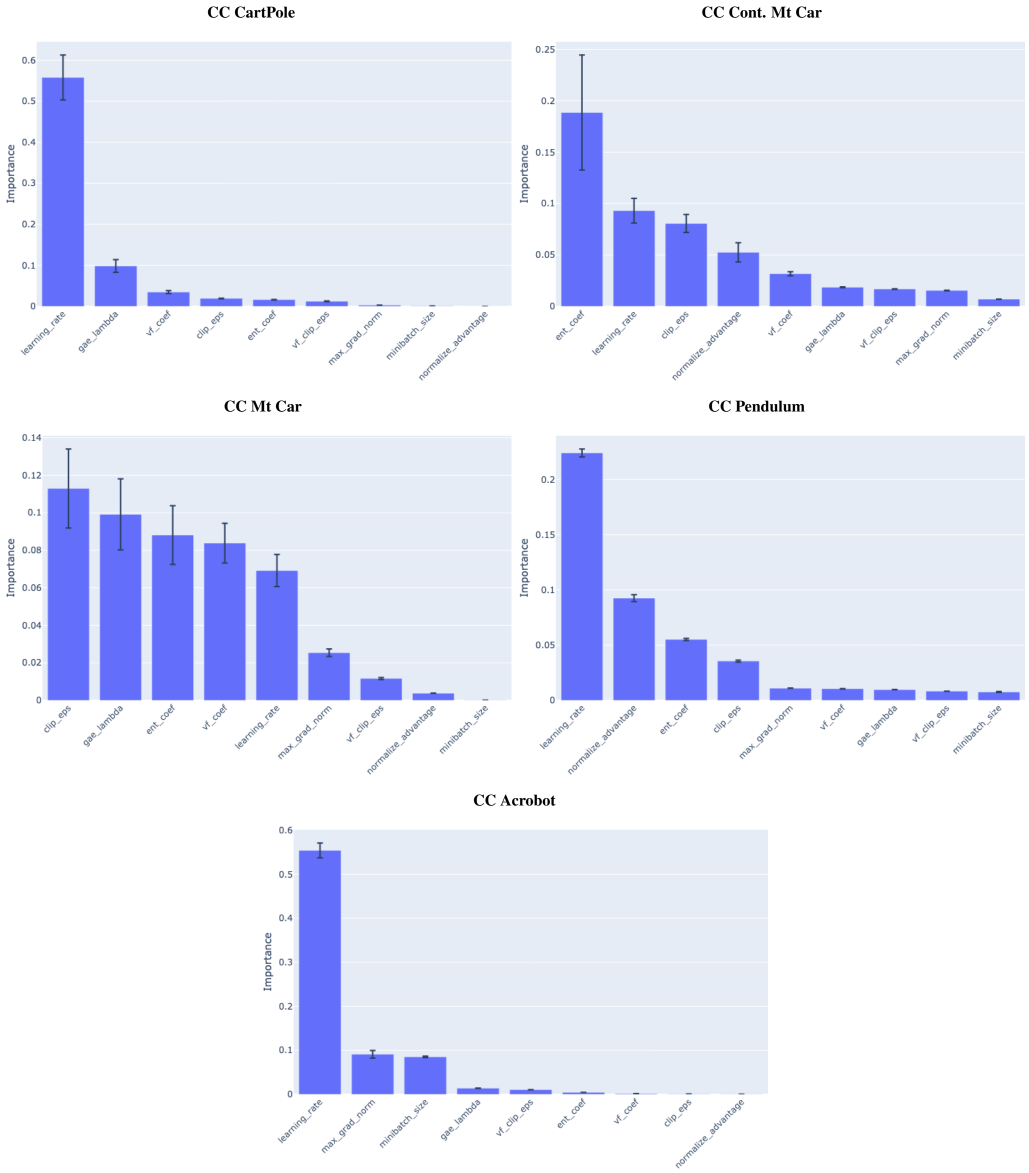


Figure 22: Hyperparameter importances for PPO: Brax and Box2D.


CC CartPole

CC Cont. Mt Car

CC Mt Car

CC Pendulum

CC Acrobot

Figure 23: Hyperparameter importances for PPO: Classic Control.

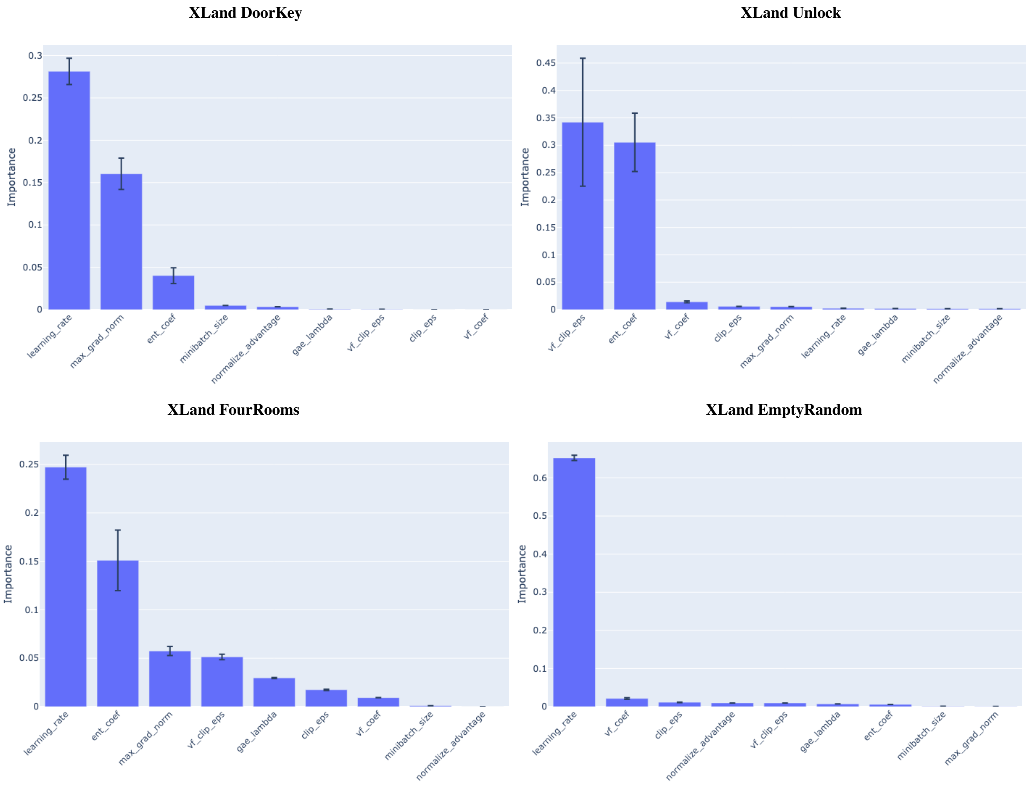


Figure 24: Hyperparameter importances for PPO: XLand.

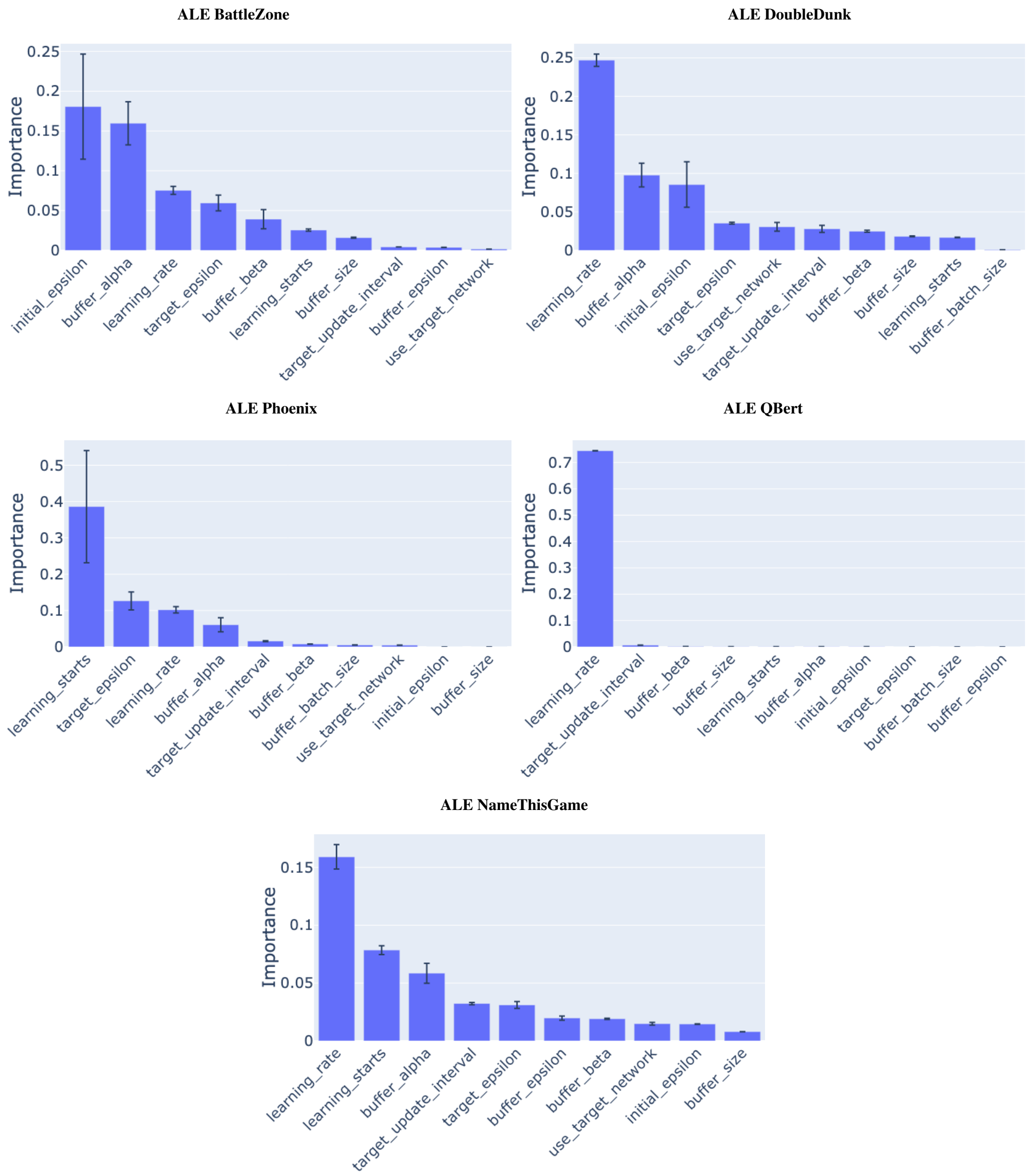


Figure 25: Hyperparameter importances for DQN: ALE.

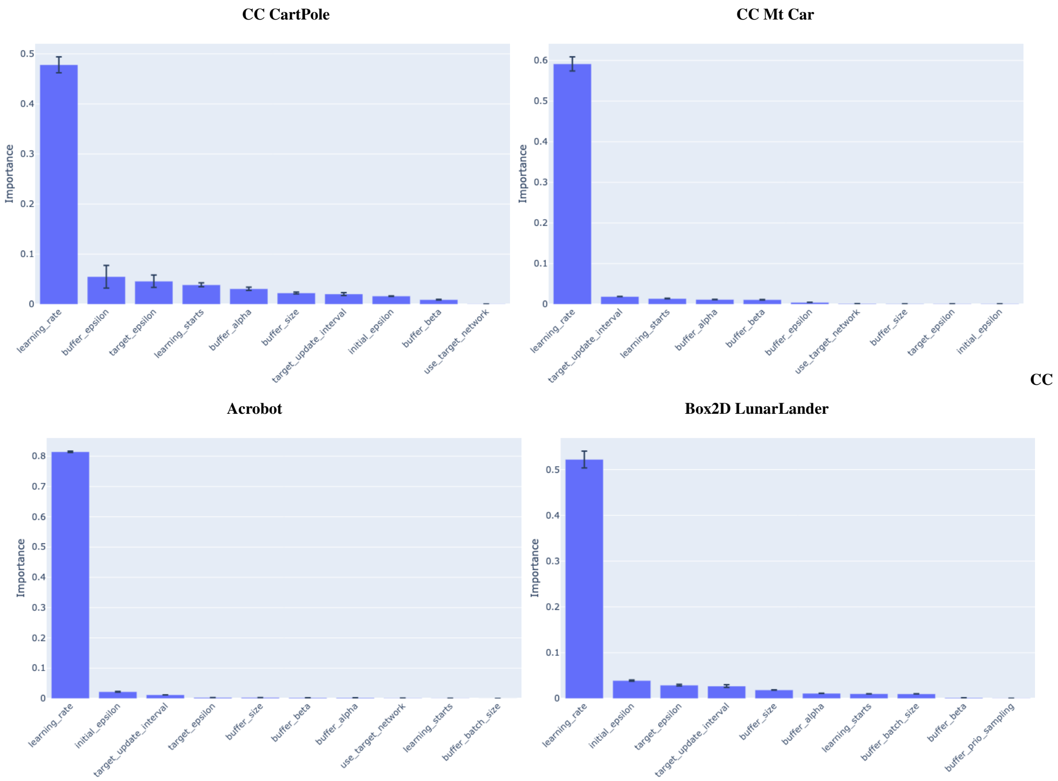


Figure 26: Hyperparameter importances for DQN: Classic Control and Box2D.

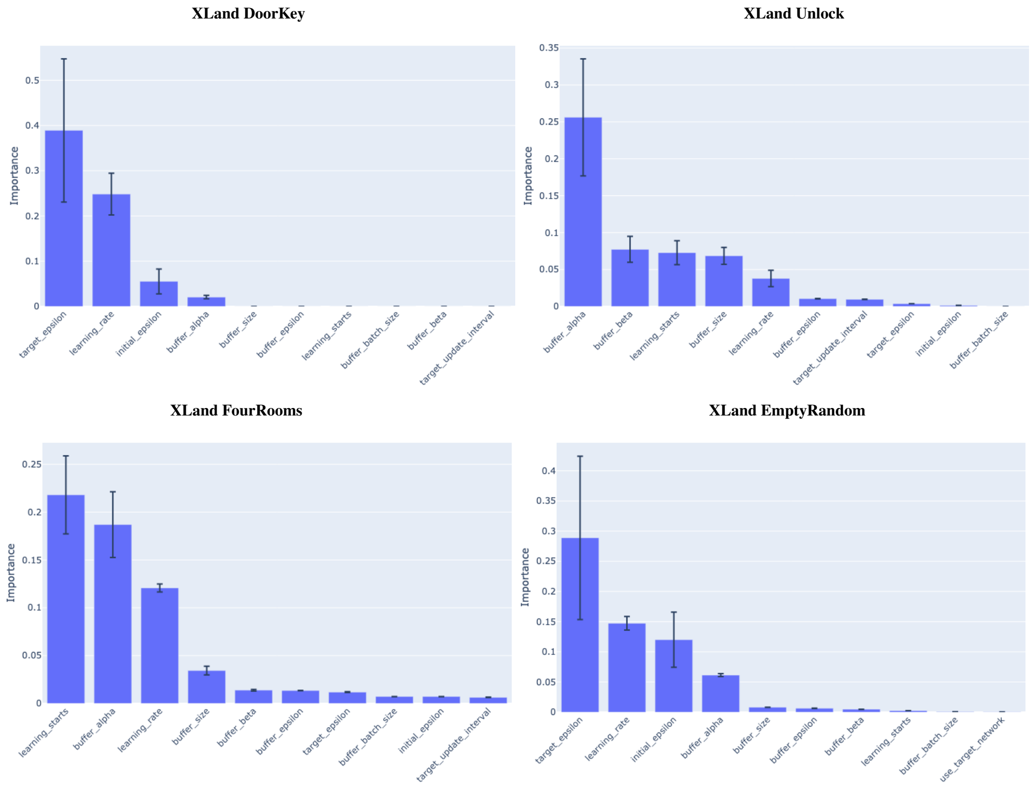


Figure 27: Hyperparameter importances for DQN: XLand.

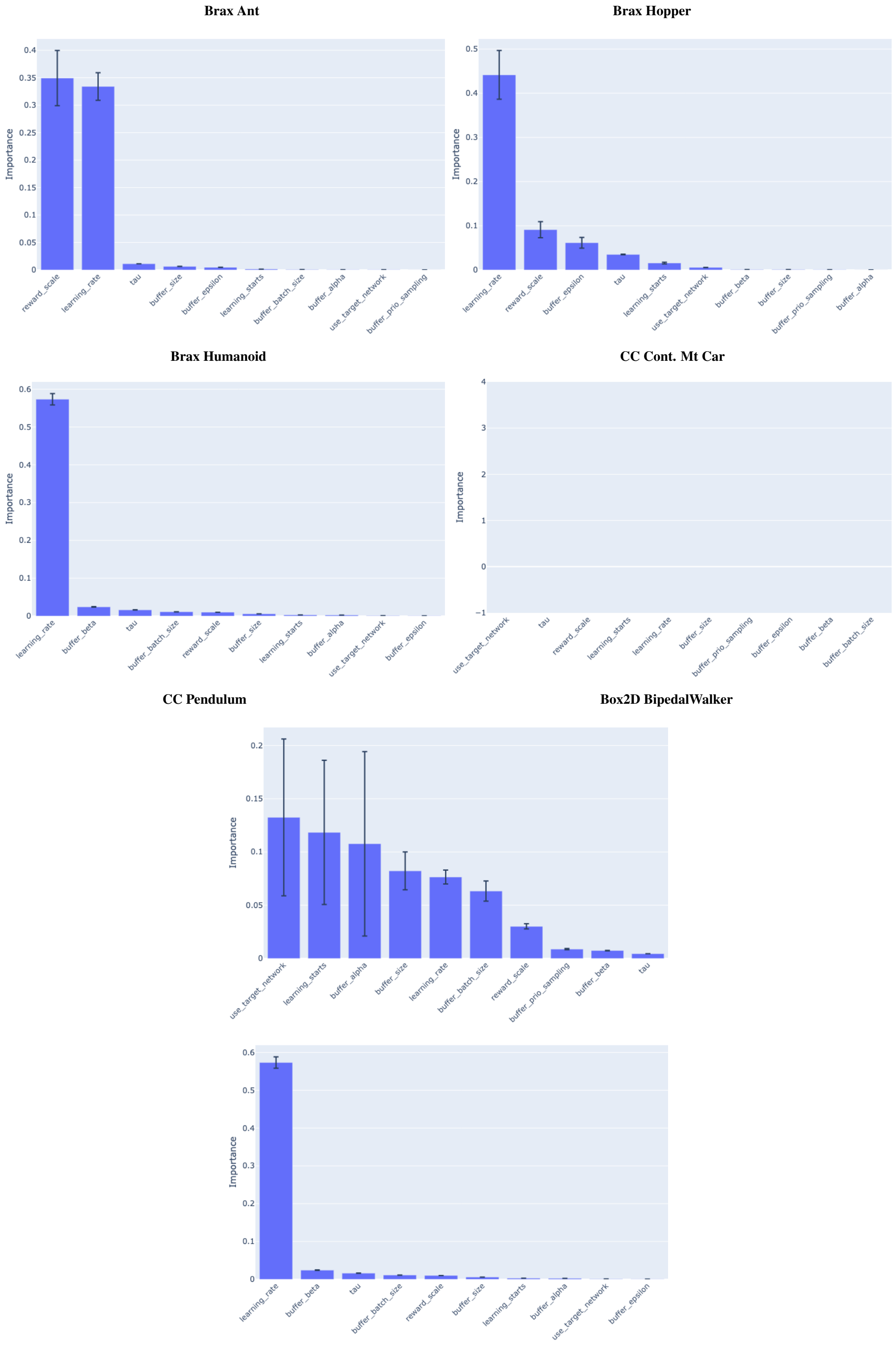


Figure 28: Hyperparameter importances for SAC.

H.4 Budget Correlations

We show the budget correlation plots for all budgets (Figures 29, 30, 31 and 32 for PPO, Figures 33, 34 and 35 for DQN and Figure 36 for SAC). We see that most correlations are strong or very strong with some numerical inconsistencies in mountaincar and brax halfcheetah. XLand is the the only domain with a strong trend over time, for DQN most correlations only become strong after about 30-40% of training while the same is true for DoorKey of PPO. These are consistent in the domains, though: we see low or no correlations for Brax (likely due to numerical issues) and in Classic Control, strong correlations otherwise. ALE and Box2D are other domains with strong correlations while XLand tends to need a warmup phase.

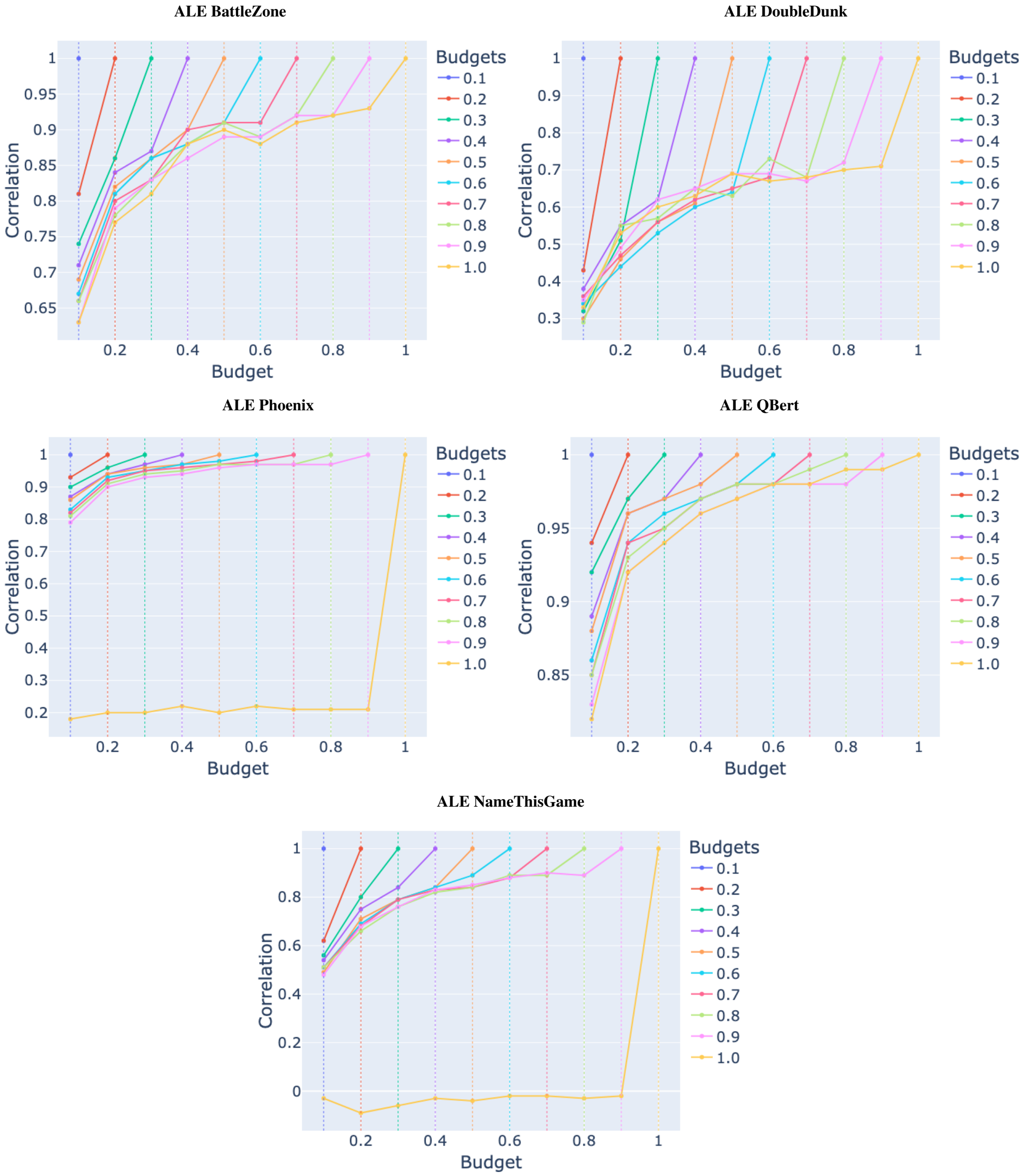


Figure 29: Budget correlations for PPO: ALE.

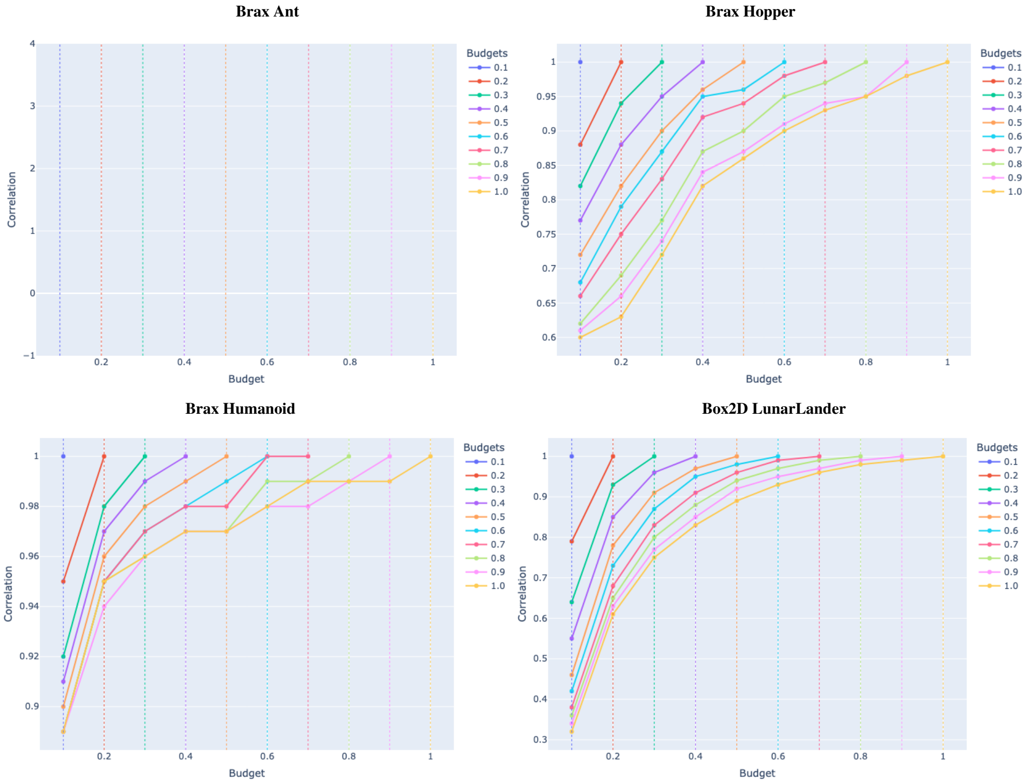
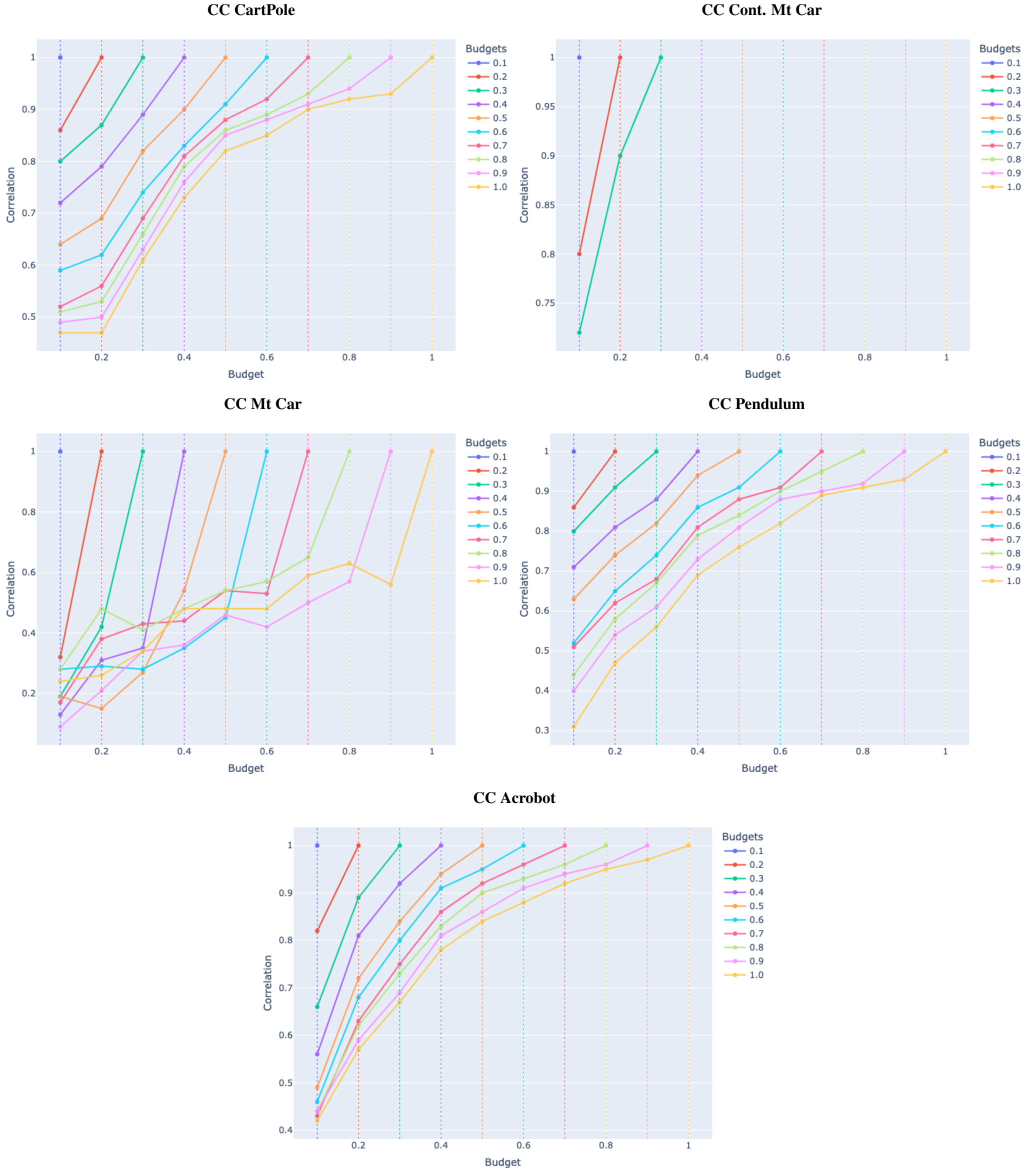


Figure 30: Budget correlations for PPO: Brax and Box2D.


CC CartPole

CC Cont. Mt Car

CC Mt Car

CC Pendulum

CC Acrobot

Figure 31: Budget correlations for PPO: Classic Control.

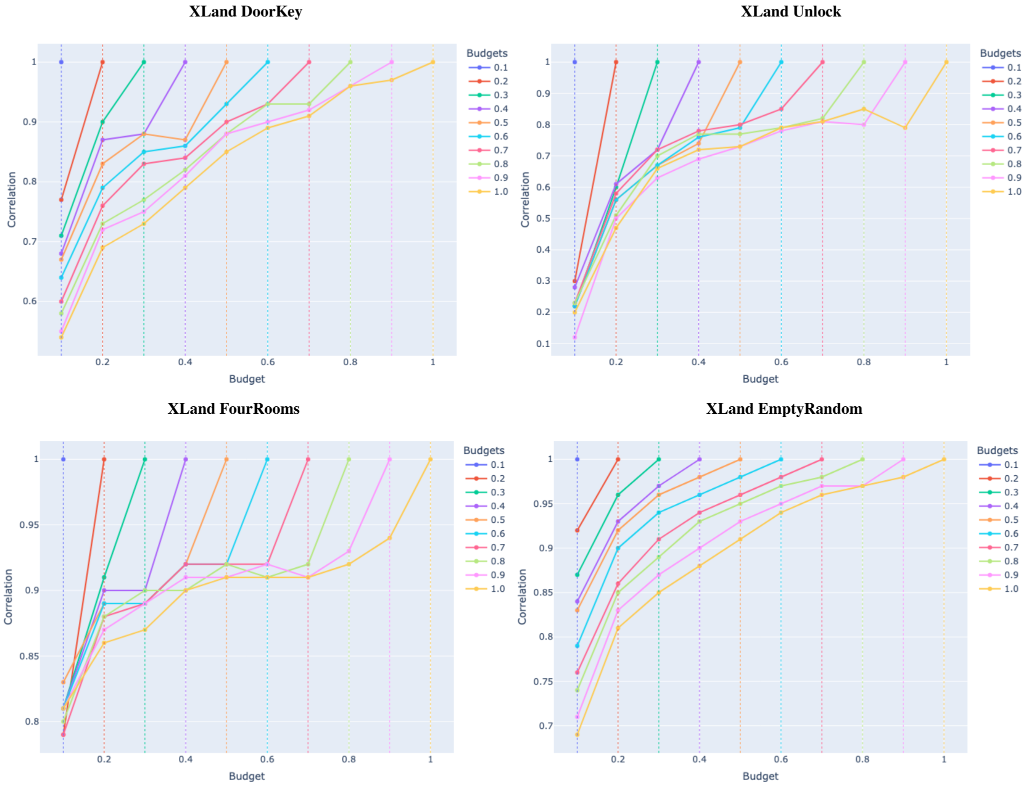


Figure 32: Budget correlations for PPO: XLand.

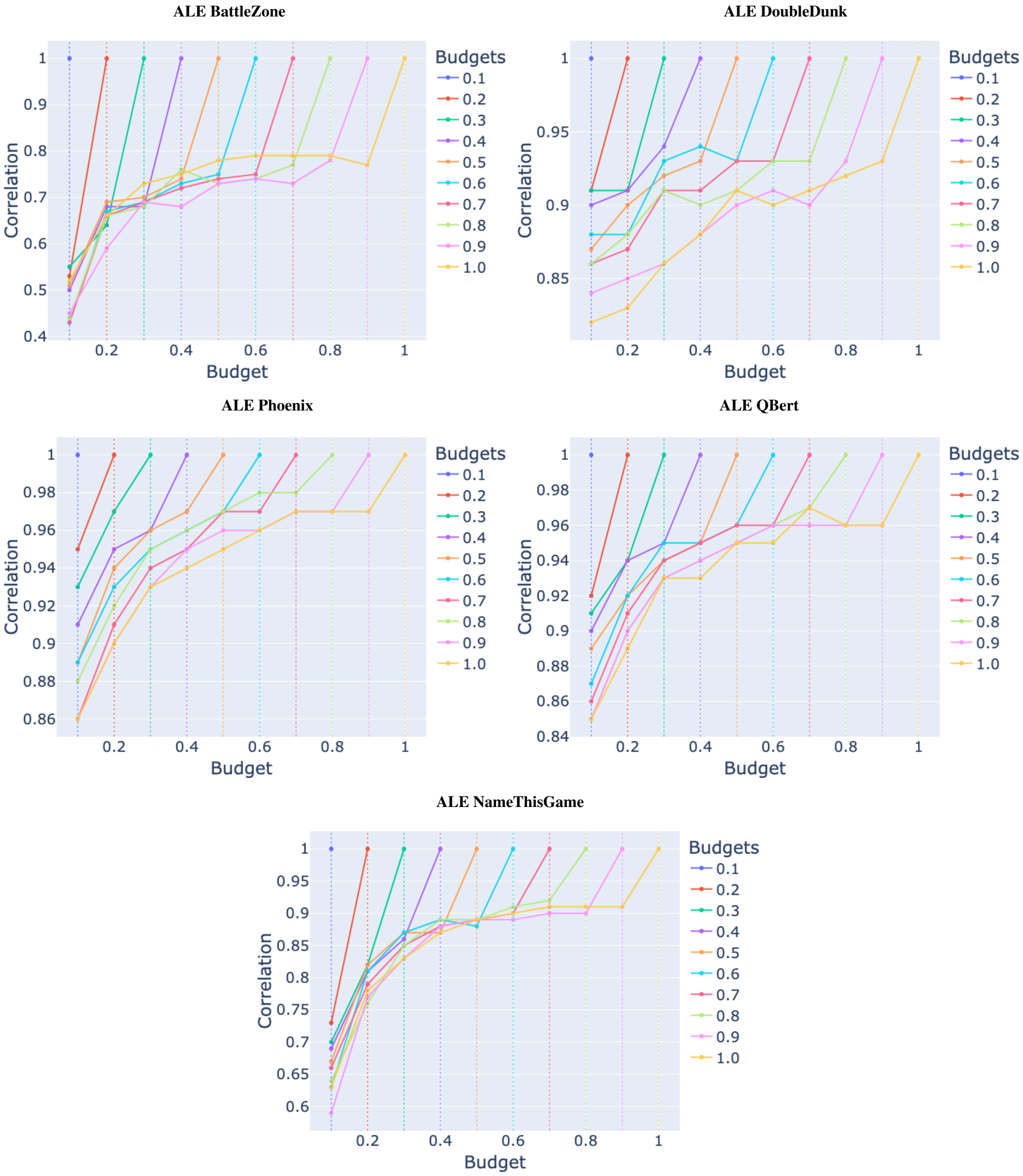


Figure 33: Budget correlations for DQN: ALE.

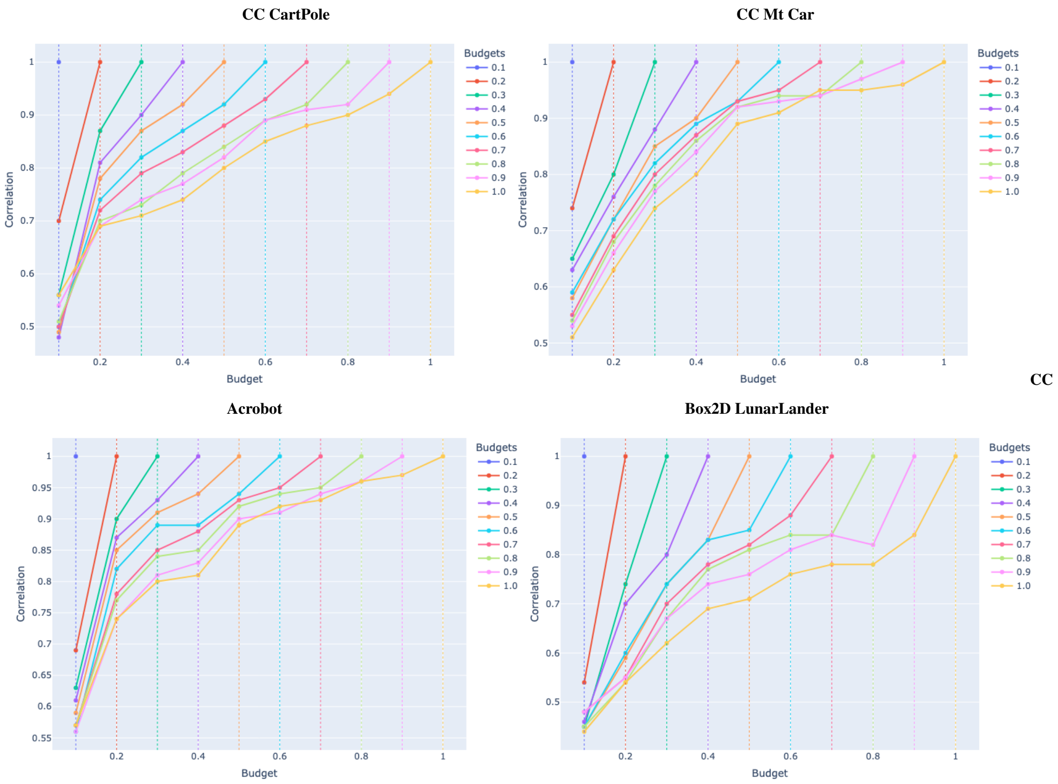


Figure 34: Budget correlations for DQN: Classic Control and Box2D.

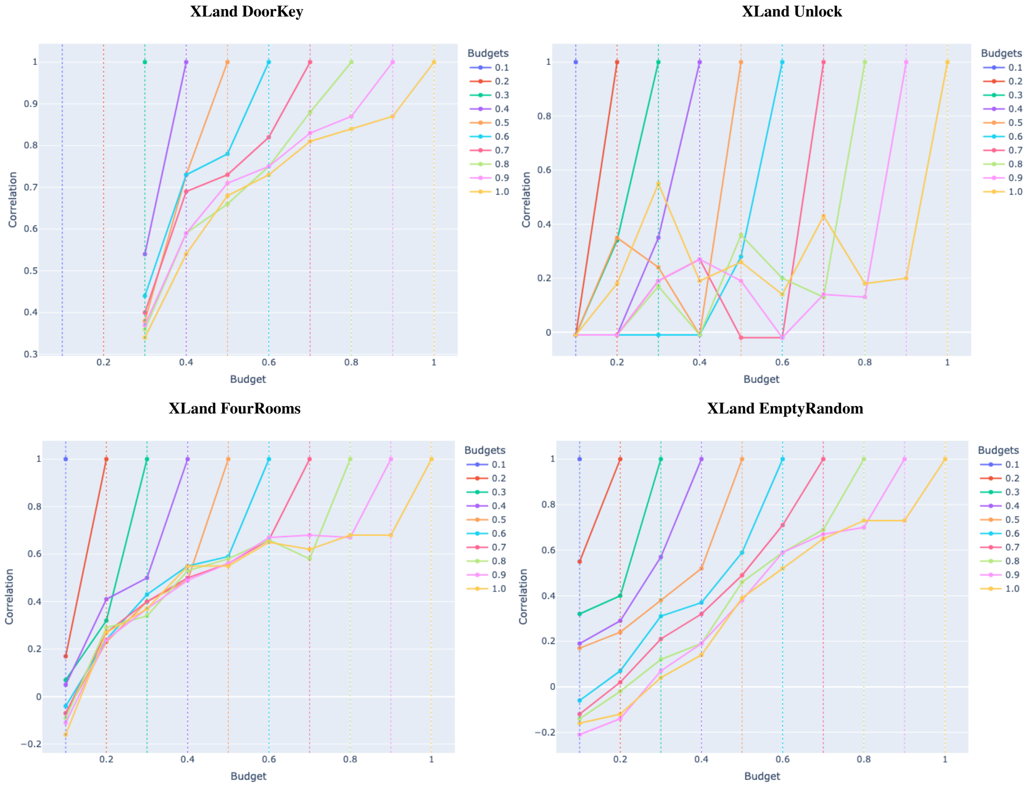


Figure 35: Budget correlations for DQN: XLand.

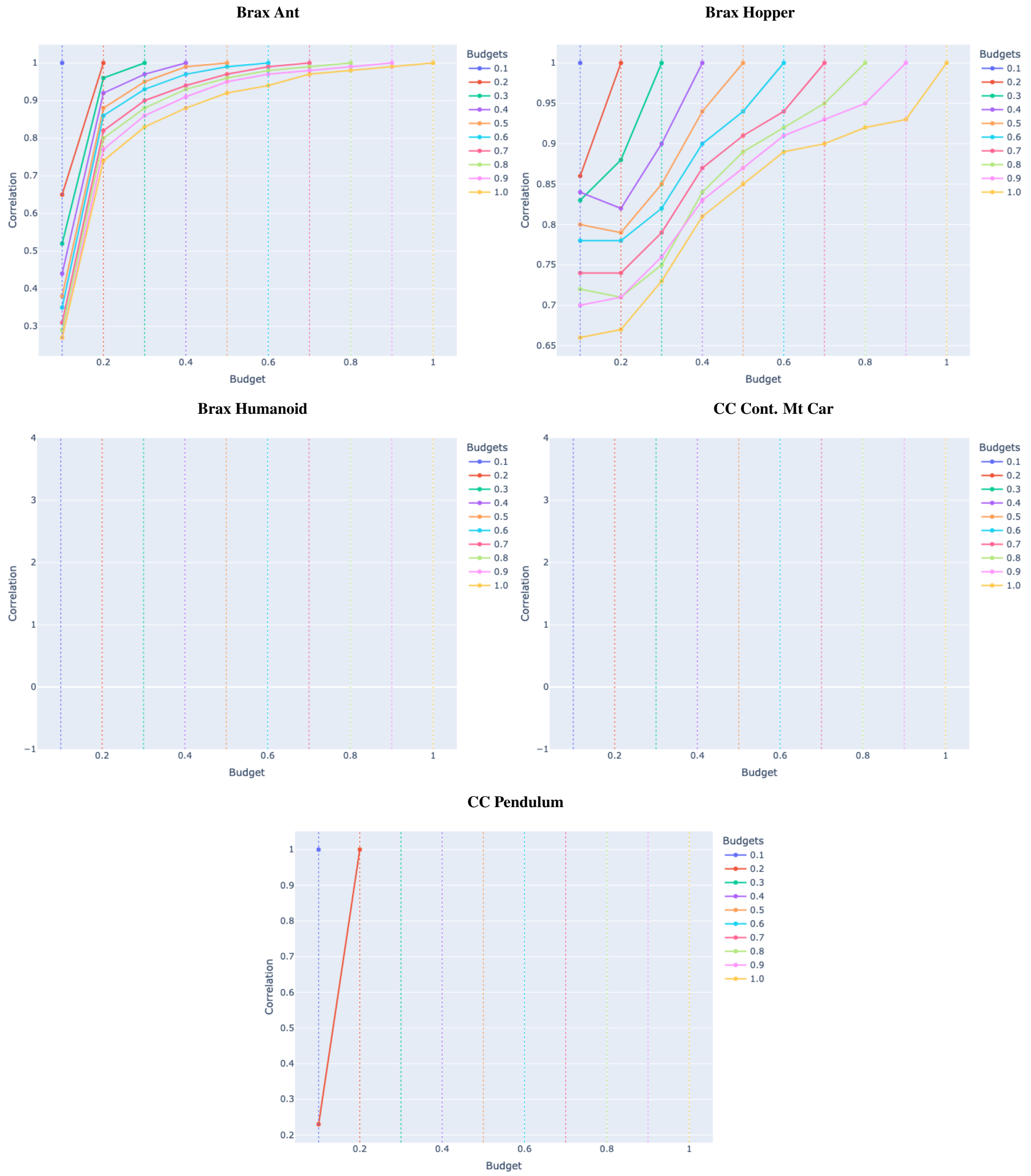


Figure 36: Budget correlations for SAC.

I Resource Consumption

All running time results are stated in Table 19 and were obtained using the same setup on the H100 cluster as described in Appendix B.1.

Algorithm	Environment	Platform	Running Time [s]	Total Running Time [h]
DQN	Acrobot-v1	CPU	26.1	37.13
DQN	BattleZone-v5	GPU	2967.69	4220.72
DQN	CartPole-v1	CPU	10.27	14.6
DQN	DoubleDunk-v5	GPU	2918.08	4150.16
DQN	LunarLander-v2	CPU	34.47	49.03
DQN	MiniGrid-DoorKey-5x5	CPU	81.44	115.82
DQN	MiniGrid-EmptyRandom-5x5	CPU	30.32	43.12
DQN	MiniGrid-FourRooms	CPU	172.31	245.07
DQN	MiniGrid-Unlock	CPU	94.68	134.65
DQN	MountainCar-v0	CPU	19.4	27.59
DQN	NameThisGame-v5	GPU	2970.15	4224.22
DQN	Phoenix-v5	GPU	2710.29	3854.64
DQN	Qbert-v5	CPU	2943.79	4186.73
PPO	Acrobot-v1	CPU	15.34	21.82
PPO	BattleZone-v5	GPU	1154.29	1641.66
PPO	BipedalWalker-v3	CPU	89.83	127.76
PPO	CartPole-v1	CPU	7.95	11.3
PPO	DoubleDunk-v5	GPU	1083.08	1540.38
PPO	LunarLander-v2	CPU	162.97	231.78
PPO	LunarLanderContinuous-v2	CPU	300.47	427.33
PPO	MiniGrid-DoorKey-5x5	CPU	81.23	115.52
PPO	MiniGrid-EmptyRandom-5x5	CPU	26.37	37.5
PPO	MiniGrid-FourRooms	CPU	179.84	255.77
PPO	MiniGrid-Unlock	CPU	112.33	159.76
PPO	MountainCar-v0	CPU	13.21	18.79
PPO	MountainCarContinuous-v0	CPU	7.68	10.93
PPO	NameThisGame-v5	GPU	1130.46	1607.76
PPO	Pendulum-v1	CPU	13.81	19.64
PPO	Phoenix-v5	GPU	955.17	1358.46
PPO	Qbert-v5	CPU	1145.07	1628.54
PPO	ant	GPU	220.87	314.13
PPO	halfcheetah	GPU	851.99	1211.73
PPO	hopper	GPU	458.43	651.98
PPO	humanoid	GPU	338.6	481.57
SAC	BipedalWalker-v3	CPU	486.32	691.66
SAC	LunarLanderContinuous-v2	CPU	381.22	542.17
SAC	MountainCarContinuous-v0	CPU	557.13	792.36
SAC	Pendulum-v1	CPU	111.76	158.95
SAC	ant	GPU	824.95	1173.26
SAC	halfcheetah	GPU	2194.59	3121.2
SAC	hopper	GPU	1263.28	1796.66
SAC	humanoid	GPU	871.83	1239.93

Table 19: Running times of algorithms and environments and respective platforms they were executed on. The column *Running Time* represents the duration, in seconds, of a single training session, while *Total Running Time* indicates the cumulative hours spent on all experiments conducted for a given environment.

Each experiment was run for 4096 total runs. This results in a total CPU running timer of $10\,105.34$ h and GPU running time of $32\,588.46$ h (including 40.54 h GPU hours for the running time experiments).

J Additional HPO Results

This section contains optimizer results per environment. In order to compare optimizer performance across environments, we apply normalization, as done in subset selection. However, to incorporate the distances between incumbents, we use explicit returns rather than ranks. In particular, we normalize using the minimum and maximum returns obtained during the initial landscaping experiments described in Section 4.1. This is done per environment, to allow for a comparison of optimizers across environments while preserving relative order and distances. This yields optimizer scores between 0 and 1 for our experiments, since no optimizer was able to find a better incumbent performance as encountered through 256 Sobol-sampled configurations. However, due to numerical instabilities in the Box2D and Brax environments, we fixed the minimum in these domains to -200 and -2000, respectively, to exclude artificially low values that distort the normalization.

Figures 37, 38, 39, 40, and 41 show results for PPO. Figures 42, 43, 42, and 44 show results for DQN. Figures 42, 45 and 46 show results for SAC.

When comparing the results shown in Figure 16, we can see that SMAC outperforms PBT and RS in terms of mean performance. However, if we would benchmark the optimizers on another, arbitrary selected set of environments, e.g. QBert-v5 (ALE), Phoenix-v5 (ALE), and MiniGrid-DoorKey-5x5 (XLand), the results would look very different from the observation on our selected subset. This, in fact, emphasizes the importance of the subset of environments on which a HPO method is evaluated. The differences show, that in order to reliable benchmark and compare HPO methods, they need to be evaluated on the same, representative subset of environments.

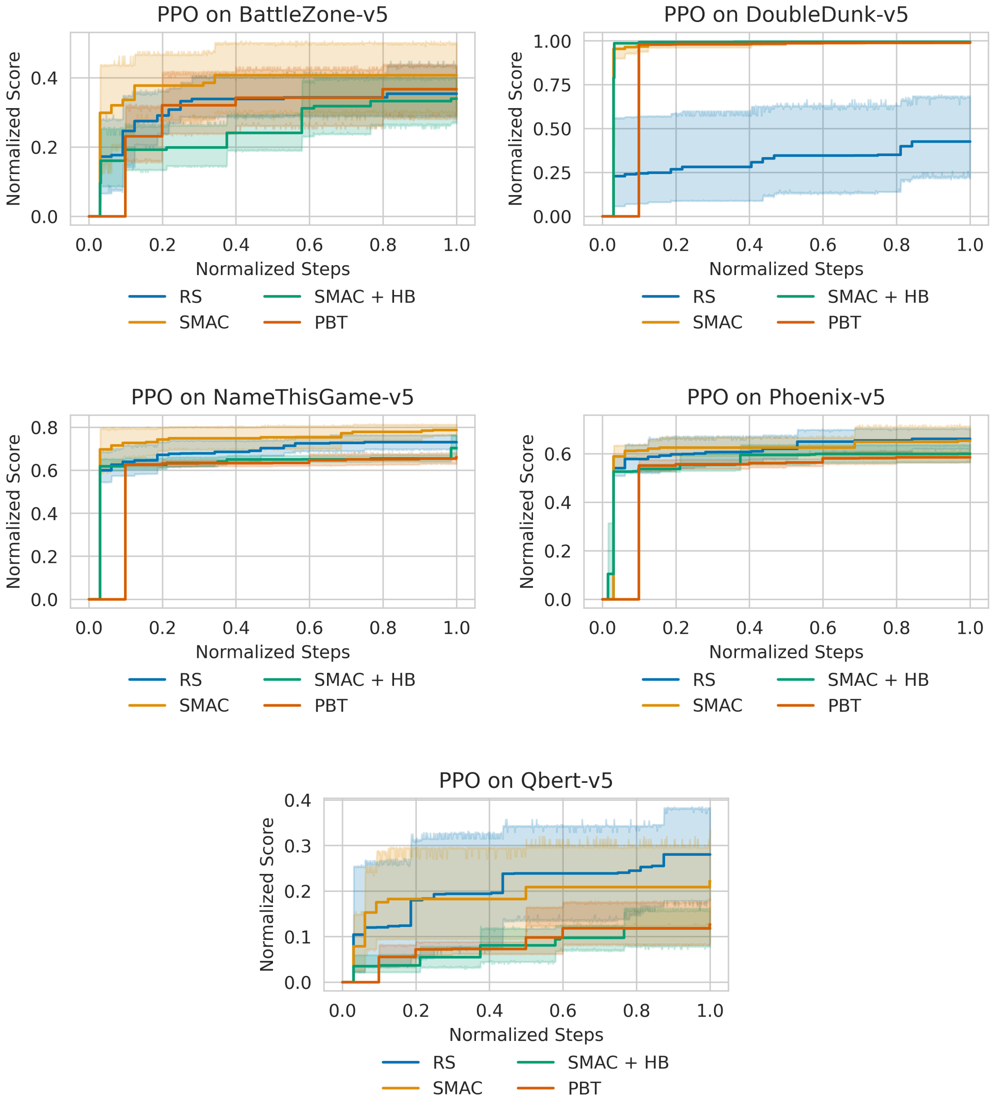


Figure 37: Anytime performance of optimizers for PPO: ALE. Figure shows 95%-confidence intervals.

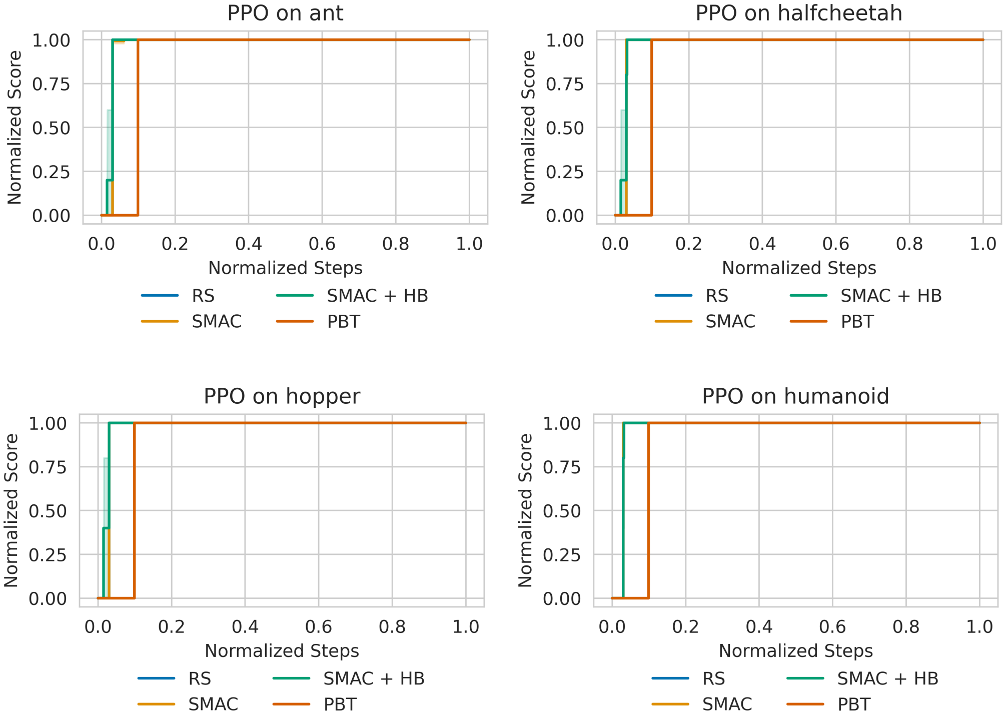


Figure 38: Anytime performance of optimizers for PPO: Brax. Figure shows 95%-confidence intervals.

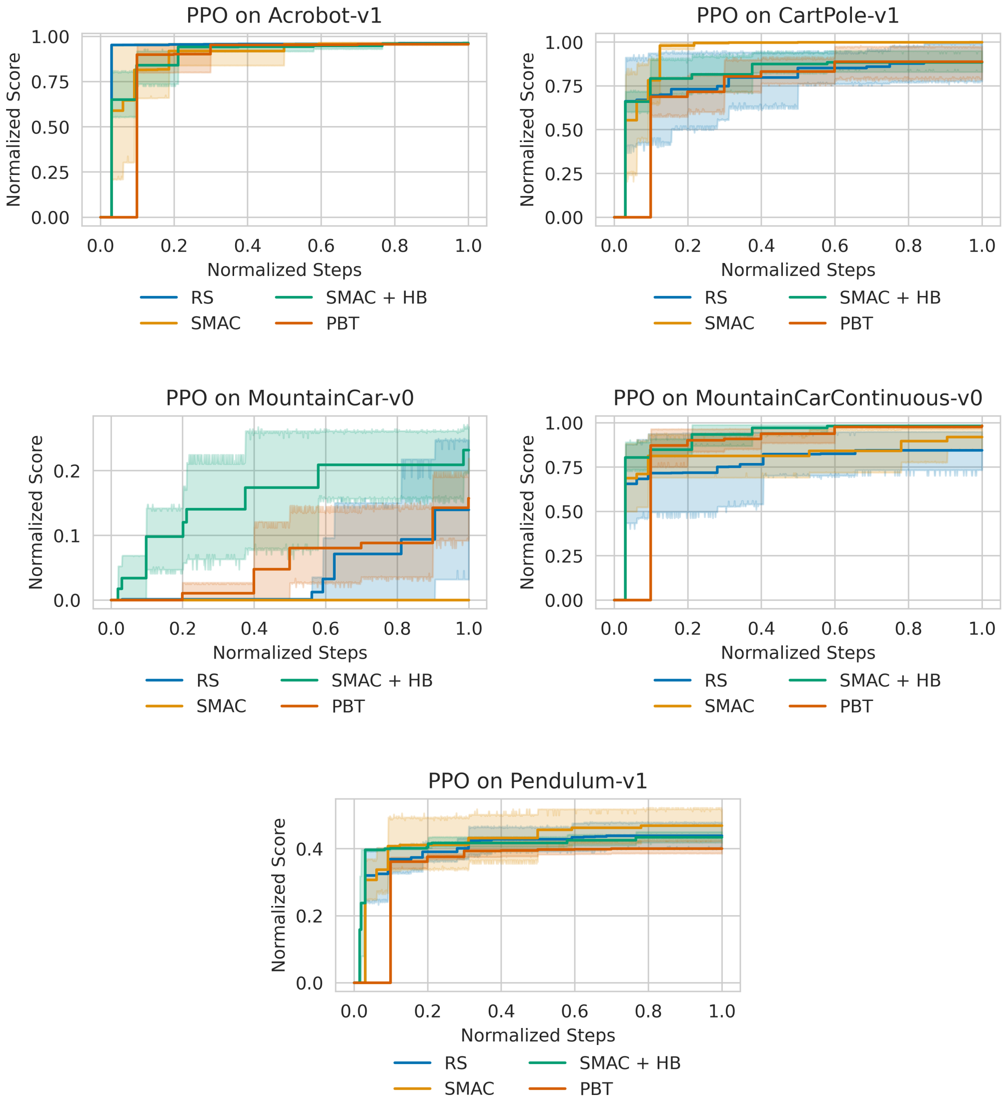


Figure 39: Anytime performance of optimizers for PPO: Classic Control. Figure shows 95%-confidence intervals.

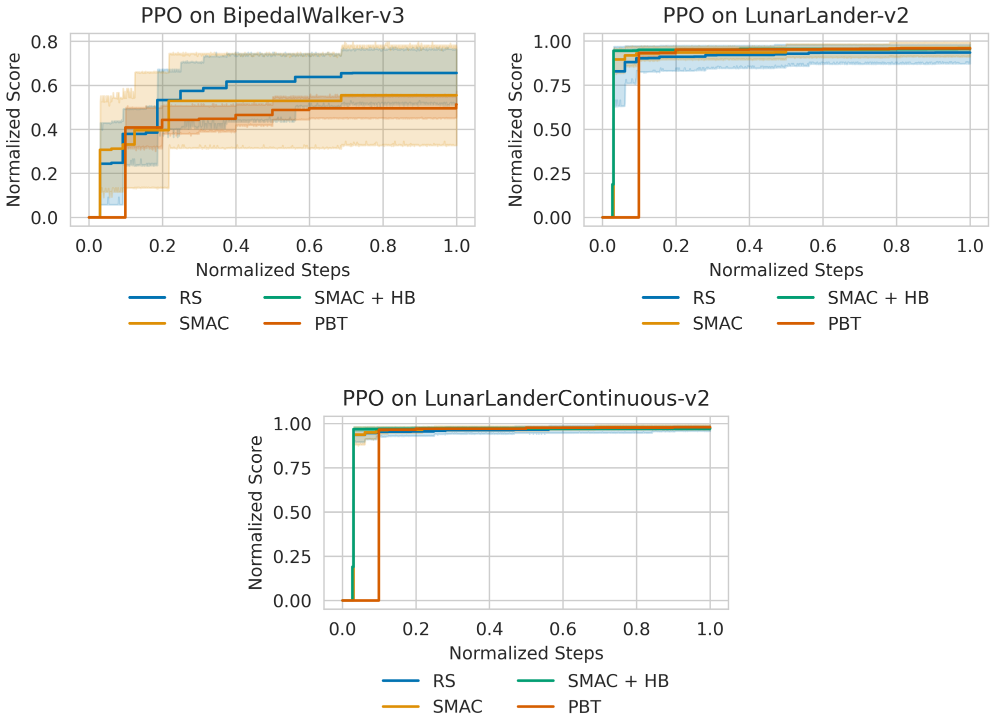


Figure 40: Anytime performance of optimizers for PPO: Box2D. Figure shows 95%-confidence intervals.

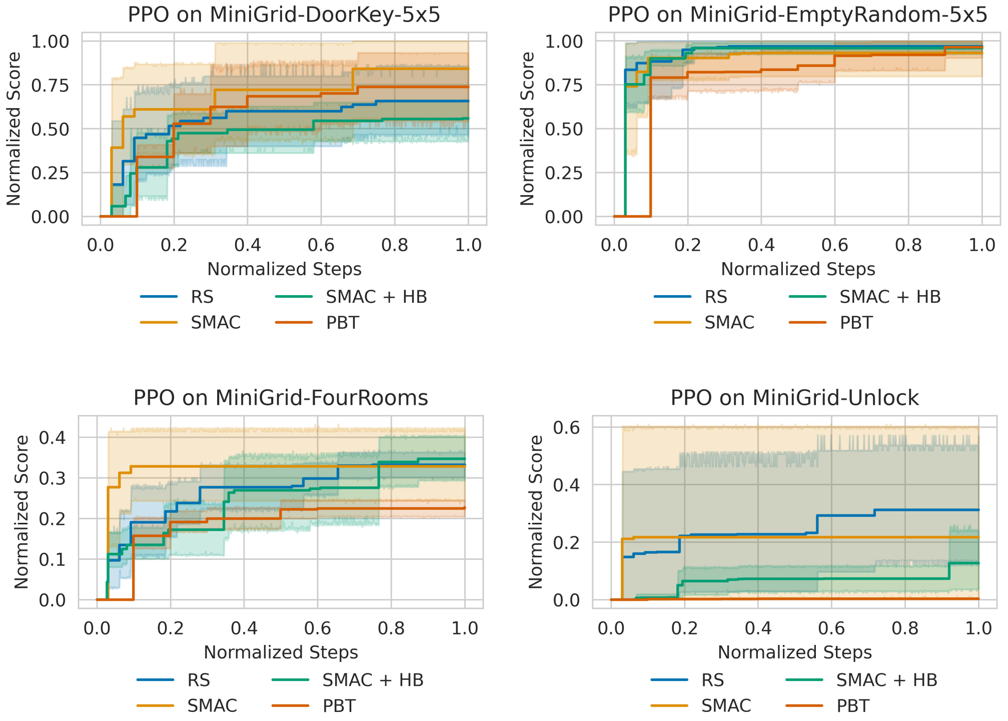


Figure 41: Anytime performance of optimizers for PPO: XLand. Figure shows 95%-confidence intervals.

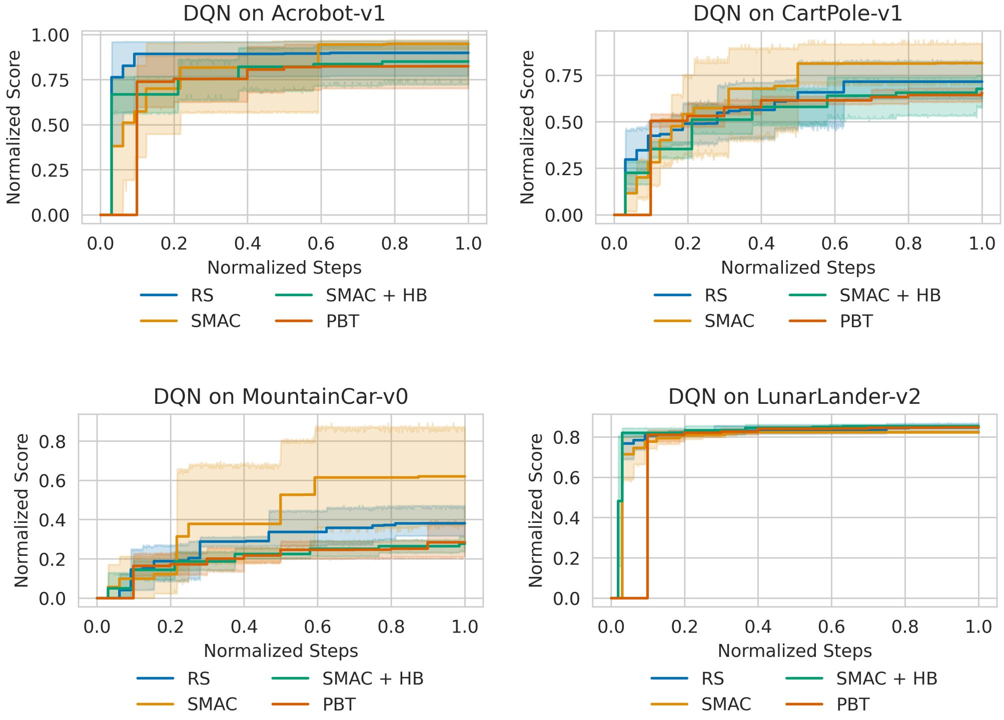


Figure 42: Anytime performance of optimizers for DQN: Classic Control and Box2D. Figure shows 95%-confidence intervals.

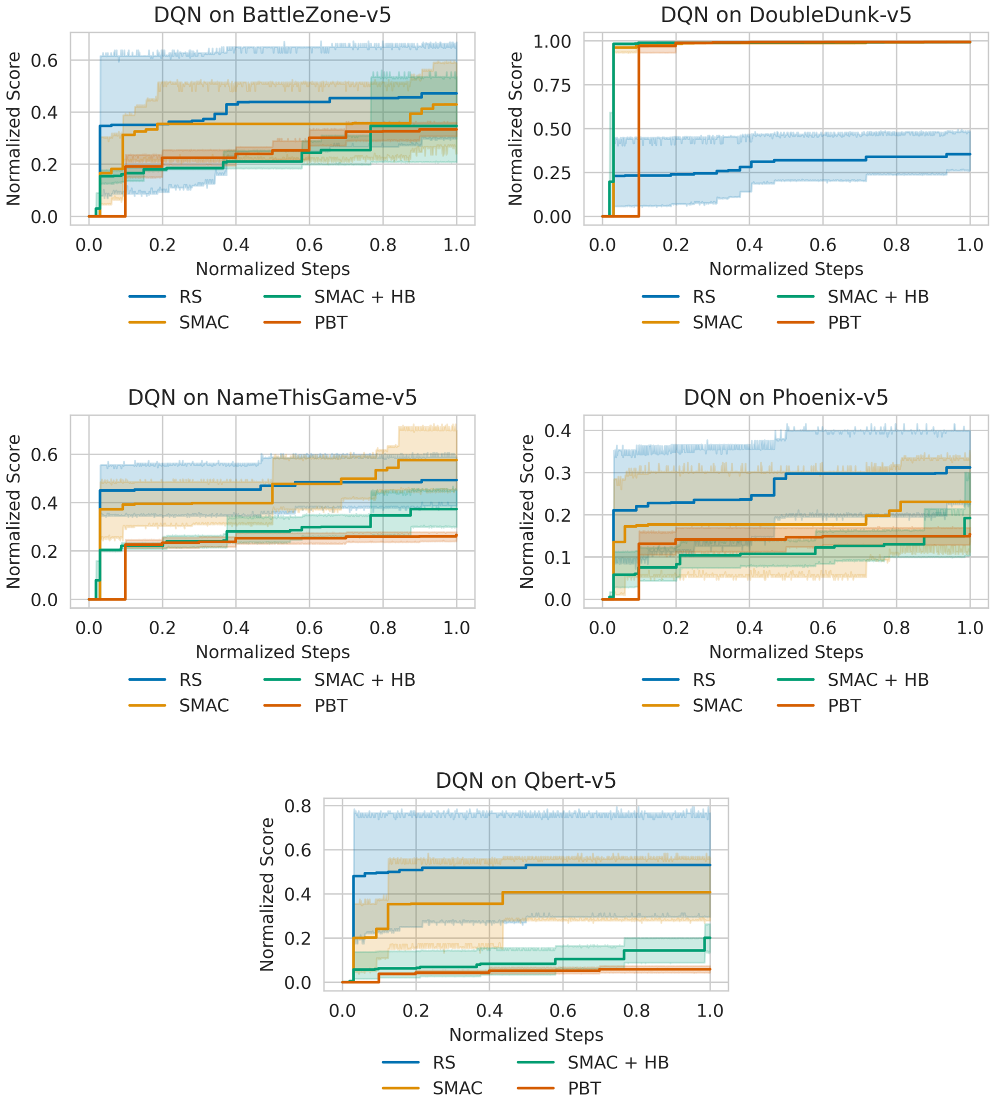


Figure 43: Anytime performance of optimizers for DQN: ALE. Figure shows 95%-confidence intervals.

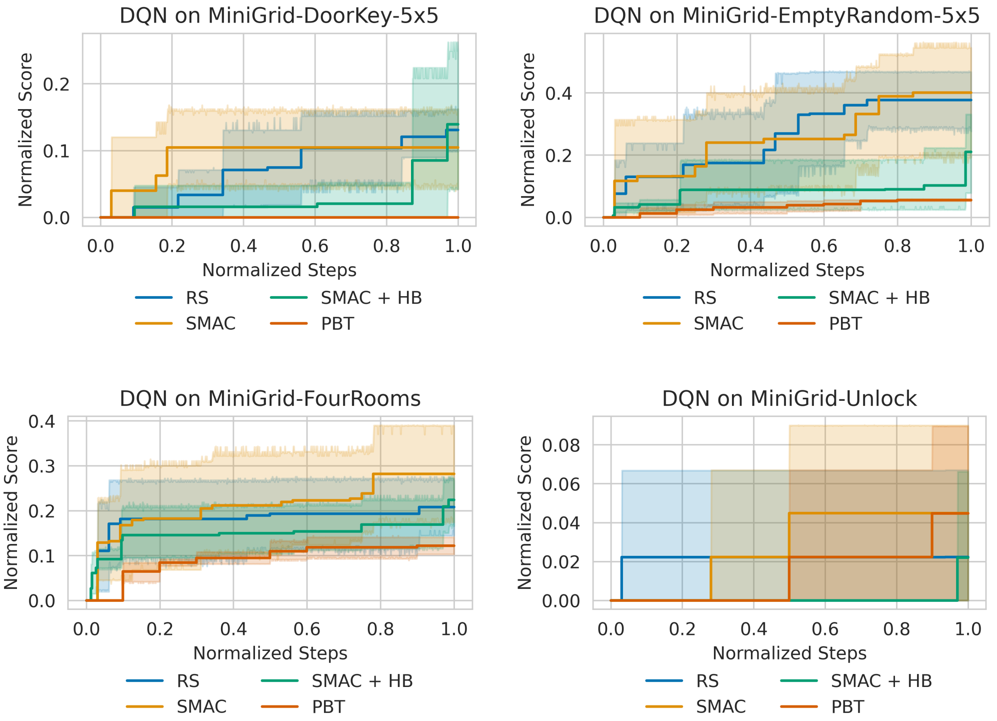


Figure 44: Anytime performance of optimizers for DQN: XLand. Figure shows 95%-confidence intervals.

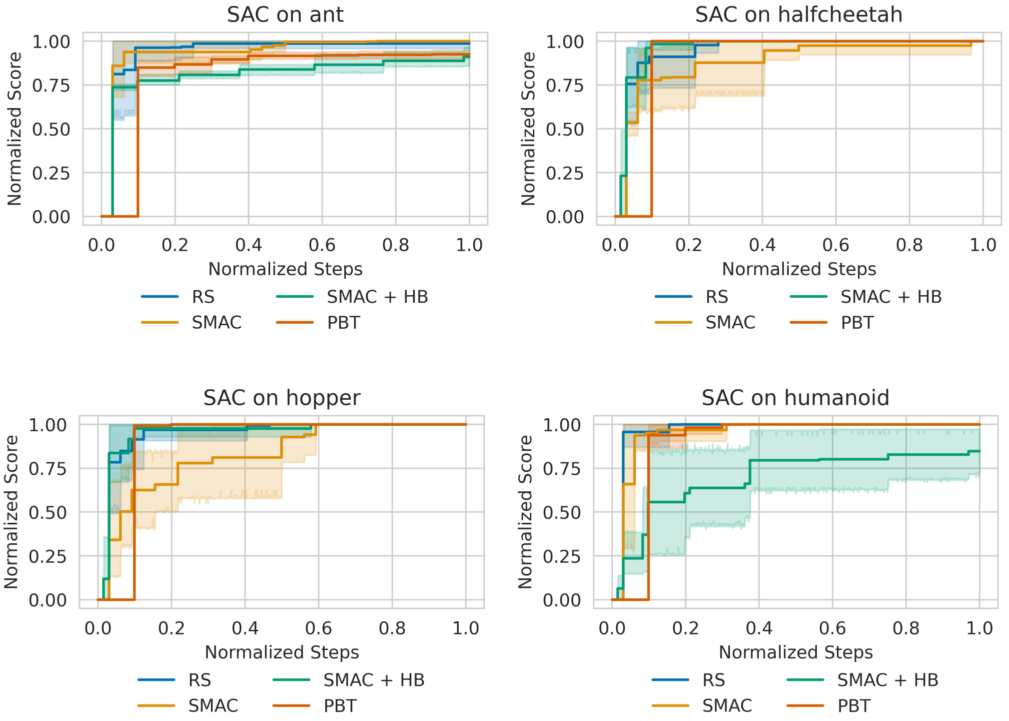


Figure 45: Anytime performance of optimizers for SAC: Brax. Figure shows 95%-confidence intervals.

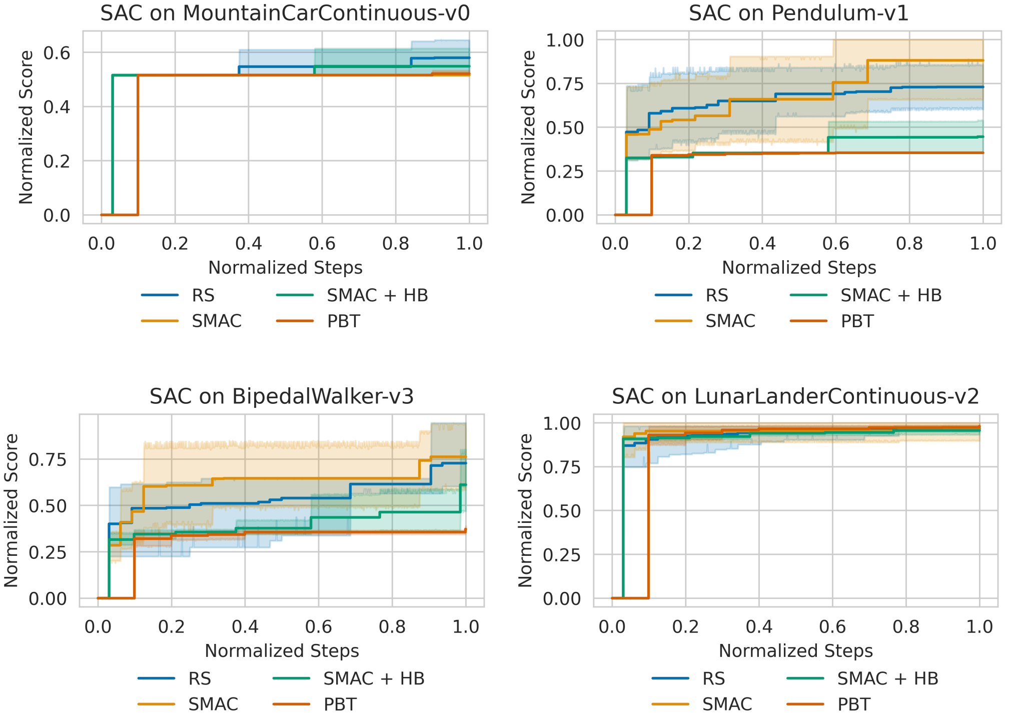


Figure 46: Anytime performance of optimizers for SAC: Classic Control and Box2D. Figure shows 95%-confidence intervals.