Hyperparameters in Reinforcement Learning and How To Tune Them

Theresa Eimer [1 *] **Marius Lindauer** [1] **Roberta Raileanu** [2]

Abstract

In order to improve reproducibility, deep reinforcement learning (RL) has been adopting better scientific practices such as standardized evaluation metrics and reporting. However, the process of hyperparameter optimization still varies widely across papers, which makes it challenging to compare RL algorithms fairly. In this paper, we show that hyperparameter choices in RL can significantly affect the agent's final performance and sample efficiency, and that the hyperparameter landscape can strongly depend on the tuning seed which may lead to overfitting. We therefore propose adopting established best practices from AutoML, such as the separation of tuning and testing seeds, as well as principled hyperparameter optimization (HPO) across a broad search space. We support this by comparing multiple state-of-the-art HPO tools on a range of RL algorithms and environments to their hand-tuned counterparts, demonstrating that HPO approaches often have higher performance and lower compute overhead. As a result of our findings, we recommend a set of best practices for the RL community, which should result in stronger empirical results with fewer computational costs, better reproducibility, and thus faster progress. In order to encourage the adoption of these practices, we provide plug-and-play implementations of the tuning algorithms used in this paper at https://github.com/facebookresearch/how-to-autorl.

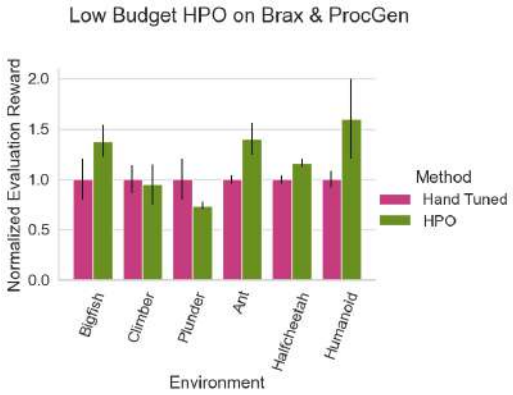


Figure 1: **Comparison of Hyperparameter Tuning Approaches:** state-of-the-art hyperparameter optimization packages match or outperform hand tuning via grid search, while using less than 1/12 of the budget.

1. Introduction

Deep reinforcement Learning (RL) algorithms contain a number of design decisions and hyperparameter settings, many of which have a critical influence on the learning speed and success of the algorithm. While design decisions and implementation details have received greater attention in the last years (Henderson et al., 2018; Engstrom et al., 2020; Hsu et al., 2020; Andrychowicz et al., 2021; Obando-Ceron & Castro, 2021), the same is less true of RL hyperparameters. Progress in self-adapting algorithms (Zahavy et al., 2020), RL-specific hyperparameter optimization tools (Franke et al., 2020; Wan et al., 2022), and meta-learnt hyperparameters (Flennerhag et al., 2022) has not yet been adopted by RL practitioners. In fact, most papers only report final model hyperparameters or grid search sweeps known to be suboptimal and costly compared to even simple Hyperparameter Optimization (HPO) baselines like random search (Bergstra & Bengio, 2012). In addition, the seeds used for tuning and evaluation are rarely reported, leaving it unclear if the hyperparameters were tuned on the test seeds, which is – as we will show – a major reproducibility issue. In this paper, we aim to lay out and address the potential causes for the lack of adoption of HPO methods in the RL community.

Underestimation of Hyperparameter Influence While it has been previously shown that hyperparameters are important to an RL algorithm's success (Henderson et al., 2018; Engstrom et al., 2020; Andrychowicz et al., 2021), the impact of even seemingly irrelevant hyperparameters is still underestimated by the community, as indicated by the fact that many papers tune only two or three hyperparameters (Schulman et al., 2017; Berner et al., 2019; Badia et al., 2020;

[*]Work was done during an internship at Meta AI. [1]Leibniz University Hannover [2]Meta AI. Correspondence to: Theresa Eimer <t.eimer@ai.uni-hannover.de>.

Proceedings of the 40^{th} International Conference on Machine Learning, Honolulu, Hawaii, USA. PMLR 202, 2023.

Hambro et al., 2022). We show that even often overlooked hyperparameters can make or break an algorithm's success, meaning that careful consideration is necessary for a broad range of hyperparameters. This is especially important for as-of-yet unexplored domains, as pointed out by Zhang et al. (2021a). Furthermore, hyperparameters cause different algorithm behaviors depending on the random seed which is a well-known fact in AutoML (Eggensperger et al., 2019; Lindauer & Hutter, 2020) but has not yet factored widely into RL research, negatively impacting reproducibility.

Fractured State of the Art in AutoRL Even though HPO approaches have succeeded in tuning RL algorithms (Franke et al., 2021; Awad et al., 2021; Zhang et al., 2021a; Wan et al., 2022), the costs and benefits of HPO are relatively unknown in the community. AutoRL papers often compare only a few HPO methods, are limited to single domains or toy problems, or use a single RL algorithm (Jaderberg et al., 2017; Parker-Holder et al., 2020; Awad et al., 2021; Kiran & Ozyildirim, 2022; Wan et al., 2022). In this work, we aim to understand the need for and challenges of AutoRL by comparing multiple HPO methods across various state-of-the-art RL algorithms on challenging environments. Our results demonstrate that HPO approaches have better performance and less compute overhead than hyperparameter sweeps or grid searches which are typically used in the RL community (see Figure 1).

Ease of Use State-of-the-art AutoML tools are often released as research papers rather than standalone packages. In addition, they are not immediately compatible with standard RL code, while easy to use solutions like Optuna (Akiba et al., 2019) or Ax (Bakshy et al., 2018) only provide a limited selection of HPO aproaches. To improve the availability of these tools, we provide Hydra sweepers (Yadan, 2019) for several variations of population-based methods, such as standard PBT (Jaderberg et al., 2017), PB2 (Parker-Holder et al., 2020) and BGT (Wan et al., 2022), as well as the evolutionary algorithm DEHB (Awad et al., 2021). Note that all of these have shown to improve over random search for tuning RL agents. As black-box methods, they are compatible with any RL algorithm or environment and due to Hydra, users do not have to change their implementation besides returning a success metric like the reward once training is finished. Based on our empirical insights, we provide best practice guidelines on how to use HPO for RL.

In this paper, we demonstrate that **compared to tuning hyperparameters by hand, existing HPO tools are capable of producing better performing, more stable, and more easily comparable RL agents, while using fewer computational resources**. We believe widespread adoption of HPO protocols within the RL community will therefore result in more accurate and fair comparisons across RL methods and in the end to faster progress.

To summarize, **our contributions** are:

1. Exploration of the hyperparameter landscape for commonly-used RL algorithms and environments;
2. Comparison of different types of HPO methods on state-of-the-art RL algorithms and challenging RL environments;
3. Open-source implementations of advanced HPO methods that can easily be used with any RL algorithm and environment; and
4. Best practice recommendations for HPO in RL.

2. The Hyperparameter Optimization Problem

We provide an overview of the most relevant formalizations of HPO in RL, Algorithm Configuration (Schede et al., 2022) and Dynamic Algorithm Configuration (Adriaensen et al., 2022). Algorithm Configuration (AC) is a popular paradigm for optimizing hyperparameters of several different kinds of algorithms (Eggensperger et al., 2019).

Definition 2.1 (AC)**.** Given an algorithm A, a hyperparameter space $\mathbf{\Lambda}$, as well as a distribution of environments or environment instances $\mathcal{I}$, and a cost function c, find the optimal configuration $\boldsymbol{\lambda}^* \in \mathbf{\Lambda}$ across possible tasks s.t.:

$$\boldsymbol{\lambda}^* \in \arg\min_{\boldsymbol{\lambda} \in \mathbf{\Lambda}} \mathbb{E}_{i \sim \mathcal{I}} \left[c(A(i; \boldsymbol{\lambda})) \right].$$

The cost function could be the negative of the agent's reward or a failure indicator across a distribution of tasks. Thus it is quite flexible and can accommodate a diverse set of possible goals for algorithm performance. This definition is not restricted to one train and test setting but aims to achieve the best possible performance across a range of environments or environment instances. AC approaches thus strive to avoid overfitting the hyperparameters to a specific scenario. Even for RL problems focusing on generalization, AC is therefore a suitable framework. Commonly, the HPO process is terminated before we have found the true $\boldsymbol{\lambda}^*$ via an optimization budget (e.g. the runtime or number of training steps). The best found hyperparameter configuration found by the optimization process is called the incumbent.

Another relevant paradigm for tuning RL is Dynamic Algorithm Configuration (DAC) (Biedenkapp et al., 2020; Adriaensen et al., 2022). DAC is a generalization of AC that does not search for a single optimal hyperparameter value per algorithm run but instead for a sequence of values.

Definition 2.2 (DAC)**.** Given an algorithm A, a hyperparameter space $\mathbf{\Lambda}$ as well as a distribution of environments or environment instances $\mathcal{I}$ with state space $\mathcal{S}$, cost function c and a space of dynamic configuration policies $\mathbf{\Pi}$ with each $\pi \in \mathbf{\Pi} : \mathcal{S} \times \mathcal{I} \to \mathbf{\Lambda}$, find $\pi^* \in \mathbf{\Pi}$ s.t.:

$$\pi^* \in \arg\min_{\pi \in \mathbf{\Pi}} \mathbb{E}_{i \sim \mathcal{I}} \, c(A(i; \pi))$$

As RL is a dynamic optimization process, it can benefit from dynamic changes in the hyperparameter values such as learning rate schedules (Zhang et al., 2021a; Parker-Holder et al., 2022). Thus HPO tools developed specifically for RL have been following the DAC paradigm in order to tailor the hyperparameter values closer to the training progress (Franke et al., 2020; Zhang et al., 2021a; Wan et al., 2022).

It is worth noting that while the model architecture can be defined by a set of hyperparameters like the number of layers, architecture search is generally more complex and thus separately defined as the NAS problem or combined with HPO to form the general AutoDL problem (Zimmer et al., 2021). While some tools include options for optimizing architecture hyperparameters, insights into how to find good architectures for RL are out of scope for this paper.

3. Related Work

While RL as a field has seen many innovations in the last years, small changes to the algorithm or its implementation can have a big impact on its results (Henderson et al., 2018; Andrychowicz et al., 2021; Engstrom et al., 2020). In an effort to consolidate these innovations, several papers have examined the effect of smaller design decisions like the loss function or policy regularization for on-policy algorithms (Hsu et al., 2020; Andrychowicz et al., 2021), DQN (Obando-Ceron & Castro, 2021) and offline RL (Zhang & Jiang, 2021). AutoRL methods, on the other hand, have focused on automating and abstracting some of these decisions (Parker-Holder et al., 2022) by using data-driven approaches to learn various algorithmic components (Bechtle et al., 2020; Xu et al., 2020; Metz et al., 2022) or even entire RL algorithms (Wang et al., 2016; Duan et al., 2016; Co-Reyes et al., 2021; Lu et al., 2022).

While overall there has been less interest in hyperparameter optimization, some RL-specific HPO algorithms have been developed. STACX (Zahavy et al., 2020) is an example of a self-tuning algorithm, using meta-gradients (Xu et al., 2018) to optimize its hyperparameters during runtime. This idea has recently been generalized to bootstrapped meta-learning, enabling the use of meta-gradients to learn any combination of hyperparameters on most RL algorithms on the fly (Flennerhag et al., 2022). Such gradient-based approaches are fairly general and have shown a lot of promise (Paul et al., 2019). However, they require access to the algorithm's gradients, thus limiting their use and incurring a larger compute overhead. In this paper, we focus on purely black-box methods for their ease of use in any RL setting.

Extensions of population-based training (PBT) (Jaderberg et al., 2017; Li et al., 2019) improvements like BO kernels (Parker-Holder et al., 2020) or added NAS components (Franke et al., 2020; Wan et al., 2022) have led to significant performance and efficiency gains, offering a RL-specific way of optimizing hyperparameters during training. A benefit of PBT methods is that they implicitly find a schedule of hyperparameter settings instead of a fixed value.

Beyond PBT methods, many general AC algorithms have proven to perform well on ML and RL tasks (Schede et al., 2022). A few such examples are SMAC (Lindauer et al., 2022) and DEHB (Awad et al., 2021) which are based on Bayesian Optimization and evolutionary algorithms, respectively. SMAC is model-based (i.e. it learns a model of the hyperparameter landscape using a Gaussian process) and both are multi-fidelity methods (i.e. they utilize shorter training runs to test many different configurations, only progressing the best ones). While these algorithms have rarely been used in RL so far, there is no evidence to suggest they perform any worse than RL-specific optimization approaches. In fact, a possible advantage of multi-fidelity approaches over population-based ones is that given the same budget, multi-fidelity methods see a larger number of total configurations, while population-based ones see a smaller number of configurations trained for a longer time.

4. The Hyperparameter Landscape of RL

Before comparing HPO algorithms, we empirically motivate why using dedicated tuning tools is important in RL. To this end we study the effect of hyperparameters as well as that of the random seed on the final performance of RL algorithms. We also investigate the smoothness of the hyperparameter space. The goal of this section is not to achieve the best possible results on each task but to gather insights into how hyperparameters affect RL algorithms and how we can optimize them effectively.

Experimental Setup To gain robust insights into the impact of hyperparameters on the performance of an RL agent, we consider a range of widely-used environments and algorithms. We use basic gym environments such as OpenAI's Pendulum and Acrobot (Brockman et al., 2016), gridworld with an exploration component such as MiniGrid's Empty and DoorKey 5x5 (Chevalier-Boisvert et al., 2018), as well as robot locomotion tasks such as Brax's Ant, Halfcheetah and Humanoid (Freeman et al., 2021). We use PPO (Schulman et al., 2017) and DQN (Mnih et al., 2015) for the discrete environments, and PPO as well as SAC (Haarnoja et al., 2018) for the continuous ones, all in their `StableBaselines3` implementations (Raffin et al., 2021). This selection is representative of the main classes of model-free RL algorithms (i.e. on-policy policy-optimization, off-policy value-based, and off-policy actor-critic) and covers a diverse set of tasks posing different challenges (i.e. discrete and continuous control), allowing us to draw meaningful and generalizable conclusions.

For each environment, we sweep over 8 hyperparameters for DQN, 7 for SAC and 11 for PPO (for a full list, see Appendix E). We run each combination of hyperparameter setting, algorithm and environment for 5 different random seeds. For brevity's sake, we focus on the PPO results in the main paper. The results on the other algorithms lead to similar conclusions and can be found in Appendix H.

For the tuning insights in this section, we use random search (RS) in its Optuna implementation (Akiba et al., 2019), a multi-fidelity method called DEHB (Awad et al., 2021) and a PBT approach called PB2 (Parker-Holder et al., 2020). Although grid search is certainly more commonly-used in RL than RS, we do not include it as a baseline due to its major disadvantages relative to RS such as its poor scaling with the size of the search space and heavy reliance on domain knowledge (Bergstra & Bengio, 2012). We choose DEHB and PB2 as two standard incarnations of multi-fidelity and PBT methods without any extensions like run initialization (Wan et al., 2022) or configuration racing (Lindauer et al., 2022) because we want to test how well lightweight vanilla versions of these algorithm classes perform on RL.We use a total budget of 10 full RL runs for all methods. For more background on these methods as well as their own hyperparameter settings, see Appendix C. A complete overview of search spaces and experiment settings can be found in Appendix D. The code for all experiments in this paper can be found at `https://github.com/facebookresearch/how-to-autorl`.

4.1. Which RL Hyperparameters Should Be Tuned?

Our goal is not to find good default hyperparameter settings (see Appendix E for our reasoning) or gain insights into why some configurations perform a certain way. Instead, we are interested in their general relevance, i.e., the effect size for hyperparameter tuning. Thus, we run sweeps over our chosen hyperparameters for each environment and algorithm to get an impression of which hyperparameters are important in each setting. See Appendix G for the full results.

In Figure 2, we can see a *large influence on the final performance of almost every hyperparameter we sweep over for each environment.* Even the rarely tuned clip range can be a deciding factor between an agent succeeding or failing in an environment such as in Ant. In many cases, hyperparameters can also have a large effect on the algorithm's stability throughout training. In total, we observed only the worst hyperparameter choice being within the best choice's standard deviation 7 times out of 126 settings and only 13 times the median performance dropping by less than 20%. At the same time, hyperparameter importance analysis using fANOVA (Hutter et al., 2014) shows that one or two hyperparameters monopolize the importance on each environment - though they tend to vary from environment to environment

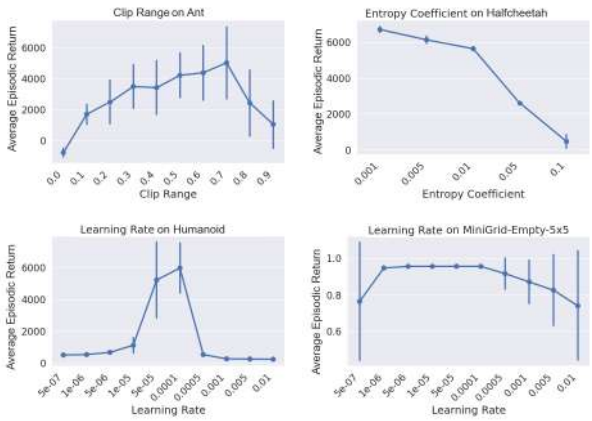


Figure 2: Hyperparameter landscapes of learning rate, clip range and entropy coefficient for PPO on Brax and MiniGrid. For each hyperparameter value, we report the average final return and standard deviation across 5 seeds.

(e.g.fro PPO learning rate on Acrobot, clip range on Pendulum and the GAE lambda on MiniGrid - see Appendix I for full results). Additionally, Partial Dependency Plots on Pendulum and Acrobot (see Appendix J) show that there are almost no complex interaction patterns between the hyperparameters which would increase the difficulty when tuning all of them at the same time. Since most hyperparameters have significant influences on performance, their importance varies across environments and there are only few interference effects, we recommend tuning as many hyperparameters as possible – as is best practice in the AutoML community (Eggensperger et al., 2019).

This result suggests that common grid search approaches are likely suboptimal as good search space coverage along many dimensions is highly expensive (Bergstra & Bengio, 2012). In order to empirically test if current HPO tools are well suited to such a set of diverse hyperparameters, we tune our algorithms using differently sized search spaces: (i) only the learning rate (which could be hand-tuned), (ii) a small space with three hyperparameters (which would be expensive but possible to tune manually) and (iii) the full search space of 7 hyperparameters for SAC, 9 for DQN, and 11 for PPO (which is too large to feasibly tune by hand - sweeping 7 hyperparameters with only three values amounts to a grid search of 2187 runs).

In Table 1 we see that RS performs well on Acrobot but it falls short on Pendulum, displaying large discrepancies across seeds, some performing well, and some failing to find a good configuration. While this is a typical failure case of RS, this does not mean RS is a weak candidate, ranking second overall by outperforming PB2 in several cases. PB2 is also quite unreliable: on Acrobot, its performance decreases with the size of the search space; on Pendulum, however, it improves with the size of the search space. As

Table 1: Tuning PPO on Acrobot (top) and SAC on Pendulum (bottom) across different search space sizes (i.e. only learning rate, {learning rate, entropy coefficient, training epochs}, and full search space). Shown is the negative evaluation reward across 5 tuning runs. Lower numbers are better, best performance on each environment is highlighted. The best final performance on a single seed from our sweeps is also reported.

		DEHB Inc.	PB2 Inc.	RS Inc.	Sweep
Acrobot	LR Only	$\mathbf{71 \pm 1}$	94 ± 22	78 ± 5	81
	Small	$\mathbf{72 \pm 1}$	193 ± 160	80 ± 6	
	Full	$\mathbf{71 \pm 3}$	305 ± 186	83 ± 5	
Pendulum	LR Only	$\mathbf{71 \pm 12}$	207 ± 126	89 ± 25	117
	Small	119 ± 12	106 ± 12	401 ± 363	
	Full	112 ± 24	78 ± 19	144 ± 48	

with RS, part of the underlying issue is the inconsistent performance of PB2. Note that the incumbent configuration is fairly static across all PB2 runs for the larger search spaces. In most cases, the configuration changes at most once during training, showing that PB2 currently does not take full advantage of its ability to find dynamic schedules. DEHB is the most stable in terms of standard deviation across seeds, even though we see a slight decrease in performance on Pendulum with larger search spaces.

Overall we see that finding well performing configurations across large search spaces is usually possible even with a simple algorithm like RS. All methods deliver reasonable hyperparameter configurations across a large search space, especially given all of them use only 10 full training runs. On this small budget, they are able to match or outperform the single best seeds in all our sweep runs which use a total of 125 runs per environment. Our experiments show that *automatically tuning a large variety of hyperparameters is both beneficial and efficient using even simple algorithms like RS or vanilla instantiations of multi-fidelity and population-based methods.*

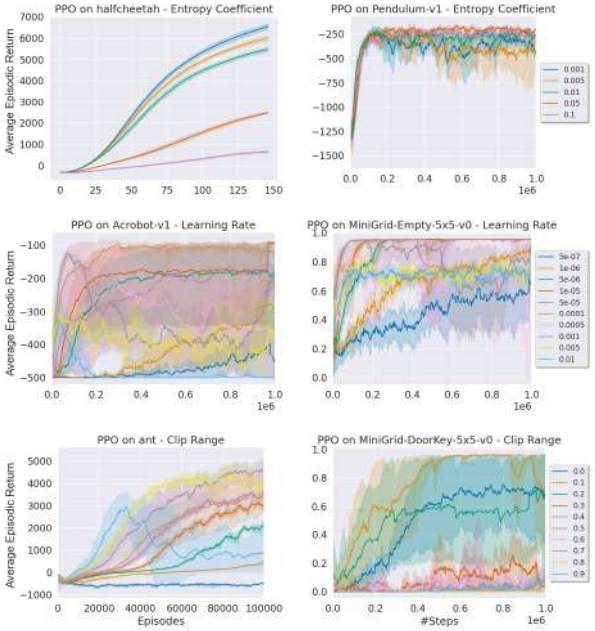


Figure 3: Hyperparameter Sweeps for PPO across learning rates, entropy coefficients and clip ranges on various environments. The mean and standard deviation are computed across 5 seeds.

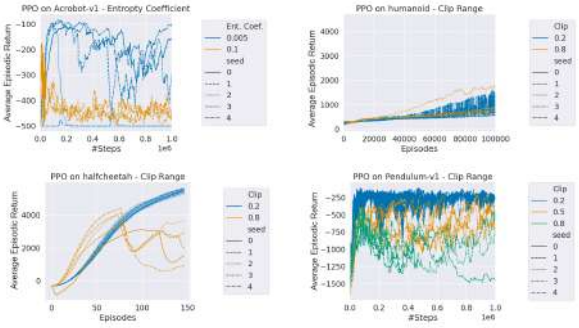


Figure 4: Individual seeds for selected clip range and entropy coefficient values of PPO across various environments.

4.2. Are Hyperparameters in RL Well Behaved?

In addition to the large number of hyperparameters contributing to an algorithm's performance, how an algorithm behaves with respect to changing hyperparameter values is an important factor in tuning algorithms. Ideally, we want the algorithm's performance to be predictable, i.e., if the hyperparameter value is close to the optimum, we want the agent to perform well and then become progressively worse the farther we move away – in essence, a smooth optimization landscape (Pushak & Hoos, 2022). As we can see in Figure 2, the transitions between different parts of the search space are fairly smooth. The configurations perform in the order we would expect them to given the best values, with the drops in performance being mostly gradual instead of sudden. Figure 3 shows configurations also performing consistently with regards to one another during the runtime, i.e., good configurations tend to learn quickly and bad configurations decay soon after training begins. This means HPO approaches utilizing partial algorithm runs to measure the quality of configurations like multi-fidelity methods or PBT should not face major issues tuning RL algorithms.

While we do see large variability in some configurations, this issue seems to occur largely in medium-well performing configurations, not in the very best or worst ones (see Figure 3). This supports our claim that hyperparameters are not only useful in increasing performance but have a significant influence on algorithm variability.

During the run itself differences between seeds can become an issue, however, especially for methods using partial runs. On many environments, when looking at each seed indi-

Table 2: Tuning PPO on Acrobot (top) and SAC on Pendulum (bottom) across the full search space and different numbers of seeds. Lower numbers are better, best test performance for each method and values within its standard deviation are highlighted. Test performances are aggregated across 10 separate test seeds using the mean for each tuning run. We report mean and standard deviation of these.

		DEHB Inc.	DEHB Test	PB2 Inc.	PB2 Test	RS Inc.	RS Test
Acrobot	1 Seed	70.6 ± 3.4	$\mathbf{341.3 \pm 183.1}$	305.3 ± 185.5	$\mathbf{353.7 \pm 134.5}$	77.8 ± 4.9	136.8 ± 70.5
	3 Seeds	76.2 ± 0.9	$\mathbf{381.1 \pm 127.6}$	301.2 ± 128.0	$\mathbf{411.3 \pm 117.9}$	88.2 ± 5.7	$\mathbf{98.8 \pm 16.3}$
	5 Seeds	79.3 ± 1.2	465.1 ± 24.6	228.5 ± 149.5	$\mathbf{471.8 \pm 19.1}$	89.2 ± 10.4	116.8 ± 43.3
	10 Seeds	156.0 ± 24.5	464.8 ± 36.5	404.9 ± 53.3	$\mathbf{474.4 \pm 23.5}$	108.3 ± 28.2	$\mathbf{100.1 \pm 20.0}$
Pendulum	1 Seed	111.5 ± 23.6	$\mathbf{150.5 \pm 13.4}$	77.8 ± 19.0	840.7 ± 580.1	88.6 ± 24.9	168.3 ± 46.4
	3 Seeds	125.0 ± 23.2	$\mathbf{144.8 \pm 9.0}$	133.3 ± 14.7	$\mathbf{171.0 \pm 35.5}$	150.7 ± 13.9	159.0 ± 21.6
	5 Seeds	127.3 ± 11.5	350.2 ± 418.2	134.0 ± 22.1	661.3 ± 586.2	134.8 ± 9.8	397.8 ± 485.5
	10 Seeds	742.4 ± 498.8	318.6 ± 281.3	282.0 ± 252.9	468.6 ± 437.9	144.5 ± 17.9	$\mathbf{150.2 \pm 4.8}$

vidually per hyperparameter as in Figure 4, we can see the previously predictable behaviour is replaced with significant differences across seeds. We observe single seeds with crashing performance, inconsistent learning curves and also exceptionally well performing seeds that end up outperforming the best seeds of configurations which are better on average. Given that we believe tuning only a few seeds of the target RL algorithm is still the norm (Schulman et al., 2017; Berner et al., 2019; Raileanu & Rocktäschel, 2020; Badia et al., 2020; Hambro et al., 2022), such high variability with respect to the seed is likely a bigger difficulty factor for HPO in RL than the optimization landscape itself.

Thus, our conclusion is somewhat surprising: *it should be possible to tune RL hyperparameters just as well as the ones in any other fields without RL-specific additions to the tuning algorithm since RL hyperparameter landscapes appear to be rather smooth.* The large influence of many different hyperparameters is a potential obstacle, however, as are interaction effects that can ocurr between hyperparameters. Furthermore, RL's sensitivity to the random seed, can present a challenge in tuning its hyperparameters, both by hand and in an automated manner.

4.3. How Do We Account for Noise?

As the variability between random seeds is a potential source of error when tuning and running RL algorithms, we investigate how we can account for it in our experiments to generate more reliable performance estimates.

As we have seen high variability both in performance and across seeds for different hyperparameter values, we return to Figure 2 to investigate how big the seed's influence on the final performance really is. The plots show that the standard deviation of the performance for the same hyperparameter configuration can be very large. While this performance spread tends to decrease for configurations with better median performance, top-performing seeds can stem from unstable configurations with low median performance (e.g. the learning rate on Humanoid). In most cases, there is an overlap between adjacent configurations, so *it is certainly possible to select a presumably well-performing hyperparameter configuration on one seed that has low average performance across others*.

As this is a known issue in other fields as well, albeit not to the same degree as in RL, *it is common to evaluate a configuration on multiple seeds in order to achieve a more reliable estimate of the true performance* (Eggensperger et al., 2018). We verify this for RL by comparing the final performance of agents tuned by DEHB and PB2 on the performance mean across a single, 3 or 5 seeds. We then test the overall best configuration on 5 unseen test seeds.

Table 2 shows that RS is able to improve the average test performance on both Acrobot and Pendulum by increasing the number of tuning seeds, as are DEHB and PB2 on Pendulum. However, this is only true up to a point as performance estimation across more than 3 seeds leads to a general decrease in test performance, as well as a sharp increase in variance in some cases (e.g. 5 seed RS or 10 seed PB2 on Pendulum). Especially when tuning across 10 seeds, we see that the incumbents suffer as well, indicating that evaluating the configurations across multiple seeds increases the difficulty of the HPO problem substantially, even though it can help avoid overfitting. The performance difference between tuning and testing is significant in many cases and we can see e.g. on Acrobot that the best incumbent configurations, found by DEHB, perform more than four times worse on test seeds. We can find this effect in all tuning methods, especially on Pendulum. This presents a challenge for reproducibility given that currently it is almost impossible to know what seeds were used for tuning or evaluation. Simply reporting the performance of tuned seeds for the proposed method and that of testing seeds for the baselines is an unfair comparison which can lead to wrong conclusions.

To summarize, we have seen that the main challenges are the size of the search space, the variability involved in training RL agents, and the challenging generalization across

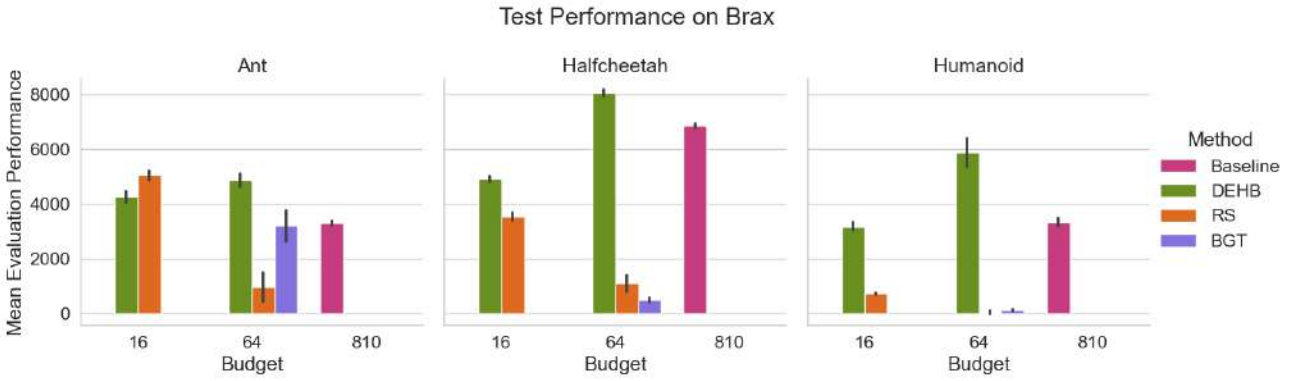


Figure 5: Tuning Results for PPO on Brax. Shown is the mean evaluation reward across 10 episodes for 3 tuning runs as well as the 98% confidence interval across tuning runs.

random seeds. Since many hyperparameters have a large influence on agent performance, but the optimization landscape is relatively smooth, RL hyperparameters can be efficiently tuned using HPO techniques, as we have shown in our experiments. Manual tuning, however, is comparatively costly as its cost scales at least linearly with the size of the search space.Dedicated HPO tools, on the other hand are able to find good configurations on a significantly smaller budget by searching the whole space. *A major difficulty factor, however, is the high variability of results across seeds, which is an overlooked reproducibility issue that can lead to distorted comparisons of RL algorithms. This problem can be alleviated by tuning the algorithms on multiple seeds and evaluating them on separate test seeds.*

5. Tradeoffs for Hyperparameter Optimization in Practice

While the experiments in the previous section are meant to highlight what challenges HPO tools face in RL and how well they overcome them, we now turn to more complex use cases of HPO. To this end, we select three challenging environments each from Brax (Freeman et al., 2021) (Ant, Halfcheetah and Humanoid) and three from Procgen (Cobbe et al., 2020) (Bigfish, Climber and Plunder) and automatically tune the state-of-the-art RL algorithms on these domains (PPO for Brax and IDAAC (Raileanu & Fergus, 2021) for Procgen). Our goal here is simple: we want to see if HPO tools can improve upon the state of the art in these domains with respect to both final performance and compute overhead. As we now want to compare absolute performance values on a more complex problems with a bigger budget, we use BGT (Wan et al., 2022) as the state-of-the-art population-based approach, and DEHB since it is among the best solver currently available (Eggensperger et al., 2021). As before, we use RS as an example of a simple-to-implement tuning algorithm with minimal overhead. In view of the results of Turner et al. (2021) on HPO for supervised machine learning, we expect that RS should be outperformed by the other approaches. For each task, we work on the original open-sourced code of each state-of-the-art RL method we test against, using the manually tuned hyperparameter settings as recommended in the corresponding papers as the baseline. All tuning algorithms will be given a small budget of up to 16 full algorithm runs as well as a larger one of 64 runs. In comparison, IDAAC's tuning budget is 810 runs. To give an idea of the reliability of both the tuning algorithm and the found configurations, we tune each setting 3 times across 5 seeds and test the best-found configuration on 10 unseen test seeds.

As shown in Figures 5 and 6, these domains are more challenging to tune on our small budgets relative to our previous environments (for tabular results, see Appendix F). While we do not know how the Brax baseline agent was tuned as this is not reported in the paper, the IDAAC baseline uses 810 runs which is 12 times more than the large tuning budget used by our HPO methods. On Brax, DEHB outperforms the baseline with a mean rank of 1.3 compared to 1.7 for the 16 run budget and a rank of 1.0 compared to the baselines's 1.3 with 64 runs. On Procgen the comparison is similar with 1.7 to 2 for 16 runs and 1.0 to 1.3 for 64 runs (see Appendix D.4 for details on how the rank is computed). We also see that DEHB's incumbent and test scores improve the most consistently out of all the tuning methods, with the additional run budget being utilized especially well on Brax. RS, as expected cannot match this performance, ranking 2.3 and 2.7 for 16 runs and 3.3 and 3 for 64 runs respectively. We also see poor scaling behavior in some cases e.g. RS with a larger budget overfits to the tuning seeds on Brax while failing to improve on Procgen. As above, we see an instance of PB2 performing around 5 times worse on the test seeds compared to the incumbent on Bigfish, further suggesting that certain PBT variants may struggle to generalize in such settings. On the other environments it does better, however, earning a Procgen rank of 2 on the 16 run budget, matching the baseline. With a budget of 64 runs, it

ranks 2.7, the same as BGT and above RS. BGT does not overfit to the same degree as PB2 but performs worse on lower budgets, ranking 3.8 on Procgen for 16 runs and 2.7 for 64. On Brax, it fails to find good configurations with the exception of a single run on Ant (rank 3). We do not restart the BGT optimization after a set amount of failures, however, in order to keep within our small maximum budgets. The original paper indicates that it is likely BGT will perform much better given a less restrictive budget.

Overall, HPO tools conceived for the AC setting, as represented by DEHB, are the most consistent and reliable within our experimental setting. Random Search, while not a bad choice on smaller budgets, does not scale as well with the number of tuning runs. Population-based methods cannot match either; PB2, while finding very well performing incumbent configurations struggles with overfitting, while BGT would likely benefit from larger budgets than used here. Further research into this optimization paradigm that prioritizes general configurations over incumbent performance could lead to additional improvements.

Across both benchmarks we see large discrepancies between the incumbent and test performance. This underlines our earlier point about the importance of using different test and tuning seeds for reporting. In terms of compute overhead, all tested HPO methods had negligible effects on the total runtime, with BGT, by far the most expensive one, utilising on average under two minutes of time to produce new configurations for the 16 run budget and less than 2 hours for the 64 run budget, with all other approaches staying under 5 minutes in each budget. Overall, we see that *even computationally cheap methods with small tuning budgets can generally match or outperform painstakingly hand-tuned configurations that use orders of magnitude more compute.*

6. Recommendations & Best Practices

Our experiments show the benefit of comprehensive hyperparameter tuning in terms of both final performance and compute cost, as well as how common overfitting to the set of tuning seeds is. As a result of our insights, we recommend some good practices for HPO in RL going forward.

Complete Reporting We still find that many RL papers do not state how they obtain their hyperparameter configurations, if they are included at all. As we have seen, however, unbiased comparisons should not take place on the same seeds the hyperparameters are tuned on. Hence, reporting the tuning seeds, the test seeds, and the exact protocol used for hyperparameter selection, should be standard practice to ensure a sound comparison across RL methods.

Adopting AutoML Standards In many ways, the AutoML community is ahead of the RL community regarding hyperparameter tuning. We can leverage this by learning from their best practices, as e.g. stated by Eggensperger et al. (2019) and Lindauer & Hutter (2020), and using their HPO tools which can lead to strong performance as shown in this paper. One notable good practice is to use separate seeds for tuning and testing hyperparameter configurations. Other examples include standardizing the tuning budget for the baselines and proposed methods, as well as tuning on the training and not the test setting.While HPO in RL provides unique challenges such as the dynamic nature of the training loop or the strong sensitivity to the random seed, we observe significant improvements in both final performance and compute cost by employing state-of-the-art AutoML approaches. This can be done by integrating multi-fidelity evaluations into the population-based framework or using optimization tools like DEHB and SMAC.

Integrate Tuning Into The Development Pipeline For fair comparisons and realistic views of RL methods, we have to use competently tuned baselines. More specifically, the proposed method and baselines should use the same tuning budget and be evaluated on test seeds which should be different from the tuning seeds. Integrating HPO into RL codebases is a major step towards facilitating such comparisons. Some RL frameworks have started to include options for automated HPO (Huang et al., 2021; Liaw et al., 2018) or provide recommended hyperparameters for a set of environments (Raffin et al., 2021) (although usually not how they were obtained). The choice of tuning tools for each library is still relatively limited, however, while provided hyperparameters are not always well documented and typically do not transfer well to other environments or algorithms. Thus, we hope our versatile and easy-to-use HPO implementations that can be applied to any RL algorithm and environment will encourage broader use of HPO in RL (see Appendix B for more information). In the future, we hope more RL libraries include AutoRL approaches since in a closed ecosystem, more sophisticated methods that go beyond black-box optimizers (e.g. gradient-based methods, neuro-evolution, or meta-learned hyperparameter agents à la DAC) could be deployed more easily.

A Recipe For Efficient RL Research To summarize, we recommend the following step-by-step process for tuning and selecting hyperparameters in RL:

1. Define a training and test set which can include:
   (a) environment variations
   (b) random seeds for non-deterministic environments
   (c) random seeds for initial state distributions
   (d) random seeds for the agent (including network initialization)
   (e) training random seeds for the HPO tool
2. Define a configuration space with all hyperparameters that likely contribute to training success;

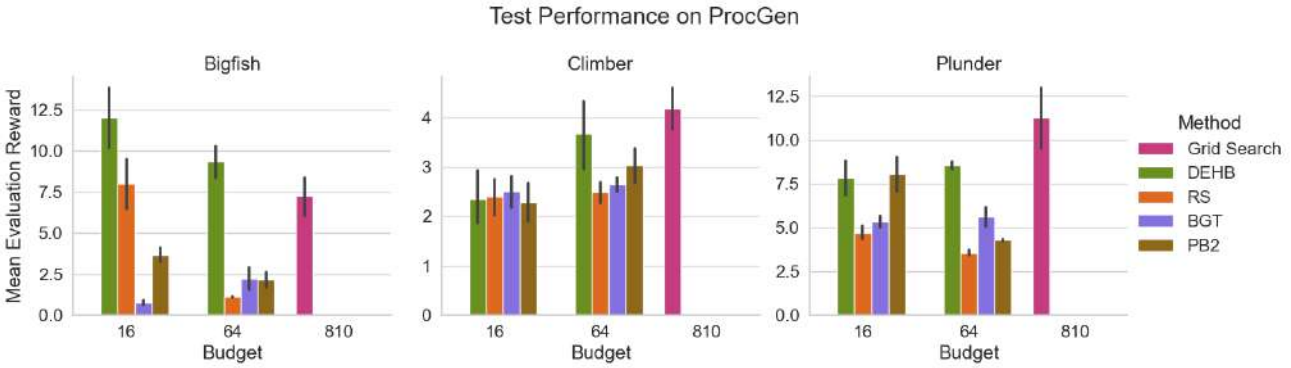


Figure 6: Tuning Results for IDAAC on Procgen. Shown is the mean evaluation reward across 10 episodes for 3 tuning runs as well as the 98% confidence interval across tuning runs.

3. Decide which HPO method to use;
4. Define the limitations of the HPO method, i.e. the budget (or use self-terminating (Makarova et al., 2022));
5. Settle on a cost metric – this should be an evaluation reward across as many episodes as a needed for a reliable performance estimate;
6. Run this HPO method on the training set across a number of tuning seeds;
7. Evaluate the resulting incumbent configurations on the test set across a number of separate test seeds and report the results.

To ensure a fair comparison, this procedure should be followed for all RL methods used, including the baselines. If existing hyperparameters are re-used, their source and tuning protocol should be reported. In addition, their corresponding budget and search space should be the same as those of the other RL methods used for comparison. In case the budget is runtime and not e.g. a number of environment steps, it is also important to use comparable hardware for all runs. Furthermore, it is important to use the same test seeds for all configurations that are separate from all tuning seeds. If this information is not available, re-tuning the algorithm is preferred. This procedure, including all information on the search space, cost metric, HPO method settings, seeds and final hyperparameters should be reported. We provide a checklist containing all of these points in Appendix A and as a `LaTeX` template in our GitHub repository.

7. Conclusion

We showed that hyperparameters in RL deserve more attention from the research community than they currently receive. Underreported tuning practices have the potential to distort algorithm evaluations while ignored hyperparameters may lead to suboptimal performance. With only small budgets, we demonstrate that HPO tools like DEHB can cover large search spaces to produce better performing configurations using fewer computational resources than hyperparameter sweeps or grid searches. We provide versatile and easy-to-use implementations of these tools which can be applied to any RL algorithm and environment. We hope this will encourage the adoption of AutoML best practices by the RL community, which should enhance the reproducibility of RL results and make solving new domains simpler.

Nevertheless, there is a lot of potential for developing HPO approaches tailored to the key challenges of RL such as the high sensitivty to the random seed for a given hyperparameter configuration. Frameworks for learnt hyperparameter policies or gradient-based optimization methods could counteract this effect by reacting dynamically to an algorithm's behaviour on a given seed. We believe this is a promising direction for future work since in our experiments, PBT methods yield fairly static configurations instead of flexible schedules. Benchmarks like the recent AutoRL-Bench (Shala et al., 2022) accelerate progress by comparing AutoRL tools without the need for RL algorithm evaluations. Lastly, higher-level AutoRL approaches that do not aim to find hyperparameter values but replace them entirely by directing the algorithm's behavior could in the long term both simplify and stabilize RL algorithms. Examples include exploration strategies (Zhang et al., 2021b), learnt optimizers (Metz et al., 2022) or entirely new algorithms (Co-Reyes et al., 2021; Lu et al., 2022).

Acknowledgements

Marius Lindauer acknowledges funding by the European Union (ERC, "ixAutoML", grant no.101041029). Views and opinions expressed are those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council Executive Agency. Neither the European Union nor the granting authority can be held responsible for them.

References

Adriaensen, S., Biedenkapp, A., Shala, G., Awad, N., Eimer, T., Lindauer, M., and Hutter, F. Automated dynamic algorithm configuration. *Journal of Artificial Intelligence Research*, 2022.

Agarwal, R., Schwarzer, M., Castro, P., Courville, A., and Bellemare, M. Deep reinforcement learning at the edge of the statistical precipice. In Ranzato, M., Beygelzimer, A., Dauphin, Y. N., Liang, P., and Vaughan, J. W. (eds.), *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS*, pp. 29304–29320, 2021.

Akiba, T., Sano, S., Yanase, T., Ohta, T., and Koyama, M. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2019.

Andrychowicz, M., Raichuk, A., Stanczyk, P., Orsini, M., Girgin, S., Marinier, R., Hussenot, L., Geist, M., Pietquin, O., Michalski, M., Gelly, S., and Bachem, O. What matters for on-policy deep actor-critic methods? A large-scale study. In *9th International Conference on Learning Representations, ICLR*. OpenReview.net, 2021.

Awad, N., Mallik, N., and Hutter, F. DEHB: evolutionary hyberband for scalable, robust and efficient hyperparameter optimization. In Zhou, Z. (ed.), *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI*, pp. 2147–2153. ijcai.org, 2021.

Badia, A., Piot, B., Kapturowski, S., Sprechmann, P., Vitvitskyi, A., Guo, Z., and Blundell, C. Agent57: Outperforming the atari human benchmark. In *Proceedings of the 37th International Conference on Machine Learning, ICML*, volume 119 of *Proceedings of Machine Learning Research*, pp. 507–517. PMLR, 2020.

Bakshy, E., Dworkin, L., Karrer, B., Kashin, K., Letham, B., Murthy, A., and Singh, S. Ae: A domain-agnostic platform for adaptive experimentation. 2018.

Bechtle, S., Molchanov, A., Chebotar, Y., Grefenstette, E., Righetti, L., Sukhatme, G., and Meier, F. Meta learning via learned loss. In *25th International Conference on Pattern Recognition, ICPR*, pp. 4161–4168. IEEE, 2020.

Bergstra, J. and Bengio, Y. Random search for hyper-parameter optimization. 13:281–305, 2012.

Berner, C., Brockman, G., Chan, B., Cheung, V., Debiak, P., Dennison, C., Farhi, D., Fischer, Q., Hashme, S., Hesse, C., Józefowicz, R., Gray, S., Olsson, C., Pachocki, J., Petrov, M., de Oliveira Pinto, H., Raiman, J., Salimans, T., Schlatter, J., Schneider, J., Sidor, S., Sutskever, I., Tang, J., Wolski, F., and Zhang, S. Dota 2 with large scale deep reinforcement learning. *CoRR*, abs/1912.06680, 2019.

Biedenkapp, A., Bozkurt, H. F., Eimer, T., Hutter, F., and Lindauer, M. Dynamic Algorithm Configuration: Foundation of a New Meta-Algorithmic Framework. In Lang, J., Giacomo, G. D., Dilkina, B., and Milano, M. (eds.), *Proceedings of the Twenty-fourth European Conference on Artificial Intelligence (ECAI'20)*, pp. 427–434, June 2020.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. Openai gym, 2016.

Chevalier-Boisvert, M., Willems, L., and Pal, S. Minimalistic gridworld environment for gymnasium, 2018. URL https://github.com/Farama-Foundation/Minigrid.

Co-Reyes, J., Miao, Y., Peng, D., Real, E., Le, Q., Levine, S., Lee, H., and Faust, A. Evolving reinforcement learning algorithms. In *9th International Conference on Learning Representations, ICLR*. OpenReview.net, 2021. URL https://openreview.net/forum?id=0XXpJ4OtjW.

Cobbe, K., Hesse, C., Hilton, J., and Schulman, J. Leveraging procedural generation to benchmark reinforcement learning. In *Proceedings of the 37th International Conference on Machine Learning, ICML*, volume 119 of *Proceedings of Machine Learning Research*, pp. 2048–2056. PMLR, 2020.

Duan, Y., Schulman, J., Chen, X., Bartlett, P., Sutskever, I., and Abbeel, P. Rl^2: Fast reinforcement learning via slow reinforcement learning. *CoRR*, abs/1611.02779, 2016.

Eggensperger, K., Lindauer, M., and Hutter, F. Neural networks for predicting algorithm runtime distributions. In Lang, J. (ed.), *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 1442–1448. ijcai.org, 2018.

Eggensperger, K., Lindauer, M., and Hutter, F. Pitfalls and best practices in algorithm configuration. pp. 861–893, 2019.

Eggensperger, K., Müller, P., Mallik, N., Feurer, M., Sass, R., Klein, A., Awad, N., Lindauer, M., and Hutter, F. Hpobench: A collection of reproducible multi-fidelity benchmark problems for HPO. In Vanschoren, J. and Yeung, S. (eds.), *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks*, 2021.

Engstrom, L., Ilyas, A., Santurkar, S., Tsipras, D., Janoos, F., Rudolph, L., and Madry, A. Implementation matters in deep RL: A case study on PPO and TRPO. In *8th International Conference on Learning Representations, ICLR*. OpenReview.net, 2020.

Flennerhag, S., Schroecker, Y., Zahavy, T., van Hasselt, H., Silver, D., and Singh, S. Bootstrapped meta-learning. In *The Tenth International Conference on Learning Representations, ICLR*. OpenReview.net, 2022.

Franke, J., Köhler, G., Biedenkapp, A., and Hutter, F. Sample-efficient automated deep reinforcement learning. In *9th International Conference on Learning Representations, ICLR*. OpenReview.net, 2021.

Franke, J. K., Köhler, G., Biedenkapp, A., and Hutter, F. Sample-efficient automated deep reinforcement learning. *arXiv:2009.01555 [cs.LG]*, 2020.

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

Haarnoja, T., Zhou, A., Abbeel, P., and Levine, S. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In Dy, J. G. and Krause, A. (eds.), *Proceedings of the 35th International Conference on Machine Learning, ICML*, volume 80 of *Proceedings of Machine Learning Research*, pp. 1856–1865. PMLR, 2018.

Hambro, E., Raileanu, R., Rothermel, D., Mella, V., Rocktäschel, T., Küttler, H., and Murray, N. Dungeons and data: A large-scale nethack dataset. 2022.

Henderson, P., Islam, R., Bachman, P., Pineau, J., Precup, D., and Meger, D. Deep reinforcement learning that matters. In McIlraith, S. and Weinberger, K. (eds.), *Proceedings of the Conference on Artificial Intelligence (AAAI'18)*. AAAI Press, 2018.

Hsu, C., Mendler-Dünner, C., and Hardt, M. Revisiting design choices in proximal policy optimization. *CoRR*, abs/2009.10897, 2020.

Huang, S., Dossa, R., Ye, C., and Braga, J. Cleanrl: High-quality single-file implementations of deep reinforcement learning algorithms. *CoRR*, abs/2111.08819, 2021.

Hutter, F., Hoos, H., and Leyton-Brown, K. An efficient approach for assessing hyperparameter importance. In Xing, E. and Jebara, T. (eds.), *Proceedings of the 31th International Conference on Machine Learning, (ICML'14)*, pp. 754–762. Omnipress, 2014.

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

Kiran, M. and Ozyildirim, B. Hyperparameter tuning for deep reinforcement learning applications. *CoRR*, abs/2201.11182, 2022. URL https://arxiv.org/abs/2201.11182.

Li, A., Spyra, O., Perel, S., Dalibard, V., Jaderberg, M., Gu, C., Budden, D., Harley, T., and Gupta, P. A generalized framework for population based training. In Teredesai, A., Kumar, V., Li, Y., Rosales, R., Terzi, E., and Karypis, G. (eds.), *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD*, pp. 1791–1799. ACM, 2019.

Li, L., Jamieson, K., DeSalvo, G., Rostamizadeh, A., and Talwalkar, A. Hyperband: A novel bandit-based approach to hyperparameter optimization. 18(185):1–52, 2018.

Liaw, R., Liang, E., Nishihara, R., Moritz, P., Gonzalez, J., and Stoica, I. Tune: A research platform for distributed model selection and training. *CoRR*, abs/1807.05118, 2018.

Lindauer, M. and Hutter, F. Best practices for scientific research on neural architecture search. *Journal of Machine Learning Research*, 21:1–18, 2020.

Lindauer, M., Eggensperger, K., Feurer, M., Biedenkapp, A., Deng, D., Benjamins, C., Ruhkopf, T., Sass, R., and Hutter, F. SMAC3: A versatile bayesian optimization package for hyperparameter optimization. *J. Mach. Learn. Res.*, 23:54:1–54:9, 2022.

Lu, C., Kuba, J., Letcher, A., Metz, L., de Witt, C., and Foerster, J. Discovered policy optimisation. *CoRR*, abs/2210.05639, 2022.

Makarova, A., Shen, H., Perrone, V., Klein, A., Faddoul, J., Krause, A., Seeger, M., and Archambeau, C. Automatic termination for hyperparameter optimization. In Guyon, I., Lindauer, M., van der Schaar, M., Hutter, F., and Garnett, R. (eds.), *International Conference on Automated Machine Learning, AutoML*, volume 188 of *Proceedings of Machine Learning Research*, pp. 7/1–21. PMLR, 2022.

Metz, L., Harrison, J., Freeman, C., Merchant, A., Beyer, L., Bradbury, J., Agrawal, N., Poole, B., Mordatch, I., Roberts, A., and Sohl-Dickstein, J. Velo: Training versatile learned optimizers by scaling up. *CoRR*, abs/2211.09760, 2022.

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

Obando-Ceron, J. and Castro, P. Revisiting rainbow: Promoting more insightful and inclusive deep reinforcement learning research. In Meila, M. and Zhang, T. (eds.), *Proceedings of the 38th International Conference on Machine Learning, ICML*, volume 139 of *Proceedings of Machine Learning Research*, pp. 1373–1383. PMLR, 2021.

Parker-Holder, J., Nguyen, V., and Roberts, S. Provably efficient online hyperparameter optimization with population-based bandits. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., and Lin, H. (eds.), *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS*, 2020.

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

Paul, S., Kurin, V., and Whiteson, S. Fast efficient hyperparameter tuning for policy gradient methods. In Wallach, H. M., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E. B., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS*, pp. 4618–4628, 2019.

Pushak, Y. and Hoos, H. H. Automl loss landscapes. *ACM Trans. Evol. Learn. Optim.*, 2(3):10:1–10:30, 2022.

Raffin, A., Hill, A., Gleave, A., Kanervisto, A., Ernestus, M., and Dormann, N. Stable-baselines3: Reliable reinforcement learning implementations. *J. Mach. Learn. Res.*, 22:268:1–268:8, 2021.

Raileanu, R. and Fergus, R. Decoupling value and policy for generalization in reinforcement learning. In Meila, M. and Zhang, T. (eds.), *Proceedings of the 38th International Conference on Machine Learning, ICML*, volume 139 of *Proceedings of Machine Learning Research*, pp. 8787–8798. PMLR, 2021.

Raileanu, R. and Rocktäschel, T. RIDE: rewarding impact-driven exploration for procedurally-generated environments. In *8th International Conference on Learning Representations, ICLR*. OpenReview.net, 2020.

Sass, R., Bergman, E., Biedenkapp, A., Hutter, F., and Lindauer, M. Deepcave: An interactive analysis tool for automated machine learning. *CoRR*, abs/2206.03493, 2022.

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

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

Shala, G., Arango, S., Biedenkapp, A., Hutter, F., and Grabocka, J. Autorl-bench 1.0. In *Workshop on Meta-Learning (MetaLearn@NeurIPS'22)*, 2022.

Storn, R. and Price, K. Differential evolution - A simple and efficient heuristic for global optimization over continuous spaces. *J. Glob. Optim.*, 11(4):341–359, 1997.

Turner, R., Eriksson, D., McCourt, M., Kiili, J., Laaksonen, E., Xu, Z., and Guyon, I. Bayesian optimization is superior to random search for machine learning hyperparameter tuning: Analysis of the black-box optimization challenge 2020. *CoRR*, abs/2104.10201, 2021.

Wan, X., Lu, C., Parker-Holder, J., Ball, P., Nguyen, V., Ru, B., and Osborne, M. Bayesian generational population-based training. In Guyon, I., Lindauer, M., van der Schaar, M., Hutter, F., and Garnett, R. (eds.), *International Conference on Automated Machine Learning, AutoML*, volume 188 of *Proceedings of Machine Learning Research*, pp. 14/1–27. PMLR, 2022.

Wang, Z., Schaul, T., Hessel, M., van Hasselt, H., Lanctot, M., and de Freitas, N. Dueling network architectures for deep reinforcement learning. In Balcan, M. and Weinberger, K. (eds.), *Proceedings of the 33rd International Conference on Machine Learning (ICML'17)*, volume 48, pp. 1995–2003. Proceedings of Machine Learning Research, 2016.

Xu, Z., van Hasselt, H., and Silver, D. Meta-gradient reinforcement learning. In Bengio, S., Wallach, H. M., Larochelle, H., Grauman, K., Cesa-Bianchi, N., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS*, pp. 2402–2413, 2018.

Xu, Z., van Hasselt, H., Hessel, M., Oh, J., Singh, S., and Silver, D. Meta-gradient reinforcement learning with an objective discovered online. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., and Lin, H. (eds.), *Advances in Neural Information Processing Systems 33: Annual*

Conference on Neural Information Processing Systems 2020, NeurIPS, 2020.

Yadan, O. Hydra - a framework for elegantly configuring complex applications. Github, 2019. URL https://github.com/facebookresearch/hydra.

Zahavy, T., Xu, Z., Veeriah, V., Hessel, M., Oh, J., van Hasselt, H., Silver, D., and Singh, S. A self-tuning actor-critic algorithm. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., and Lin, H. (eds.), *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS*, 2020.

Zhang, B., Rajan, R., Pineda, L., Lambert, N., Biedenkapp, A., Chua, K., Hutter, F., and Calandra, R. On the importance of hyperparameter optimization for model-based reinforcement learning. In Banerjee, A. and Fukumizu, K. (eds.), *The 24th International Conference on Artificial Intelligence and Statistics, AISTATS*, volume 130 of *Proceedings of Machine Learning Research*, pp. 4015–4023. PMLR, 2021a.

Zhang, S. and Jiang, N. Towards hyperparameter-free policy selection for offline reinforcement learning. In Ranzato, M., Beygelzimer, A., Dauphin, Y. N., Liang, P., and Vaughan, J. W. (eds.), *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS*, pp. 12864–12875, 2021.

Zhang, T., Xu, H., Wang, X., Wu, Y., Keutzer, K., Gonzalez, J., and Tian, Y. Noveld: A simple yet effective exploration criterion. In Ranzato, M., Beygelzimer, A., Dauphin, Y. N., Liang, P., and Vaughan, J. W. (eds.), *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS*, pp. 25217–25230, 2021b.

Zimmer, L., Lindauer, M., and Hutter, F. Auto-pytorch: Multi-fidelity metalearning for efficient and robust autodl. *IEEE Trans. Pattern Anal. Mach. Intell.*, 43(9): 3079–3090, 2021.

A. Reproducibility Checklist for Tuning Hyperparameters in RL

Below is a checklist we recommend for conducting experiments and reporting the process in RL. It is hard to give general recommendations for all RL settings when it comes to questions of budget, number of seeds or configuration space size. For guidance on an appropriate number of testing seeds, as well as recommendations on how to report them, see Agarwal et al. (2021). The ideal number of tuning seeds will likely depend heavily on the domain, though we recommend to use at least 5 to avoid overtuning on a small number of seeds. As for configuration space size, we have seen successful tuning across up to 14 hyperparameters in this paper and only small differences between 3 and up to 9 hyperparameters in Section 4, so we believe there is no reason be too selective for search spaces of around this size unless hyperparameter importances on the algorithm and domain are fairly well known. Much larger search spaces could benefit from pruning, potentially after an initial analysis of hyperparameter importance.

1. Are there training and test settings available on your chosen domains?
   If yes:
   - Is only the training setting used for training? ✓✗
   - Is only the training setting used for tuning? ✓✗
   - Are final results reported on the test setting? ✓✗
2. Hyperparameters were tuned using `<package-name>` which is based on `<an-optimization-method>`
3. The configuration space was: `<algorithm-1>`:
   - `<a-continuous-hyperparameter>`: (`<lower>`, `<upper>`)
   - `<a-logspaced-continuous-hyperparameter>`: log((`<lower>`, `<upper>`))
   - `<a-discrete-hyperparameter>`: [`<lower>`, `<upper>`]
   - `<a-categorical-hyperparameter>`: `<choice-a>`, `<choice-b>`
   - ...

   `<algorithm-2>`:
   - `<an-additional-hyperparameter>`: (`<lower>`, `<upper>`)
   - ...
4. The search space contains the same hyperparameters and search ranges wherever algorithms share hyperparameters ✓✗
   If no, why not?
5. The cost metric(s) optimized was/were `<a-cost-metric>`
6. The tuning budget was `<the-budget>`
7. The tuning budget was the same for all tuned methods ✓✗
   If no, why not?
8. If the budget is given in time: the hardware used for all tuning runs is comparable ✓✗
9. All methods that were reported were tuned with this the methods and settings described above ✓✗
   If no, why not?
10. Tuning was done across $< n >$ tuning seeds which were: $[< 0 >, < 1 >, < 2 >, < 3 >, < 4 >]$
11. Testing was done across $< m >$ test seeds which were: $[< 5 >, < 6 >, < 7 >, < 8 >, < 9 >]$
12. Are all results reported on the test seeds? ✓✗
    If no, why not?
13. The final incumbent configurations reported were:
    `<algorithm-1-env-1>`:
    - `<a-hyperparameter>`: `<value>`
    - ...

    `<algorithm-1-env-2>`:
    - `<a-hyperparameter>`: `<value>`

- ...

<algorithm-2-env-1>:

- <a-hyperparameter>: <value>
- ...

14. The code for reproducing these experiments is available at: <a-link>
15. The code also includes the tuning process ✓✗
16. Bundled with the code is an exact version of the original software environment, e.g. a conda environment file with all package versions or a docker image in case some dependencies are not conda installable ✓✗
17. The following hardware was used in running the experiments:

- <n> <gpu-types>
- ...

B. Our AutoRL Hydra Sweepers

We provide implementations of DEHB and PBT variations to supplement existing options like Optuna (Akiba et al., 2019) or ray (Liaw et al., 2018) with state-of-the-art HPO tools that still have relatively high barriers of use, particularly in RL. Our goal was to provide a tuning option with as little human overhead as possible but as much flexibility for applying it to different RL codebases as possible. Hydra allows us to do both by using the tuning algorithms as sweepers that launch different configurations either locally or as parallel cluster jobs. In practice, this means minimal code changes are necessary to use our sweepers: the return value will need to be a cost metric and for PBT checkpointing and loading is mandatory. In this way, these plugins are compatible with any RL algorithm and environment.

Once these changes are implemented, a sweeper Hydra configuration that includes a search space definition can be used to run the whole optimization process in one go or resume existing runs (e.g. if the optimization was terminated accidentally or if more tuning budget becomes available after the fact). We include the option of using tuning seeds, which is so far uncommon except for CleanRL (Huang et al., 2021) where they are user specified. Furthermore, we extended the option of initial runs for PBT variations to the original PBT and PB2 instead of just BGT in order to stabilize those methods. In comparison to existing user-friendly tuners like Optuna, we provide different tuning algorithms that are not BO-based and include the option of using multi-fidelity tuning in hydra directly instead of having to implement a separate script.

Figure 7 shows an example Hydra configuration file turned into a ready-to-run tuning configuration file for tuning with DEHB. The corresponding configuration file, here for the full search space of PPO in StableBaselines3, is shown in Figure 8.

C. Additional Background on Tuning Methods Used

Since many in the RL community might be unfamiliar with the state of the art in AutoML and AutoRL, we provide brief descriptions of the RS, DEHB and PBT approaches we use in this paper.

C.1. Random Search

Random Search for hyperparameter optimization commonly refers to the method of sampling from a configuration space in a pseudo-random fashion (Bergstra & Bengio, 2012). The resulting configurations are then evaluated on full algorithm runs and the best performing one selected as incumbent. While RS is not as reliable with small budgets and larger search spaces as other tuning options, it has proven to be a better alternative to grid search due to its scaling properties. As Grid Search exhaustively evaluates all combinations of the given hyperparameter value set, it needs n^m algorithm evaluations, with n being the number of hyperparameters per dimension and m the number of dimensions in the search space. Still we only evaluate n values for each dimension, irrelevant of how important this dimension actually is. RS implicitly shifts more importance to the more relevant hyperparameters by varying the whole configuration at once, producing good results on smaller budgets. Furthermore, Grid Search relies entirely on the domain knowledge of the user since they provide all configurations. This is of course an issue with new methods, domains or if the optimal hyperparameter configuration falls outside of the norm.

```yaml
defaults:
  - algorithm: ppo
  - override hydra/launcher: submitit_slurm

hydra:
  run:
    dir: tuning_test_seeds/${env_name}_${hydra.job.override_dirname}/seed=${seed}
  job:
    config:
      override_dirname:
        exclude_keys:
          - seed
  launcher:
    partition: partition-name
    mem_gb: 10
    timeout_min: 1720
  sweep:
    dir: tuning_test_seeds/${env_name}_${hydra.job.override_dirname}/seed=${seed}

log_dir: ./agent_logs
env_name: Pendulum-v1
load: false
save: false
reward_curves: false
seed: 0
wandb: true
wandb_tags: [base]
logging_interval: 1e5
```

```yaml
defaults:
  - algorithm: ppo
  - search_space: ppo
  - override hydra/sweeper: DEHB
  - override hydra/launcher: submitit_slurm

hydra:
  sweeper:
    dehb_kwargs:
      mutation_factor: 0.2
      max_budget: ${algorithm.total_timesteps}
      min_budget: 1e3
      cost: "steps"
      wandb_project: autorl-benchmarks
      eta: 5
      seeds: [5,6,7,8,9]
    slurm: true
    slurm_timeout: ${hydra.launcher.timeout_min}
    search_space: ${search_space}
    total_brackets: 3
    budget_variable: algorithm.total_timesteps
    n_jobs: 1050
  run:
    dir: tuning_output/dehb_seed_${seed}_${algorithm.agent_class}_${env_name}_seeds_5
  launcher:
    partition: partition-name
    mem_gb: 10
    timeout_min: 1720
  sweep:
    dir: tuning_output/dehb_seed_${seed}_${algorithm.agent_class}_${env_name}_seeds_5

log_dir: ./agent_logs
env_name: Pendulum-v1
load: false
save: false
reward_curves: false
seed: 0
wandb: false
wandb_tags: [base]
logging_interval: 1e5
```

Figure 7: A base Hydra configuration file (left) and the changes necessary to tune this algorithm with DEHB (right).

```yaml
hyperparameters:
  algorithm.model_kwargs.learning_rate:
    type: uniform_float
    lower: 0.000001
    upper: 0.01
    log: true
  algorithm.model_kwargs.batch_size:
    type: categorical
    choices: [16, 32, 64, 128]
  algorithm.model_kwargs.n_steps:
    type: categorical
    choices: [256, 512, 1024, 2048, 4096]
  algorithm.model_kwargs.n_epochs:
    type: uniform_int
    lower: 5
    upper: 20
    log: false
  algorithm.model_kwargs.gae_lambda:
    type: uniform_float
    lower: 0.8
    upper: 0.9999
    log: false
  algorithm.model_kwargs.clip_range:
    type: uniform_float
    lower: 0.0
    upper: 0.5
    log: false
  algorithm.model_kwargs.clip_range_vf:
    type: uniform_float
    lower: 0.0
    upper: 0.5
    log: false
  algorithm.model_kwargs.normalize_advantage:
    type: categorical
    choices: [True, False]
  algorithm.model_kwargs.ent_coef:
    type: uniform_float
    lower: 0.0
    upper: 0.5
    log: false
  algorithm.model_kwargs.vf_coef:
    type: uniform_float
    lower: 0.0
    upper: 1.0
    log: false
  algorithm.model_kwargs.max_grad_norm:
    type: uniform_float
    lower: 0.0
    upper: 1.0
    log: false
```

Figure 8: Example definition of a search space for PPO in a separate configuraion file.

C.2. DEHB

DEHB is the combination of the evolutionary algorithm Differential Evolution (DE) (Storn & Price, 1997) and the multi-fidelity method HyperBand (Li et al., 2018). HyperBand as a multi-fidelity method is based on the idea of running many configurations with a small budget, i.e. only a fraction of training steps, and progressing promising ones to the next higher budget level. In this way we see many datapoints, but avoid spending time on bad configurations. DEHB starts with a full set of HyperBand budgets, from very low to full budget, and runs it in its first iteration. For each budget, DEHB runs the equivalent of one full algorithm run in steps, e.g. if the current budget is $\frac{1}{10}$ of the full run budget, 10 configurations will be evaluated. For the second one, the lowest budget is left out and the second lowest is initialised with a population of configurations evolved by DEHB from the previous iteration's results. This procedure continues until either a maximum number of iterations is reached or only the full run budget budget is left. The number of budgets is decided by a hyperparameter η.

In our experiments in Section 4 we run 3 iterations with $\eta = 5$ so only 3 budgets, and in our larger DEHB experiments in Section 5 we use 2 iterations with $\eta = 1.9$ so 8 budgets. We set the minimum budget as $\frac{1}{100}$ of the full run training steps in each case.

C.3. PBT Variants

PBT is based on the idea of maintaining a population of agents in parallel, each with its own hyperparameter configuration. These agents are then trained for n steps, after which their performance is evaluated and a checkpoint of their training state is created. Now a portion of the worst agents are replaced by the best ones and the rest of the configurations are refined for the next iteration. This will result in a hyperparameter schedule utilizing the best performing configurations at each iteration.

The original PBT (Li et al., 2019) randomly samples the initial configurations and then subsequently perturbes them by either randomly increasing or decreasing each hyperparameter by a constant factor. Categorical values are randomly resampled with a fixed probability.

This undirected sampling proved successful, but only with large population sizes upwards of 64 agents, therefore newer iterations of PBT often use a model to select new hyperparameter configurations, as e.g. PB2 uses Bayesian Optimization with a Gaussian Process (Parker-Holder et al., 2020). This enables optimization with a significantly smaller population size of as little as 4 agents.

As we have seen, however, the result can be volatile, therefore Wan et al. (2022) suggested two main extensions on top of PB2, in addition to the ability to tune the architecture with the PBT framwork, forming the current state of the art across PBT methods. The extensions are (1) the use of periodic kernel restarts in case no improvements are visible and (2) the use of full budget initial runs to warmstart the Gaussian Process with high quality datapoints.

In our experiments we use 20 configuration changes for each method with a population size of 8 for PB2 in Section 4 and $16/64$ for PB2 in Section 5. For BGT, we use 8 initial runs and a population size of 8 for the smaller budget and 48 and 16, respectively for the larger one. In both PB2 and BGT, we always replace the worst 12.5% of agents with the best 12.5%.

D. An Overview of Hyperparameter Configurations & Search Spaces

D.1. Stable Baselines Default Configurations

Table 3 shows the default hyperparameters we use throughout Section 4.

D.2. Stable Baseline Sweep Values

We sweep over the same hyperparameter values for each environment one dimension at a time. For PPO, these are `learning rate` $\in \{1e{-}2, 5e{-}3, 1e{-}3, 5e{-}4, 1e{-}4, 5e{-}5, 1e{-}5, 5e{-}6, 1e{-}6, 5e{-}7\}$, `entropy coefficient` $\in \{0.1, 0.05, 0.01, 0.005, 0.001\}$ and `clip range` $\in \{0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9\}$.

For SAC, `learning rate` $\in \{1e-2, 5e-3, 1e-3, 5e-4, 1e-4, 5e-5, 1e-5, 5e-6, 1e-6, 5e-7\}$, `tau` $\in \{1.0, 0.9, 0.8, 0.7, 0.6, 0.5, 0.4, 0.3, 0.2, 0.1\}$ and `training frequency` $\in \{1, 2, 4, 8, 16\}$.

For DQN, `learning rate` $\in \{1e-2, 5e-3, 1e-3, 5e-4, 1e-4, 5e-5, 1e-5, 5e-6, 1e-6, 5e-7\}$, `epsilon` $\in \{1.0, 0.9, 0.8, 0.7, 0.6, 0.5, 0.4, 0.3, 0.2, 0.1\}$ and `training frequency` $\in \{1, 2, 4, 8, 16\}$.

		Acrobot & Pendulum	Brax	MiniGrid
PPO	Policy Class	`MlpPolicy`	`MlpPolicy`	`CnnPolicy`
	leaning_rate	1e-3	1e-4	7e-4
	batch_size	64	512	64
	gamma	0.9	0.99	0.999
	n_steps	1024	1024	256
	n_epochs	10	16	4
	gae_lambda	0.95	0.96	0.95
	clip_range	0.2	0.2	0.2
	clip_range_vf	null	null	null
	normalize_advantage	`True`	`True`	`True`
	ent_coef	0.01	0.01	0.01
	vf_coef	0.5	0.5	0.5
	max_grad_norm	0.5	0.5	0.5
	use_sde	`False`	`False`	`False`
	sde_sample_freq	4	4	4
SAC	Policy Class	`MlpPolicy`	`MlpPolicy`	
	leaning_rate	1e-4	1e-4	
	batch_size	256	512	
	gamma	0.99	0.99	
	tau	1.0	1.0	
	learning_starts	100	100	
	buffer_size	1000000	1000000	
	train_freq	1	1	
	gradient_steps	1	1	
	use_sde	`False`	`False`	
	sde_sample_freq	-1	-1	
DQN	Policy Class	`MlpPolicy`		`CnnPolicy`
	learning_rate	1e-3		5e-7
	batch_size	64		64
	tau	1.0		1.0
	gamma	0.9		0.999
	learning_starts	50000		100
	train_freq	4		4
	gradient_steps	1		1
	exploration_fraction	0.1		0.1
	exploration_initial_eps	1.0		1.0
	exploration_final_eps	0.05		0.05
	buffer_size	1000000		1000000

Table 3: StableBaselines hyperparameter defaults for different environments.

	Hyperparameter	Full Space	Small Space	LR Only
PPO	leaning_rate	log(interval(1e-6, 0.1))	log(interval(1e-6, 0.1))	log(interval(1e-6, 0.1))
	ent_coef	interval(0.0, 0.5)	interval(0.0, 0.5)	
	n_epochs	range[5,20]	range[5,20]	
	batch_size	{16, 32, 64, 128}		
	n_steps	{256, 512, 1024, 2048, 4096}		
	gae_lambda	interval(0.8, 0.9999)		
	clip_range	interval(0.0, 0.5)		
	clip_range_vf	interval(0.0, 0.5)		
	normalize_advantage	{True, False}		
	vf_coef	interval(0.0, 1.0)		
	max_grad_norm	interval(0.0, 1.0)		
SAC	leaning_rate	log(interval(1e-6, 0.1))	log(interval(1e-6, 0.1))	log(interval(1e-6, 0.1))
	train_freq	range[1,1e3]	range[1,1e3]	
	tau	interval(0.01, 1.0)	interval(0.01, 1.0)	
	batch_size	{64, 128, 256, 512}		
	learning_starts	range[0,1e4]		
	buffer_size	range[5e3,5e7]		
	gradient_steps	range[1,10]		
DQN	learning_rate	log(interval(1e-6, 0.1))	log(interval(1e-6, 0.1))	log(interval(1e-6, 0.1))
	batch_size	{4, 8, 16, 32}	{4, 8, 16, 32}	
	exploration_fraction	interval(0.005, 0.5)	interval(0.005, 0.5)	
	learning_starts	range[0,1e4]		
	train_freq	range[1,1e3]		
	gradient_steps	range[1,10]		
	exploration_initial_eps	interval(0.5, 1.0)		
	exploration_final_eps	interval(0.001, 0.2)		
	buffer_size	range[5e3,5e7]		

Table 4: StableBaselines search spaces.

D.3. Stable Baselines Search Spaces

Table 4 shows the search spaces we used for the experiments in Section 4. The search spaces are the same for all tuning methods and across environments. We denote floating point intervals as `interval(lower,upper)`, integer ranges as `range[lower,upper]`, categorical choices as {`choice 1, choice 2`} and add `log` if the search space for this hyperparameter is traversed logarithmically.

D.4. Brax Experiment Settings

We base our implementations the training code provided with Brax with minor additions like only a single final evaluation and agent loading. The GitHub commit ID for the code version we use is `3843d433050a08cb492c301e039e04409b3557fc`. The cost metric we optimize is the evaluation reward across one episode of the environment batch. We tune on seeds $0-4$ and evaluate on seeds $5-14$. The baseline hyperparameters are taken from this commit as well and are shown in 5 together with our search space.

For both Procgen and Brax, we compute the rank of a method as follows: the best performing method on the test seeds and all other methods within its standard deviation receive rank one. The method with the next best mean (and all methods in its standard deviation) receive the next free rank – 2 in case there was a single best method, 3 if there were two and so on. These ranks are determined for each environment from which we can compute a mean across the whole domain.

Hyperparameter	Search Space	Defaults
leaning_rate	log(interval(1e-6, 0.1))	3e-4
num_update_epochs	range[1,15]	4
batch_size	{128,256,512,1024,2048}	1024
num_minibatches	range[0,7]	6
entropy_cost	interval(0.0001, 0.5)	1e-2
gae_lambda	interval(0.5, 0.9999)	0.95
epsilon	interval(0.01, 0.9)	0.3
vf_coef	interval(0.01, 0.9)	0.5
reward_scaling	interval(0.01, 1.0)	0.1

Table 5: Search Space and baseline hyperparameters for Brax. Actual number of minibatches is $2^{num\_minibatches}$. Epsilon refers to the clip range in this implementation.

Hyperparameter	Search Space	Bigfish	Climber	Plunder
lr	log(interval(1e-6, 0.1))	5e-4	5e-4	5e-4
eps	log(interval(1e-6, 0.1))	1e-5	1e-5	1e-5
hidden_size	-	256	256	256
clip_param	interval(0.0, 0.5)	0.2	0.2	0.2
num_mini_batch	-	8	8	8
ppo_epoch	range[1, 5]	3	3	3
num_steps	-	256	256	256
max_grad_norm	interval(0.0, 1.0)	0.5	0.5	0.5
value_loss_coef	interval(0.0, 1.0)	0.5	0.5	0.5
entropy_coef	interval(0.0, 0.5)	0.01	0.01	0.01
gae_lambda	interval(0.8, 0.9999)	0.95	0.95	0.95
gamma	-	0.999	0.999	0.999
alpha	interval(0.8, 0.9999)	0.99	0.99	0.99
clf_hidden_size	-	4	64	4
order_loss_coef	interval(0.0, 0.1)	0.01	0.001	0.1
use_nonlinear_clf	{`True`, `False`}	`True`	`True`	`False`
adv_loss_coef	interval(0.0, 1.0)	0.05	0.25	0.3
value_freq	range[1, 5]	32	1	8
value_epoch	range[1, 10]	9	9	1

Table 6: Search Space and baselines hyperparameters for IDAAC on Procgen.

D.5. Procgen Experiment Settings

We use the open-source code provided by (Raileanu & Fergus, 2021) with minor additions like loading agents. The GitHub commit ID for the code version we use is `2fe30202942898b1b09d76e5d8c71d5a7db3686b`. The cost metric we optimize is the evaluation reward across ten episodes of the training environment. We tune on seeds $0-4$ and evaluate on seeds $5-14$. Our baseline are the provided best hyperparameters per environment (see 6 for the configuration and our search space).

D.6. Hardware

All of our experiments were run on a compute cluster with two Intel CPUs per node (these were used for the experiments in Section 4) and four different node configurations for GPU (used for the experiments in Section 5). These configurations are: 2 Pascal 130 GPUs, 2 Pascal 144 GPUs, 8 Volta 10 GPUs or 8 Volta 332. We ran the CPU experiments with 10GB of memory on single nodes and the GPU experiments with 10GB for Procgen and 40GB for Brax on a single GPU each.

	Hyperparameter	Default	DEHB	PB2	RS
PPO Acrobot	leaning_rate	1e-3	5e-05	5e-3 for 90e5 steps, then 3e-6	1e-6
	batch_size	64	64	64 for 90e5 steps, then 128	64
	n_steps	1024	1024	256	256
	n_epochs	10	14	8 for 90e5 steps, then 20	6
	gae_lambda	0.95	0.82	0.85 for 90e5 steps, then 0.82	0.81
	clip_range	0.2	0.14	0.06 for 90e5 steps, then 0.27	0.05
	clip_range_vf	null	0.43	0.06	0.11
	normalize_advantage	`True`	`True`	`True`	
	ent_coef	0.01	0.21	0.42 for 90e5 steps, then 0.29	0.01
	vf_coef	0.5	0.02	0.5 for 90e5 steps, then 0.7	0.07
	max_grad_norm	0.5	0.75	0.24 for 90e5 steps, then 0.5	0.97

Table 7: StableBaselines hyperparameter defaults for different environments.

E. Details on the Tuned Configurations

We want to give some insights into how much the incumbents of our HPO methods differ from the baselines and one another. We show an example comparisons between different incumbent configurations on the full search space of PPO on Acrobot in Table 7. This result is consistent with what we find across other algorithms and environments: the differences between incumbents as well as between incumbents and baseline are fairly large. The result of HPO in our experiment has not meant small changes to only a subset of the search space, but usually significant deviations from the baseline in most of them. Still, we can see some similarities at times, in this case the batch size stays consistently at 64 across all configurations (with the exception of the final training phase of PB2). We can also see common trends among the incumbents at times, e.g. in the value of the GAE λ which is between 0.81 and 0.85 for the incumbents, but at 0.95 in the default configuration. Unfortunately, the other hyperparameter values do not seem to share any trends and often have significantly different values as e.g. in the entropy coefficient which varies between 0.01 and 0.42.

Why do we then see such similar performance from all of these configurations? We believe three main factors are at play: hyperparameter importance, the algorithm's sensitivity to a hyperparameter value and interaction effects between hyperparameters. It is likely that not all hyperparameters are crucial to optimize in this setting, so seeing very different values for unimportant hyperparameters can make the configurations appear more different than they are. We know from our experiments, however, that a mistake in the entropy coefficient can be highly damaging to the algorithm's performance in Acrobot (see Figure 21 below). Comparing the entropy coefficient curve of Acrobot and Ant in this figure, however, reveals that the median performance across different entropy coefficients degrades much less quickly for Acrobot than for Ant – on Acrobot, PPO is less sensitive to changes in hyperparameter values. To put it another way: hyperparameter values may look different between configurations but result in the same algorithm behaviour as long as they are within a similar range. Lastly, since we optimize many hyperparameters and the agent's behaviour depends on these hyperparameters, it is possible that hyperparameters interact with each other to produce a similar outcome as long as their relation stays similar. This could be an explanation for combining lower learning rates with more update epochs as DEHB does. Analysing hyperparameter configurations on their own, however, will not provide enough information to determine each of these factors for each hyperparameter. They have to be explored through separate experiments first before we can draw conclusions on how similar the configurations we found in HPO tools are and what that means for optimal hyperparameter values in our settings.

F. Tuning Results on Brax & Procgen in Tabular Form

For ease of comparison, we provide the results of Section 5 in tabular form.

Table 8: Tuning PPO on Brax's Ant, Halfcheetah and Humanoid environments. Shown are tuning results across 3 runs across 5 seeds each, tested on 10 different test seeds.

	Ant	Halfcheetah	Humanoid
Baseline	3448 ± 343	6904 ± 377	$\mathbf{3235 \pm 758}$
DEHB Inc.	5745 ± 878	3993 ± 871	1788 ± 718
DEHB Test	$\mathbf{4288 \pm 1017}$	4928 ± 500	$\mathbf{3167 \pm 874}$
RS Inc.	2515 ± 1750	2978 ± 1007	763 ± 317
RS Test	$\mathbf{5165 \pm 896}$	3646 ± 699	753 ± 209
DEHB Inc. (64)	7170 ± 1045	8202 ± 445	4338 ± 1655
DEHB Test (64)	$\mathbf{4696 \pm 1252}$	$\mathbf{8039 \pm 636}$	$\mathbf{5205 \pm 2781}$
BGT Inc. (64)	1119 ± 1321	1051 ± 752	434 ± 15
BGT Test (64)	3196 ± 3307	456 ± 461	132 ± 0
RS Inc. (64)	6344 ± 654	7891 ± 386	2932 ± 798
RS Test (64)	669 ± 2447	950 ± 1461	325 ± 162

Table 9: Tuning IDAAC on Procgen's Bigfish, Climber and Plunder. Results are across 3 runs using 5 seeds each, and tested on 10 different test seeds.

	Bigfish	Climber	Plunder
Baseline	6.8 ± 3.2	$\mathbf{4.1 \pm 1.4}$	$\mathbf{11.8 \pm 5.5}$
DEHB Inc.	7.3 ± 2.0	3.7 ± 0.2	5.8 ± 0.2
DEHB Test	$\mathbf{11.9 \pm 4.3}$	$\mathbf{2.7 \pm 1.5}$	$\mathbf{8.6 \pm 2.6}$
BGT Inc.	1.3 ± 0.2	2.5 ± 0.4	4.5 ± 0.3
BGT Test	0.9 ± 0.4	2.4 ± 1.1	5.3 ± 1.0
PB2 Inc.	26.1 ± 2.2	3.2 ± 0.3	4.7 ± 0.3
PB2 Test	3.4 ± 1.9	2.6 ± 1.1	$\mathbf{8.3 \pm 2.7}$
RS Inc.	4.4 ± 2.1	5.5 ± 0.6	6.2 ± 1.3
RS Test	7.4 ± 5.0	2.6 ± 1.0	4.7 ± 1.0
DEHB Inc. (64 runs)	11.5 ± 0.7	6.0 ± 1.0	7.3 ± 0.9
DEHB Test (64 runs)	$\mathbf{9.4 \pm 2.5}$	$\mathbf{3.9 \pm 1.9}$	$\mathbf{8.7 \pm 0.7}$
BGT Inc. (64 runs)	4.7 ± 5.1	3.2 ± 1.5	5.3 ± 0.1
BGT Test (64 runs)	2.1 ± 1.9	2.6 ± 0.4	5.9 ± 1.6
PB2 Inc. (64 runs)	10.5 ± 10.1	3.4 ± 0.3	5.2 ± 0.2
PB2 Test (64 runs)	2.1 ± 1.1	3.0 ± 0.9	4.3 ± 0.2
RS Inc. (64 runs)	1.9 ± 0.3	5.9 ± 0.8	6.7 ± 0.4
RS Test (64 runs)	1.1 ± 0.1	2.3 ± 0.7	3.6 ± 0.5

G. Hyperparameter Sweeps for PPO, DQN and SAC

The full set of PPO sweeps can be found in Figures 9, 10, 11 and 12, the SAC sweeps in Figures 15 and 16 and the DQN sweeps in Figures 13 and 10.

G.1. PPO Sweeps

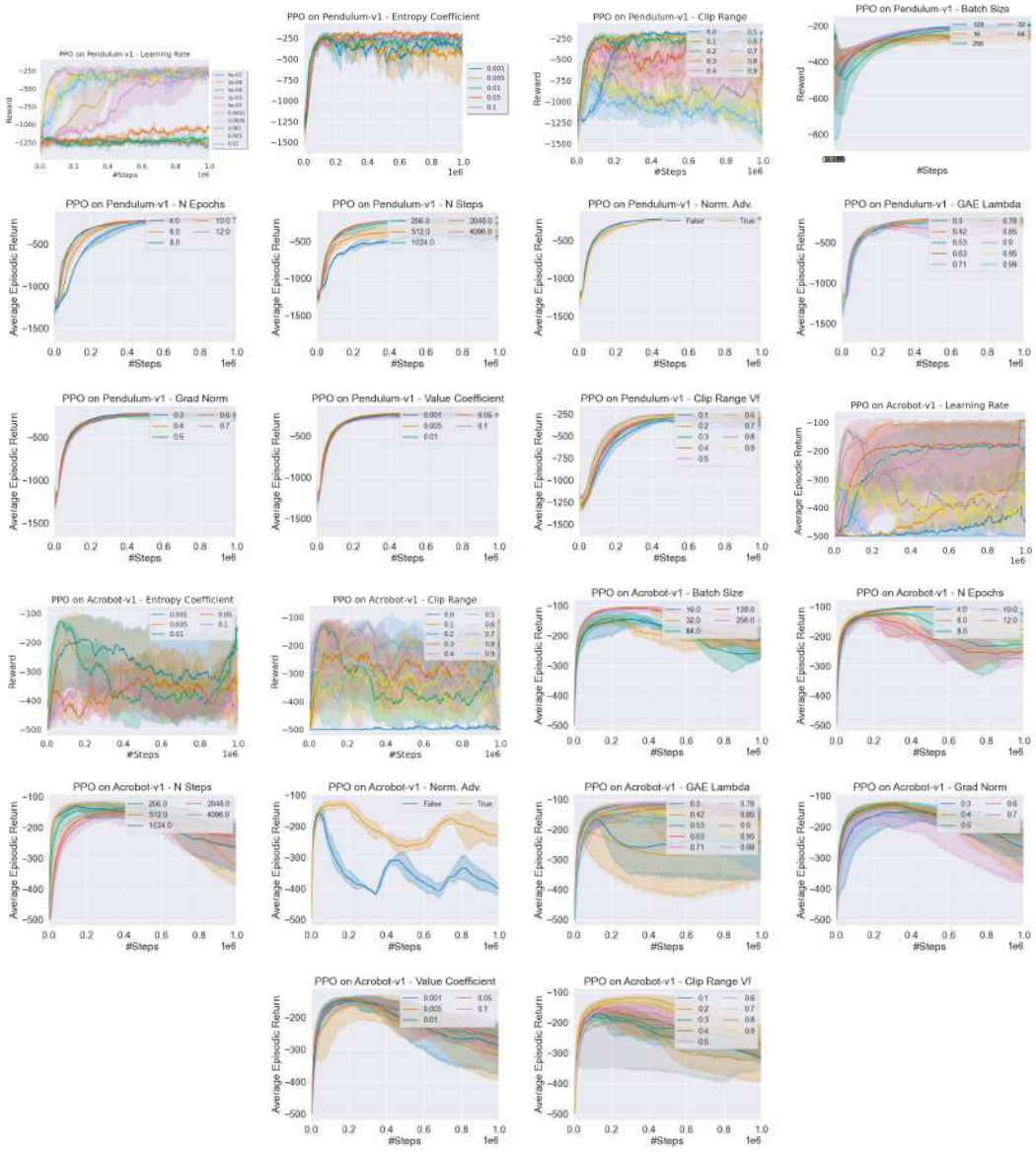


Figure 9: Hyperparameter Sweeps for PPO on Pendulum and Acrobot.

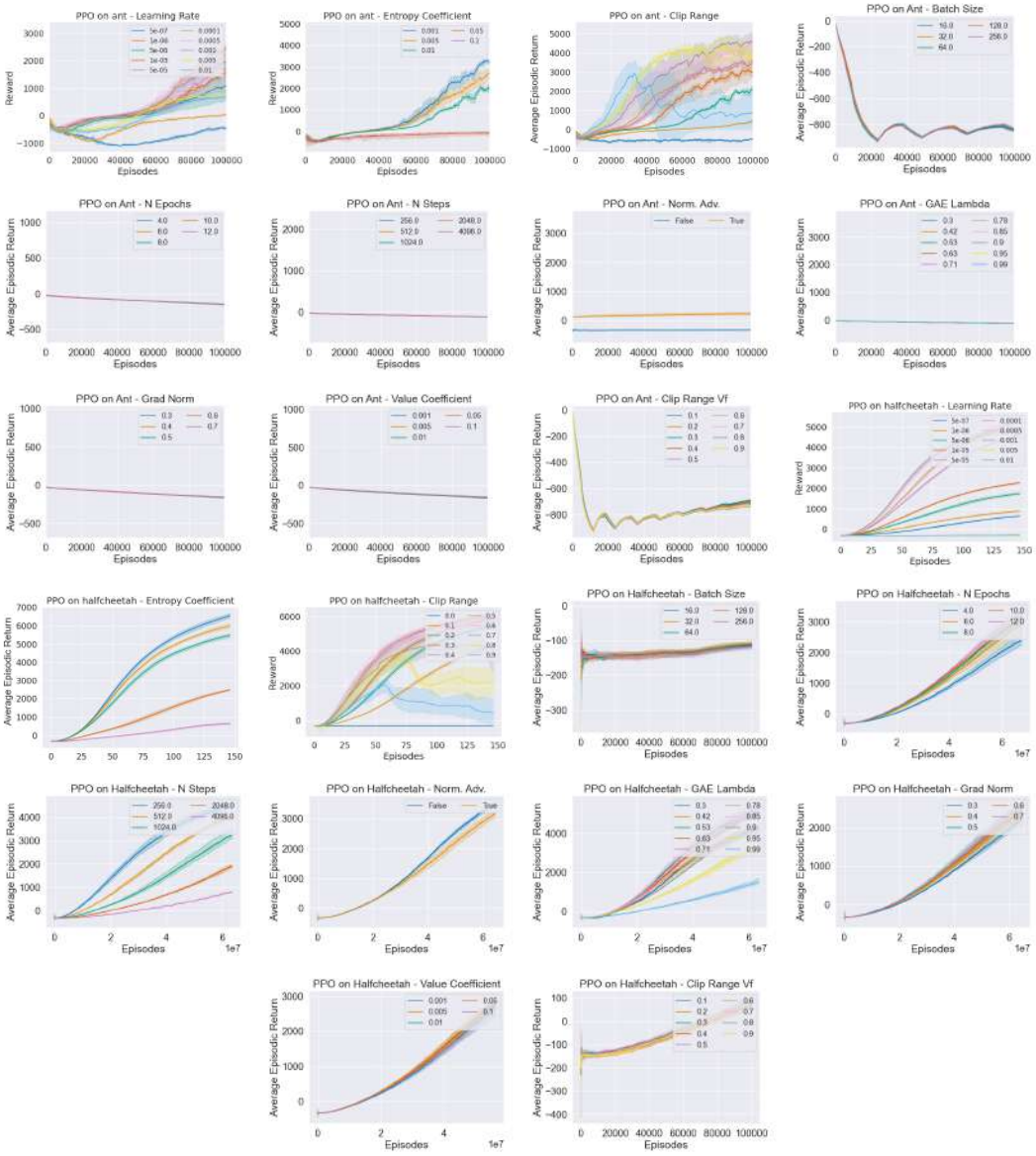


Figure 10: Hyperparameter Sweeps for PPO on Ant and Halfcheetah.

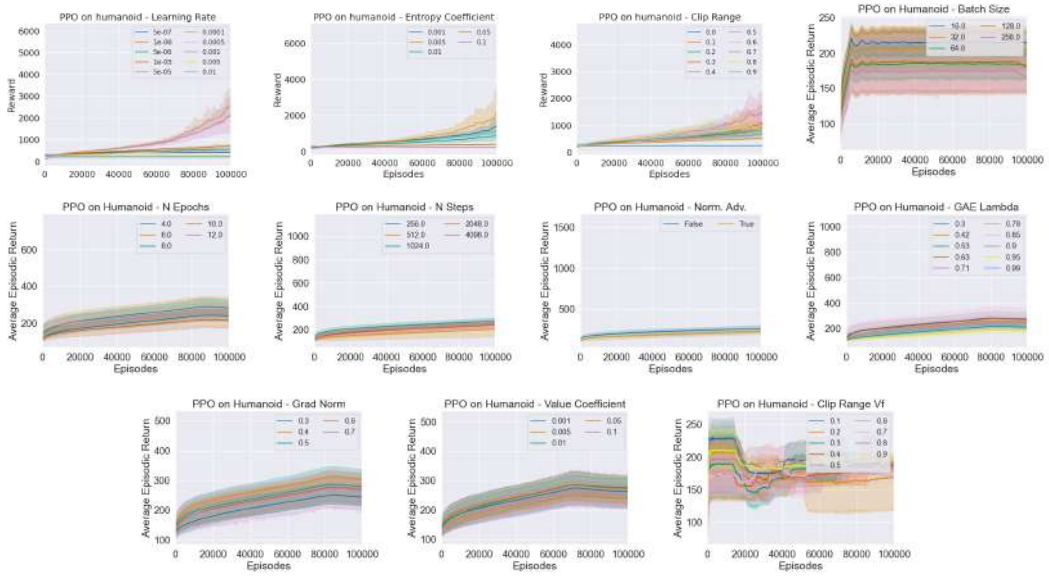


Figure 11: Hyperparameter Sweeps for PPO on Humanoid.

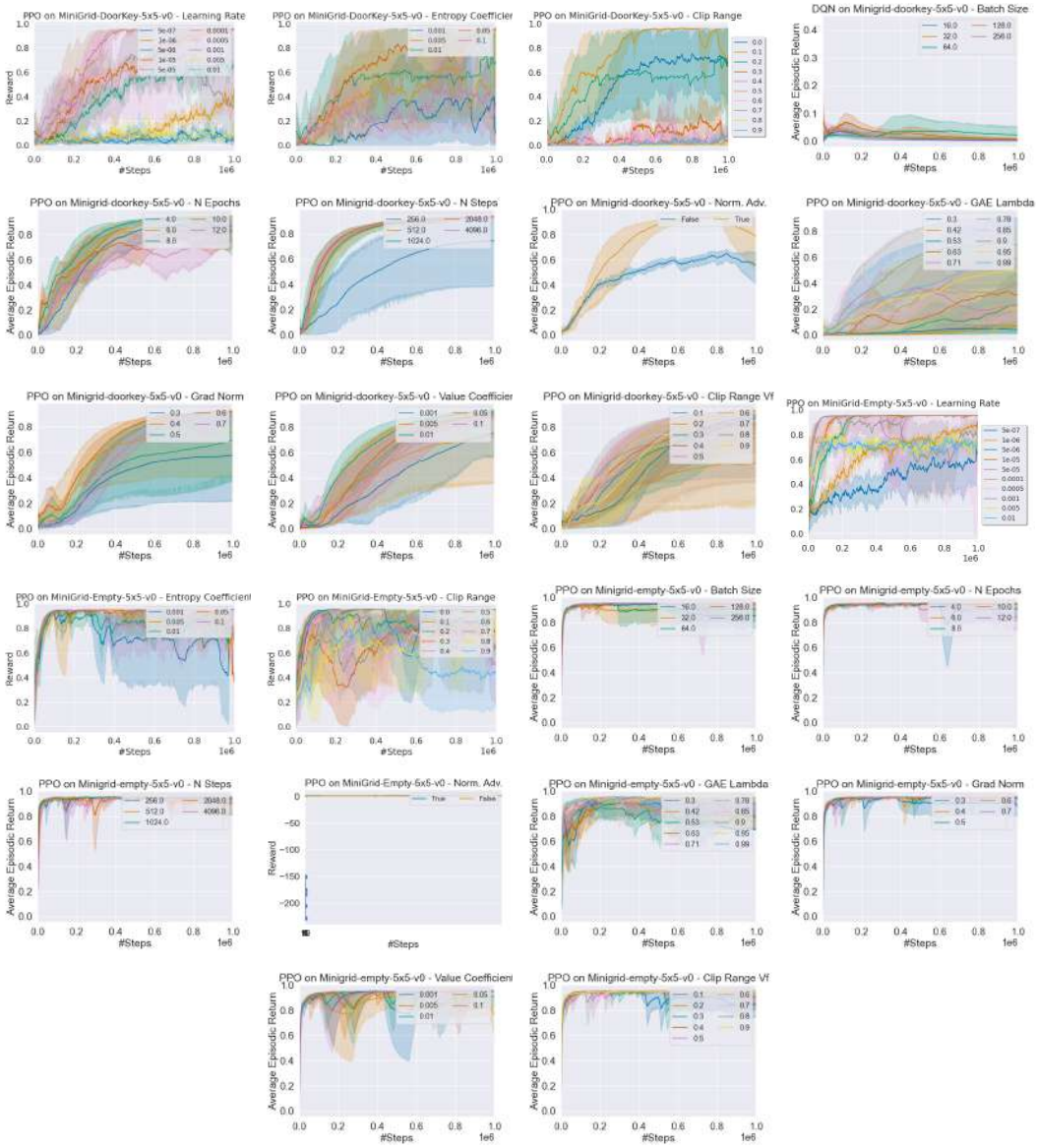


Figure 12: Hyperparameter Sweeps for PPO on MiniGrid.

G.2. DQN Sweeps

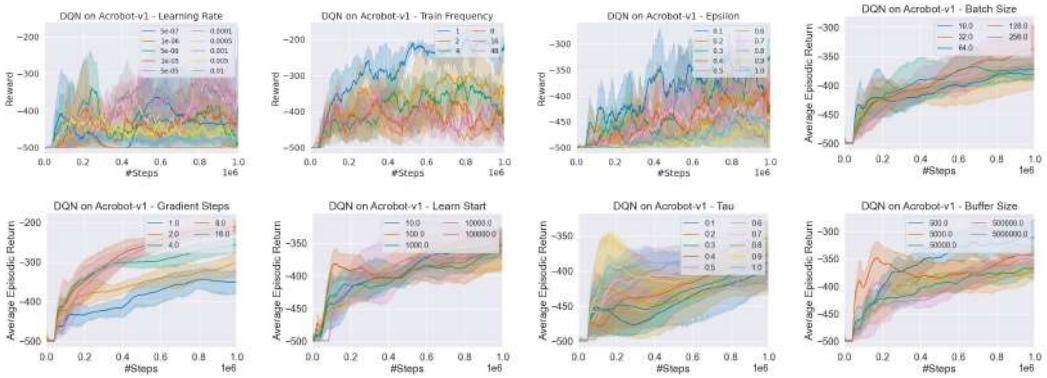


Figure 13: Hyperparameter Sweeps for DQN on Acrobot.

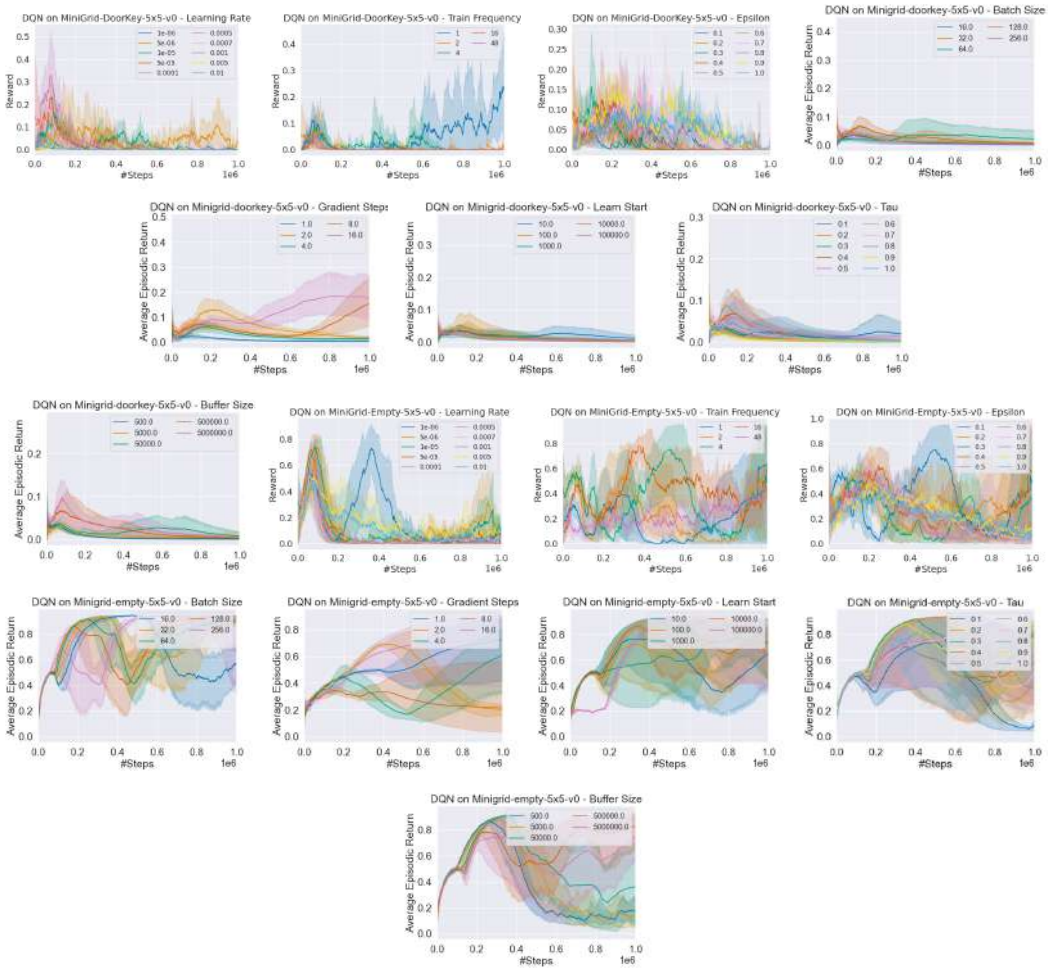


Figure 14: Hyperparameter Sweeps for DQN on MiniGrid.

G.3. SAC Sweeps

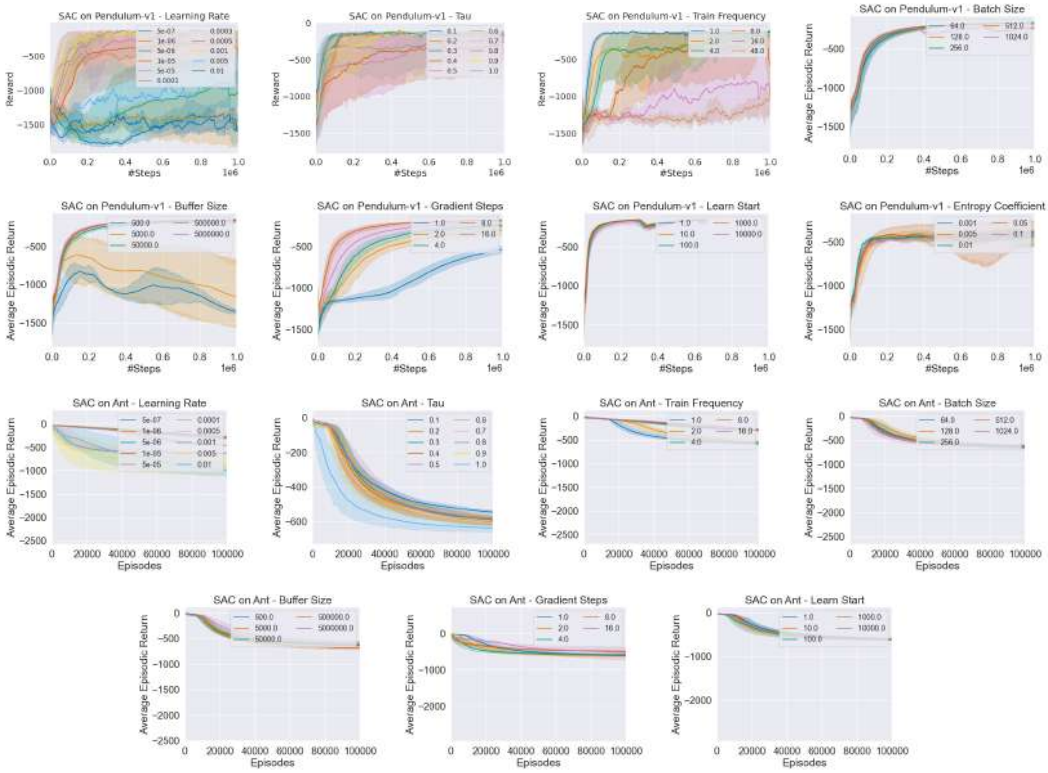


Figure 15: Hyperparameter Sweeps for SAC on Pendulum and Ant.

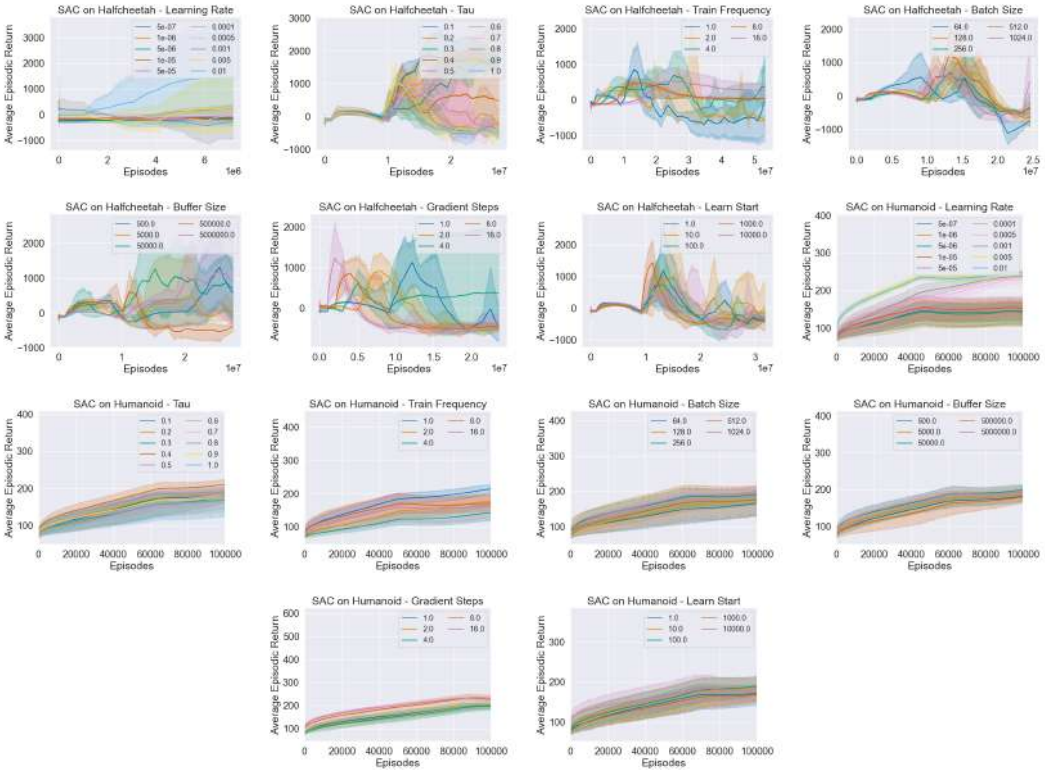


Figure 16: Hyperparameter Sweeps for SAC on Halfcheetah and Humanoid.

H. Full Performance Pointplots

The full set of SAC pointplots can be found in Figures 17 and 18, the DQN pointplots in Figures 19 and 20 and the PPO pointplots in Figures 21, 22, 23 and 24.

H.1. SAC Pointplots

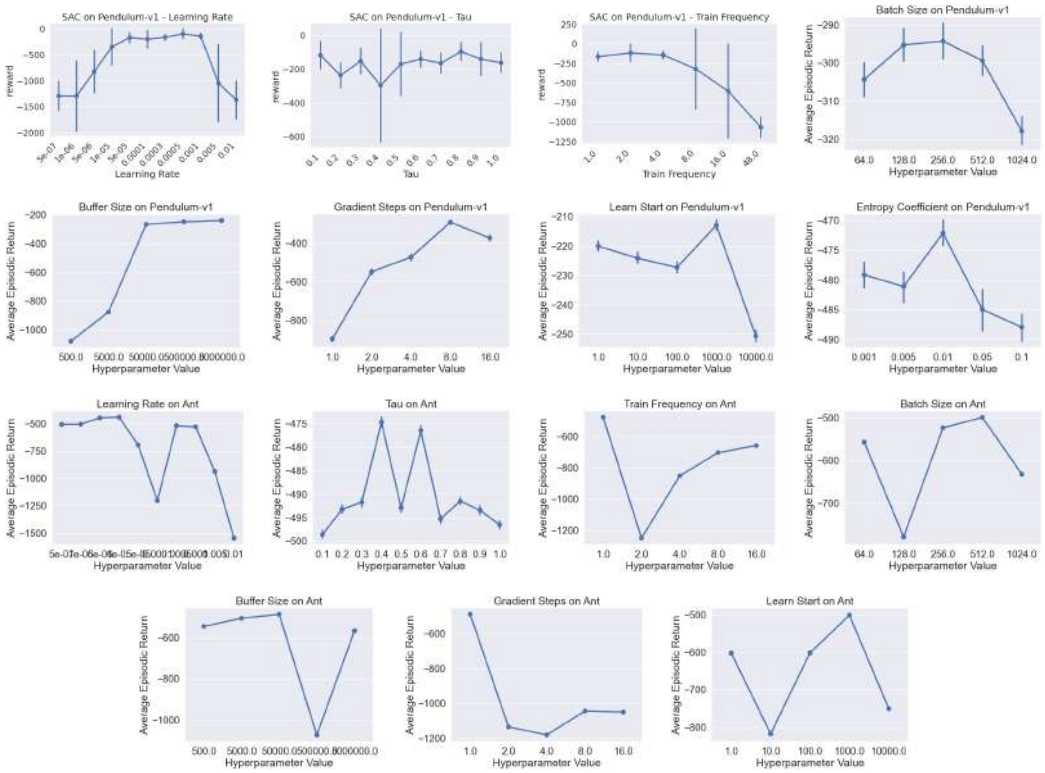


Figure 17: Final returns across 5 seeds for different hp variations of SAC on Pendulum and Ant.

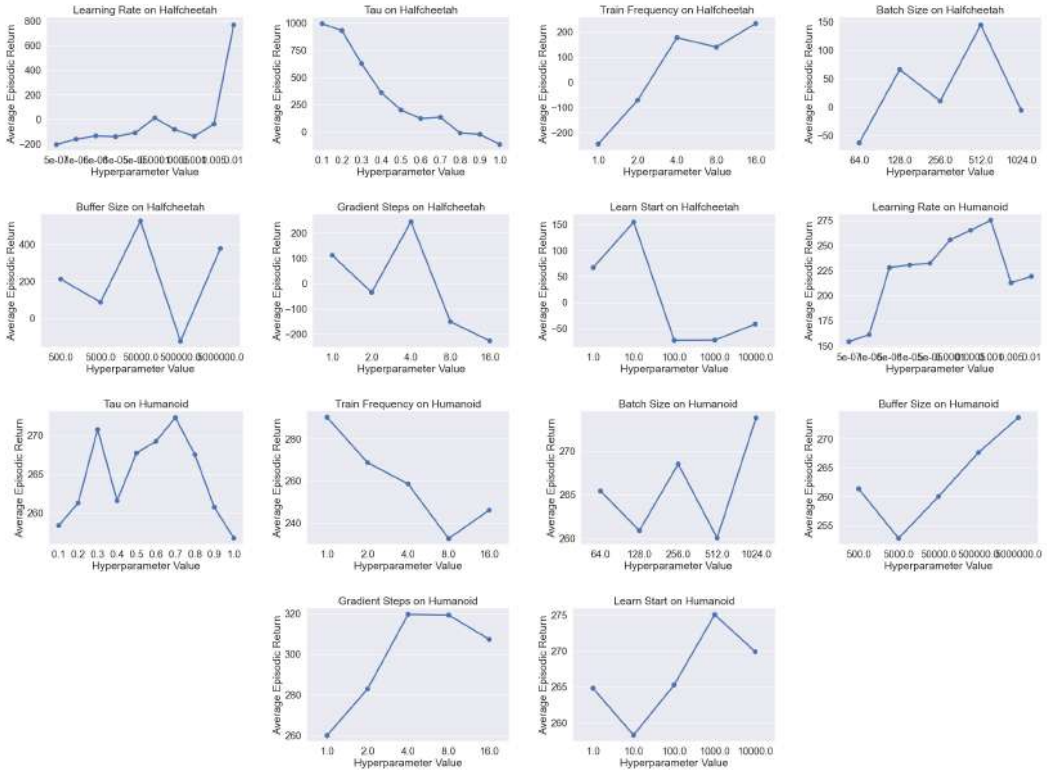


Figure 18: Final returns across 5 seeds for different hp variations of SAC of Halfcheetah and Humanoid.

H.2. DQN Pointplots

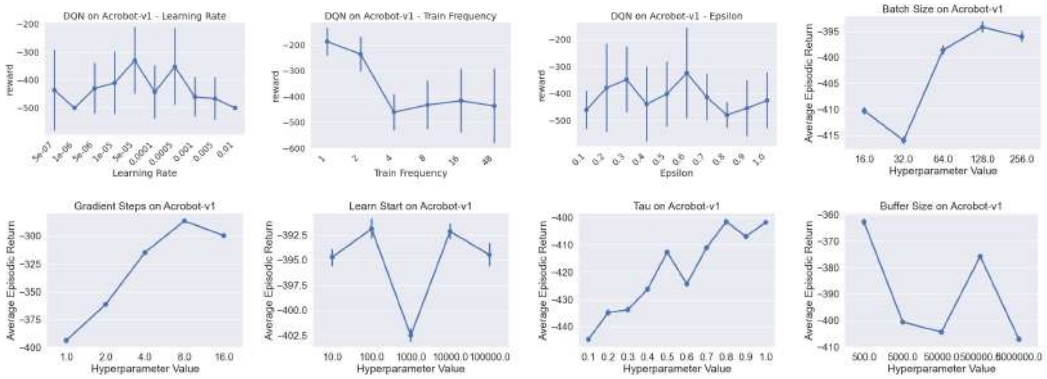


Figure 19: Final returns across 5 seeds for different hp variations of DQN on Acrobot.

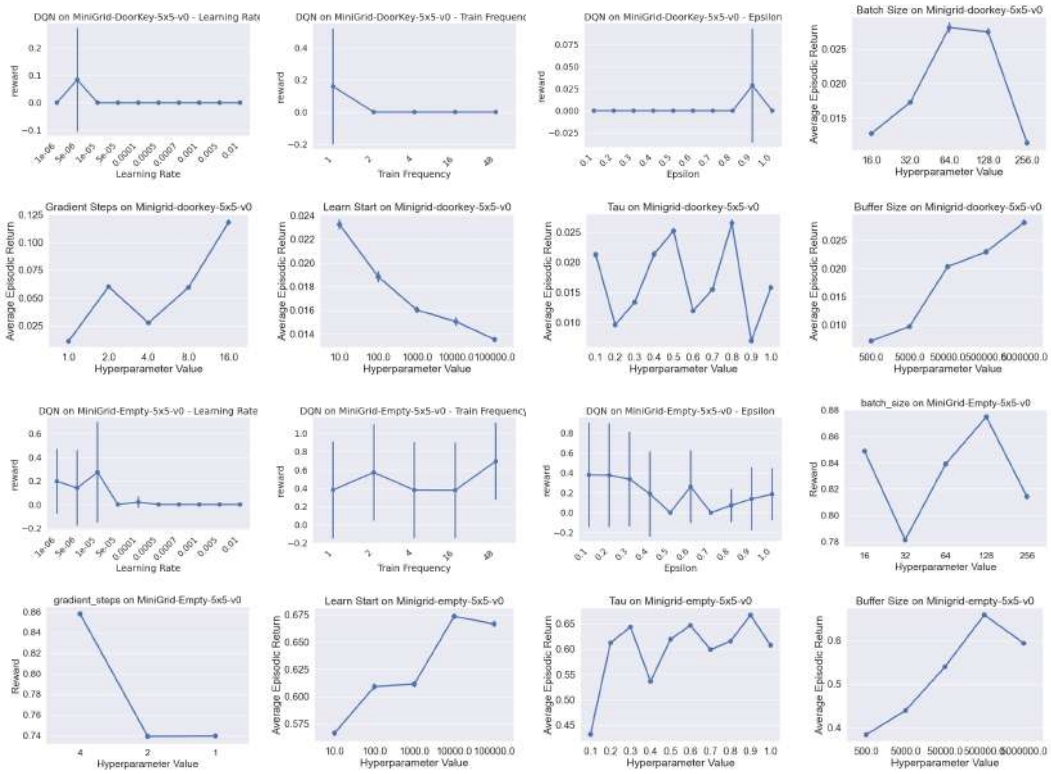


Figure 20: Final returns across 5 seeds for different hp variations of DQN on MiniGrid.

H.3. PPO Pointplots

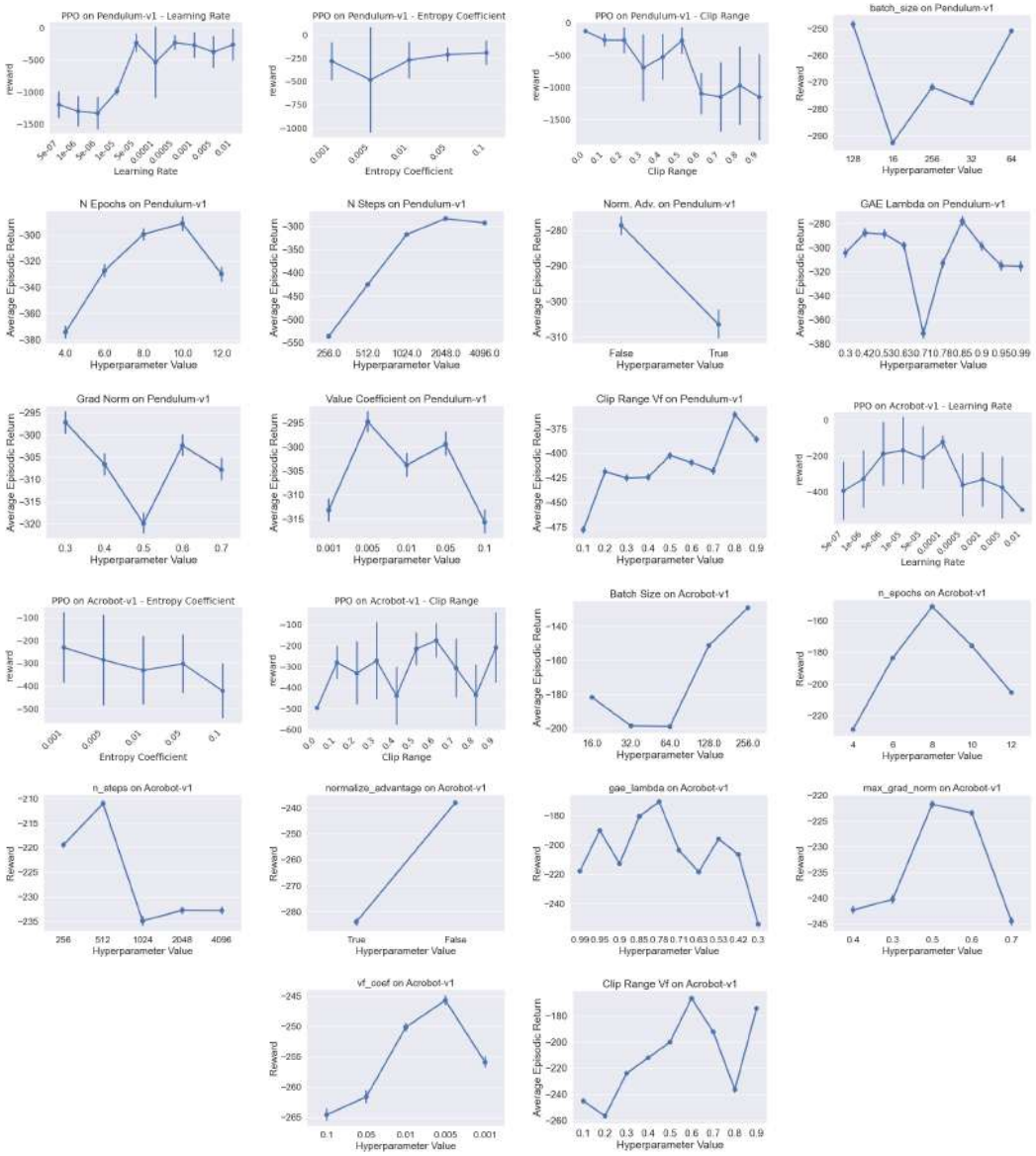


Figure 21: Final returns across 5 seeds for different hp variations of PPO on Pendulum and Acrobot.

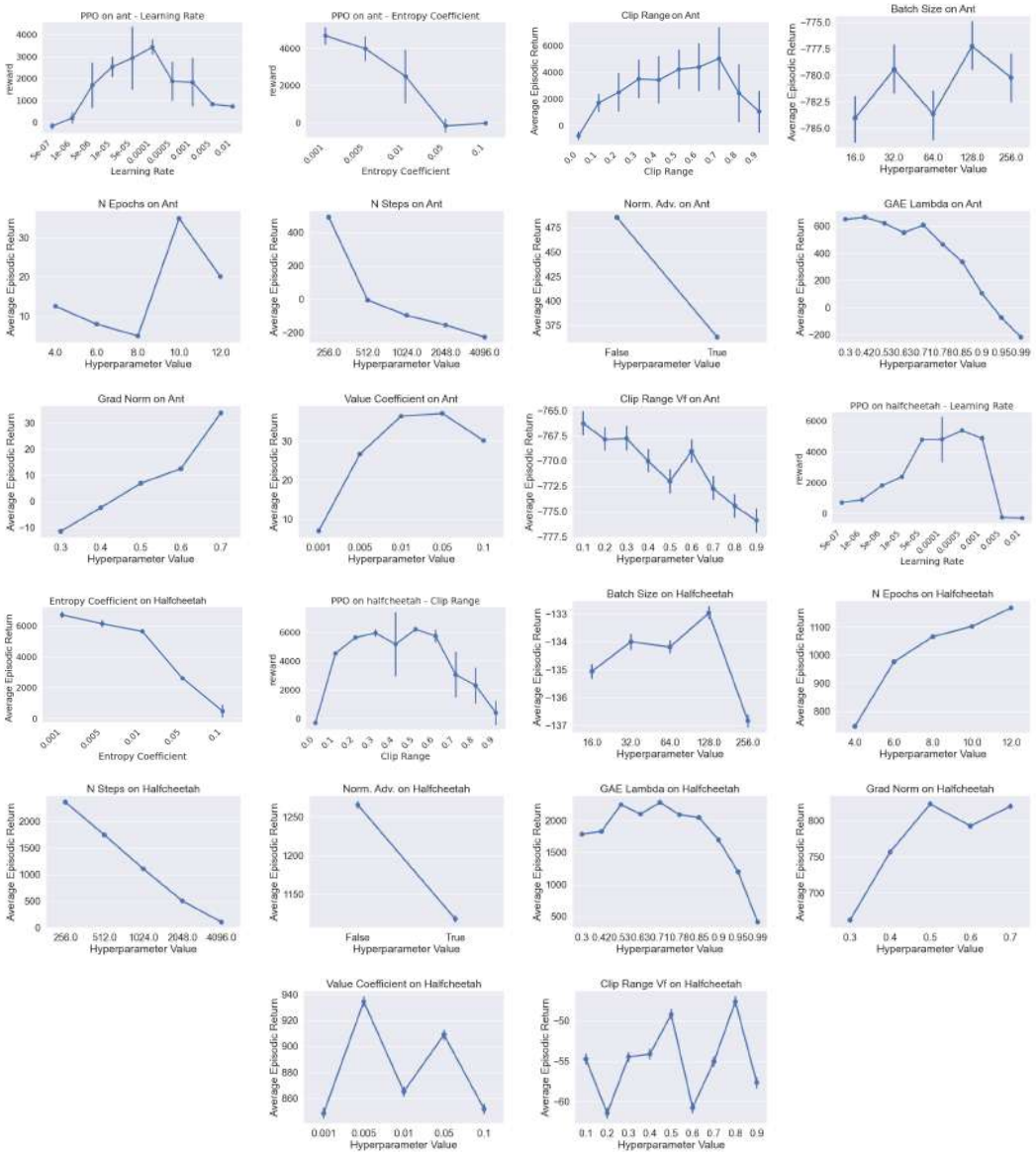


Figure 22: Final returns across 5 seeds for different hp variations of PPO on Ant and Halfcheetah.

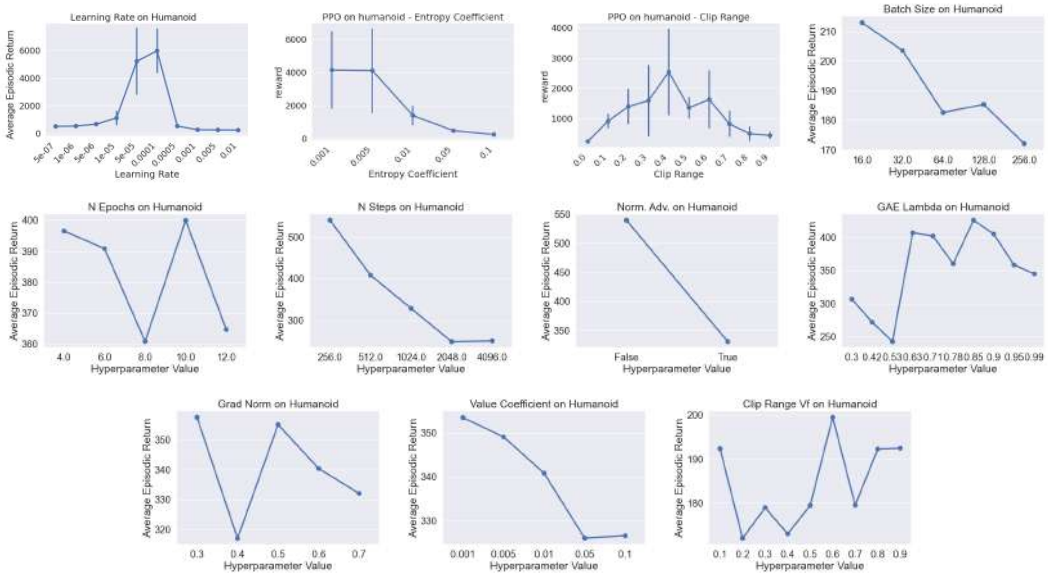


Figure 23: Final returns across 5 seeds for different hp variations of PPO on Humanoid.

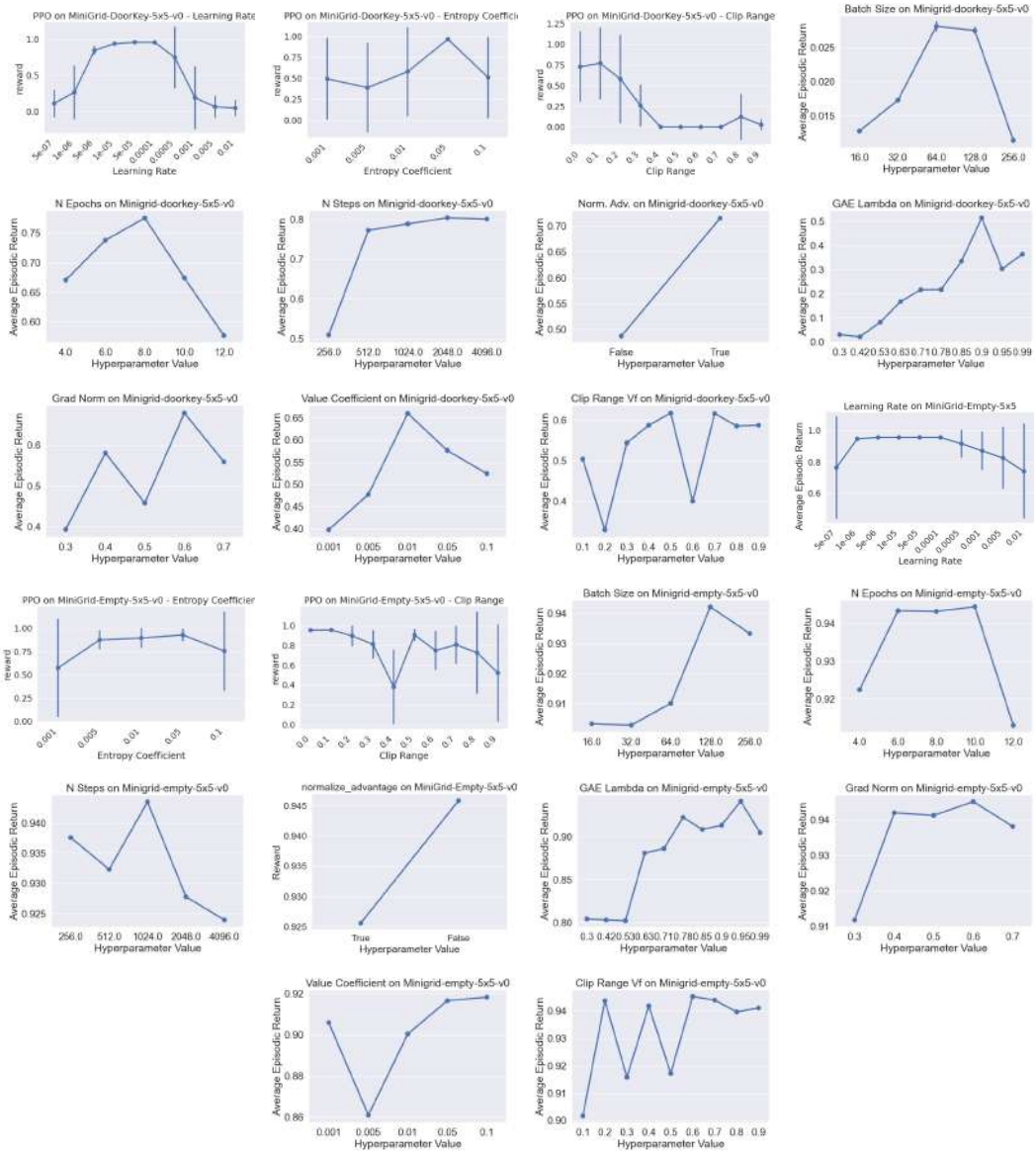


Figure 24: Final returns across 5 seeds for different hp variations of PPO on MiniGrid.

I. Hyperparameter Importances using fANOVA

These hyperparameter importance plots were made using the fANOVA (Hutter et al., 2014) plugin of DeepCAVE (Sass et al., 2022).

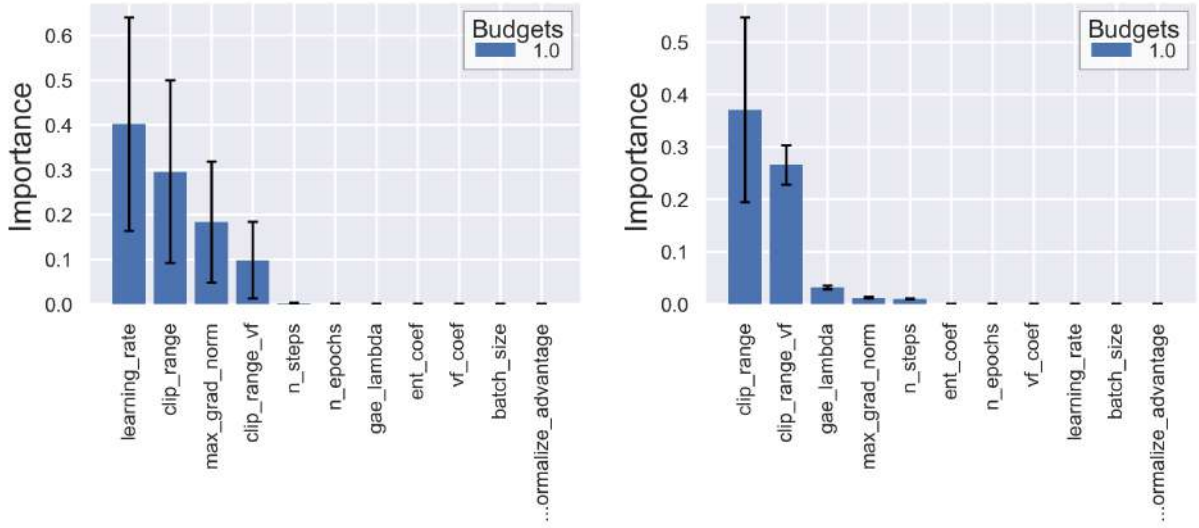


Figure 25: PPO Hyperparameter Importances on Acrobot (left) and Pendulum (right).

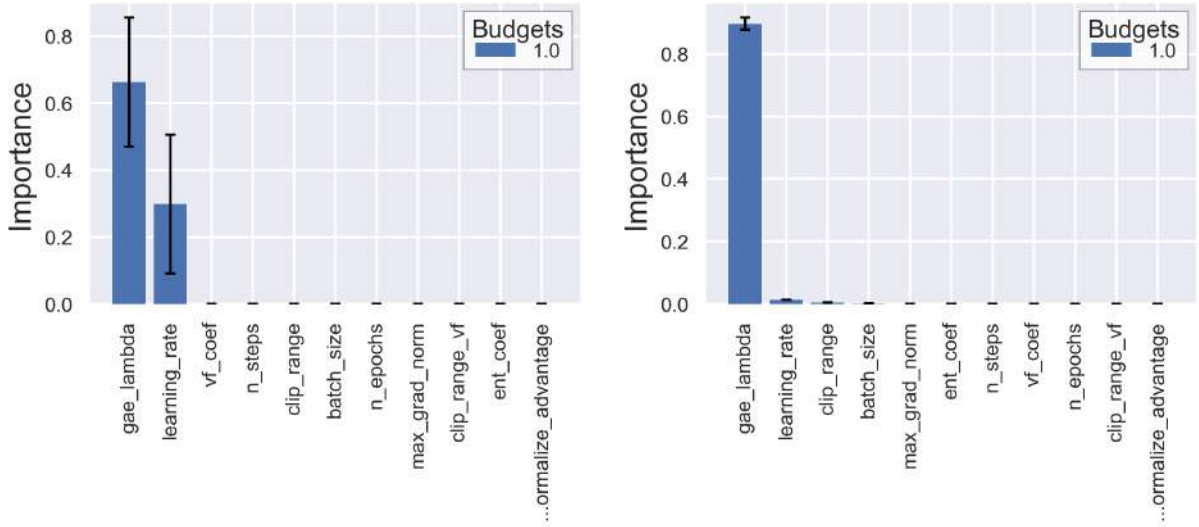


Figure 26: PPO Hyperparameter Importances on MiniGrid Empty (left) and Minigrid DoorKey(right).

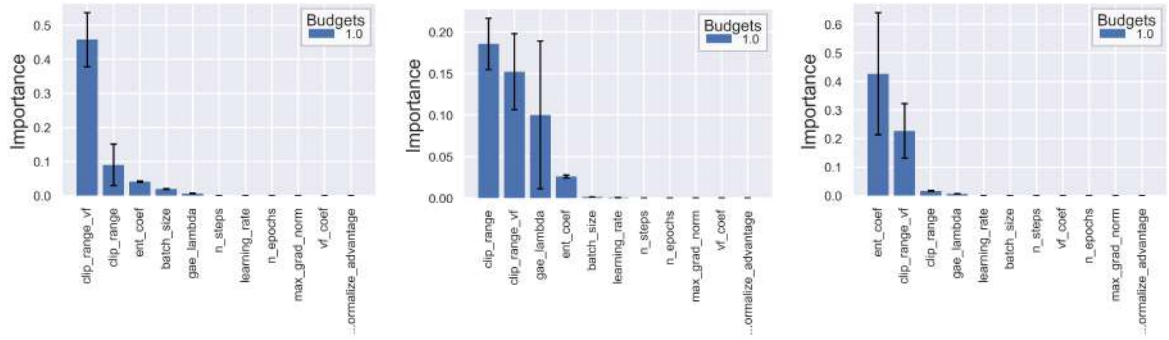


Figure 27: PPO Hyperparameter Importances on Brax Ant (left), Halfcheetah (middle) and Humanoid (right).

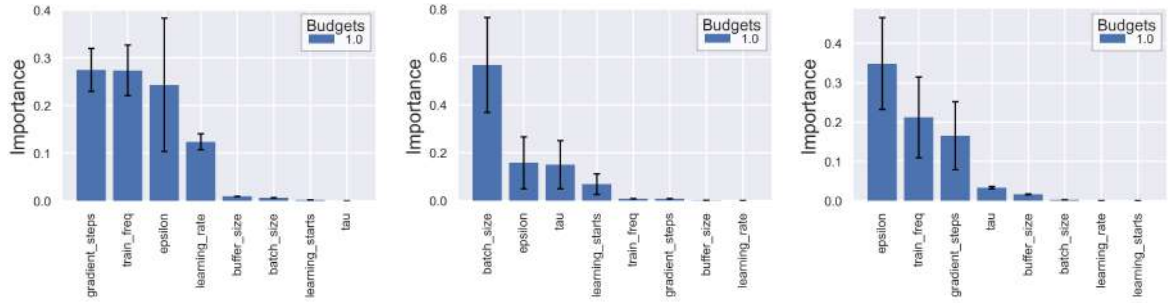


Figure 28: DQN Hyperparameter Importances on Acrobot (left), MiniGrid Empty (middle) and MiniGrid DoorKey (right).

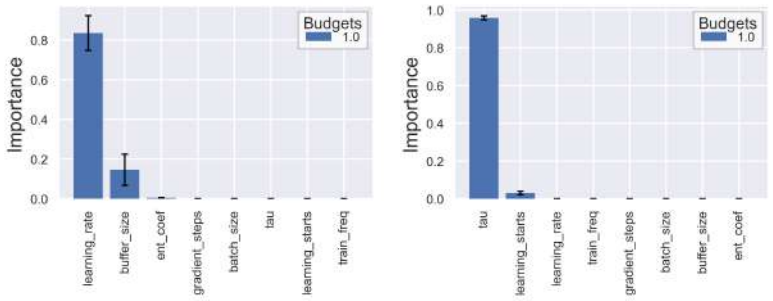


Figure 29: SAC Hyperparameter Importances on Pendulum (left) and Brax Ant(right).

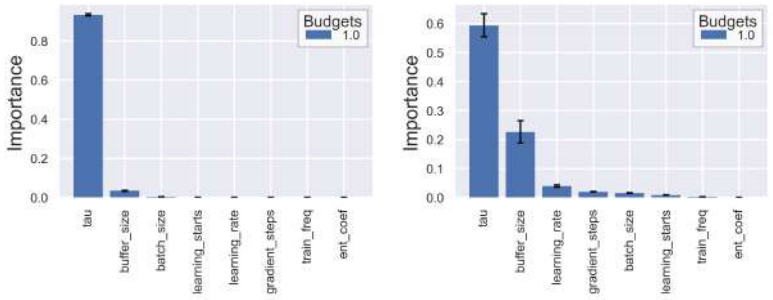
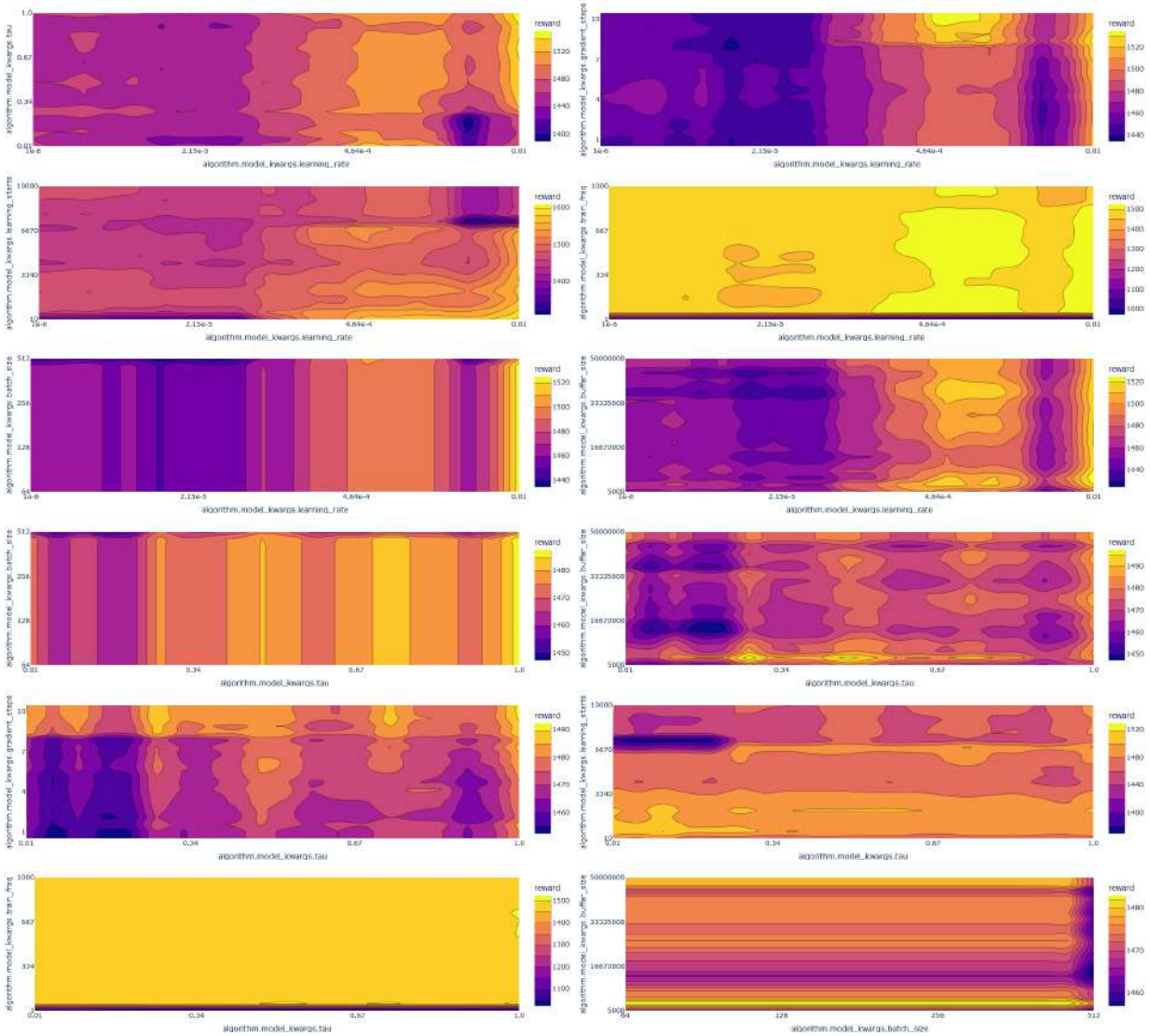


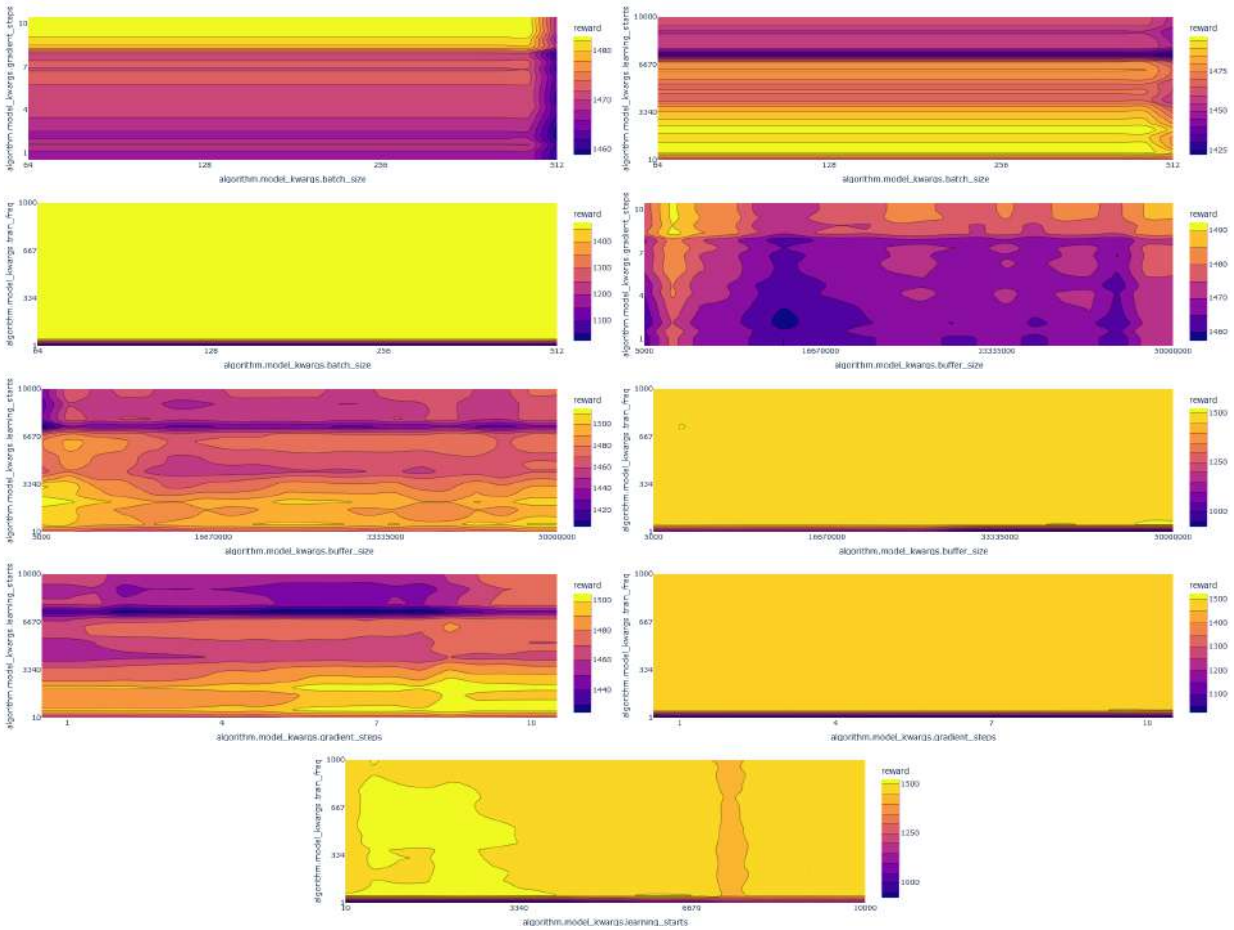
Figure 30: SAC Hyperparameter Importances on Brax Halfcheetah (left) and Humanoid (right).

J. Partial Dependency Plots

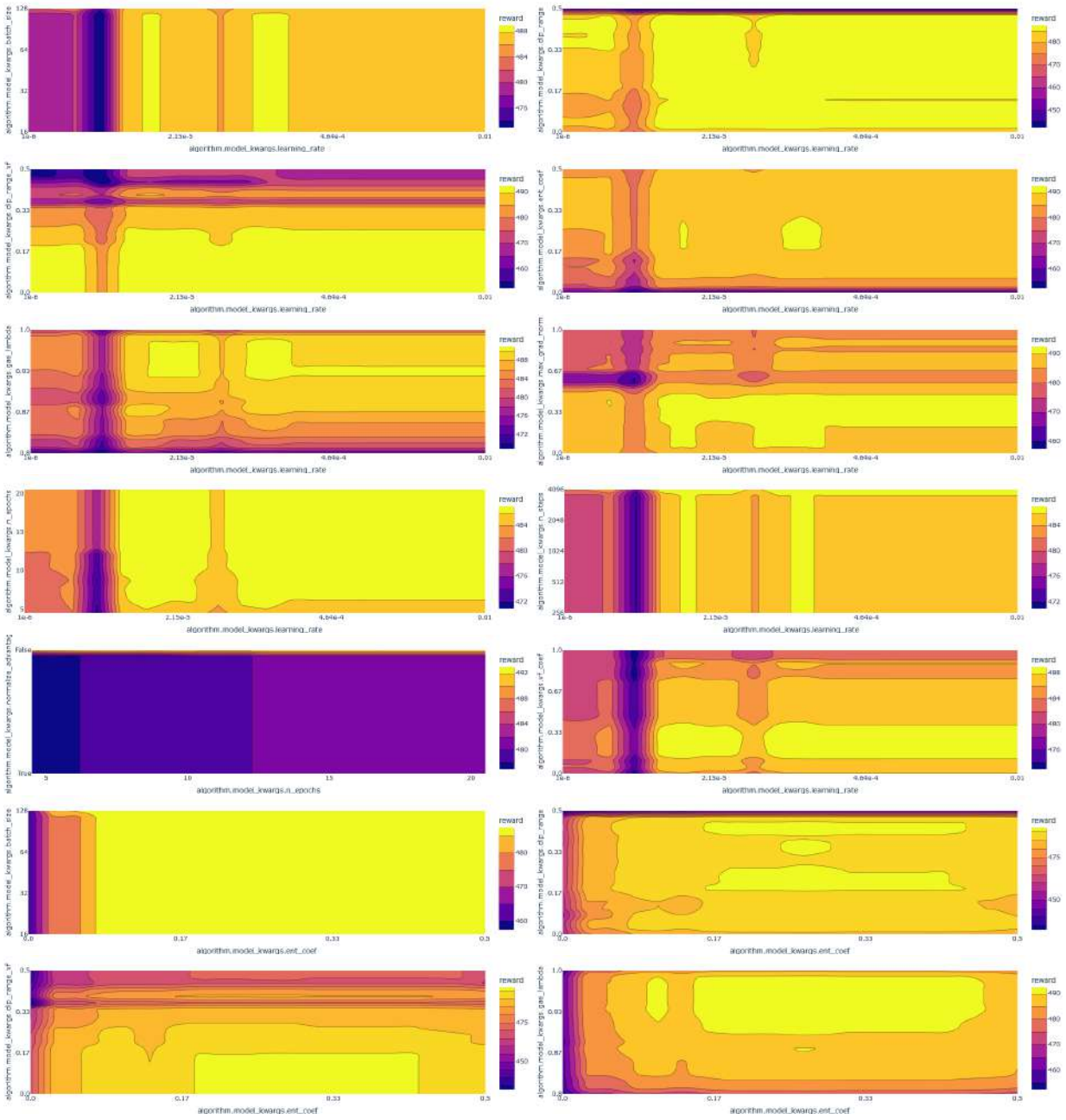
These plots show performance (lighter is better) across the value ranges of two hyperparameters.

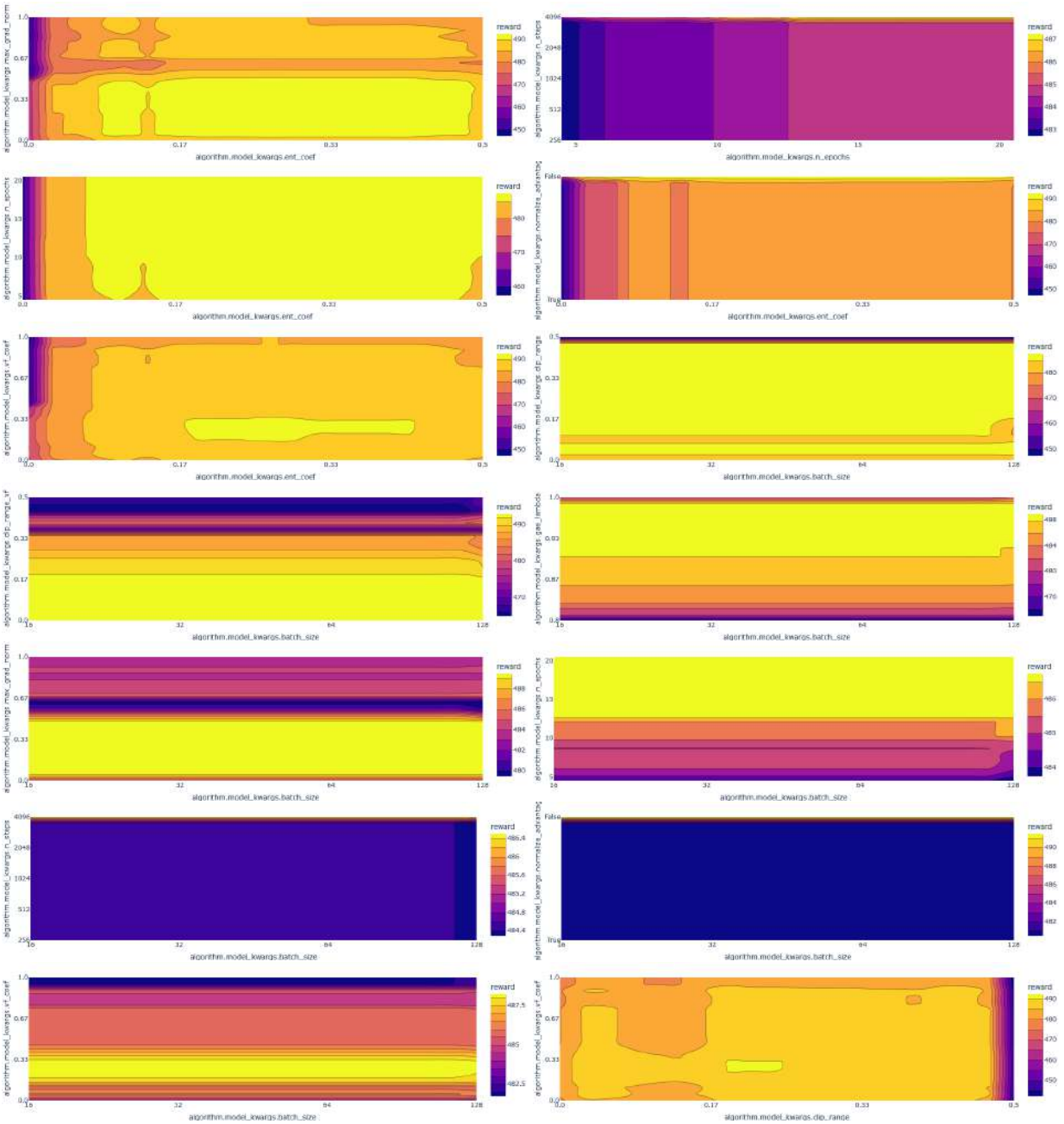
J.1. SAC on Pendulum

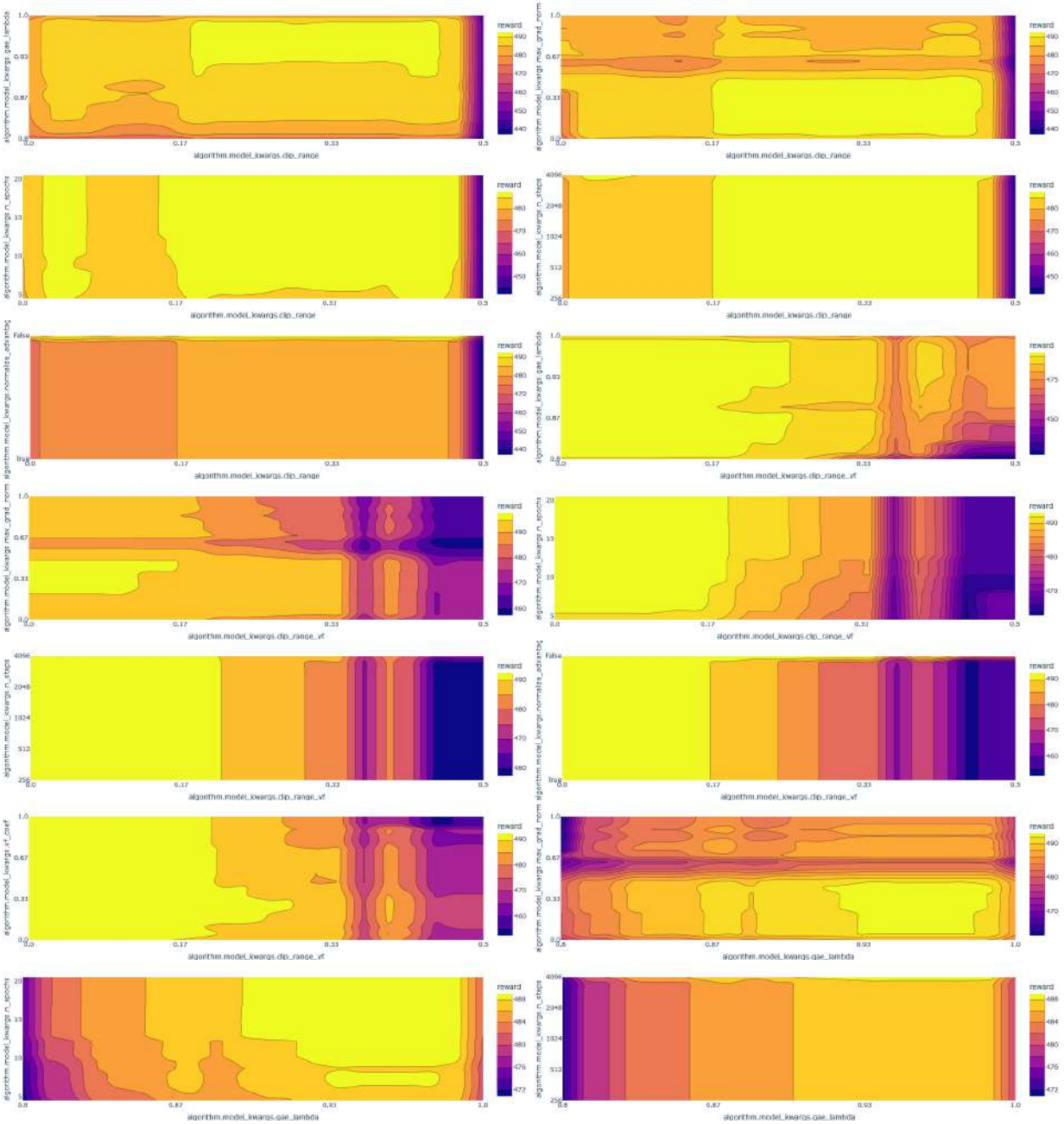


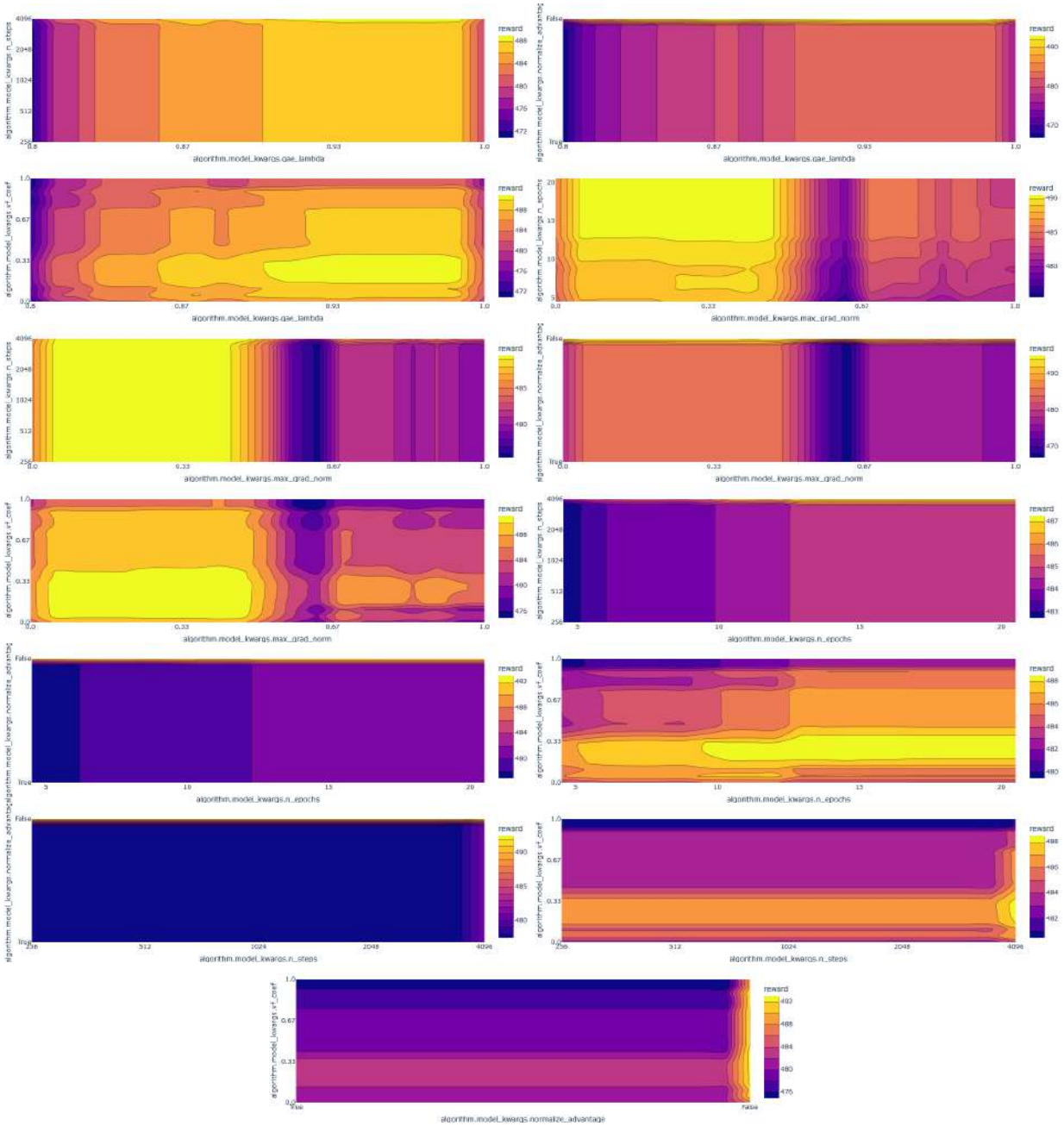

algorithm.model_kwargs.batch_size
algorithm.model_kwargs.batch_size
algorithm.model_kwargs.batch_size
algorithm.model_kwargs.buffer_size
algorithm.model_kwargs.buffer_size
algorithm.model_kwargs.buffer_size
algorithm.model_kwargs.gradient_steps
algorithm.model_kwargs.gradient_steps
algorithm.model_kwargs.learning_starts
reward

J.2. PPO on Acrobot




algorithm.model_kwargs.max_grad_norm
algorithm.model_kwargs.ent_coef
reward
algorithm.model_kwargs.n_steps
algorithm.model_kwargs.n_epochs
algorithm.model_kwargs.normalize_advantage
algorithm.model_kwargs.vf_coef
algorithm.model_kwargs.clip_range
algorithm.model_kwargs.batch_size
algorithm.model_kwargs.gae_lambda
algorithm.model_kwargs.clip_range_vf


algorithm.model_kwargs.gae_lambda
algorithm.model_kwargs.max_grad_norm
algorithm.model_kwargs.n_epochs
algorithm.model_kwargs.n_steps
algorithm.model_kwargs.normalize_advantage
algorithm.model_kwargs.vf_coef
algorithm.model_kwargs.clip_range
algorithm.model_kwargs.clip_range_vf
reward


algorithm.model_kwargs.gae_lambda
algorithm.model_kwargs.max_grad_norm
algorithm.model_kwargs.n_epochs
algorithm.model_kwargs.n_steps
algorithm.model_kwargs.normalize_advantage
reward