Accounting for Hyperparameter Tuning in Online Reinforcement Learning

by

Anna Hakhverdyan

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

Abstract

Most work in online reinforcement learning (RL) tunes hyperparameters in an offline phase without accounting for the said interaction. This empirical methodology is a reasonable approach 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 incompatible with exhaustive hyperparameter searches, and typical evaluations do not characterize how much data one must use for such searches. We investigate how to do *online tuning*, where the agent must select hyperparameters during interactions. Hyperparameter tuning is part of the agent rather than a separate *hidden* phase. We layer the Bayesian optimizer over standard RL algorithms and assess behaviour when tuning hyperparameters online. We show the expected result - this strategy's success depends on the environment and the algorithm. We introduce a way of tuning that mitigates wasteful resetting and shows that it can achieve comparable but not better performance levels than the default values, highlighting the need for further development.

Preface

No parts of this thesis have been published yet.

The bird fights its way out of the egg. The egg is the world.

– *Demian*, Hermann Hesse.

To my parents, brother, and little Levon
for being the anchor to my existence.

Acknowledgements

I would like to begin my acknowledgments by expressing my gratitude to my supervisor, Martha White, for her unwavering support and mentorship during the past two years at the University of Alberta. Her contributions to our project and guidance regarding my future career have been invaluable. Through her detailed feedback, I have greatly improved my ability to approach questions and enhance my writing skills. I am incredibly thankful to have had her as my supervisor. I would also like to thank Adam White for his assistance in deepening my understanding of the empirical design of Reinforcement Learning algorithms.

I want to extend my gratitude to my unofficial supervisor and a great friend, Andrew Patterson, who had a profound influence on shaping my abilities as both a researcher and an engineer. He taught me how to approach, assess, and start to answer research questions. He was a constant help and support for the duration of my degree, leaving me excited by the possibilities after all our meetings. Thank you for being patient with me, and I look forward to working with you again in the future.

I also want to thank Marlos Machado, Rich Sutton, and Mike Bowling for giving me their time and energy to answer my mostly non-trivial inquiries. Your patience and unbounded knowledge helped me clarify the questions I had in my mind while teaching me ways to view a problem more intuitively and from different perspectives.

I am incredibly thankful to my lab mates and many great friends in the Reinforcement Learning and Artificial Intelligence (RLAI) Lab and Alberta Machine Intelligence Institute (AMII) for the community, guidance, and support during my time as a graduate student at the University of Alberta. I want to

thank Manan Tomar, Rohini Das, Parham Panahi, Shibhansh Dohare, Yazeed Mahmoud, Abrar Fahim, Vlad Tkachuk, Jordan Coblin, Suyog Chandramouli, Farzane Aminmansour, Khurram Javed, Esraa Elelimy, Golnaz Mesbahi, Alex Lewandowski, Olya Mastikhina, Haseeb Shah, Han Wang, Kushagra Chandak, Prabhat Nagarajan, Fernando Hernandez-Garcia, Aidan Bush, Jiamin He, Esraa Saleh, Jacob Adkins, Mohamed Ayman Mohamed, Abhishek Naik and many more people that impacted me one way or another.

I am also incredibly grateful for two particular lifelong friends, Kevin Roice and Diego Gomez, for their friendship and support during my time in Edmonton. Thank you for being a family to me, which I am sure will stay the same even when we are miles apart.

And finally, I want to thank my family for all the love and support they gave me during our time apart. You were the reason for me to enjoy the good days and push through the hard ones. Thank you for being by my side.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

Reinforcement learning (RL) (Sutton & Barto, 2018) is a field of machine learning focusing on sequential decision-making: how agents learn to make decisions by learning from the experience of interaction with some environment. The agent receives a state from the environment and takes action in response. The environment then provides the agent with a new state conditioned on the chosen action and a reward signal, which the agent uses to learn how to act in the environment. The agent's goal is to maximize the cumulative reward by making decisions that lead to high rewards.

There are different ways one can approach the designing of RL agents. In this study, we consider the *online reinforcement learning* setting, where the agent learns as it "lives" in the environment. Its life starts at some point with no prior knowledge about the world it interacts with, and it navigates it by making a series of decisions. The agent does not have access to a simulator, so it cannot learn in parallel on multiple copies of the environment simultaneously nor reset arbitrarily. In this setting, the evaluation of the agents starts from the beginning of the learning, as we prefer agents that start performing well early, as opposed

to only caring about producing agents that perform optimally at the end of the training phase, as in this case, the training phase is the agent's lifetime. This single lifetime-focused problem of the online RL setting reflects many real-world deployment scenarios, such as recommendation systems, robot learning, or process control (Afsar et al., 2022; Gu et al., 2017; Luo et al., 2022; Janjua et al., 2023; Lawrence et al., 2024). Even though there are many potential use cases of online RL, many limitations prevent us from deploying agents in this manner.

One of the prominent bottlenecks for the agents that learn online is dealing with the numerous hyperparameters that most algorithms have, like the learning rate, target network refresh rate, or exploration parameters, to name a few. A natural question arises: *how should one select these hyperparameters?* One option is to use the default hyperparameter values from publicly available packages (Raffin et al., 2021; Castro et al., 2018). However, there is no guarantee that these hyperparameters will perform well for the setting we are interested in - hyperparameters that performed well in one problem may not achieve good results in the other one (Obando-Ceron et al., 2024). Imagine two different problem settings: one where the agent starts in a state where it can collect rewards right away and another where the agent needs to explore its surroundings before it can start collecting rewards. The hyperparameters that work well for the first problem may result in undermining the performance of the second one. The first agent may need a higher learning rate and lower exploration rate, while the second may need a lower learning rate and higher exploration rate. If we use the default hyperparameters, we may set the same hyperparameters for both agents, which may lead to poor performance for either. These default hyperparameters were most likely carefully 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 problem at hand - radically different hyperparameters may be required for

different benchmark problems (Patterson et al., 2024b).

Let us consider tuning one of the hyperparameters that most agents have, the learning rate, to control the priority distribution of users on a High-performance computing (HPC) cluster. Testing different learning rates requires restarting the RL agent's learning for each value and letting it run for some time to understand whether it is a performant hyperparameter value. During that time, the RL agent may encounter a value that is not performing well, thus doing a poor job of assigning priorities to the users' schedules, potentially messing with the workflow of many for days.

Ideally, this hyperparameter tuning process should be a part of the agent, making it easier to account for the hyperparameter tuning and use all the environment interactions for learning in the environment. Especially if the RL agents can be more sample efficient, they won't need too many data points to find a performant hyperparameter configuration, letting them start controlling the system effectively immediately and receive as much reward as possible. Though obvious, *online tuning* is not the current standard - we do not design algorithms for this setting. Instead, most tune the hyperparameters in a separate non-observable phase - one not reported nor accounted for in the final performances stated. Such *hidden tuning* can be acceptable empirical practice if the goal is to understand our algorithms' final performances in a scenario that can be run numerous times in parallel copies while doing an exhaustive hyperparameter search, which is not a feasible expectation in the real world.

Moreover, several inadvertent consequences are coming from doing the hidden tuning. The main issue is that it allows the researchers to avoid developing deployable algorithms, consequently encouraging the use of an increasing number of hyperparameters because adding more hyperparameters improves performance

at no cost. However, the more hyperparameters we have to tune, the harder it becomes to use these algorithms in an online manner: for example, adding an entropy coefficient to the algorithms aids in exploration, but it is another hard-to-tune hyperparameter in the pool of many. Further, the hidden tuning obscures how much data one needs to get high performance, making it hard to use the results from the literature to decide which algorithms will work in real-world problems. Standard empirical results end up being less pertinent.

In contrast, in online tuning, all environment interactions count, and the agents are evaluated based on how quickly they begin to perform well without a separate hidden phase. There can be a few reasons why agents will start performing better fast. First, an agent may have fewer hyperparameters to tune and can settle on a performant setting of those hyperparameters faster. If an agent has no hyperparameters, which is the ideal case, we won't need any tuning, and it will focus on learning right away! Second, 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., 2018b). Third, an agent might reuse prior data, like one gathered under previous hyperparameter configurations, to better infer what hyperparameters to try next. In the online setting, this sequential optimization paradigm would be the one to consider the most, as we want the agents to improve continually using all the data gathered throughout their lifetime.

In the online tuning setting, the hyperparameter tuning phase is an *explicit part* of the overall agent interaction with the environment - no additional interaction with the environment, only the amount of the given budget. This online tuning setting (intentionally) blurs the line between tuning and learning. We illustrate the difference between online and hidden tuning in fig. 1.1.

Figure 1.1: Contrasting the typical hidden tuning setting (left) with the proposed online tuning setting (right). The online tuning setting requires the tuning to be a part of the agent, as it must tune the hyperparameters online. Hidden tuning layers the tuning of the hyperparameters outside of the agent-environment interaction, allowing a separate search to be performed.

Throughout the thesis, we investigate how to tune hyperparameters in online reinforcement learning. We introduce a generic online hyperparameter tuning approach that can be layered on any existing algorithm. We go through the details of different tuning methods and motivate standard Bayesian optimization as the tuner to convert RL algorithms with numerous hyperparameters into one that tunes its hyperparameters online. Although this approach can be a naive layering, which leads to suboptimal behaviors, it provides a default strategy to test algorithms in this new setting, enabling comparisons to previous algorithms in a budgeted way. This approach allows us to assess the state of the field and understand how sensitive our algorithms are for online tuning while also providing a baseline approach for online tuning algorithms. This budgeted approach is also a way to fairer and more reproducible comparisons (Khetarpal et al., 2018) between different algorithms, as it is not clear how much data is used to tune the hyperparameters in the hidden tuning setting.

We show the behavior of Soft Actor-critic (SAC) (Haarnoja et al., 2018) and Proximal Policy Optimization (PPO) (Schulman et al., 2017) algorithms in sev-

eral popular Mujoco (Todorov et al., 2012) environments. For the first part of the experiments, we use some portion of the given total budget to tune the hyperparameters by resetting the agent and the environment before trying a new hyperparameter configuration, mimicking the hidden tuning. We use a Bayesian optimizer as a meta-learner to give the agent the next set of hyperparameter values to try. We find that given small enough ranges and hyperparameter trials, SAC can start performing as well 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, which is a limitation of current algorithms as we discuss in chapter 3. However, using different environments helps us see that the hyperparameters also depend on the environments, proving the unreliability of default hyperparameters.

We also provide a simple modification to the previous experiment setup to mitigate the wasteful resetting of the agents. We use the total budget to tune and learn simultaneously by sharing the learned knowledge of the agent from configuration to configuration using some simple conditions. We show that this naive approach can achieve similar performance levels to the case where we reset the agents in the same set of Mujoco environments when using specific algorithms. We also tried this simple methodology in a continuing reinforcement learning environment, where there is no end, and we saw similar results to the episodic cases.

The contributions of the thesis can be summarized as follows:

- We introduce a new evaluation paradigm that eliminates the hidden hyperparameter tuning phase by tuning hyperparameters online within a given budget.

- With online tuning, we can achieve results comparable to or better than default settings in Mujoco environments, but these performance levels depend on environment specifics, algorithms, and the number of hyperparameters those algorithms have.
- We propose a simple non-resetting methodology, which performs similarly to resetting approaches but only works for the SAC agent in episodic and continuing environments.

Chapter 2

Background

In this chapter, we define Reinforcement Learning (RL), emphasizing the online setting, where the agent interacts with the environment directly. We then review the techniques used in the literature to tune the hyperparameters for the RL agents and illustrate the pros and cons of each approach and its use case in the online setting. We will dive into the details of the hyperparameter tuning methods, including hyperparameter sweeping, random search, Bayesian optimization, and other methods used in most machine learning fields. We will also discuss specific methods like population-based training, meta-gradient methods, and other techniques. Finally, we will discuss which methods may be a good fit for the online setting despite the challenges of tuning the hyperparameters online.

2.1 Reinforcement Learning

Reinforcement learning embodies the problem of learning to make decisions by interacting with an environment modeled as a reward-agnostic Markov-decision process (MDP). MDPs are defined by a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P} \rangle$, where $\mathcal{S}$ is the state

space, $\mathcal{A}$ is the action space and transition probability map $\mathcal{P}$, which maps a state-action pair, (s, a), to a state distribution, $\mathcal{P}(\cdot|s, a)$. With this formulation, the agent interacts with the environment, seeing state $s_t \in \mathcal{S}$, taking action $a_t \in \mathcal{A}$, seeing new state $s_{t+1} \in \mathcal{S}$, and receiving reward r_{t+1}. Its goal is to learn a policy function $\pi : \mathcal{S} \to \mathcal{A}$ that maps state to actions, which maximizes the expected cumulative reward. We can see the agent-environment interaction in fig. 2.1.

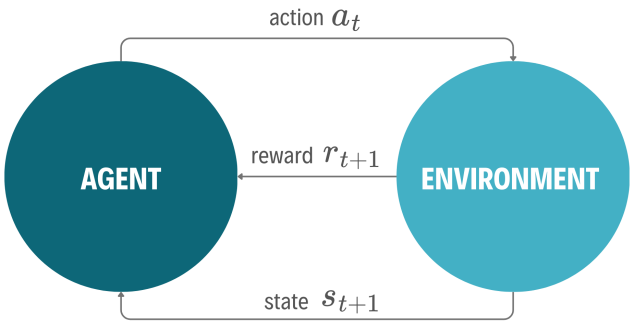

Figure 2.1: The agent-environment interaction loop in reinforcement learning. The agent takes action a_t in the environment given the current state s_t, receives reward r_{t+1}, and sees the next state s_{t+1}. The agent then uses this information to update its policy π and continues interacting with the environment.

2.1.1 Value Function and Policy Learning

To learn the policy π that maps states to actions, we usually learn value functions - functions of states that estimate the expected cumulative reward the agent can get at that given state when it follows the current policy π after that, which is called *policy evaluation*.

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s\right], \text{for all } s \in \mathcal{S}$$

One can also estimate the value functions by learning action-value functions, which estimate the expected return one can get after taking action a in state s:

$$q_\pi(s,a) \doteq \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s, A_t = a\right]$$

The value of the state is the expectation of the action-value function over all possible actions:

$$v_\pi(s) \doteq \sum_a \pi(s,a) q_\pi(s,a), \text{for all } s \in \mathcal{S}$$

Having an updated value function, we can apply *policy improvement* to update our current policy, acting greedily according to the updated value function. The new greedy policy takes at each state the action that appears best according to $q_\pi(s,a)$.

$$\pi'(s) \doteq \arg\max_a q_\pi(s,a) = \arg\max_a \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s, A_t = a]$$

Generalized policy iteration (GPI), which iteratively does policy evaluation and policy improvement, improves the agent's policy. Although there are numerous GPI algorithms in the literature, the most used and popular one is the Q-learning. Q-Learning uses TD-error, usually denoted with δ, to update the value functions. TD error is the difference between the current estimate and the estimate of the next state. So, it tells us how much the agent's estimate was off for the value of the current state. With an approximate value function $V \approx v_\pi$, we can calculate TD-error as follows:

$$\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$$

Q-learning is an off-policy method where we learn the value function from a policy that is different from the one we want to improve. Q-learning update rule with approximation of action-value function $Q \approx q_\pi$ is given by:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_{a \in \mathcal{A}} Q(S_{t+1}, a) - Q(S_t, A_t) \right]$$

Sarsa is an example of an on-policy method, where we learn the value function from the same policy we want to improve. Sarsa update rule is given by:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t) \right]$$

The difference between the two methods is how they update the value function. Sarsa updates the value function based on the next action the agent will take while Q-learning updates the value function based on the best action the agent can take.

2.1.2 Online Reinforcement Learning

In online reinforcement learning, the agent interacts with the environment directly, updating its policy based on the observations and rewards it receives. The agent learns online, meaning it learns while interacting with the environment, without a separate training phase.

We separate the online setting into two categories: episodic and continuing. In the episodic setting, the agent interacts with the environment for a fixed number of episodes, while in the continuing setting, the agent interacts indefinitely. In the episodic case, the agent has a total budget of interaction T (global step count), generating a trajectory τ of interaction over this lifetime (in this case,

the trajectories are the episodes) and is evaluated based on some performance measure $g(\tau)$ over the entire lifetime, which can be the average (discounted) return per step of the performance measure over the lifetime. Namely, if the agent is currently in episode n that started at timestep t_n with a duration of L, the (discounted) return is $G_n = \sum_{t=t_n}^{L} \gamma^t R_{t+1}$ for that episode, then the overall performance $g(\tau) = \frac{1}{N} \sum_{n=1}^{N} G_n$ where N is the number of episodes seen in the lifetime T.

For the continuing setting, a typical measure of performance is the average reward $g(\tau) = \frac{1}{T} \sum_{t=1}^{T} r_t$, where T is the number of steps taken in the environment.

2.1.3 RL-Glue Interface

As shown in fig. 2.1, the agent interacts with the environment by acting and receiving observations and rewards. Most of the time, these parts are separate, and the agent and the environment need a way to communicate by developing interactive programs. RL-Glue (Tanner & White, 2009) helps with achieving that.

RL-Glue is a software framework designed to help researchers create and evaluate reinforcement learning algorithms in a standardized manner. It focuses on online, single-agent reinforcement learning problems, specifying how agents and environments should interact by organizing their components to follow specific communication rules, the problem set we are considering throughout the thesis. The RL-Glue interface includes four main components: the agent, the environment, the RL Glue interface, and the experiment. The agent chooses actions based on the observations received from the environment, while the latter generates new observations and rewards based on the agent's actions. The experiment

part manages the experiment's execution, including the sequence of interactions between the agent and the environment, and evaluates the agent's performance. The RL-Glue program facilitates communication between the agent and environment in response to commands from the experiment program. This interface allows researchers to easily compare different algorithms and environments and test their algorithms on various problems. The RL-Glue interface is in fig. 2.2.

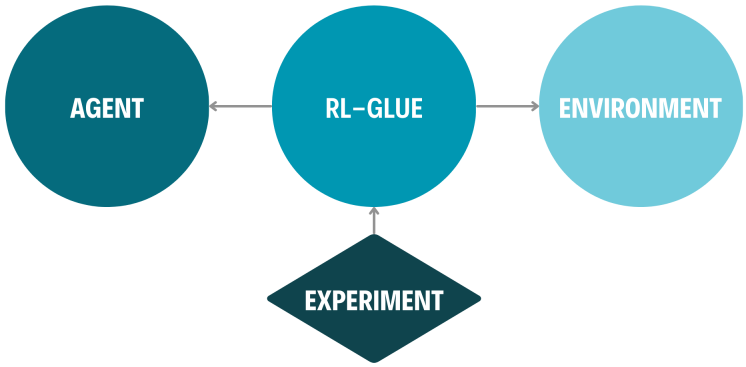

Figure 2.2: The RL Glue interface between the agent and the environment. The agent selects actions based on the states it receives from the environment, and the environment generates states and rewards based on the actions taken by the agent. The RL Glue interface provides a standardized communication path for the agent and the environment.

We use this interface to evaluate the agent's performance online, where the agent interacts with the environment directly. The interface allows us to assess the agent's performance based on the interaction with the environment and to tune the hyperparameters online.

2.2 Hyperparameter Tuning

Hyperparameter tuning (Feurer & Hutter, 2019) is an essential step in training all machine learning models, including reinforcement learning agents, which, in

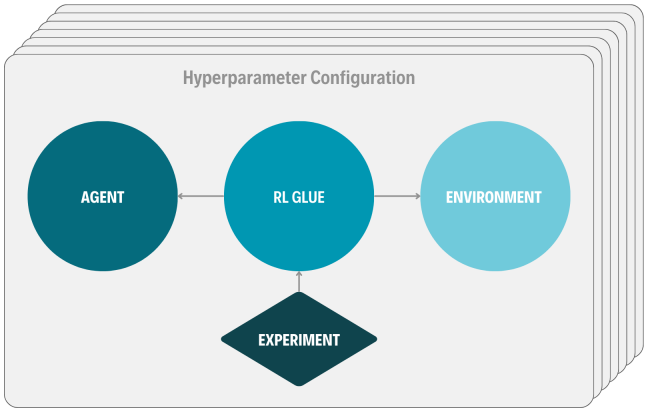

Figure 2.3: The usual hyperparameter tuning process. The agent uses different hyperparameter configurations and evaluates performance for each set of hyperparameters (the gray rectangle) independently. The best hyperparameter configuration is selected based on the final performance.

most cases, takes place as the most computationally expensive part of the training process. Hyperparameters are values set before or during the learning process, such as the learning rate α, which controls the speed of learning, the discount factor γ used in calculating the return G_i, which the agents try to maximize, or the exploration rate ϵ in the case of Q-learning-based algorithms, which controls the agent's exploration-exploitation trade-off. Mostly, these parameters are not learned during training but are set before the learning starts and can significantly affect the model's performance. A good set of hyperparameters can get to near-optimal performance, while a poor set of hyperparameters can lead to bad performance or even failure to learn (Obando-Ceron et al., 2024). Thus, agent designers often spend significant time and computation tuning numerous hyperparameters to get the best possible agents in their limited budget. A depiction of the hyperparameter tuning process is in fig. 2.3.

There are numerous methods for tuning hyperparameters in the literature, including ones used in many machine learning disciplines, like hyperparameter

sweeping or grid search, random search, Bayesian optimization, and other more specific tuning methods. Each method has advantages and disadvantages, and the choice of method depends on the problem we face and the resources available, which we will discuss next.

2.2.1 Hyperparameter Sweeping or Grid Search

Hyperparameter sweeping, or grid search is a straightforward method for tuning hyperparameters. It involves defining a grid of hyperparameters to search over and evaluate the model's performance for each combination of hyperparameters. The grid search method is easy to implement and can search over a wide range of hyperparameters. However, grid search is computationally expensive, especially when searching over many hyperparameters or when the hyperparameters have a wide range of values. If we want to evaluate N hyperparameters, each with K possible values, we would need to try out K^N combinations, which can become computationally impractical for large N and K. Grid search is also inefficient, as it does not consider the results of previous evaluations when selecting the next set of hyperparameters to evaluate, even though this weakness makes it easy to parallelize.

2.2.2 Random Search

Random search (Bergstra & Bengio, 2012) is an alternative to grid search that, instead of splitting the hyperparameter space into a grid and evaluating all possible combinations, randomly samples hyperparameters from a predefined distribution, usually uniform, in a given range. Random search can explore a different set of hyperparameter values than the ones specified by the grid. Random search is

easy to implement and can run parallel to speed up the search process. However, a random search is still inefficient, as it does not consider the results of previous evaluations when selecting the next set of hyperparameters to evaluate, requiring numerous hyperparameter configurations to find a performant set of hyperparameter values. So, grid and random search are inefficient in computational resources and time, especially in the online setting, where the agent must learn while interacting with the environment, and all the data is valuable.

2.2.3 Bayesian Optimization

Bayesian optimization (BO) (Snoek et al., 2012) is a more advanced method for tuning hyperparameters that uses a probabilistic model to model the relationship between hyperparameters and some objective function, which in our case is the average performance of the agent given the hyperparameter configuration. Bayesian optimization is an iterative process that uses the results of previous evaluations to select the next set of hyperparameters to evaluate.

Assume we run an algorithm with hyperparameter $h \in \mathcal{H}$, a set of all possible combinations of the hyperparameters, in the given environment to generate a trajectory τ. Let $g(\tau)$ be the average performance for that trajectory, and $G(h) \doteq g(\tau)$ is the random variable. Let's define the true $v(h) = \mathbb{E}[G(h)]$, the expected value across all trajectories that we could have seen when using h due to stochasticity in the environment and the policy, and other factors, like the hardware used. The Bayesian optimizer tries to model the expected return $\mathbb{E}[G|h]$ and chooses a point in the objective function we want to approximate. In the case of RL, it is to pick a hyperparameter configuration that will maximize the return $v(h)$:

$$h^* = \arg\max_{h \in \mathcal{H}} v(h)$$

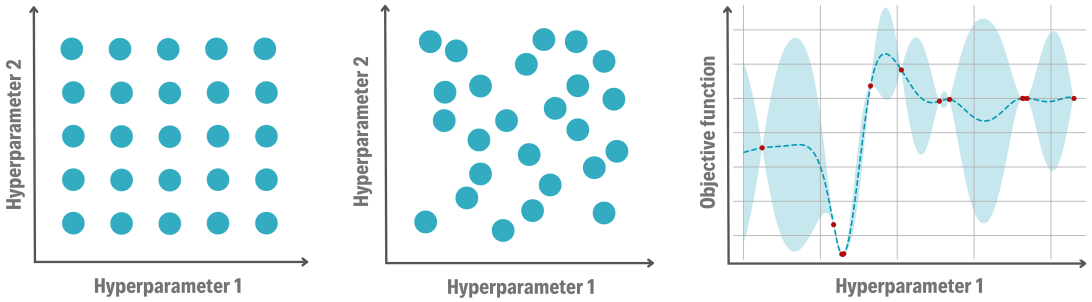

Figure 2.4: Visual representation of the hyperparameter tuning methods. Grid search (left) evaluates all possible combinations of hyperparameters, random search (middle) samples hyperparameters randomly from a predefined distribution, and the Bayesian optimization process (right) models the relationship between hyperparameters and the objective function to select the next set of hyperparameters to evaluate.

As we can see in fig. 2.4, Bayesian optimization is more computationally efficient than grid search and random search, as it can explore the hyperparameter space more effectively and requires fewer evaluations to find the best set of hyperparameters. Its most prominent advantage is that it can model the relationship between hyperparameters and the objective function and use this information to select the next set of hyperparameters to evaluate. Sometimes, only a few samples are enough for the BO to model the function well. However, Bayesian optimization has its hyperparameters to tune, like the kernel function, the acquisition function, and the number of samples to evaluate, which can affect the performance of the optimization process. However, thanks to the extensive literature on Bayesian optimization (Balandat et al., 2020; Falkner et al., 2018; Springenberg et al., 2016; Lizotte et al., 2007), many libraries provide easy-to-use interfaces for implementing Bayesian optimization (Bergstra et al., 2013; Akiba et al., 2019; Lindauer et al., 2022). Overall, for online tuning, Bayesian optimization can be the most effective method for tuning hyperparameters, as it can find the best set of hyperparameters with fewer evaluations in a sequential process, which we need in the online setting.

We use the TPESampler (Watanabe, 2023) from the Optuna (Akiba et al., 2019) software package in all of our experiments in this thesis. TPE, or Tree-structured Parzen Estimator, optimizes hyperparameters by dividing the search space into high-performing and low-performing groups. It then uses Parzen estimators, a technique for estimating probability distributions by placing "bumps" on data points, to model the likelihood of hyperparameters belonging to each group. Based on these models, TPE selects new hyperparameters to evaluate, focusing on those with the highest expected improvement. This process iteratively refines its estimates as it explores the hyperparameter space. One of the benefits of this method is that it works with all types of hyperparameters - integers, floats, and categorical variables, making tuning all the hyperparameters possible.

2.2.4 Other HPO Methods

There is extensive literature on using HPO methods specifically targeted for RL, a subfield often called Auto RL (Parker-Holder et al., 2022). The non-stationary nature of the RL setting makes the hyperparameter tuning process even more difficult. As the agent learns, the optimal hyperparameters may change, and the hyperparameters that were performant at the beginning of the training may not be optimal at the end. This non-stationarity can make finding the best set of hyperparameters difficult, as the agent's performance may change over time. Thus, AutoRL methods are the starting point for developing HPO methods that address issues present in RL.

One of the well-known methods is evolutionary algorithms, which mimic the process of natural selection to search for the best set of hyperparameters. These methods use a population of agents to search for the best set of hyperparameters, and they evolve the population over time to find the best set of hyperparameters.

A popular evolutionary algorithm is the population-based training (PBT) (Jaderberg et al., 2017; Parker-Holder et al., 2020; Parker-Holder et al., 2024; Wan et al., 2022; Parker-Holder et al., 2020) methods. These methods use a population of agents to search for the best set of hyperparameters. They parallelize the computation of hyperparameters by running different configurations simultaneously for some interval, rank the agents according to their return, replace the worst ones with the best ones, and perturb the hyperparameters on the newly replaced ones. After some time, these methods converge to a set of performant hyperparameter configurations. Although parallelizable and easy to use, these methods do not apply to the online tuning paradigm as this setting is sequential and non-parallelizable. Also, there is no guarantee that the algorithm can find a good set of hyperparameters in the set budget.

Some studies consider the interaction between the noise and instability in the RL setting and standard HPO methods (Eimer et al., 2022, 2023; Hertel et al., 2020). One work adapts PBT to the online setting by sharing experience (replay buffers) across the population of agents to find an optimal policy (Franke et al., 2021) more quickly. We similarly reuse experience for tuning the hyperparameters online in chapter 5, sequentially, not in parallel.

Some specific ideas could benefit the online tuning setting. Nguyen et al. (2020) uses information during training deep learning systems to assess performance and accounts for the cost of running additional training steps when deciding whether to test a hyperparameter. Other work similarly tries to avoid running expensive learning systems to completion by predicting performance using smaller datasets (Klein et al., 2017) or terminating early (Makarova et al., 2022). Such approaches still suffer from issues like sample inefficiency and tuning the agents for the starting point of the training, which is not ideal for the online RL, especially in the never-ending learning setting. Similar issues arise in one

of our experiments, Bayesian optimization with resets, which we will discuss in depth in chapter 4.

Some work avoids hidden tuning, either by developing methods that don't have hyperparameters (Jacobsen & Cutkosky, 2022) or exploiting offline data and simulators (Wang et al., 2022). Ultimately, a hyperparameter-free algorithm is an ideal approach for the online tuning setting because it can immediately learn from all available interactions rather than wasting interaction with the environment to find hyperparameters. A long-standing goal in RL is to develop automatic approaches for hyperparameters, typically for specific hyperparameters like the learning rate (Jacobsen et al., 2019; Kearney et al., 2018, 2019; Mahmood et al., 2012; Degris et al., 2024) or the eligibility trace parameter λ(Javed et al., 2024). There are a few works considering more general approaches to tune multiple hyperparameters at once, some using meta gradient descent (Xu et al., 2018a; Flennerhag et al., 2022), using off-policy learning to assess multiple policies (Paul et al., 2020), or methods that tune themselves (Zahavy et al., 2020), which is unfortunately limited to a few hyperparameter values. The other set of works that avoid hidden tuning is typically motivated by real-world deployment and leverages offline data or simulators to tune before deploying a fully specified agent into the real world (Letham & Bakshy, 2019; Zhang et al., 2021; Wang et al., 2022; Kiran & Ozyildirim, 2018). These approaches are essentially hyperparameter-free because we can deploy a fully specified agent without tuning. Though laudable, these algorithms need more development, and in fact, the online tuning setting is the right place to develop and test these approaches, which in turn will allow for a fair comparison of existing methods that have many hyperparameters.

Chapter 3

Tuning is Hard Under Many Hyperparameters

Hyperparameter tuning is a grueling problem in any field of Machine learning, which is more of a headache in Reinforcement learning as there are a lot more hyperparameters that affect the performance of RL agents by quite a large margin - the problem is even more evident as the data used by the agents differs due to the stochasticity of both the environment and the agent.

One of the hard-to-tune and environment-specific hyperparameters is the learning rate α - a hyperparameter responsible for how much information from the gradients will pass to the networks and how fast or slow the networks learn. Thus, if the learning rate value is too small, the networks will take ages to learn, and too large of a value will make the networks diverge. Besides, the discount factor γ and the exploration rate ϵ are also hard to tune as they affect the learning of the agent immensely - how much an agent can explore is proportional to how well and fast it can find a proper solution, same happens with how much into the future can the agent see. Thus, in this section, we tune each of these

hyperparameters and their different combinations to see how they interplay with each other and if there is any difference in the final performance of the agent when we add more and more hyperparameters to tune. The overall goal of the experiments is to see whether the current tuning practices are good contenders for finding a satisfactory hyperparameter value on a limited budget.

Let us look at how hyperparameters of RL algorithms affect the performance in the Mountain Car environment. The environment is simple - the agent has to learn to drive a car up a hill. The agent receives a reward of -1 on each step it takes to reach the goal, and the episode ends when the agent reaches the goal at the top of the hill. The agent has three actions to choose from - push left, push right, or do nothing. The agent receives a reward of 0 when it reaches the goal. The optimal behavior of the agent is to drive up the hill by using the momentum it gets from the push-right action. We consider the agent performs well in this environment when the agent reaches an average return of -150 over 100 episodes (Patterson et al., 2024a). The agent takes 500 as the maximum episode length.

As an agent, we use the DDQN (Van Hasselt et al., 2016) algorithm, a variant of the DQN algorithm (Mnih et al., 2015) that uses two separate networks to estimate the Q-values, one for target Q-values and one for current Q-values, mitigating the overestimation issue present in DQN. The default DDQN agent has a lot of hyperparameters - namely, 13 that we account for in our studies. As you can imagine, tuning an agent with that many hyperparameters can be difficult, and we show that in this section. All the default hyperparameter values and ranges for all algorithms in all environments are in table A.1.

We consider tuning up to 4 different hyperparameters - learning rate α, discount factor γ, exploration rate ϵ, and the batch size. All the other hyperparameters have the default values used in Ceron & Castro (2021). We con-

sider the following hyperparameter ranges for the DDQN agent: learning rate α: $[0.0001, 0.001]$, discount factor γ: $[0.9, 0.99]$, exploration rate ϵ: $[0.1, 0.9]$, and batch size: $[32, 128]$. We compare the average performance when we tune only one hyperparameter, two hyperparameters, and three hyperparameters. We use Random search and Bayesian optimization to tune the hyperparameters.

We plot the performance of the DDQN agent in four different scenarios - the agent will try 5, 10, 20, and 50 different hyperparameter configurations using five different seeds while keeping track of the best performance it saw during those trials. It has 100K timesteps to evaluate each hyperparameter configuration and keep the score it got in the final 50% of its steps - we only consider the average return of the last 50K steps. For the Bayesian optimizer (BO), we use Optuna (Akiba et al., 2019). For each of the seeds, we allow the BO to have 3 test or warmup runs, where it usually tries out the values at the boundaries. After we evaluate a hyperparameter configuration in the environment, we give the data to the BO, which uses it to suggest better configurations to maximize the final return.

In fig. 3.1, we show the distribution plots of the maximum returns the DDQN agent got for separate runs. As we can see, tuning just one hyperparameter is an easy task, which, of course, depends on the assumption that other hyperparameters with default values are reasonably good and that the hyperparameter selection range is relatively small. In this case, BO finds a performant hyperparameter configuration even within five trials, outperforming random search, which is a consistent result across all the environments we tried, presented in appendix (fig. B.1). If we look closely at the results, we can see that the Bayesian optimizer finds a better hyperparameter configuration than the random search in all the cases. Also, interestingly, depending on the hyperparameter, even when we tune only one, the number of configurations to try to get to good performance

increases, meaning that the hyperparameter is quite sensitive to tune, like ϵ, for which random search struggles to find a plausible solution for, but Optuna finds a performant solution from the very start. These results reiterate the assumption that using Bayesian optimization algorithms fits hyperparameter tuning in RL agents.

But when we increase the number of hyperparameters for the search algorithms to tune, it requires more and more trials to find a well-performing hyperparameter configuration - it needs more than 20 trials to achieve the expected behavior of a good-performing agent shown in the red dotted line. These results suggest that hyperparameter tuning gets increasingly complex as we grow the number of hyperparameters to tune. Even though the Bayesian optimizer finds a better hyperparameter configuration faster than the random search, it still requires many trials to find a well-performing hyperparameter configuration when we increase the number of hyperparameters to be tuned. These results suggest that hyperparameter tuning is a problem in RL, and we need to think of better ways to tune hyperparameters in RL agents or spend time and energy on developing agents with fewer or no hyperparameters to tune.

In the online setting, in which we are interested in the thesis, we care about sequential performance more than the final performance. Thus, according to the results in fig. 3.1, we can see that the Bayesian optimizer finds a better hyperparameter configuration than the random search in all the cases. The results in this section suggest that using Bayesian optimization algorithms is a better choice for hyperparameter tuning in RL agents. Thus, we will use Optuna search to conduct online hyperparameter tuning experiments in the upcoming sections.

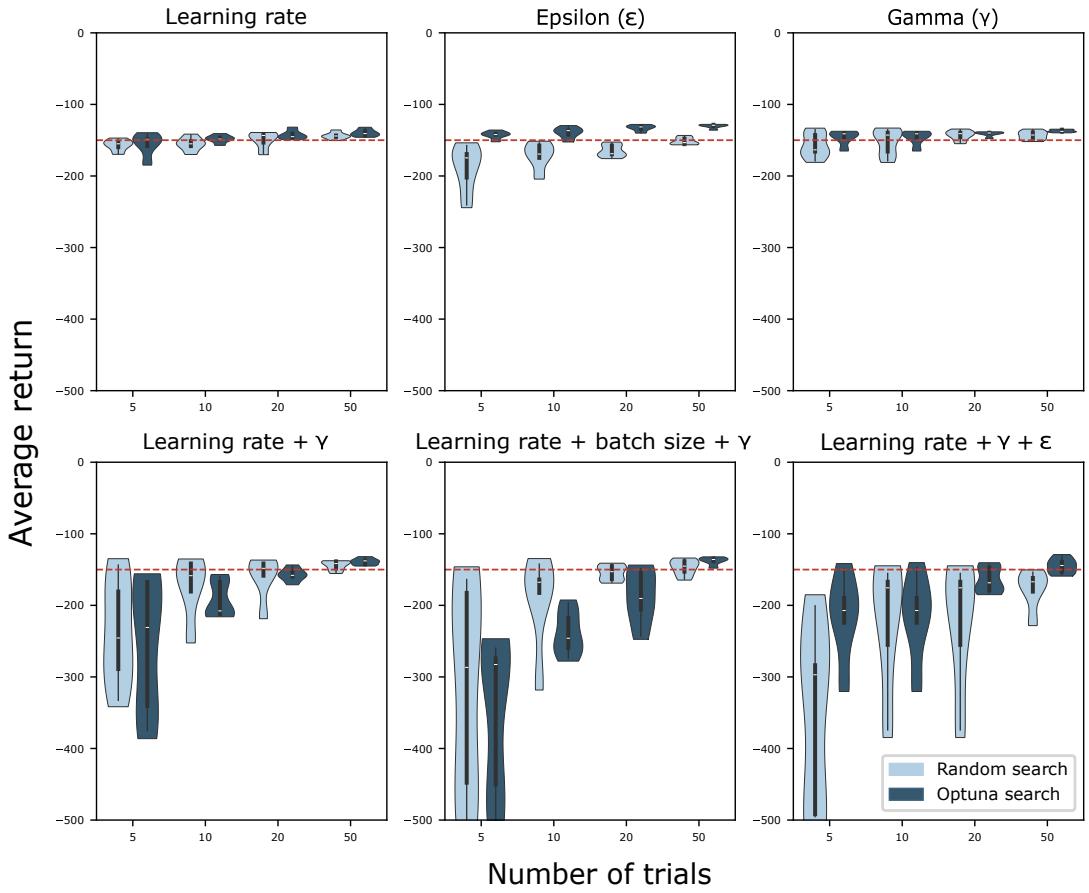

Figure 3.1: Distribution plots of the highest average return achieved by the DDQN agent in the Mountain Car environment over five seeds. The agent tries 5, 10, 20, and 50 different hyperparameter configurations. The hyperparameters tuned are the learning rate α, discount factor γ, exploration rate ϵ, batch size, and different combinations using those hyperparameters. The light blue depicts the random search, while the dark blue shows the Bayesian optimization. The red dotted line shows the expected performance of a well-performing agent, which is -150. The x-axis shows the number of hyperparameters tuned. The y-axis shows the average return of the agent. The maximum episode length is 500.

Chapter 4

Online Tuning with Resets

In this chapter, we tune the hyperparameters in an online setting as described in chapter 2. Before the agent begins interacting with the environment, it has a budget for environment interaction. The agent must sequentially select and evaluate hyperparameters within this budget. The agent uses initial interaction to tune hyperparameters and settles on a choice for the remainder of learning. This approach differs from hidden tuning, where one tunes the hyperparameters in one phase and evaluates the agent in another.

For this chapter, we consider the version of online tuning where we reset the agent and the environment after each hyperparameter evaluation. It is a simplified and sequential version of the hidden tuning phase, where we train the agent with a hyperparameter setting for a fixed number of steps, then reset the agent and the environment and evaluate it with a new hyperparameter setting.

As the hyperparameters are evaluated sequentially in the online tuning setting, the obvious and the most straightforward way to tune the hyperparameters is to use Bayesian Optimization (BO), as we can reuse the performance data of the previous runs to select a better hyperparameter configuration that will lead to

better performance. Of course, we can also use other methods mentioned in chapter 2, but, as we established, most are not as efficient and suited for sequential decision-making as BO.

The key to applying BO to the online tuning setting is how much interaction we should use to tune the hyperparameters. When searching for hyperparameters in the usual hidden tuning setting, this trade-off does not arise because all the allocated online interaction steps use the hyperparameter configuration found during the hidden tuning phase. But, in the online setting, the agent or agent designer has to select (a) how long to test each hyperparameter before resetting the agent and the environment and testing a new hyperparameter and (b) the maximum percentage of the lifetime to dedicate to testing different hyperparameter settings. The summary of this generic approach is in algorithm 1.

Algorithm 1 Online tuning **with resets**

Input: RL Algorithm `Alg`, hyperparameter set $\mathcal{H}$, global steps T, tuning steps M, trial length L
Initialize: Bayesian Optimizer (BO), max-perf $= -\infty$, best-h = None, max tuning iterations $= M/L$
$h \leftarrow$ random point from $\mathcal{H}$
for $i = 1$ to max tuning iterations **do**
  Run `Alg` with h for L steps to get performance G
  Add (h, G) to the Optimizer
  If max-perf $< G$, then set best-h $= h$ and max-perf $= G$
  Reset `Alg` (reinitialize weights, clear buffer, clear the optimizer state, reset environment, etc.)
  Get next $h \in \mathcal{H}$ from the BO
end for
Run `Alg` with best-h for the remaining $T - M$ steps

Let us consider an example. The agent designer has a budget of 3M steps to give for interaction with the environment, and they want to tune the hyperparameters for 2M steps, as the agent designer believes that 2M steps are enough to find a good hyperparameter setting. They will use the last 1M steps to de-

ploy the best hyperparameter setting found. The agent has 5 hyperparameters to tune, and they decide to test each setting for 200K steps. Combining these two, the agent has 10 hyperparameter settings in total to test in the given 2M steps budget. This is a small number of hyperparameter configurations to test, which may not be enough to find an optimal value. Doing a grid search on a cross-product of even 2 choices per hyperparameter would already take testing $2^5 = 32$ hyperparameter settings, but the agent designer has no budget to try this method. However, with correlations between hyperparameters, meaning the ease of tuning hyperparameters together, the agent designer hopes that testing as few as 10 hyperparameter settings will be enough to find reasonable hyperparameters.

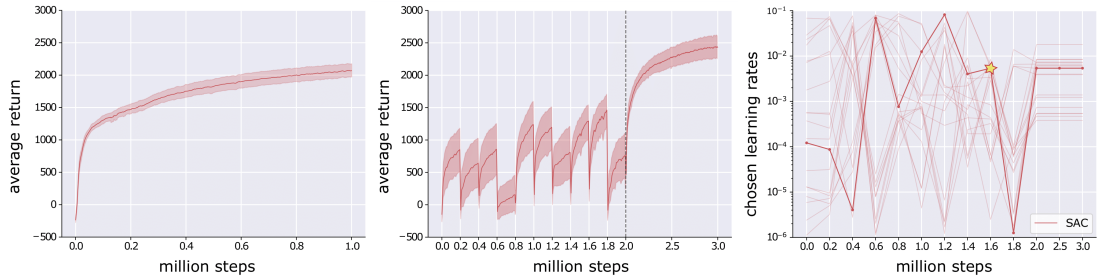

Figure 4.1: SAC in HalfCheetah using default hyperparameters versus online tuning of the stepsize using BO with resets. The leftmost figure depicts the average performance of the SAC agent with the default hyperparameters provided in the literature. In the middle, we show an example of our proposed online tuning strategy, where we have 3M steps as a budget. We stop tuning the hyperparameters after the 2M mark (the dotted line) and run the last million steps with the best hyperparameter configuration found. The last plot shows the learning rates chosen by the optimizer in the first 2M steps for each of the individual runs of the second plot. The dark line shows the values tested for one seed and the yellow star points to the best learning rate picked. The shaded area is a 95% bootstrapped confidence interval of 20 independent runs.

We visualize such an experiment in fig. 4.1, where we show the performance of the SAC agent in HalfCheetah, a well-used environment in the Mujoco physics engine where the agent has to learn to control a half-cheetah to run as fast as possible. We compare the performance of the SAC agent with the default hy-

perparameters provided in the literature and also show the values of the learning rate hyperparameter chosen during the interaction. As we can see, when tuning only the learning rate, the agent can find a better configuration than the default hyperparameters in just 2M steps. We set the budget of the default hyperparameters runs to the same amount as the evaluation steps given to the SAC agent to make the comparison fair. We also visualize the sequence of values tested in each run. It is worth noting how much the chosen learning rate value can vary between runs. Unlike hidden tuning, there is not one value used for all the independent runs; each run may have a unique stepsize. This is one of the key differences between hidden and online tuning, as we treat each agent individually rather than as a part of the population, which is an important feature in real-world applications.

Now, we evaluate the online tuning with the resets using SAC and PPO agents in five different Mujoco environments - Ant, HalfCheetah, Hopper, Walker2D, and Reacher. For this set of experiments, we let Optuna do no warmup trials - it starts with a random value in that given range and starts the optimization process.

As we consider the online setting, we don't train PPO with parallel copies of the environment; it is single-stream, an interaction with just one instance of the environment. As in the previous example, we give all the agents an overall budget of 3 million online steps, $L = 200$K evaluation steps or trial length for each hyperparameter setting, and use two different stopping conditions: after 1 million steps and after 2 million steps. In other words, the agent can test 5 hyperparameter settings in the first setting, and in the second, it tests 10. We compare the agent's performance with the default hyperparameters and the online tuning strategy.

We additionally consider the effect of the number of hyperparameters that are tuned. For SAC, we test two scenarios: tuning only the learning rate (one hyperparameter) and tuning five hyperparameters: the learning rate, the discount factor γ, the entropy coefficient, the target refresh coefficient τ, and the reward scale. PPO, on the other hand, has 7 hyperparameters to tune: the learning rate, the discount factor γ, the GAE parameter λ, the entropy coefficient, the value function coefficient, the clip parameter, and the gradient normalization clipping factor. When tuning only the stepsize, we leave the remaining hyperparameters at the defaults. For details on the hyperparameter ranges and additional experiment details, see appendix A.

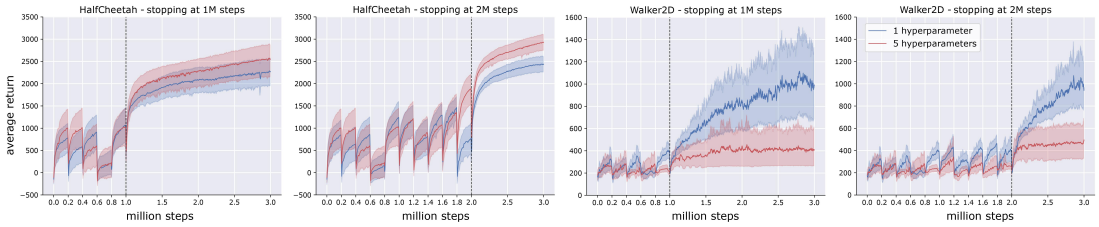

Figure 4.2: SAC in HalfCheetah and Walker2D environments using two different stopping conditions while tuning one and many hyperparameters. Each agent had an overall 3 million steps budget, and each hyperparameter had a 200K trial length. The gray dotted line depicts the timestep when the agent stops testing different hyperparameters and deploys the best configuration found. The blue line corresponds to the agents that had to tune one hyperparameter, while the red line depicts the agents with many hyperparameters. Note that SAC with default hyperparameters reaches scores of approximately 2000 and 800 in HalfCheetah and Walker2D, respectively. The shaded area is a 95% bootstrapped confidence interval of 20 independent runs.

We first examine the effect of tuning one hyperparameter versus many for SAC on HalfCheetah and Walker2D environments, shown in fig. 4.2. We selected these two from the 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 doesn't need to be close to 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 more stark when tuning for 2 million steps. The further improvement makes sense, given that the agent can test 2x as many configurations, getting even more performance gains. In Walker2D, on the other hand, the increased flexibility is detrimental.

Next, we investigate the performance of both PPO and SAC agents in all five Mujoco environments. From the results in fig. 4.3, we can see 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 learning rate, PPO's performance is comparable to SAC, but when we tune seven hyperparameters, the performance drops significantly, making the gap between the agents noticeable. These results can be due to PPO having more hyperparameters to tune, the hyperparameters being more sensitive to the environment, or the given trial length being too short to find good hyperparameters. SAC, on the other hand, appears 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. 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. 4.3 and the additional results for stopping at 2 million steps in the appendix, in fig. C.1.

One of the hard choices made in the previous experiment is the trial length for each hyperparameter configuration we got from the Bayesian optimizer. It can be considered another hyperparameter added to the list of many hyperparam-

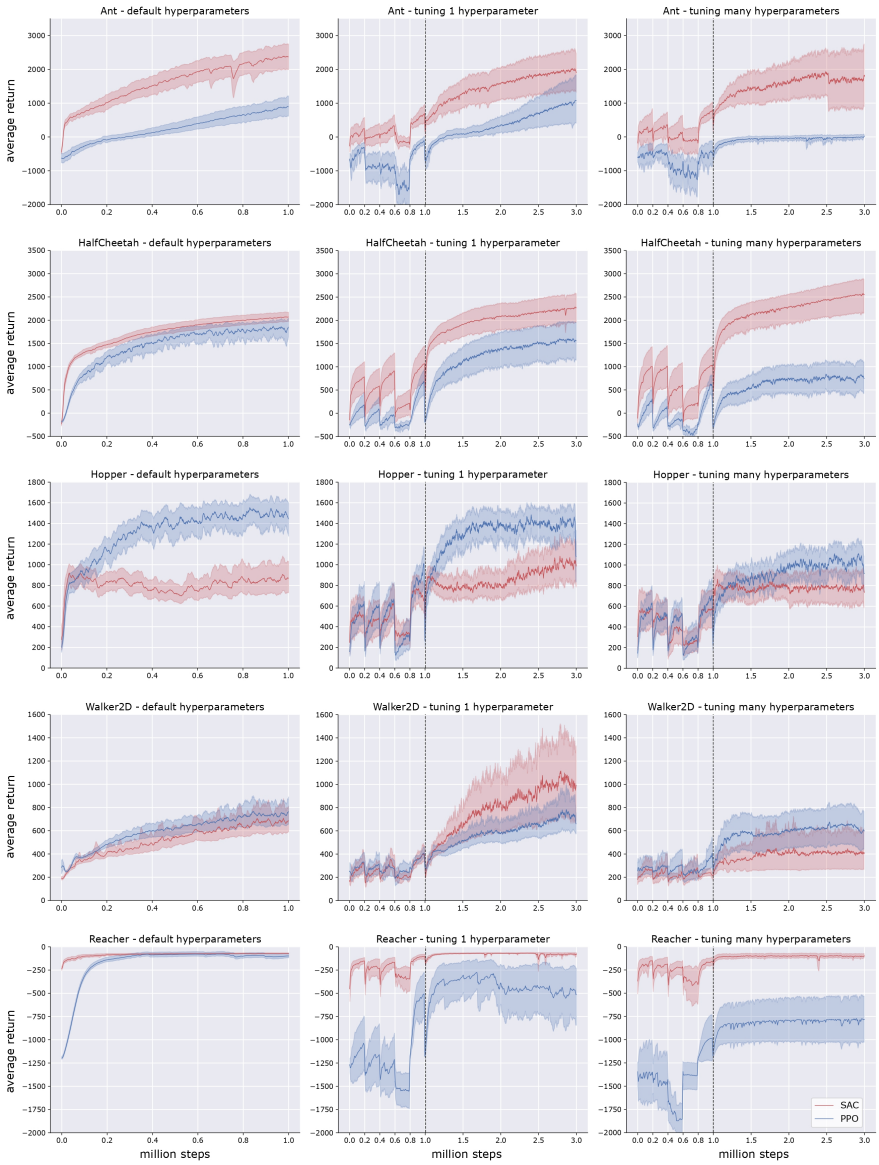

Figure 4.3: SAC (red) and PPO (blue) algorithms in multiple Mujoco environments. In the first column, we have the performance of the algorithms with the default hyperparameters. The second and third columns show the algorithms' performances within the 3 million budget, where we stop hyperparameter optimization after 1M steps with the difference of tuning one and many hyperparameters. The dotted line depicts the timestep the agent started deployment with the best hyperparameter configuration found. The shaded area is a 95% bootstrapped confidence interval of 20 independent runs.

eters that the agent designer should choose before starting the agent's training process. Thus, with the upcoming experiments, we want to see whether there is any impact on the performance if we let the agent try each hyperparameter for 500K steps instead of 200K with the same budget as in the previous experiment. This idea is motivated by the fact that depending on the algorithm we train the agents, the time to get to good performance can vary significantly. As SAC updates its policy more frequently than PPO, it might need less time to get to a good performance for the Bayesian optimizer to find better hyperparameter configurations. PPO agents, on the other hand, might need more time to get to comparable performance, and the 200K trial length might not be enough for the optimizer to suggest a better hyperparameter configuration. Thus, we opted to test the same experiment with an equivalent budget but with a longer trial length. We chose the trial length of 500K, as PPO agents achieve reasonable performance after 500K steps in the environment (Schulman et al., 2017; Freeman et al., 2021b).

As before, we consider two scenarios: one with enough budget to test 5 hyperparameters and the other with 10. Thus, as we give the trial length of 500K, the budget for the agent will be 3.5M and 6M steps, respectively, where the agent uses the chosen hypers for the last 1M steps. We compare the performance of 500K trial length agents with the 200K ones and show the results in fig. 4.4.

We observe that agents with a trial length of 500K steps perform slightly better than those with a 200K trial length across all environments. This difference is particularly noticeable for the PPO agent, which, as a trajectory-based algorithm, requires more time to update its networks with additional trajectories to achieve good performance. Despite this, the PPO agent still cannot outperform the SAC agent, which updates with every environment step. This finding aligns with previous results, where the SAC agent was less sensitive to hyperparameters

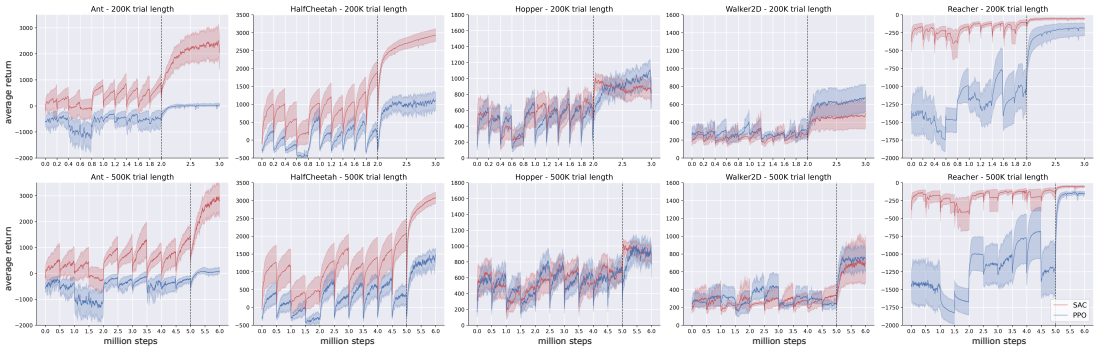

Figure 4.4: SAC (red) and PPO (blue) agents in multiple Mujoco environments with 200K and 500K trial lengths. The first row shows the agents' performance in all five Mujoco environments with the 200K trial length. The second row shows the performance with the 500K trial length. To make the comparison fair, we give an equivalent budget in both cases and let them deploy the best hyperparameter configuration found in the last 1M steps. The dotted line depicts the timestep agents start using the best hyperparameter configuration found. The shaded area is a 95% bootstrapped confidence interval of 20 independent runs.

than the PPO agent. Although the SAC agent also experiences a slight performance boost with increased trial length, the improvement is insignificant. These results indicate that while trial length is a hyperparameter, it does not impact performance as much as the number of hyperparameters to tune. Consequently, agent designers may prefer shorter trial lengths to test more hyperparameters within a given budget.

Next, we will examine whether increasing the agent's budget affects performance. We will conduct the same experiment as before, but now the agent will have a budget of 10M steps, with the final 1M steps used to deploy the best hyperparameter configuration found. The results are in fig. 4.5.

The results for 10M steps are comparable to the one in fig. 4.4, and only in some environments do we see slight differences. After letting the agents try more hyperparameters, we can see that the SAC agent gets better than the default performance on Ant and HalfCheetah environments. We also see a slight

Figure 4.5: SAC (red) and PPO (blue) agents in multiple Mujoco environments with 200K and 500K trial lengths with a total budget of 10M steps. The first row shows the agents' performance in all five Mujoco environments with the 200K trail length. The second row shows the performance with the 500K trail length. We use the last 1M steps to deploy the best hyperparameter configuration found. The dotted line depicts the timestep agents start the deployment with the best hyperparameter configuration found. The shaded area is a 95% bootstrapped confidence interval of 10 independent runs.

improvement in Walker2D and Reacher environments, even in PPO agents, suggesting what we had in mind so far: the hardness of the tuning depends on both the environment and the agent algorithms, and the more hyperparameters we have, the harder it is for us to tune all of them in a given budget. One thing to notice is that on the later trials of 200K trial length runs, the optimizer starts suggesting values that most likely will lead to better performance gains, which we want in the long run.

In this chapter, we discussed what can be a naive but effective way to tune the many hyperparameters that the agents we design have. We showed that we can tune them sequentially and get comparable or better performances than the default hyperparameters provided in the literature. But, these gains depend on the number of hyperparameter configurations we try and the length of each trial. As we showed in chapter 3, we need to try many configurations of hyperparameters to get to performant ones, which is quite visible when we compare the results

of fig. 4.2 with fig. 4.5. However, this naive extension of tuning the hyperparameters can be effective in most simulators we use to report the performance of new agents we design, and using this extension will limit the consumption of time and computation, leading to fairer comparisons as it comes with particular problems that make the shortcomings of the algorithms visible.

Chapter 5

Online Tuning without Resets

In this chapter, we explore the idea of online hyperparameter tuning in reinforcement learning, where we do not reset the agent's and environment's state after each hyperparameter setting. From the above chapter, you may have felt that resetting the agent's state after each hyperparameter setting is wasteful - we throw away all the learning and the data in the buffer accumulated by the agent throughout its lifetime. In the online setting, such data inefficiency is unacceptable - we want the agent to continually improve by adapting its hyperparameters. Also, resetting the agent's and the environment's state may not be feasible in some settings, like never-ending learning, where the agent must learn continuously.

Thus, we present a naive extension of the algorithm from the previous section to a setting where we do not reset the agent's and environment's state after each hyperparameter setting. The approach is simple but not optimal - a mere starting point going forward in the online setting. Despite its many limitations, this extension can perform as well as the standard offline tuning approaches. It is generalizable to different agents and environments, making it useful when the

agent cannot reset its state or the environment has no start state.

The idea is simple: we don't reset the agent's and environment's state after each hyperparameter setting, and it continues to learn with the given weights and buffer, allowing the agent to adapt and improve continually. In the pseudocode algorithm 1, this would involve removing the line that resets the agent and the environment and the maximum number of tuning iterations. However, there is an issue with this naive extension: some hyperparameters may result in poor performance, especially in the early stages of learning, as the optimizer may select speculative hyperparameters like the edge values to approximate the underlying function, which can lead to poor weights. It may be harder to continue learning from this poor state for a new hyperparameter setting, as it may prevent the agent from further improvement and lead to an unfair assessment of the new hyperparameter setting. We modify the algorithm slightly in an attempt to mitigate this problem.

The modification involves reverting to the previous weights if the new hyperparameter setting causes a drop in performance. In the first step, the agent selects a hyperparameter configuration, runs for L steps, and gets a performance estimate G. If this performance is below an acceptable threshold for the problem, the agent reinitializes the weights and the buffer. Otherwise, it continues from the last weights and buffer and selects a new hyperparameter setting. If, after running again for L steps, the agent obtains a performance estimate lower than the previous one by some threshold (e.g., 10% worse), then it reverts to those previous weights and buffer. The role of the threshold is to avoid resetting simply due to some stochasticity. Also, in early learning, it is unlikely for the performance of a reasonable hyperparameter setting to be worse than the one in the resetting case as it gets to learn starting from a better initial point (policy, buffer, weights).

Algorithm 2 Online tuning **without resets**

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

This modification is not perfectly robust to reverting the agent's state 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 the future development of algorithms for this online tuning setting. Our goal is to understand the benefits of starting to tailor algorithms, moving from the naive application of BO to one more specifically designed for the online setting. The pseudocode for this naive extension is in algorithm 2.

Now, let's look at the results of applying this approach to the same Mujoco environments as in the previous chapter. We hypothesize that avoiding resetting should allow the agent to obtain comparable or better performance than the resetting one. The experiment setup and details are the same as the resetting case.

We first examine the difference in the performance of agents that reset (algorithm 1) and don't reset (algorithm 2) in fig. 5.1. We also include a variant of an agent that doesn't reset - one that shares all the data from hyperparameter configuration to configuration. We can see that the naive variant (the middle figure) fails at 500k steps and does not recover, proving that bad hyperparameters can lead to poor weights that prevent further learning. The agent that

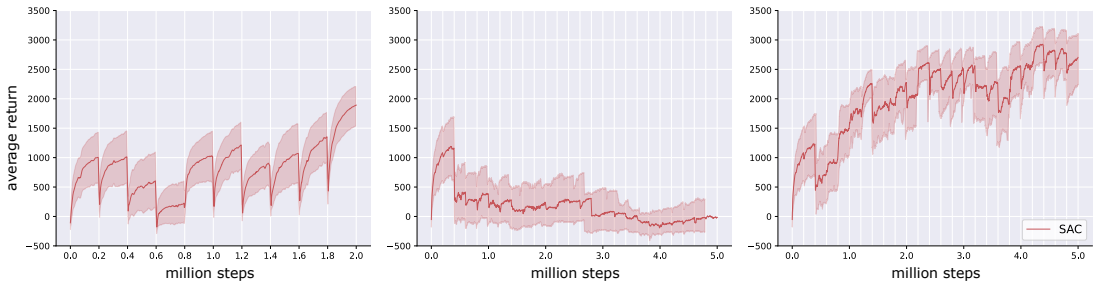

Figure 5.1: SAC in HalfCheetah, with three different online tuning strategies: agent state resets with no deployment period (Algorithm 1), a simple sharing of the agent state, and smarter sharing of the agent state (Algorithm 2). The shaded area is a 95% bootstrapped confidence interval over 20 different runs.

doesn't reset (rightmost figure) significantly outperforms the resetting results, steadily improving with time. However, this algorithm does need to recognize if a hyperparameter choice has led to poor weights to continue from an early set of weights, and this check is a limitation of this naive extension as it can be very environment-specific.

Next, we present SAC's performance for all five Mujoco environments in fig. 5.2 tested in the previous chapter. For this plot, we show the total area under the curve (AUC) over 3 million steps for agents that reset when stopping after 1 million steps and after 2 million steps (fig. C.1) and agents that don't. To make the AUC comparable across the environments, we normalize everything between 0 and 1. We normalize it according to the highest and the lowest values we can get on the given budget in each environment. For example, for all of the runs in the HalfCheetah environment, we can see that no run went above 3500 and lower than -500, which we use as the maximum and minimum numbers to normalize the data according to this equation:

$$X = \frac{X - \text{min\_value}}{\text{max\_value} - \text{min\_value}}$$

Figure 5.2: Box plots of the total AUC for the 3M evaluation budget for SAC agents tuning many hyperparameters in three different settings for all Mujoco environments showing the results for agents that reset with 1M and 2M stopping conditions and agents that share their state after each hyperparameter configuration.

Figure 5.2 shows that the proposed approach gets comparable overall performance as both resetting evaluation settings we tried in the previous chapter in all environments for 3M steps tried. SAC agents that don't reset with even the most naive augmentation on top increase their performance as they live on, performing comparably to the performance of the default hyperparameters.

Now, let's investigate if different trial lengths for hyperparameters impact performance similar to our approach in the previous chapter. We compare the agents' performance in the Mujoco environments using 200K and 500K trial lengths, providing an equivalent budget for both sets of agents. This experiment aims to determine whether allowing agents to test the hyperparameters for longer improves overall performance. We hypothesize that agents with shorter trial lengths will perform better, as they can evaluate more hyperparameters within the same time frame and quickly recover from poor hyperparameter settings.

As shown in fig. 5.3, agents with a 200K trial length outperform those with a 500K trial length across all environments, though only by a small margin, consistent results from the previous section. These intriguing results indicate

Figure 5.3: SAC agent in multiple Mujoco environments with 200K and 500K trial lengths. The first row shows the average performance in all five Mujoco environments with the 200K trail length. The second row shows the performance with the 500K trail length. In both cases, we give the same budget of 5M steps. The shaded area is a 95% bootstrapped confidence interval of 20 independent runs.

that even though agents with shorter trial lengths can test more hyperparameters and quickly discard poor ones, they do not significantly outperform agents with fewer hyperparameter trials. It may suggest a tradeoff: agents with a 500K trial length might spend more time with suboptimal hyperparameters but also have more time to learn from good hyperparameter settings.

To further understand the impact of trial length on agent performance in this non-resetting setting, we do another experiment where agents interact with the environment for twice as long, 10M steps, and compare performance with 200K and 500K trial lengths. The results of this experiment are in fig. 5.4.

As anticipated, when the agent tests more hyperparameter values, it can adjust more quickly, preventing a drop in performance. Although performance drop does not occur in the 500K trial length scenario either, visually, it may appear so due to the inclusion of interactions with poor hyperparameters. Overall, agents with shorter trial lengths seem to correct themselves more, as evident in the Hopper and Walker2d environments. However, the performance variance is quite

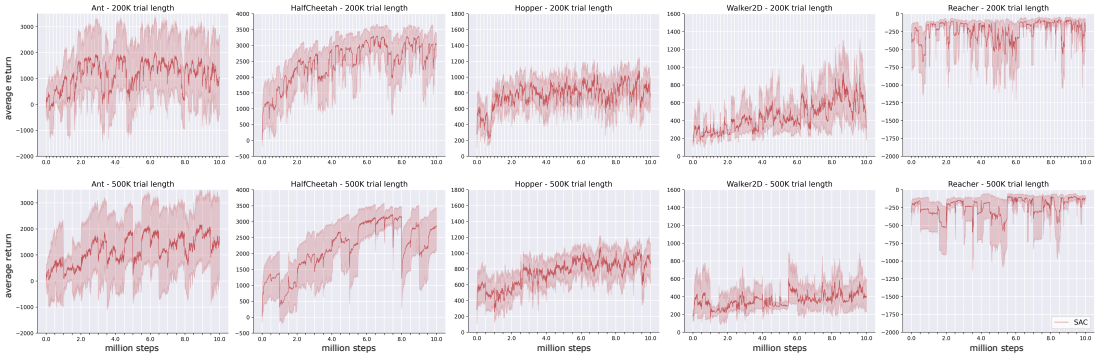

Figure 5.4: SAC agent in multiple Mujoco environments with 200K and 500K trial lengths with a total budget of 10M steps. The first row shows the agents' performance in all five Mujoco environments with the 200K trail length. The second row shows the performance with the 500K trail length. The shaded area is a 95% bootstrapped confidence interval of 10 independent runs.

large, as indicated by the substantial shaded area in the plots. To better understand individual agent performance, we plot them individually. Good single-seed performance is crucial because, in real-world applications, we typically rely on a single agent to interact with the environment, whether a robot or an intelligent agent in a water treatment plant.

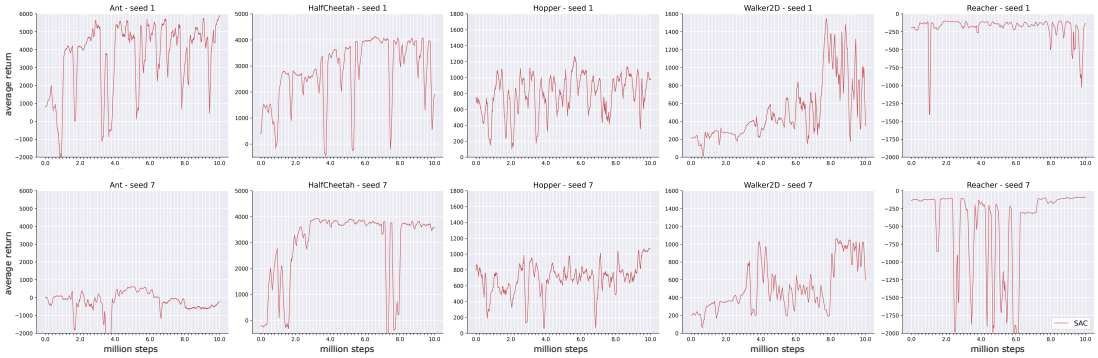

Figure 5.5: Individual runs of 200K trial length runs for 10M steps. The first row has the results for all Mujoco environments tried for seed 1, and the second row has the results for seed 7.

Figure 5.5 illustrates the performance of SAC using two different seeds: 1 and 7. Even when it is the same agent in the same environment, depending on

the seed, the performance can differ due to different initialization according to the seed. It is especially critical in the non-resetting case, where initial weight configurations play a role. If the agent starts with poorly performing weights, achieving good performance becomes difficult in this non-stationary Bayesian Optimization (BO) setting, leading to discrepancies in the independent runs, highlighting that, while this naive extension might work in some instances, it is not consistently reliable.

You may have also noticed that we didn't include the results for PPO agents in this section, as this approach does have the same impact on PPO agents as on SAC agents. Several factors might contribute to the cause of PPO not benefitting from this strategy, but two significant reasons stand out. First, PPO has more hyperparameters, making tuning more challenging. Second, PPO uses a clipped surrogate objective for training, which prevents the network weights from drifting too far from each other. It, in response, can cause the weights to remain close to their initial values, making PPO more sensitive to the initial weights and hindering performance in this naive sharing setting. The results for PPO are presented in the appendix fig. D.2, fig. D.3, and fig. D.4.

In conclusion, this sharing algorithm is the most straightforward extension of the resetting setting to be more apt to the online RL. Although this method is imperfect and only works in specific cases, it shows some progress toward designing algorithms or tuning methods that can adapt to environmental changes and tune and evaluate simultaneously.

Chapter 6

Never-ending Reinforcement Learning

Continual or never-ending RL (Khetarpal et al., 2022) is a subfield of RL that focuses on agents that learn continually without any resets. In this chapter, we explore the idea of online hyperparameter tuning in this never-ending setting, where the agent lives on and continues to learn in an environment that never ends. This setting is critical for the future of RL, as it is more realistic and closer to real-world applications.

Despite the importance of this setting, there is no good testbed. Some environment benchmarks can work in a never-ending setting include Crafter (Hafner, 2021), Nethack (Küttler et al., 2020), MineCraft (Guss et al., 2021), and robotics simulators (Wołczyk et al., 2021). Unfortunately, these domains are either costly to run or require massive agent architectures, posing problems in terms of computational access and compatibility with the long-running experiments necessary for never-ending learning.

Another option is Jelly Bean World (JBW) (Platanios et al., 2019), a foraging

domain where an agent explores an infinite grid world collecting jellybeans and avoiding onions. However, as new patches of the environment are procedurally generated, it results in an unbounded memory footprint, making long-running learning experiments impractical.

Given the lack of fast continuing environments, we developed a configurable and lightweight testbed for never-ending RL called Flower-picker: a finite grid-world where the agent collects flowers and avoids thorns. In this chapter, we apply the online tuning approach to this new environment to demonstrate the challenges of online tuning in a never-ending setting.

6.1 Flower-picker Environment

We introduce the Flower-picker environment, a never-ending grid world that requires exploration and adaptation to non-stationarity. It is a seemingly infinite environment containing collectible items.

The *world size* defines the grid size of the entire world. The environment wraps around itself to simulate an infinite world from the agent's point of view. Instead of being a simple grid with boundaries, it is a torus where the agent can travel indefinitely in any direction.

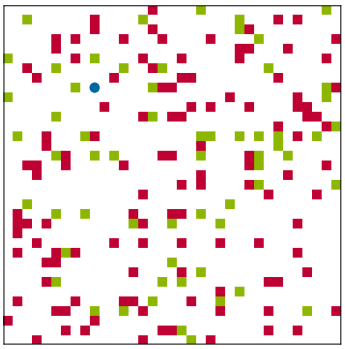

Figure 6.1: Example of the environment. The blue dot is the agent. The red and green squares are objects that give some reward when collected.

Figure 6.1 shows example configurations of the Flower-picker environment. The blue dot represents the agent. The green squares represent rewarding objects, referred to as *flowers*, while the red squares represent harmful

objects, referred to as *thorns*. Both flowers and thorns are collectible items that provide rewards of R_{flower} and R_{thorn}, set to 1 and -1, respectively, in our experiments. The actions are {*up, down, left, right*}, moving the agent in the corresponding cardinal direction. The agent gets 0 as a reward for every step taken in the environment.

The agent's observation at each step is a tensor of binary values corresponding to the state of each cell in its field of view, i.e., whether a cell is occupied or empty. The observations are agent-centered, meaning the agent is always at the center of the observation. The observation tensor has a shape of (*world size*, *world size*, *unique items*), where each layer corresponds to a different item type, making the environment fully observable to the agent.

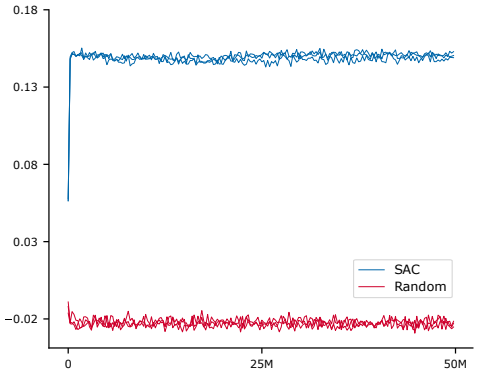

Figure 6.2: The performance of 3 individual seeds of SAC agent trained for 50M steps.

We use a simple configuration of the environment to test the online-tuning paradigm. In this setup, the ratio of flowers is relatively high, making it easier for the agent to find flowers by simply exploring the environment. Additionally, after collection, objects respawn at random, unoccupied locations. Despite being a simple configuration, it remains a challenging environment for the agent, testing its ability to explore different areas of the world.

We use a 15x15 environment, where 5% of the grid cells contain flowers and 10% thorns. The agent receives +1 as a reward for collecting a flower and -1 for collecting a thorn. The observation is a tensor of binary values representing the status of each cell in its field of view, i.e., whether a cell is occupied or empty.

The observation tensor has a shape of (15, 15, 2), where each layer corresponds to a different item type. The performance of the SAC agent for this environment configuration is around 0.15, as shown in fig. 6.2.

6.2 Online Tuning in Flower-picker

We use the online tuning method outlined in chapter 5 for the Flower-picker environment introduced earlier. The results, shown in fig. 6.3, indicate that this method successfully identifies nearly optimal hyperparameters for the SAC agent in the environment. However, as observed in previous experiments, the PPO agent has difficulty improving the performance, consistent with the findings in chapter 5.

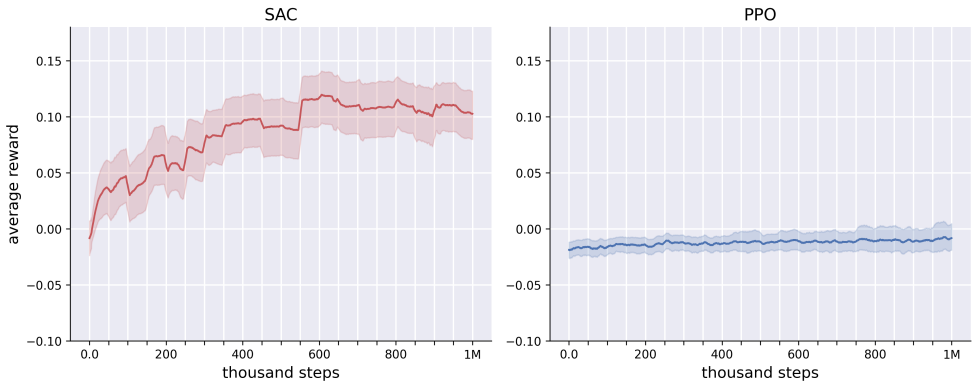

Figure 6.3: In the Flower-picker environment, we evaluate SAC and PPO agents for over 1 million steps in a 15x15 grid environment using the non-resetting paradigm, with each trial consisting of 50K steps. The results are 95% bootstrapped confidence intervals based on 20 independent runs.

Next, we evaluate the online tuning approach in the environment with an extended training duration of 10 million steps. The outcomes of this experiment are in fig. 6.4 where we showcase the individual runs of the agents.

The results are consistent with 1M results - the SAC agent gets to near-

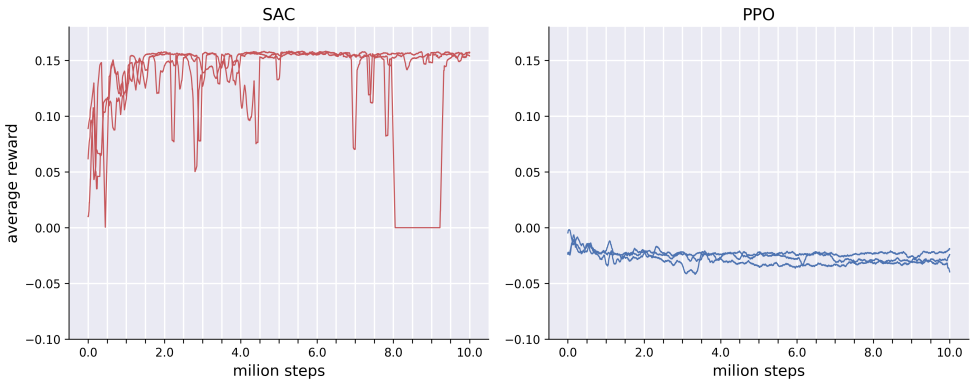

Figure 6.4: SAC and PPO agents are evaluated for 10 million steps in a 15x15 grid. Each trial consists of 50K steps. The results show the individual runs of 3 independent seeds.

optimal results after 10M steps, while PPO agents can't seem to learn much, being any better than a random agent. If we look closely, the individual runs displayed in the fig. 6.4 show that the SAC agents get the same value as in the fig. 6.2. These results illustrate that even though this naive extension of online tuning works on selective agents, namely SAC agents from our experiments, seeing their separate runs in this continuing environment where we share the weights, the buffer, and the optimizer state shows that developing methods that tune and learn simultaneously using online tuning methodology is necessary in the environments where the agent has a single lifetime and that we should investigate and make better algorithms.

Chapter 7

Conclusions and Future Work

We introduced 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 or even better than 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, making the algorithms with more hyperparameters to tune, like PPO, struggle 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 within a single lifetime. We also introduce a configurable and lightweight testbed for never-ending RL called Flower-picker and apply the online tuning approach to show the benefits of online tuning in a never-ending setting.

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 to outperform the current approach. Additionally, the Bayesian Optimization algorithms are not 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 agent's state. We also want to contribute to the reproducibility and fairness of the results by budgeting the interactions with the environment.

In the future, we plan to explore how the algorithm will behave if we add additional constraints to the Bayesian optimizer in the form of changes in the network or the agent's performance. We also plan to explore more advanced Bayesian Optimization algorithms that can handle non-stationarity better, like using neural networks as an acquisition function. Lastly, we plan to use more complex environments and algorithms to see how the online tuning approach behaves in more challenging scenarios.

References

Afsar, M. M., Crump, T., and Far, B. Reinforcement learning based recommender systems: A survey. *ACM Computing Surveys*, 2022.

Akiba, T., Sano, S., Yanase, T., Ohta, T., and Koyama, M. Optuna: A next-generation hyperparameter optimization framework. In *International Conference on Knowledge Discovery & Data Mining*, 2019.

Balandat, M., Karrer, B., Jiang, D., Daulton, S., Letham, B., Wilson, A. G., and Bakshy, E. Botorch: A framework for efficient monte-carlo bayesian optimization. *Advances in Neural Information Processing Systems*, 33, 2020.

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, 2013.

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

Bergstra, J., Yamins, D., and Cox, D. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In *International Conference on Machine Learning*, volume 28, 2013.

Castro, P. S., Moitra, S., Gelada, C., Kumar, S., and Bellemare, M. G.

Dopamine: A research framework for deep reinforcement learning. *arXiv preprint arXiv:1812.06110*, 2018.

Ceron, J. S. O. and Castro, P. S. Revisiting rainbow: Promoting more insightful and inclusive deep reinforcement learning research. In *International Conference on Machine Learning*, 2021.

Degris, T., Javed, K., Sharifnassab, A., Liu, Y., and Sutton, R. Step-size optimization for continual learning. *arXiv preprint arXiv:2401.17401*, 2024.

Eimer, T., Benjamins, C., and Lindauer, M. Hyperparameters in contextual rl are highly situational. *arXiv preprint arXiv:2212.10876*, 2022.

Eimer, T., Lindauer, M., and Raileanu, R. Hyperparameters in reinforcement Learning and how to tune them. In *International Conference on Machine Learning*, 2023.

Falkner, S., Klein, A., and Hutter, F. Bohb: Robust and efficient hyperparameter optimization at scale. In *International Conference on Machine Learning*, 2018.

Feurer, M. and Hutter, F. Hyperparameter optimization. *Automated machine learning*, 2019.

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

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

Freeman, C. D., Frey, E., Raichuk, A., Girgin, S., Mordatch, I., and Bachem, O. Brax - a differentiable physics engine for large scale rigid body simulation, 2021a.

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

Gu, S., Holly, E., Lillicrap, T., and Levine, S. Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates. In *2017 IEEE international conference on robotics and automation (ICRA)*, 2017.

Guss, W. H., Castro, M. Y., Devlin, S., Houghton, B., Kuno, N. S., Loomis, C., Milani, S., Mohanty, S., Nakata, K., Salakhutdinov, R., et al. The minerl 2020 competition on sample efficient reinforcement learning using human priors. *arXiv preprint arXiv:2101.11071*, 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 *International Conference on Machine Learning*, 2018.

Hafner, D. Benchmarking the spectrum of agent capabilities. In *International Conference on Learning Representations*, 2021.

Hertel, L., Baldi, P., and Gillen, D. L. Quantity vs. Quality: On hyperparameter optimization for deep reinforcement learning. *arXiv preprint arXiv:2007.14604*, 2020.

Jacobsen, A. and Cutkosky, A. Parameter-free mirror descent. In *Conference on Learning Theory*, 2022.

Jacobsen, A., Schlegel, M., Linke, C., Degris, T., White, A., and White, M. Meta-descent for online, continual prediction. In *Association for the Advancement of Artificial Intelligenc*, volume 33, 2019.

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 rraining of neural networks. *arXiv preprint arxiv: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*, 2023.

Javed, K., Sharifnassab, A., and Sutton, R. S. Swifttd: A fast and robust algorithm for temporal difference learning. In *Reinforcement Learning Conference*, 2024.

Kearney, A., Veeriah, V., Travnik, J. B., Sutton, R. S., and Pilarski, P. M. TIDBD: adapting temporal-difference step-sizes through stochastic meta-descent. In *Advances on Neural Information Processing Systems*, 2018.

Kearney, A., Veeriah, V., Travnik, J. B., Pilarski, P. M., and Sutton, R. S. Learning feature relevance through step size adaptation in temporal-difference learning. *arXiv preprint arxiv:1903.03252*, 2019.

Khetarpal, K., Ahmed, Z., Cianflone, A., Islam, R., and Pineau, J. Re-evaluate: Reproducibility in evaluating reinforcement learning algorithms. *International Conference on Machine Learning Reproducibility in ML Workshop*, 2018.

Khetarpal, K., Riemer, M., Rish, I., and Precup, D. Towards continual reinforcement learning: A review and perspectives. *Journal of Artificial Intelligence Research*, 75, 2022.

Kiran, M. and Ozyildirim, M. Hyperparameter tuning for deep reinforcement learning applications, 2018.

Klein, A., Falkner, S., Bartels, S., Hennig, P., and Hutter, F. Fast bayesian optimization of machine learning hyperparameters on large datasets. In *International Conference on Artificial Intelligence and Statistics*, 2017.

Küttler, H., Nardelli, N., Miller, A., Raileanu, R., Selvatici, M., Grefenstette, E., and Rocktäschel, T. The nethack learning environment. *Advances in Neural Information Processing Systems*, 2020.

Lange, R. T. gymnax: A JAX-based reinforcement learning environment library, 2022.

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, 2024.

Letham, B. and Bakshy, E. Bayesian optimization for policy search via online-offline experimentation. *Journal of Machine Learning Research*, 20, 2019.

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. *Journal of Machine Learning Research*, 23, 2022.

Lizotte, D., Wang, T., Bowling, M., and Schuurmans, D. Automatic gait optimization with gaussian process regression. In *International Joint Conference on Artifical Intelligence*, 2007.

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.

Mahmood, A. R., Sutton, R. S., Degris, T., and Pilarski, P. M. Tuning-free step-size adaptation. In *International Conference on Acoustics, Speech and Signal Processing*, 2012.

Makarova, A., Shen, H., Perrone, V., Klein, A., Faddoul, J. B., Krause, A., Seeger, M., and Archambeau, C. Automatic termination for hyperparameter optimization. In *International Conference on Automated Machine Learning*, 2022.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., 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, 2015.

Nguyen, V., Schulze, S., and Osborne, M. Bayesian optimization for iterative learning. *Advances in Neural Information Processing Systems*, 33, 2020.

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

Parker-Holder, J., Nguyen, V., and Roberts, S. One-shot bayes opt withprobabilistic population based training. *arXiv preprint arxiv:2002.02518*, 2020.

Parker-Holder, J., Nguyen, V., and Roberts, S. J. Provably efficient online hyperparameter optimization with population-based bandits. *Advances in Neural Information Processing Systems*, 33, 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. *Journal of Artificial Intelligence Research*, 74, 2022.

Parker-Holder, J., Nguyen, V., Desai, S., and Roberts, S. Tuning mixed input hyperparameters on the fly for efficient population based autorl. In *Advances in Neural Information Processing Systems*, 2024.

Patterson, A., Neumann, S., Kumaraswamy, R., White, M., and White, A. The cross-environment hyperparameter setting benchmark for reinforcement learning. In *Reinforcement Learning Conference*, 2024a.

Patterson, A., Neumann, S., White, M., and White, A. Empirical design in reinforcement learning. *Journal of Machine Learning Research*, 2024b.

Paul, S., Kurin, V., and Whiteson, S. Fast efficient hyperparameter tuning for policy gradients. *Advances in Neural Information Processing Systems*, 33, 2020.

Platanios, E. A., Saparov, A., and Mitchell, T. Jelly bean world: A testbed for never-ending learning. In *International Conference on Learning Representations*, 2019.

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

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. Proximal policy optimization algorithms. *arXiv preprint arxiv:1707.06347*, 2017.

Snoek, J., Larochelle, H., and Adams, R. P. Practical bayesian optimization

of machine learning algorithms. *Advances in Neural Information Processing Systems*, 25, 2012.

Springenberg, J. T., Klein, A., Falkner, S., and Hutter, F. Bayesian optimization with robust bayesian neural networks. *Advances in Neural Information Processing Systems*, 29, 2016.

Sutton, R. S. Adapting bias by gradient descent: An incremental version of delta-bar-delta. In *The Association for the Advancement of Artificial Intelligence*, volume 92, 1992.

Sutton, R. S. and Barto, A. G. *Reinforcement learning: An introduction*. MIT press, 2018.

Tanner, B. and White, A. Rl-glue: Language-independent software for reinforcement-learning experiments. *Journal of Machine Learning Research*, 10, 2009.

Todorov, E., Erez, T., and Tassa, Y. Mujoco: A physics engine for model-based control. In *International Conference on Intelligent Robots and Systems*, 2012.

Van Hasselt, H., Guez, A., and Silver, D. Deep reinforcement learning with double q-learning. In *Association for the Advancement of Artificial Intelligenc*, volume 30, 2016.

Wan, X., Lu, C., Parker-Holder, J., Ball, P. J., Nguyen, V., Ru, B., and Osborne, M. Bayesian generational population-based training. In *International Conference on Automated Machine Learning*, 2022.

Wang, H., Sakhadeo, A., White, A. M., Bell, J. M., Liu, V., Zhao, X., Liu, P., Kozuno, T., Fyshe, A., and White, M. No more pesky hyperparameters: Offline

hyperparameter tuning for RL. *Transactions on Machine Learning Research*, 2022.

Watanabe, S. Tree-structured parzen estimator: Understanding its algorithm components and their roles for better empirical performance. *arXiv preprint arXiv:2304.11127*, 2023.

White, M. and White, A. A greedy approach to sdapting the trace parameter for temporal difference learning. In *International Conference on Autonomous Agents & Multiagent Systems*, 2016.

Wołczyk, M., Zajac, M., Pascanu, R., Kuciński, Ł., and Miłoś, P. Continual world: A robotic benchmark for continual reinforcement learning. *Advances in Neural Information Processing Systems*, 2021.

Xu, Z., van Hasselt, H., and Silver, D. Meta-gradient reinforcement learning. In *Advances on Neural Information Processing Systems*, 2018a.

Xu, Z., van Hasselt, H. P., and Silver, D. Meta-gradient reinforcement learning. *Advances in Neural Information Processing Systems*, 31, 2018b.

Zahavy, T., Xu, Z., Veeriah, V., Hessel, M., Oh, J., van Hasselt, H. P., Silver, D., and Singh, S. A self-tuning actor-critic algorithm. *Advances in Neural Information Processing Systems*, 33, 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 *International Conference on Artificial Intelligence and Statistics*, 2021.

Zhou, H., Lin, Z., Li, J., Fu, Q., Yang, W., and Ye, D. Revisiting discrete soft actor-critic. *arXiv preprint arXiv:2209.10081*, 2022.

Appendix

A Design Choices and Hyperparameter Values Used in the Experiments

All the hyperparameter ranges and values for all algorithms in all environments can be seen in table A.1. We consider 3 different RL algorithms - SAC, PPO, and DDQN. SAC is for continuous action spaces, but we extended it to work with discrete action spaces (Zhou et al., 2022). All the hyperparameter ranges listed below are chosen according to the standard values used in hyperparameter sweeping, like not too big of a learning rate or small discount factor γ. We applied log-uniform scaling to the learning rate and γ 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. For example, we let the buffer size or the network architecture stay the same as this hyperparameter depends on the budget the experiment designer can give. 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 A.1: Hyperparameter values and ranges for SAC, PPO, and DDQN.

Parameter	Value	Ranges
Shared		
optimizer	Adam	
nonlinearity	ReLU	
learning rate	$1e-2/\ 3\cdot10^{-4}$	$\log([1e-6, 0.1])$
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	$[1e-4, 0.1]$
target update interval	1	
update frequency	1	
replay buffer size	10^6 / 10^3	
start updates	500 / 10^3	
normalize observations	False	
normalize rewards	False	

A.1 Details on the Environments

For all Mujoco and Classic control experiments, we used jit-compiled versions of the environments, namely the Brax (Freeman et al., 2021a) and Gymnax (Lange, 2022) packages, which made our experiments quite fast - it took about 2.5 hours to get 5M steps in Mujoco using brax.

B Tuning is Hard in the Cartpole Environment

In this section, we present the results for the same experiment as in the section for the DDQN algorithm in the Cartpole environment. The Cartpole environment is another classic reinforcement learning problem where an agent must balance a pole on a moving cart. It is an episodic environment, with each episode having a maximum length of 200. If we do not limit the episodes to some steps, this environment can be used as a continuing environment. In the Cartpole environment, the agent receives a reward of +1 for each step the pole remains upright, with the episode ending if the pole falls beyond a certain angle or the cart moves off-screen. The maximum possible reward is 200 per episode, corresponding to the 200 environment step limit.

As in the fig. 3.1, we show the distribution plots of the highest average return achieved by the DDQN agent over 5 different seeds. The x-axis shows the average return, while the y-axis is the number of trials. The dotted line depicts the average performance, which in this case is 180. As we can see from the plot, we get consistent results for the Mountain car environment in the chapter 3. The more hyperparameters we have, the more hyperparameter configurations we need to try to achieve good overall performance over all the seeds.

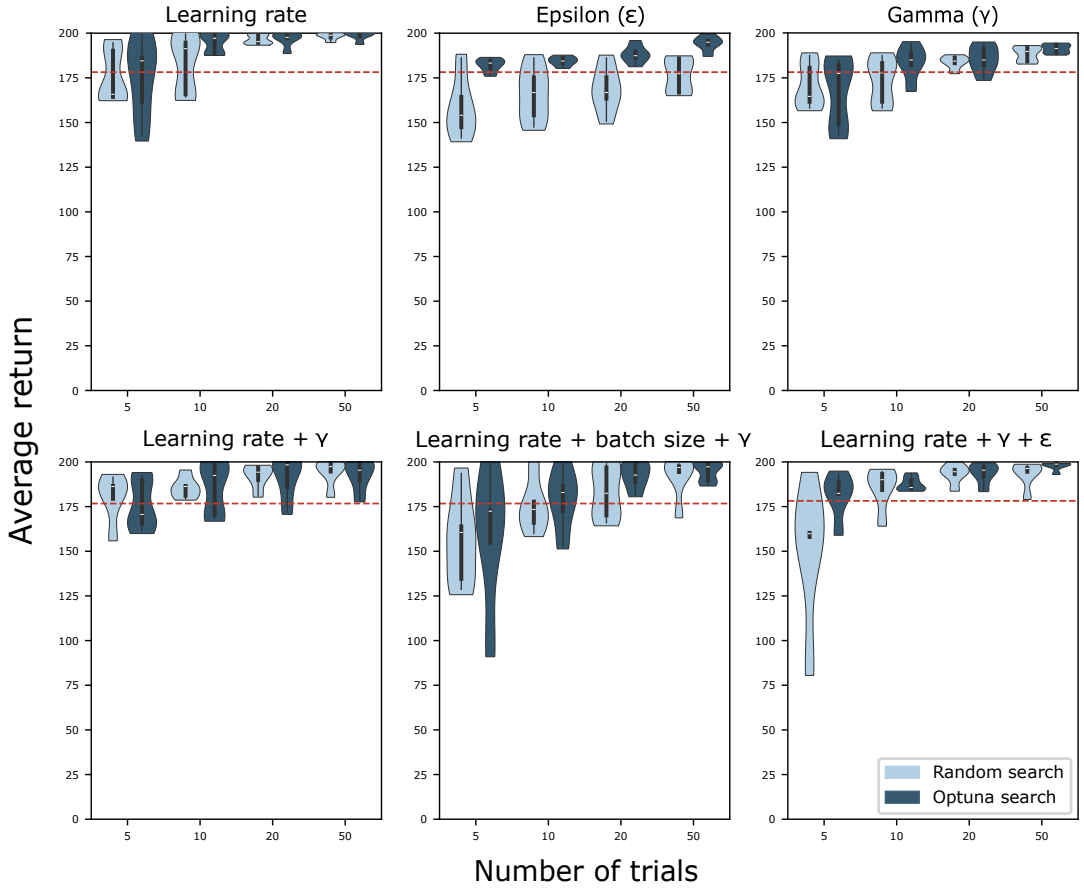

Figure B.1: Distribution plots of the highest average return achieved by the DDQN agent in the Cartpole environment over 5 seeds. The y-axis shows the average return, while the x-axis shows the number of trials. The dotted line depicts the average good performance, which is 180.

C Results with Resetting

The results for the SAC and PPO algorithms for both 1M and 2M stopping conditions tuning one and many hyperparameters. The blue curve depicts PPO's performance over 20 seeds, while the red curve is SAC.

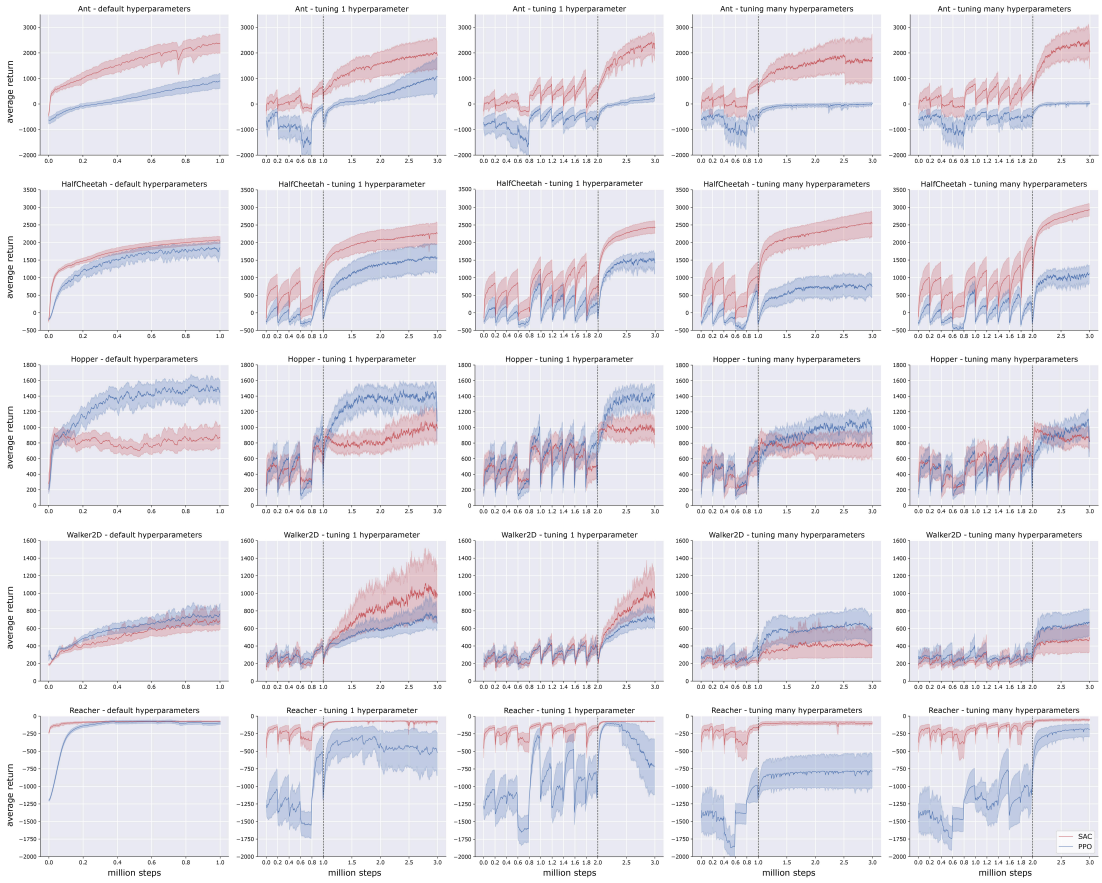

Figure C.1: SAC (red) and PPO (blue) algorithms in multiple Mujoco environments. In the first column, we have the performance of the algorithms with default hyperparameters. The second and third columns show the algorithms' performances within the 3 million budget, where we stop hyperparameter optimization 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. The dotted line depicts the timestep agent started deploying the best hyperparameters it has seen. The shaded area is a 95% bootstrapped confidence interval over 20 different runs.

D Results without Resetting

In this section, we present the plots for SAC and PPO algorithms in the non-resetting paradigm we introduced in the chapter 5. In fig. D.1, we show the

results of the SAC agent tuning its hyperparameters 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.

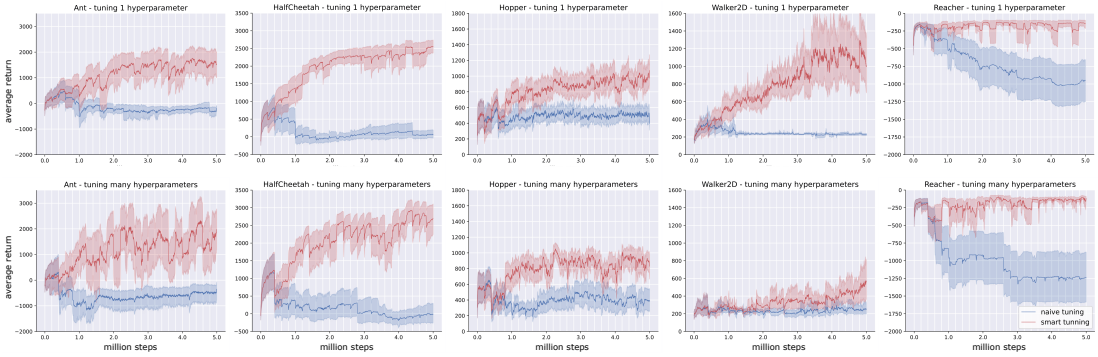

Figure D.1: The results for the SAC algorithm in Mujoco environments using the non-resetting paradigm. The red line shows the performance of agents for a 5M online interaction budget where we share the weight on some conditions, while the blue line is the performance of the agents that share their knowledge naively from configuration to configuration in the same budget. The plots in the first row correspond to the setting where we only tune the learning rate, and in the second row when we tune all 5 hyperparameters.

Even though conditional sharing helps SAC to perform better over time, the same results are not visible in PPO. Interestingly, in PPO, it seems like both sharing methods behave similarly. This can result from having a clipped surrogate objective that doesn't let the parameters drift too far away from each other, even when we change the hyperparameters. This means that we can achieve better performance if we design algorithms that take into account the change from one hyperparameter to another, or PPO may have a harder time tuning its many hyperparameters. The same can be seen in PPO results where the agents had 10M steps of total budget as depicted in fig. D.3.

And the plots of the individual runs of the PPO agents for seeds 1 and 7. We only plot the performances for the Reacher environment, as PPO seems to have some benefits from sharing the accumulated knowledge on a condition only here.

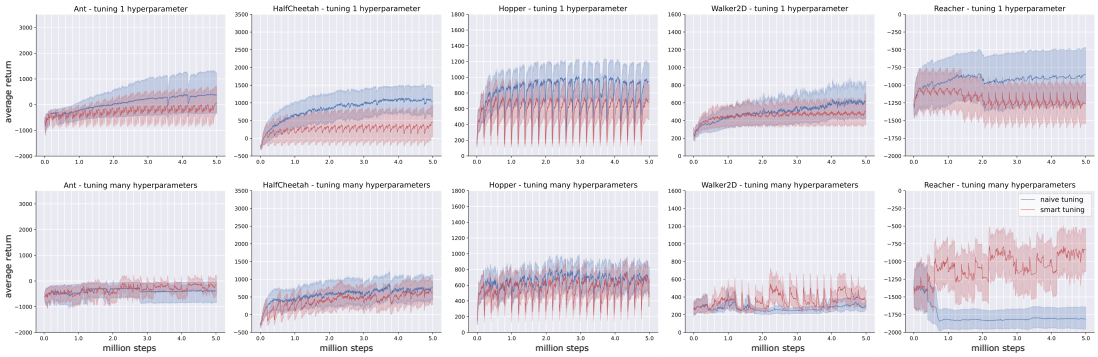

Figure D.2: PPO in a variety of Mujoco environments. The red line shows the performance of smart sharing agents for a 5M budget, while the blue line is the performance of the naive sharing agents for the same budget. The plots in the first row correspond to tuning only the learning rate, and in the second row when we tune all 7 hyperparameters.

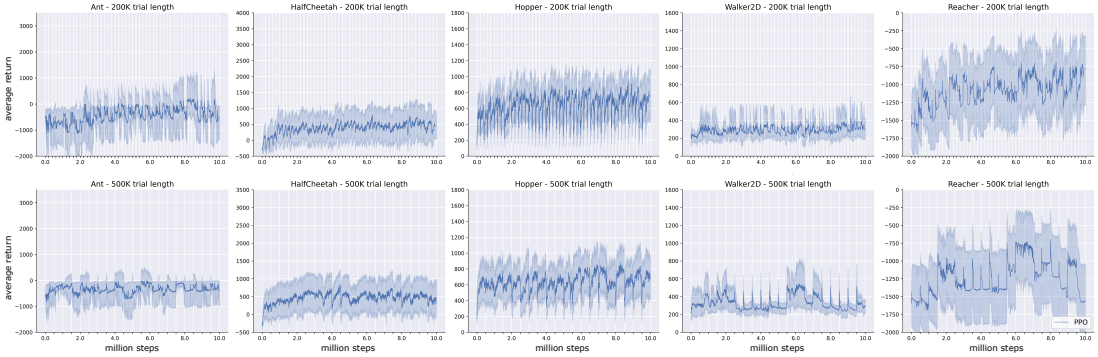

Figure D.3: PPO agent in multiple Mujoco environments with 200K and 500K trial lengths with a total budget of 10M steps. The first row shows the performance of the agents in all five Mujoco environments with the 200K trail length. The second row shows the performance with the 500K trail length. The shaded area is a 95% bootstrapped confidence interval of 10 independent runs.

This result may indicate that, for PPO, we may need longer trial lengths as it may get to good performance levels later than SAC due to its trajectory-based learning rules.

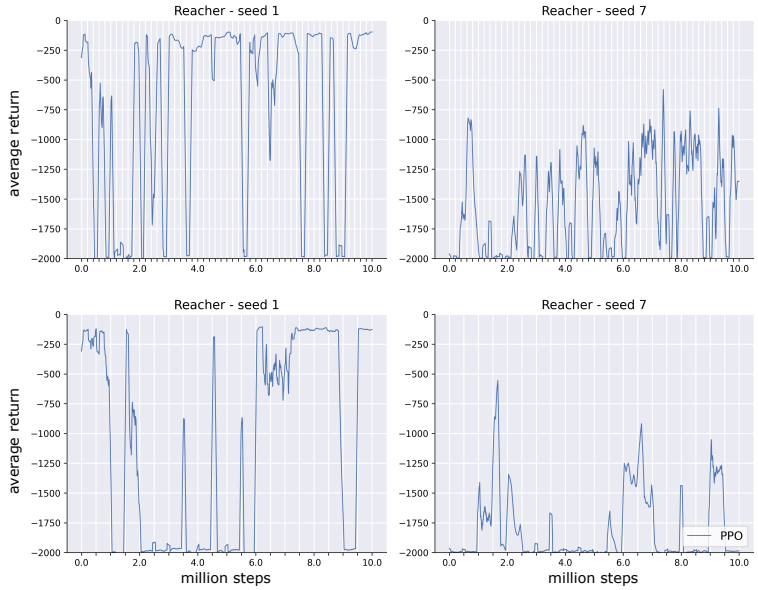

Figure D.4: Depiction of individual runs of 200K trial length runs (first row) and 500K trial length (second row) for 10M steps. The first row has the results for all Mujoco environments tried for seed 1, and the second row has the results for seed 7.

D.1 Total AUC Results for PPO

The total AUC for PPO in Mujoco environments. In all plots, we evaluate for 3M steps tuning 7 hyperparameters. 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.

As we can see from the plots, the smart sharing approach is not as effective in PPO as it is in SAC. Especially in the Hopper and Walker environments, the smart sharing approach does not perform as well as the resetting agents. This again shows that the algorithm's design is crucial in the online tuning setting.

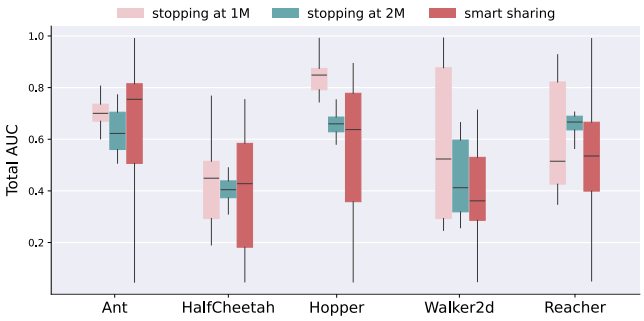

Figure D.5: Box plots of the total AUC of the PPO algorithm for the 3M evaluation budget tuning 7 hyperparameters in three different settings for all the Mujoco environments.

D.2 Final Performance Results for SAC and PPO

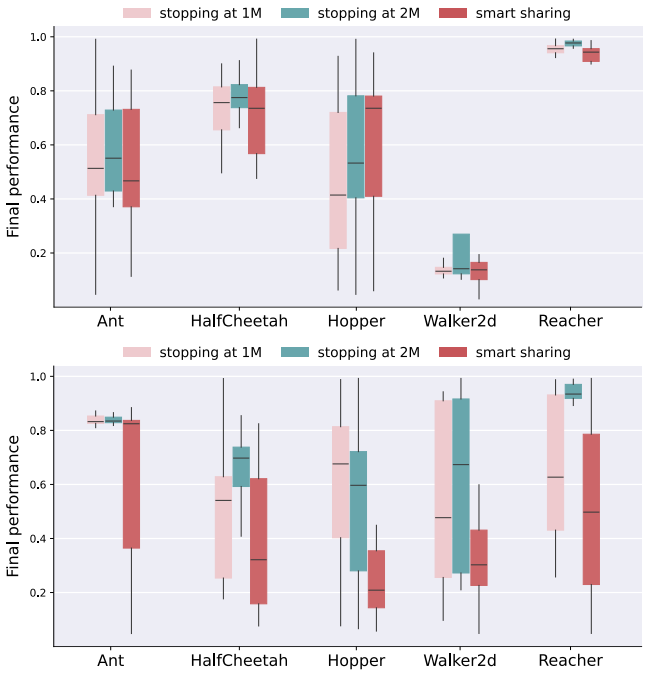

Figure D.6: Box plots of the final performances of the SAC(top) and PPO (bottom) algorithms for the 3M evaluation budget tuning 7 hyperparameters in three different settings for all the Mujoco environments testbeds.