Accounting for hyperparameter tuning for online reinforcement learning

Anonymous Author(s)
Affiliation
Address
email

Abstract

Most work in online reinforcement learning (RL) tunes hyperparameters in an offline phase without accounting for the interaction. This empirical methodology is reasonable to assess how well algorithms *can perform*, but is limited when evaluating algorithms for practical deployment in the real world. In many applications, the environment is not compatible with exhaustive hyperparameter searches, and typical evaluations do not characterize how much data is required for such searches. In this work, we explore *online tuning*, where the agent must select hyperparameters during online interaction. Hyperparameter tuning is part of the agent rather than done in a separate (hidden) tuning phase. We layer sequential Bayesian optimization on standard RL algorithms and assess behavior when tuning hyperparameters online. We show the expected result - this strategy's success depends on the environment and algorithm. In an attempt to address this issue, we introduce a "naive" smart way of tuning online, which mitigates wasteful resetting and shows that it can achieve comparable results, highlighting the benefits of smarter online tuning approaches.

1 Introduction

In this paper we consider the online reinforcement learning (RL) setting. The agent does not have access to a simulator, so it cannot learn in parallel on multiple copies of the environment nor reset arbitrarily. Further, as this is online RL, we evaluate the accumulated reward from the beginning of learning, preferring agents that learn quickly and start performing well early, as opposed to only caring about producing an optimal policy at the end of training. The online RL setting reflects many real-world deployment scenarios, such as recommendation systems or process control (Luo et al., 2022; Janjua et al., 2023; Lawrence et al., 2024).

One of the largest barriers to deploying online RL in the real world is dealing with hyperparameters. Most RL algorithms have a variety of hyperparameters, including stepsizes, target network refresh rate, and exploration parameters, to name a few. How should a practitioner select these hyperparameters? One option is to use the defaults from publicly available packages. But, there is absolutely no guarantee that these hyperparameters will perform well for the practitioner's problem. Those default hyperparameters were likely tuned for a popular RL benchmark, like Mujoco (Todorov et al., 2012) or Atari (Bellemare et al., 2013), which may not look anything like the practitioner problem. Radically different hyperparameters are required for different benchmark problems (Patterson et al., 2023). Ideally, the practitioner would tune the hyperparameters to their problem, but in general, this is not possible or at least very time-consuming. Consider tuning the stepsize for an RL agent controlling the flow rate for cleaning a filter in a water treatment plant. Testing different stepsizes requires restarting the RL agent's learning for each stepsize and letting it run for potentially multiple days for each value. During that time, the RL agent may encounter stepsize value that is not performing that well,

Submitted to 38th Conference on Neural Information Processing Systems (NeurIPS 2024). Do not distribute.

thus doing a poor job of cleaning the filter - or even worse, may damage the equipment. Further, the practitioner may need to constantly babysit this agent, monitoring what it is doing and deciding when to stop or what hyperparameter to test next.

A natural conclusion is that hyperparameter tuning should be part of the agent, essentially automating what a practitioner might do. Even better is if the RL agent can be more sample efficient so that the agent can start controlling the system effectively right away, receiving as much reward as possible. Though an obvious idea, *online tuning* is currently not the standard practice - we do not design agents for this setting. Instead, most work tunes the hyperparameters in a separate non-observable phase; this phase is not reported nor accounted for in performance curves. Such *hidden tuning* is an acceptable empirical practice if the goal is to understand how well our algorithms can perform in a near best-case scenario.

There are, however, several inadvertent consequences of this hidden tuning in the online setting. The main issue is that hidden tuning allows the researcher to avoid developing algorithms that are easy to deploy - it can even encourage using hyperparameters because adding more hyperparameters improves performance at no cost. However, these algorithms with more hyperparameters become even more difficult to use online. Further, hidden tuning obscures the fact that simpler algorithms may perform better than these methods with more hyperparameters when we take into account the true cost in terms of rewards lost during tuning. Hidden tuning obscures how much data is needed to get high performance with the new algorithms, making it hard to use results from the literature to decide which algorithms may be effective in real-world problems. Standard empirical results end up being less pertinent to a practitioner.

In online tuning, all interactions count, and agents are evaluated based on how quickly they begin to perform well without a separate hidden tuning phase. An agent could perform better quicker than another agent for a variety of reasons. One agent may have fewer hyperparameters to tune and can settle on a performant setting of those hyperparameters faster. If an agent has no hyperparameters, even better! It can focus exclusively on learning right away. An agent might have less sensitivity to its hyperparameters, making identifying a reasonable setting easier. Related to this, the agent might leverage meta-learning strategies, relying on meta-hyperparameters that could be easier to tune (Sutton, 1992; White & White, 2016; Xu et al., 2018). An algorithm might reuse prior data, like one gathered under different hyperparameter configurations, to better infer what hyperparameters to try next. Moving to the online tuning setting opens up many avenues for algorithm development.

In this work, we investigate online tuning in reinforcement learning. We start by motivating the use of standard Bayesian optimization to easily convert any RL algorithm with hyperparameters into one that tunes its own hyperparameters online. Though this naive layering is clearly a suboptimal approach, it provides a default strategy to test algorithms in this new setting. This approach is critical to facilitate new algorithm development for this online setting by enabling comparisons to previous algorithms in a budgeted way. We show the behavior of Soft Actor-critic (SAC) (Haarnoja et al., 2018) and Proximal Policy Optimization (PPO) (Schulman et al., 2017) in several classic control and Mujoco environments. We find that given small enough ranges and hyperparameter trials, SAC can start performing as good as or sometimes better than the performance we get when using the default hyperparameters, while PPO struggles to find a good set of hyperparameter values within the same budget, suggesting that more hyperparameters make it harder to find a performant solution. We then provide a "smarter" way to do online tuning that avoids resetting and reuses prior data from the sequential search. The approach is simple but is a first step towards designing for the online tuning setting. With this approach, we find that with SAC, one can achieve the same performance levels as the ones tuned offline. Even though they get similar performances, these results open up a new avenue of algorithms that will make this tuning approach more sample-efficient and performant than the one we propose here.

2 Related Work

One of the biggest challenges to tuning hyperparameters in RL is the inherent non-stationarity. Eimer et al. (2022) shows that hyperparameters are highly environment-dependent and seed dependent (Eimer et al., 2023) and changing the values of hyperparameters can have a significant impact on the performance (Obando-Ceron et al., 2023). A lot of work has been done to address the hyperparameter optimization (HPO) issues in RL. Most of the techniques are designed and used by the Automated

Reinforcement Learning (AutoRL) community, which looks into how to automate expensive and potentially even error-prone choices of RL algorithms without human intervention (Parker-Holder et al., 2022). AutoRL is the contributor of many packages (Lindauer et al., 2021) on sequential tuning with Bayesian optimization, as introduced by Snoek et al. (2012). There is a line of work showing the benefits of using BO as a sequential hyperparameter tuner. It can be robust to noise (Hertel et al., 2020) and can be combined with artificial neural networks (Springenberg et al., 2016) to scale to high dimensions and many function evaluations. BO approaches are sequential: they use a single numeric score (Nguyen et al., 2020; Klein et al., 2017) to approximate the underlying function or follow some termination criterion (Makarova et al., 2021) to stop the tuning at some point.

Another set of HPO methods is population-based evolutionary approaches, namely Population-based Training (PBT) algorithms (Jaderberg et al., 2017; Parker-Holder et al., 2020, 2021; Wan et al., 2022). They parallelize the computation of hyperparameters by running different configurations simultaneously for some interval, rank the agents according to their performances, and replace the worst ones with copies of the best ones with perturbed hyperparameters. After some time, these methods converge to a set of good hyperparameter configurations. Faster variations of the PBT method (Li et al., 2018; Falkner et al., 2018), treat HPO as a random search, and use early stopping to allocate resources to try more hyperparameters. Others use probabilistic models to guide the search (Parker-Holder et al., 2020), while in Franke et al. (2020) they share the collected experience replay data between the population leading to sample efficient tuning.

Other works explore the idea of using offline interactions to tune the hyperparameters online (Letham & Bakshy). Here they use offline data and BO to tune live systems. Other works keep a model of the environment learned from offline or online data to tune the agent (Wang et al., 2022; Zhang et al., 2021) or use data seen so far to efficiently tune the hyperparameters (Paul et al., 2019). A promising avenue of HPO methods are the meta-learning approaches (Zahavy et al., 2021) that tune a subset of their hyperparameters while learning in the environment.

3 Problem Formulation

We consider a standard online learning setting, where the agent is evaluated as it learns in the environment. The agent interacts with the environment, seeing observation $o_t \in \mathcal{O}$, taking actions $a_t \in \mathcal{A}$, seeing new observation $o_{t+1} \in \mathcal{O}$, and receiving reward r_{t+1}. It has a total budget of interaction T (global step count), generating a trajectory τ of interaction over this lifetime, and is evaluated based on some performance measure $g(\tau)$ over the entire lifetime. For the continuing setting, a typical measure of performance is the average reward $g(\tau) = \frac{1}{T}\sum_{t=1}^{T} r_t$. In the episodic setting, a typical measure of performance is the average return per step. Namely, if at time step t, the agent is currently in episode i with (discounted) return Return_i for that episode, then we set $g_t = \text{Return}_i$ and obtain performance $g(\tau) = \frac{1}{T}\sum_{t=1}^{T} g_t$.

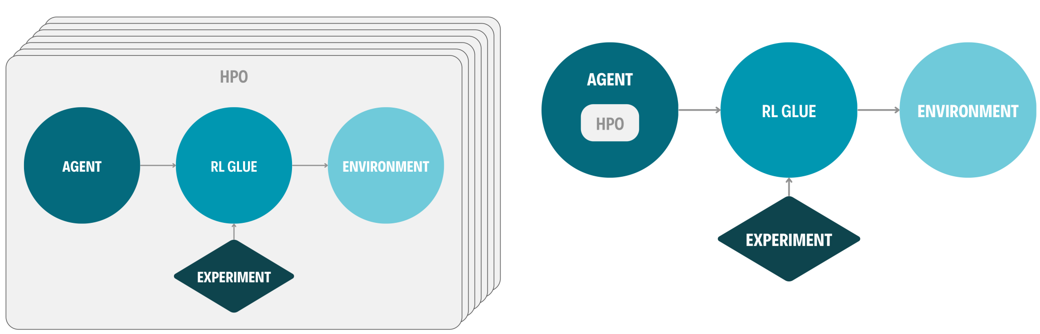

Figure 1: Contrasting the typical hidden tuning setting (left) versus the proposed online tuning setting (right). The online tuning setting requires that tuning is a part of the agent, as it must tune the hyperparameters online, during interaction. Hidden tuning layers optimize the hyperparameteers outside of the agent-environment interaction, allowing a separate search to be performed.

The additional nuance for the *online-tuning* setting is that the hyperparameter tuning phase is explicitly part of the overall agent interaction with the environment. We are not allowed to use any additional interaction with the environment. We depict the difference between online and hidden tuning in Fig. 1. This online-tuning setting (intentionally) blurs the line between tuning and learning. Once tuning is part of the learning phase, it can naturally be considered part of the agent. The ultimate goal of this problem setting is to encourage the development of hyperparameter-free agents, or ones that can quickly learn and adapt their hyperparameters.

As yet, though, we do not have such hyperparameter-free algorithms. In the interim, our algorithms have hyperparameters, and often many of them. To start investigating the online-tuning phase, we need generic online hyperparameter tuning approaches that can be layered on existing algorithms. This allows us to both assess the state-of-the-field and understand just how sensitive our algorithms are for online tuning, while also providing a baseline approach for smarter online-tuning algorithms. The goal of this paper is to provide such a simple generic approach and begin to assess the state of the field.

We assume that the algorithm has a set of hyperparameters $\mathcal{H}$ and that it picks an $h \in \mathcal{H}$ during interaction, like the stepsize, to do updates. Assume we run the agent with hyperparameters $h \in \mathcal{H}$ and let $G(h) \doteq g(\tau)$ be a stochastic sample of the performance of that hyperparameter, which is random because the trajectory generated by one lifetime of interaction is stochastic due to the environment or the agent, or both. This online tuning can be done in many ways; the remainder of this paper outlines basic, generic approaches that can be layered on top of many RL algorithms.

4 Tuning Hyperparameters Online

In this section, we outline first how to use standard Bayesian optimization approaches, often used for hyperparameter optimization approaches, in the online tuning setting.

4.1 Sequential BO for Online Tuning with Resetting

The key question for applying BO to the online tuning setting is how much interaction do we use for tuning. When searching for hyperparameters in hidden tuning, this trade-off does not arise, because all online interaction is done with the hyperparameter found during the hidden tuning. But, in the online setting, the agent or agent designer has to select (a) how long each hyperparameter is tested before resetting the agent and testing a new hyperparameter and (b) the maximum percentage of the lifetime that can be used to test different hyperparameters. BO can stop earlier than this maximum time with smart early stopping approaches, though, for simplicity, we use a fixed length of time. We summarize this generic approach, BO for Online Tuning with Resetting, in Algorithm 2.

Algorithm 1 BO Agent for Online Tuning **with Resetting**

Input: RL Algorithm `Alg`, hyperparameter set $\mathcal{H}$, number evaluation steps M, max tuning iterations
Initialize Bayesian Optimizer (e.g., using package like Optuna), max-perf = $-\infty$, best-h = None
for $i = 1$ to max tuning iterations **do**
    Get next $h \in \mathcal{H}$ from Bayesian Optimizer
    Run `Alg` with h for M steps to get performance G, send G to Bayesian Optimizer
    If max-perf $< G$, then set best-h $= h$ and max-perf $= G$
    Reset `Alg` (reinitialize weights, clear buffer, etc.)
end for
Run `Alg` with best-h for the remaining steps

Let us consider an example. An agent will be deployed for 3 million steps, and has 5 hyperparameters to tune. The agent designer specifies the ranges for these hyperparameters and decides to test each setting for 200k steps for 2 million steps. This allows for 10 hyperparameter settings to be tested, which is unlikely to find the optimal choice when there are 5 hyperparameters to set. Doing a grid search on a cross-product of even 2 choices per hyperparameter would already take testing $2^5 = 32$ hyperparameter settings. But, with correlations between hyperparameters, the agent designer can hope testing 10 hyperparameter settings will be enough to find reasonable hyperparameters.

We visualize such an experiment in Figure 2, showing how much learning time is spent testing hyperparameters the agent. We also visualize the sequence of stepsizes tested in each run. It is worth noting how much the chosen stepsize can vary between runs. Unlike hidden tuning, there is not one stepsize chosen; rather, each run may have a unique stepsize. Details for this experiment are given in Section 5.

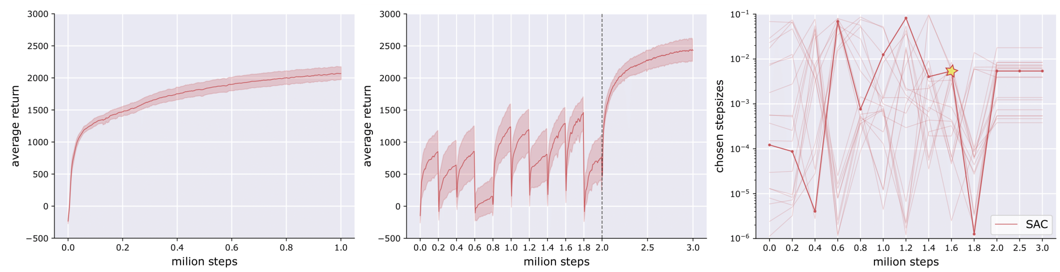

Figure 2: SAC in HalfCheetah using default hyperparameters versus online tuning of the stepsize using BO with resetting, with mean performance over 20 independent runs. The leftmost figure depicts the average performance of the SAC agent with the default hyperparameters provided in the literature. The middle one is an example of our proposed online tuning strategy, where we have 3M steps as a provided budget. We stop tuning the hyperparameters after the 2M mark (the dotted line) and use the last million steps to evaluate the best hyperparameter configuration found. The last plot shows the stepsizes chosen by the optimizer in the first 2M steps for each of the individual runs of the second plot. The dark line shows the chosen values for one seed and the yellow star points to the best stepsize picked. The shaded area is a 95% bootstrapped confidence interval.

4.2 Sequential BO for Online Tuning without Resetting

From the above figure, it is clear how much data is wasted when doing a hyperparameter search. In the online setting, such data inefficiency is not acceptable - we want the agent to continually improve by adapting its hyperparameters. In this section, we provide a basic strategy using the same BO approaches but now without resetting. The approach is simple, and absolutely not optimal, but we hope for it to provide a starting point going forward in the online setting.

The idea is simple: the agent is not reset after each hyperparameter setting and continues to learn with the given weights and buffer. In the pseudocode above, this would involve removing the line that resets the agent while also removing the maximum number of tuning iterations. Instead, because the agent is learning, without resetting, the hyperparameters can also be continually adapted, hopefully being continually improved.

However, there is an important issue with this naive extension: some hyperparameters may result in poor performance. By allowing BO to test potentially speculative hyperparameters, it is even likely that at some point the weights will become bad. It may be difficult to learn continuing from these bad weights for a new hyperparameter setting, both preventing the agent from further improvement and also not providing a fair assessment of the new hyperparameter setting.

The small modification involves reverting back to the previous weights if the new hyperparameter setting causes a drop in performance. On the first step, the agent selects hyperparameters and runs for M steps, getting back a performance estimate. If this performance is below an acceptable threshold for the problem, the agent reinitializes the weights and the buffer. Otherwise, it continues from these weights and buffer and selects a new hyperparameter setting. If, after running again for M steps, the agent obtains a performance estimate lower than the previous one by some threshold (e.g., 10% worse), then it reverts back to those previous weights and buffer. The role of the threshold is to avoid resetting simply due to some stochasticity. Further, in early learning, it is unlikely for the performance of a reasonable hyperparameter setting to be worse than the previous one as it gets to learn starting from a better initial point (policy, buffer, weights). This modification is not perfectly robust to resetting the agent's state back to a set of weights and buffers that make learning hard. But, again, our goal here is not to provide an optimal algorithm, but a simple default to facilitate future development of smarter algorithms for this online tuning setting.

Algorithm 2 BO Agent for Online Tuning **without Resetting**

Input: RL Algorithm `Alg`, hyperparameter set $\mathcal{H}$, number evaluation steps M
Initialize Bayesian Optimizer, RL internal state `B`, max-perf = $-\infty$
while Interacting with the environment **do**
  Get next $h \in \mathcal{H}$ from Bayesian Optimizer
  Run `Alg` starting from `B` with h for M steps to get performance G and new RL internal state `B`$'$
  Send G to Bayesian Optimizer
  If $G > 0.9 *$ max-perf then set `B` = `B`$'$, max-perf $= G$
end while

5 Experiments using Bayesian Optimization with Resets

In this section, we evaluate BO for online tuning with resetting in five Mujoco environments, for SAC and PPO. We use the package Optuna (Akiba et al., 2019) to do sequential BO. As we are in the online setting, PPO is not run with parallel copies of the environment; it is run single-stream, by interacting with just one environment. We give an overall budget of 3 million online steps, $M = 200$k evaluation steps for each hyperparameter setting and use two different stopping conditions: after 1 million steps, and after 2 million steps. In other words, in the first setting, the agent is able to test 5 hyperparameter settings and in the second it tests 10. We also compare to an agent using the default hyperparameter settings.

We additionally consider the effect of the number of hyperparameters that are tuned. For SAC, we test two scenarios: tuning only the stepsize (one hyperparameter) and tuning five hyperparameters. When tuning only the stepsize, we leave the remaining hyperparameters at the SAC defaults. PPO on the other hand has 7 hyperparameters to tune. For details on the hyperparameter ranges and additional experiment details, including agent and Optuna details, see the Appendix A.

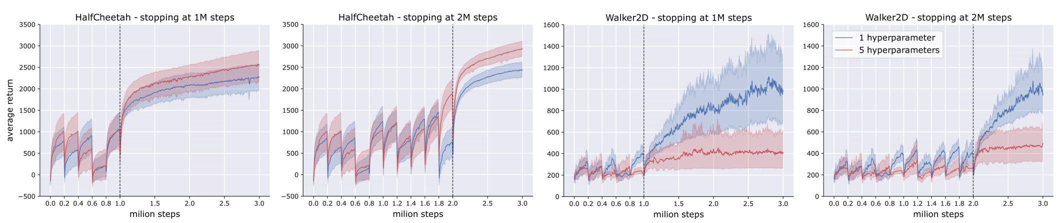

Figure 3: SAC in HalfCheetah and Walker2D using two different stopping conditions while tuning one or many hyperparameters. In all the plots, each agent had an overall 3 million steps budget, and each hyperparameter had a 200K evaluation period. The gray dotted line depicts the timestep when the agent stops testing different hyperparameters and deploys best configuration found. The blue line corresponds to the agents that had to tune one hyperparameter, while the red line shows the performance of agents with many hyperparameters. Note that SAC with default hyperparameters reaches a scores of approximately 2000 and 800 in HalfCheetah and Walker2D respectively.

We first examine the effect of tuning one hyperparameter versus many for SAC on two environments, shown in Fig. 3. We selected these two from our five Mujoco environments to highlight a case where tuning more hyperparameters was slightly better and a case where it was notably worse. In some cases, the flexibility to tune more hyperparameters can improve performance because the agent is not stuck at the defaults; however, this tuning has to be feasible within the allocated time online. If the agent needs to test many hyperparameter settings to find a good one, then the increased flexibility can be harmful. We see in HalfCheetah that there is a slight performance improvement even when tuning for 1 million steps, and this effect is even larger when tuning for 2 million steps. The further improvement makes sense, given the agent can test 2x as many configurations, getting even more performance gains. In Walker2D, on the other hand, the increased flexibility is clearly detrimental.

Next we investigate the performance of both PPO and SAC, in all five Mujoco environments. The results are qualitatively similar for stopping at 1 million and 2 million steps, so we include only 1 million in the main body in Fig. 4 and the result for stopping at 2 million steps in the appendix, in Fig. 7. It is apparent that it is generally more difficult to tune the hyperparameters for PPO online, and it often performs substantially worse than SAC. When only tuning the stepsize, the performance is more comparable to SAC, but when we tune seven hyperparameters, PPO's performance drops significantly. SAC appears to be less sensitive to its hyperparameters than PPO, and this online tuning regime makes this advantage apparent. Hidden tuning, on the other hand, might mask this difference and potentially even give preference to PPO which exposes more hyperparameters to tune during a hidden tuning phase.

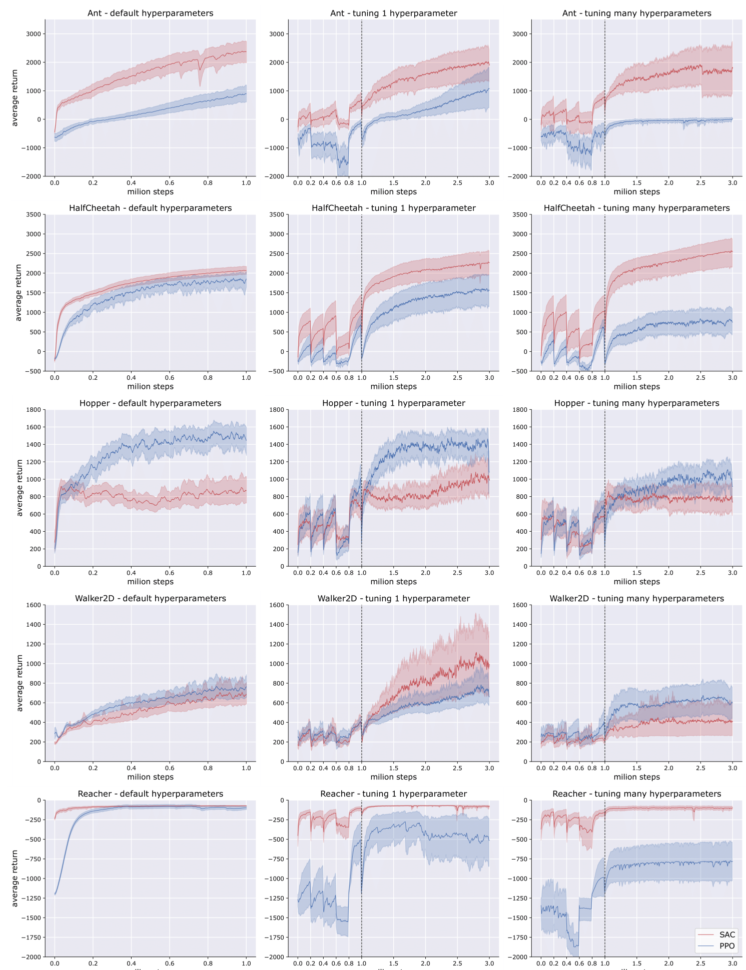

Figure 4: SAC (red) and PPO (blue) algorithms in a variety of Mujoco environments. In the first column, we have the performance of the algorithms with the default hyperparameters. The second and third columns show the algorithms' performances within the 3 million budget, where we stop hyperparameter optimization after 1M steps with the difference of tuning one and many hyperparameters. The dotted line depicts the timestep that the agent started evaluation of the best hyperparameters it has seen.

6 Experiments using Bayesian Optimization without Resets

We now test the behavior of the BO algorithm that does not reset the agent's state. We hypothesize that avoiding resetting should allow the agent to obtain comparable or better performance overall. Our goal is to understand the benefits of starting to tailor algorithms for the online tuning setting, moving from the naive application of BO to one more specifically designed for the online setting. We use the same environments and experimental details as in the previous section.

We first examine the difference in the performance of agents that use BO with resetting (Algorithm 1) and BO without resetting (Algorithm 2), which carefully considers the weights and buffer to use for the next hyperparameter configuration in the Fig. 5. We also include a naive variant of BO without resetting, which simply shares all the data from hyperparameter configuration to configuration. This naive variant essentially corresponds to Algorithm 1, but by removing the resetting line and tuning continually (not using the max tuning iterations). We can see that BO without resetting significantly outperforms BO with resetting, steadily improving with time because it is not constantly reset. However, this algorithm does need to recognize if a hyperparameter choice has led to poor weights to continue from an early set of weights. We can see the naive variant fails at 500k steps and does not recover.

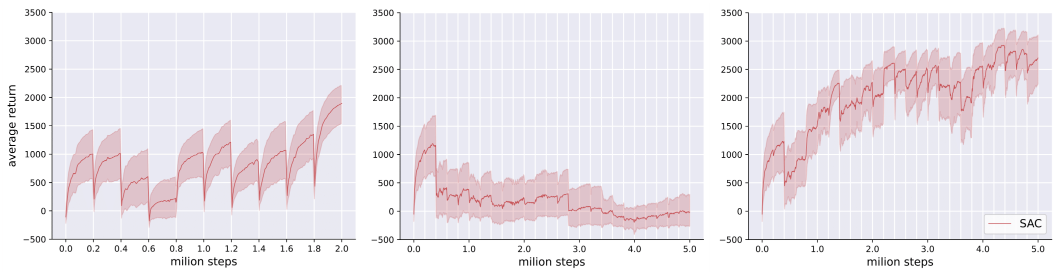

Figure 5: SAC in HalfCheetah with three different online tuning strategies: BO with resetting (Algorithm 1), a naive variant of BO without resetting, and BO without resetting (Algorithm 2). The shaded area is a 95% bootstrapped confidence interval over 20 different run.

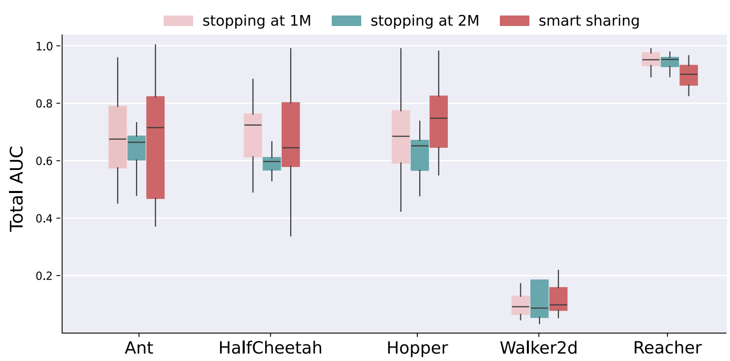

Figure 6: Box plots of the total AUC for the 3M evaluation budget for SAC tuning 5 hyperparameters in four different settings for all the Mujoco environments shwoing the results for resetting with 1M and 2M stopping conditions (first 2 box plots) and the smart sharing of the agent's state of each hyperparameter configuration.

We next present SAC's performance for all five Mujoco environments in Fig. 6. For this plot, we show the total area under the curve (AUC) over 3 million steps for BO with resetting when stopping after 1 million steps and after 2 million steps, and BO without resetting. To make the AUC comparable across the environments, we normalize all the AUCs between 0 and 1.

Figure 6 shows that using the "smart" tuning proposed above gets comparable overall performance as both resetting evaluation settings we tried in section 5 in all environments for 3M steps tried. The box plots for the final performance of all three agents are presented in Fig. 10. Overall, SAC agents that don't reset with even the most naive augmentation on top increase their performance as they live on and can perform comparably to the performance of the default hyperparameters. However, PPO, as presented in the Fig. 9 in the appendix, doesn't take advantage of this sharing mechanism, leading to comparable performances for both sharing and naive-sharing cases. This is an interesting phenomenon worth further investigation.

7 Conclusions and future work

In this paper, we introduce a novel evaluation paradigm that eliminates the hidden hyperparameter tuning phase typically used in reinforcement learning. Instead, we tune the agent's hyperparameters online within a given budget, dynamically adjusting them as the process unfolds. We use Bayesian Optimization as a meta-learner to provide the RL algorithm with hyperparameter configurations to try out sequentially. We found that, even with periodic resets, our approach achieves results comparable to the default settings in multiple Mujoco environments. However, our results also show that hyperparameters depend on the environment, the ranges we define, the number of hyperparameters, and the trial counts, thus, algorithms with more hyperparameters to tune, like PPO, struggled in this paradigm. Lastly, we propose a basic methodology for not resetting, which achieves performance levels similar to resetting approaches, marking a step towards effective data utilization in scenarios where learning is done in a single lifetime.

Even though we showcase that the proposed online paradigm is a great starting point, this is still the beginning of using hyperparameter tuning as part of online evaluation. The non-resetting approach we propose does not perform better than the resetting ones, opening up an avenue to try and make better algorithms that will outperform the current approach. Addiitonally, the Bayesian Optimization algorithms are not designed for non-stationary cases, and as we add more non-stationarity by sharing the weights and buffer, this raises the need to develop sequential decision-making algorithms that account for the changes in the state. With this methodology, we also want to contribute to fairness in evaluation, and using budgets as a reporting mechanism should be a step towards better empirical practices.

References

Akiba, T., Sano, S., Yanase, T., Ohta, T., and Koyama, M. Optuna: A next-generation hyperparameter optimization framework. *CoRR*, abs/1907.10902, 2019.

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

Eimer, T., Benjamins, C., and Lindauer, M. Hyperparameters in Contextual RL are Highly Situational, 2022.

Eimer, T., Lindauer, M., and Raileanu, R. Hyperparameters in Reinforcement Learning and How To Tune Them. In *Proceedings of the 40th International Conference on Machine Learning*. PMLR, 2023.

Falkner, S., Klein, A., and Hutter, F. BOHB: Robust and Efficient Hyperparameter Optimization at Scale. In *Proceedings of the 35th International Conference on Machine Learning*. PMLR, 2018.

Franke, J. K. H., Köhler, G., Biedenkapp, A., and Hutter, F. Sample-Efficient Automated Deep Reinforcement Learning. https://arxiv.org/abs/2009.01555v3, 2020.

Haarnoja, T., Zhou, A., Abbeel, P., and Levine, S. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *CoRR*, abs/1801.01290, 2018.

Hertel, L., Baldi, P., and Gillen, D. L. Quantity vs. Quality: On Hyperparameter Optimization for Deep Reinforcement Learning, 2020.

Jaderberg, M., Dalibard, V., Osindero, S., Czarnecki, W. M., Donahue, J., Razavi, A., Vinyals, O., Green, T., Dunning, I., Simonyan, K., Fernando, C., and Kavukcuoglu, K. Population based training of neural networks. *CoRR*, abs/1711.09846, 2017.

Janjua, M. K., Shah, H., White, M., Miahi, E., Machado, M. C., and White, A. Gvfs in the real world: making predictions online for water treatment. *Machine Learning*, pp. 1–31, 2023.

Kingma, D. and Ba, J. Adam: A method for stochastic optimization. *International Conference on Learning Representations*, 12 2014.

Klein, A., Falkner, S., Bartels, S., Hennig, P., and Hutter, F. Fast Bayesian Optimization of Machine Learning Hyperparameters on Large Datasets. *arXiv:1605.07079 [cs, stat]*, 2017.

Lawrence, N. P., Damarla, S. K., Kim, J. W., Tulsyan, A., Amjad, F., Wang, K., Chachuat, B., Lee, J. M., Huang, B., and Gopaluni, R. B. Machine learning for industrial sensing and control: A survey and practical perspective. *Control Engineering Practice*, 145:105841, 2024.

Letham, B. and Bakshy, E. Bayesian Optimization for Policy Search via Online-Offline Experimentation.

Li, L., Jamieson, K., DeSalvo, G., Rostamizadeh, A., and Talwalkar, A. Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization. *arXiv:1603.06560 [cs, stat]*, 2018.

Lindauer, M., Eggensperger, K., Feurer, M., Biedenkapp, A., Deng, D., Benjamins, C., Ruhopf, T., Sass, R., and Hutter, F. SMAC3: A Versatile Bayesian Optimization Package for Hyperparameter Optimization. https://arxiv.org/abs/2109.09831v2, 2021.

Luo, J., Paduraru, C., Voicu, O., Chervonyi, Y., Munns, S., Li, J., Qian, C., Dutta, P., Davis, J. Q., Wu, N., et al. Controlling commercial cooling systems using reinforcement learning. *arXiv preprint arXiv:2211.07357*, 2022.

Makarova, A., Shen, H., Perrone, V., Klein, A., Faddoul, J. B., Krause, A., Seeger, M., and Archambeau, C. Automatic Termination for Hyperparameter Optimization. https://arxiv.org/abs/2104.08166v4, 2021.

Nguyen, V., Schulze, S., and Osborne, M. Bayesian Optimization for Iterative Learning. In *Advances in Neural Information Processing Systems*, volume 33. Curran Associates, Inc., 2020.

Obando-Ceron, J., Bellemare, M. G., and Castro, P. S. Small batch deep reinforcement learning, 2023.

Parker-Holder, J., Nguyen, V., and Roberts, S. Provably Efficient Online Hyperparameter Optimization with Population-Based Bandits. *NeurIPS*, 2020.

Parker-Holder, J., Nguyen, V., and Roberts, S. One-shot bayes opt with probabilistic population based training. *CoRR*, abs/2002.02518, 2020.

Parker-Holder, J., Nguyen, V., Desai, S., and Roberts, S. J. Tuning mixed input hyperparameters on the fly for efficient population based autorl. *CoRR*, abs/2106.15883, 2021.

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. *CoRR*, abs/2201.03916, 2022.

Patterson, A., Neumann, S., White, M., and White, A. Empirical design in reinforcement learning. *arXiv preprint arXiv:2304.01315*, 2023.

Paul, S., Kurin, V., and Whiteson, S. *Fast Efficient Hyperparameter Tuning for Policy Gradients*. 2019.

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

Snoek, J., Larochelle, H., and Adams, R. P. Practical bayesian optimization of machine learning algorithms, 2012.

Springenberg, J. T., Klein, A., Falkner, S., and Hutter, F. Bayesian Optimization with Robust Bayesian Neural Networks. In *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc., 2016.

Sutton, R. S. Adapting bias by gradient descent: An incremental version of delta-bar-delta. In *AAAI*, volume 92, pp. 171–176. Citeseer, 1992.

Todorov, E., Erez, T., and Tassa, Y. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ international conference on intelligent robots and systems*, pp. 5026–5033. IEEE, 2012.

Wan, X., Lu, C., Parker-Holder, J., Ball, P. J., Nguyen, V., Ru, B., and Osborne, M. A. Bayesian generational population-based training, 2022.

Wang, H., Sakhadeo, A., White, A., Bell, J., Liu, V., Zhao, X., Liu, P., Kozuno, T., Fyshe, A., and White, M. No More Pesky Hyperparameters: Offline Hyperparameter Tuning for RL, 2022.

White, M. and White, A. A greedy approach to adapting the trace parameter for temporal difference learning. In Jonker, C. M., Marsella, S., Thangarajah, J., and Tuyls, K. (eds.), *Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems, Singapore, May 9-13, 2016*, pp. 557–565. ACM, 2016.

Xu, Z., van Hasselt, H. P., and Silver, D. Meta-gradient reinforcement learning. *Advances in neural information processing systems*, 31, 2018.

Zahavy, T., Xu, Z., Veeriah, V., Hessel, M., Oh, J., van Hasselt, H., Silver, D., and Singh, S. A self-tuning actor-critic algorithm, 2021.

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. https://arxiv.org/abs/2102.13651v1, 2021.

Zhou, H., Lin, Z., Li, J., Fu, Q., Yang, W., and Ye, D. Revisiting discrete soft actor-critic, 2023.

A Design choices and hyperparameters used in the experiments

All the hyperparameter ranges and values for all algorithms in all environments can be seen in table 1. In this paper, we consider 3 different RL algorithms - SAC, PPO, and DDQN. SAC is used with continuous action spaces, but we extended it to work with discrete action spaces using the modifications proposed in Zhou et al. (2023). All the hyperparameter ranges listed below are chosen according to the standard values used in hyperparameter sweeping, like not too big of a stepsize or small discount factor γ. We applied log-uniform scaling to γ and stepsize values for the tuner to prefer values at the edges of the given ranges more.

We tune only a subset of all the hyperparameters as we separate algorithm-specific and compute-specific hyperparameters. We let the buffer size or the network architecture stay the same as this hyperparameter usually depends on the budget of the experimenter, which in turn makes the no-resetting pipeline easier to handle. Meanwhile, depending on the environment, the algorithm-specific values may change - like the amount of gradient clipping in PPO - so we tune the ones the experimenter may not have enough intuition about.

Table 1: Hyperparameter values and ranges for SAC, PPO and DDQN.

Parameter	Value	Ranges
Shared		
optimizer	Adam (Kingma & Ba, 2014)	
nonlinearity	ReLU	
stepsize	$1e-2/3\cdot 10^{-4}$	$\log([1e-6, 0.1])$
discount (γ)	0.99	$\log([0.9, 1])$
number of hidden layers (all networks)	2	
number of hidden units per layer	64	
number of samples per minibatch	64	
SAC		
target smoothing coefficient (τ)	0.005	$[1e-4, 0.1]$
target update interval	1	
update frequency	1	
reward scale	1 / 5	[1, 20]
entropy coefficient	0.2	$[1e-4, 0.3]$
replay buffer size	10^6 / 10^3	
start updates	500 / 10^3	
normalize observations	False	
normalize rewards	False	
PPO		
nonlinearity	Tanh	
GAE λ	0.8 / 0.95	[0.7, 1]
PPO clip ϵ	0.1 / 0.2	[0.1, 0.8]
value loss coefficient	0.5	[0.1, 1]
entropy coefficient	0.0	[0.0, 0.5]
gradient clip	0.5	[0.1, 1]
update epochs	4	
rollout steps	256 / 2048	
normalize observations	True	
normalize rewards	True	
DDQN		
target smoothing coefficient (τ)	0.005	$[1e-4, 0.1]$
epsilon	0.05	[1e-4, 0.3]
update frequency	4	
replay buffer size	10^3	
start updates	500	

A.1 Details on the search methods

As a Bayesian optimizer, we chose Optuna as our main algorithm to try with different settings. We use the TPESampler with Hyperband Pruner to select the hyperparameters on each trial. We use these as they work with all types of hyperparameters - integers, floats, and categorical variables and the Hyperband pruner helps in more efficiently exploring the hyperparameter space by allocating resources dynamically to promising trials and stopping less promising trials early in the optimization process. The objective of the Bayesian optimizer is to maximize the overall performance of the agent, thus, we give it input the total AUC of the agent's performance using the proposed hyperparameter configuration after running it for one trial. We let Optuna do no warmup trials - it starts with a random value in that given range and then starts the optimization process.

B Results with resetting

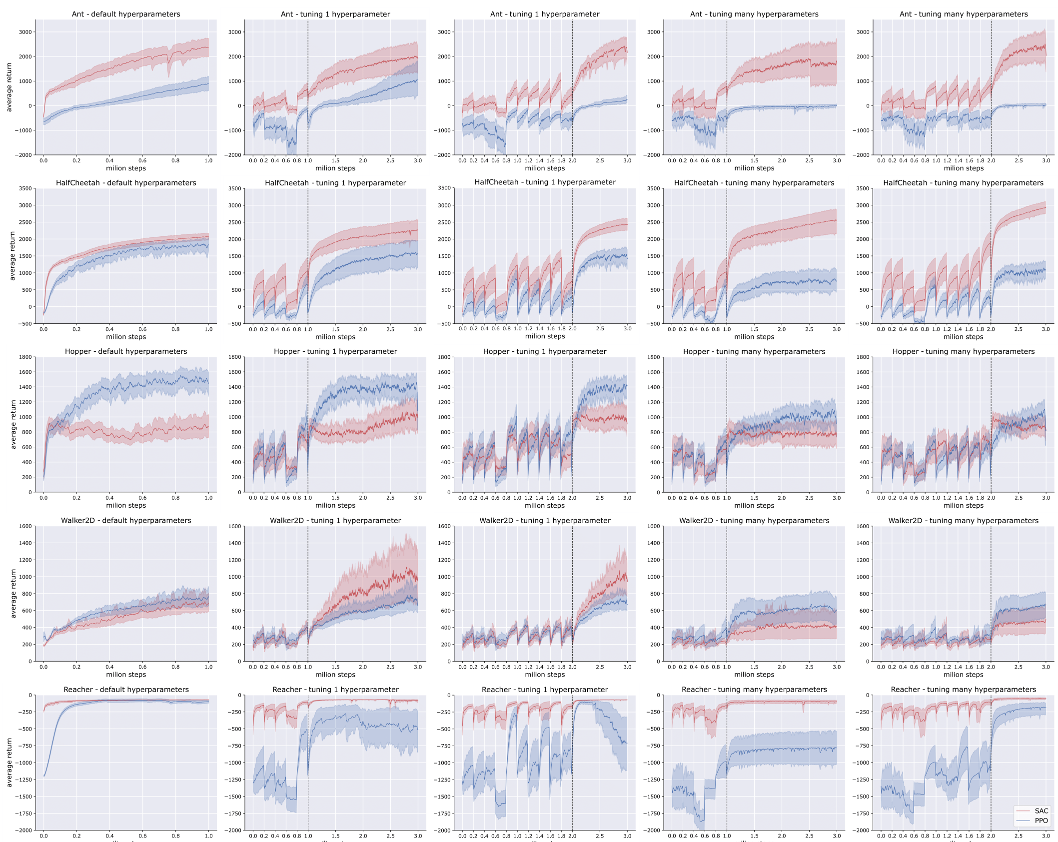

Figure 7: SAC (red) and PPO (blue) algorithms in a variety of Mujoco environments. In the first column, we have the performance of the algorithms with default hyperparameters. The second and third columns show the algorithms' performances within the 3 million budget, where we stop hyperparameter optimization after 1M steps with the difference of tuning one and many hyperparameters. The last two columns are the performance of the algorithms when we stop at 2M steps, letting it try 10 hyperparameter configurations instead of 5 as in the 1M stopping condition case. The dotted line depicts the timestep that the agent started evaluation of the best hyperparameters it has seen. The shaded area is a 95% bootstrapped confidence interval over 20 different runs.

C Results without restting

In this section, we present the plots for SAC and PPO algorithms in the smart sharing paradigm we introduced in the main paper. In Fig. 8, we show the results of the SAC agent to evaluate while tuning its hyperparameters at the same time, without resetting, for 5M global steps. As we can see from the plot, in almost all cases, the agent gets to good performance after 3M steps but keeps slowly increasing its performance. It is not comparable with the performance that one gets from the default hyperparameters, but after 5M steps, it gets to the same level of performance.

Figure 8: Smart sharing results for the SAC algorithm in a variety of Mujoco environments. In the red line, we show the performance of smart sharing agents for a 5M online interaction budget, while the blue line is the performance of the naive sharing agents for the same budget. The plots in the first row correspond to the setting where we only tune the stepsize, and the second row when we tune all 5 hyperparameters.

Even though smart sharing helps to make SAC perform better over time, the same results are not visible in PPO. Interestingly, in PPO, it seems like naive and smart sharing behave similarly. This can be the result of having a clipped surrogate objective that doesn't let the parameters drift too far away from each other - even when we change the hyperparameters, the results are the same. But only in the Reacher environment, do we see a difference between the two approaches. If we look closely, the performance of the smart and the resetting with both one and many hyperparameters are similar in some environments, the only difference is that these agents don't oscillate as much as in the resetting case. This once again proves that we can achieve the performance of default hyperparameters if we design better algorithms than this "naive" smart approach and that PPO may have a harder time tuning its many hyperparameters.

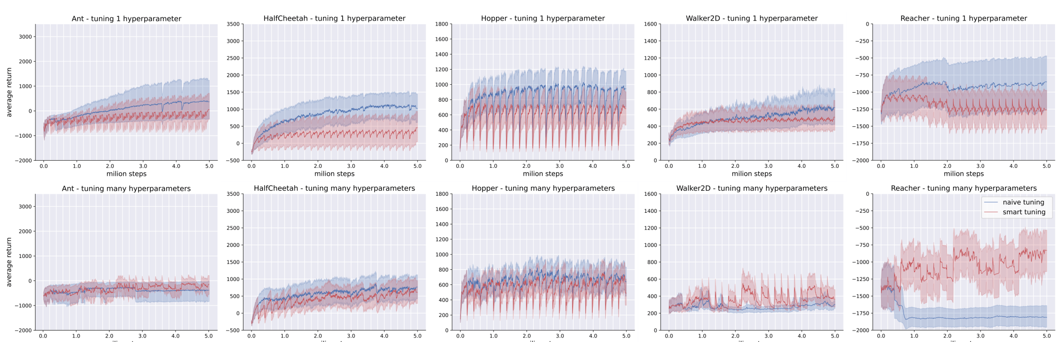

Figure 9: Smart sharing results for the PPO algorithm in a variety of Mujoco environments. In the red line, we show the performance of smart sharing agents for a 5M online interaction budget, while the blue line is the performance of the naive sharing agents for the same budget. The plots in the first row correspond to the setting where we only tune the stepsize, and the second row when we tune all 7 hyperparameters.

D Total AUC and final performance results for SAC and PPO

Here, we show the total AUC for SAC and PPO and their final performances in Mujoco environments. In all plots, we evaluate for 3M steps tuning 5 hyperparameters in SAC and 7 in PPO. The first 2 box plots are the performances for stopping at 1M (light pink) and 2M (blue) steps. The last (bright pink) boxplot is where we share the agent's state while considering the performance of the previous agent.

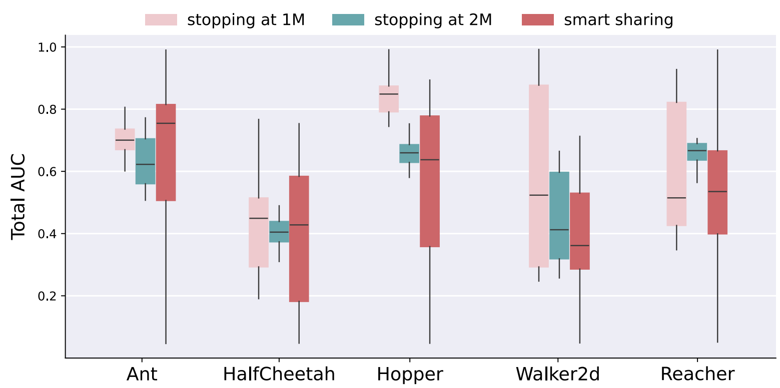

Figure 10: Box plots of the total AUC of the PPO algorithm for the 3M evaluation budget tuning 7 hyperparameters in three different settings for all the Mujoco environments.

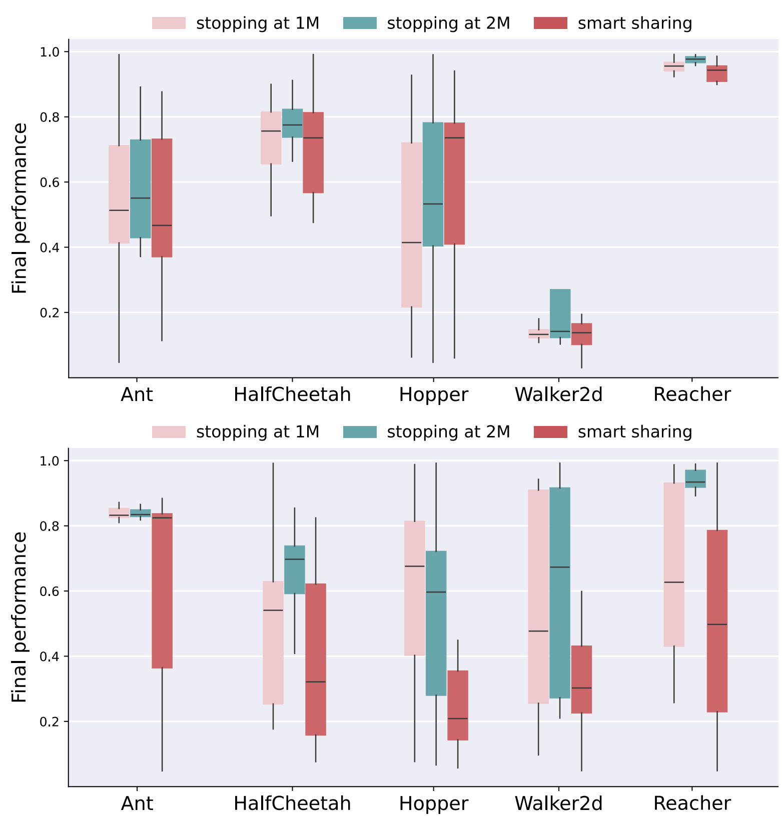

Figure 11: Box plots of the final performances of the SAC (top) and PPO (bottom) algorithms for the 3M evaluation budget tuning 5 and 7 hyperparameters respectively in three different settings for all the Mujoco environments testbeds.