Tuning for the Unknown: Revisiting Evaluation Strategies for Lifelong RL

Golnaz Mesbahi [*1] **Olya Mastikhina** [*1] **Parham Mohammad Panahi** [1] **Martha White** [1] **Adam White** [1]

Abstract

In continual or lifelong reinforcement learning access to the environment should be limited. If we aspire to design algorithms that can run for long-periods of time, continually adapting to new, unexpected situations then we must be willing to deploy our agents without tuning their hyperparameters over the agent's entire lifetime. The standard practice in deep RL—and even continual RL—is to assume unfettered access to deployment environment for the full lifetime of the agent. This paper explores the notion that progress in lifelong RL research has been held back by inappropriate empirical methodologies. In this paper we propose a new approach for tuning and evaluating lifelong RL agents where only one percent of the experiment data can be used for hyperparameter tuning. We then conduct an empirical study of DQN and Soft Actor Critic across a variety of continuing and non-stationary domains. We find both methods generally perform poorly when restricted to one-percent tuning, whereas several algorithmic mitigations designed to maintain network plasticity perform surprising well. In addition, we find that properties designed to measure the network's ability to learn continually indeed correlate with performance under one-percent tuning.

1. Introduction

Continual or lifelong reinforcement learning (RL) arises in many applications.[1] In HVAC control, agents learn to adapt the set-points daily, with deployment lasting for weeks, or months, but the agent does not exploit knowledge of length of the deployment (Luo et al., 2022). Similar situations arise in data-center cooling (Lazic et al., 2018), water treatment (Janjua et al., 2023), and many other industrial control settings. Even our popular deep RL benchmarks could naturally be treated as lifelong learning tasks: Atari agents could play games forever, switching to a new game when they die or completed each game (similar to the Switching ALE benchmark (Abbas et al., 2023)). Mujoco tasks are naturally continuing, but common practice is to truncate experiments after a fixed number of interactions, reseting to some initial configuration. In lifelong learning tasks, we should design, tune, and test our agents with limited access to the environment and then deploy the learning system as-is without further tuning of its hyperparameters during the rest of its lifetime.

The vast majority of algorithmic progress in deep RL has focused on the non-continual setting. Agent designers test algorithmic variations and hyperparameter combinations in the deployment environment for the the full lifetime of the agent and then report the best performance across these deployments. For example, if one were to develop a new exploration algorithm for Atari, then this new algorithm would be extensively tested over 200 million frames, tuning any new hyperparameters introduced by evaluating each over 200 million frames. In this sense the standard methodology is to design and tune our agents given access to the full lifetime of the agent.

There has been increased focus on extending or modifying existing deep RL agents for lifelong RL, with limited success. With the goal of enabling extended learning, these approaches can be roughly categorized into three groups: 1) resetting, 2) regularization, and 3) normalization. In the first, parts of the agent's network are reset to random initial values, causing large drops in performance, but eventually leading to improved final performance (Nikishin et al., 2022; 2023; D'Oro et al., 2022). Regularization balances error reduction with keeping the agent's network parameters close to initialization (Kumar et al., 2023); this helps because the random initial parameters help the network learn quickly. Finally, recent work has found that layer normalization can help maintain the ability to learn (Lyle et al., 2023). All these approaches are mitigations: algorithmic fixes applied to a base agent that is not designed for life-

[*]Equal contribution [1]Department of Computing Science, University of Alberta, Alberta, Canada. Correspondence to: Golnaz Mesbahi <mesbahi@ualberta.ca>, Olya Mastikhina <mastikhi@ualberta.ca>.

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

[1]In this paper we use the term lifelong learning because (1) this avoids the confusing terminology clash with continuing MDPs and (2) the name reflects the fact that agent-environment interaction will eventually end we just don't know precisely when.

long learning. In addition, in all these works the ultimate empirical demonstrations were conducted in non-continual testbeds like Atari and Mujoco, where the proposed new lifelong learning agents were tuned for the agent's entire lifetime—there is no sense in which it is continual. Many of these approaches are promoted to address *loss of plasticity*, which although important for the success of lifelong RL agents, also arises in standard episodic non-continual benchmarks like Mujoco and Atari (Nikishin et al., 2023; D'Oro et al., 2022).

There are several algorithms designed from first principles for lifelong RL. Continual backprop (Dohare et al., 2021), for example, was designed for and tested in never-ending regression and RL control tasks. This algorithm randomly re-initializes neurons in the network to promote continual adaption in the face of non-stationarity. Similarly inspired, Permanent-transient networks (Anand & Precup, 2023) use a pair of neural-networks to ensure a deep Q-learning agent is able to distill key information from a sequence of tasks, while adapting to new ones.

This paper explores the notion that progress in lifelong RL research has been held back by inappropriate empirical methodologies. Inspired by the constraints of real-world applications of RL, we propose a new methodology for tuning and evaluating lifelong RL agents. The framework is based on a simple idea: lifelong RL agents may be deployed for an unknown amount of time and thus agent designers should not be allowed to tune their agents for their entire lifetime. Instead, we introduce a tuning phase: one percent of the total lifetime. Only one percent of the experiment data can be used for hyperparameter tuning, after that the hyperparameters must be fixed and deployed for the remainder of the agent's lifetime. This setup closely matches real-world deployment scenarios where (a) we cannot tune for the agent's full lifetime, and (b) we may have limited knowledge and experience with the dynamics and state distribution of the deployment environment.

In our first set of experiments, we verify that popular and performant deep RL agents like DQN and Soft Actor-critic perform poorly across a suite continual RL task irrespective of what metric is used to select the best hyperparameters during the one-percent tuning phase.

We then use our new lifelong RL evaluation protocol to revisit recent findings from the lifelong RL literature. In particular, we investigate several mitigation strategies, including regularizing to the initial weights, Concatinated ReLu, and weight normalization, under one-percent tuning finding most actually improve performance compared to the base algorithms. However, in all cases as the deployment lifetime is extended, performance eventually drops. We also revisit many metrics proposed in the literature as potentially predictive of catastrophic performance collapse in lifelong RL, such as stable rank (Kumar et al., 2020), dormant (Sokar et al., 2023) or dead neurons (Dohare et al., 2021; Abbas et al., 2023; Lyle et al., 2022), and weight norms (Nikishin et al., 2022). Under one-percent tuning we see that some of these metrics actually correlate well with performance, disagreeing with recent results that found no correlations (Lyle et al., 2023). Finally, we find that Permanent-transient networks (PT-DQN) are the robust and compatible with one-percent tuning.

2. Background and Problem Formulation

In this paper we consider lifelong problems formulated as Markov Decision Processes (MDPs). On each discrete time step, $t = 1, 2, 3, ...$ the agent selects an action A_t from a finite set of actions $\mathcal{A}$ based, in part, on the current state of the environment $S_t \in \mathcal{S}$. In response the environment transition to a new state $S_{t+1} \in \mathcal{S}$ and emits a scalar reward $R_{t+1} \in \mathbb{R}$. The agent's action selection are determined by its policy $A_t \sim \pi(\cdot|S_t)$. Episodic problems are ones where the agent-environment interaction naturally breaks up into sub-sequences where the agent reaches a terminal and then is teleported to a start state $S_0 \sim \mu(\mathcal{S})$. A continuing problem is one where the agent-environment interaction never end.

The agent's task is to find a policy π that maximizes the expected discounted sum of rewards: $\mathbb{E}_\pi[G_t|S_t = s, A_t = a]$ where $G_t \doteq R_{t+1} + \gamma_{t+1}G_{t+1}$. We use transition-based discounting to unify episodic and continuing problems where $\gamma_{t+1} = \gamma(S_t, A_t, S_{t+1}) \in [0, 1]$—see White (2017) for further details. A lifelong RL problem is one where the agent-environment interaction, either one long episode as in a continuing task or many episodes as in an episodic task, is eventually truncated at time T but neither the agent nor the agent designer can exploit this information because it is unknown. This appears similar to how the Atari benchmark is used: at the beginning of a trial the agent is initialized and interacts with the environment for a fixed number of steps T (200 million frames) and T is unrelated to the agent's performance in the game and the agent does not make use of T (i.e., the underlying learning algorithm is not designed for finite-horizon MDPs). The key difference, as outlined in the next section, is that in lifelong RL the agent designer does not exploit knowledge of T in the design or evaluation of the agent.

In most interesting tasks the underlying state cannot be directly observed by the agent, instead only an observation, $\mathbf{x}_t$ of S_t is available to the agent. In the case of discrete action, the policy is constructed using a neural network, outputting estimates of the value of each action: $\hat{q}_\theta(S_t, A_t) \approx \mathbb{E}_\pi[G_t|S_t = s, A_t = a]$, where θ are the learnable parameters of a neural network. In this work

we use the DQN algorithm (Mnih et al., 2015) to learn $\hat{q}_\theta$ and select actions. In the case of continuous actions, we learn a parameterized policy $\pi_{\mathbf{w}}$ where $\mathbf{w}$ are the parameters of a network which are adapted to maximized the expected discounted return from the start state, using an update derived from the policy gradient theorem (Sutton et al., 1999). One such algorithm is Soft Actor-critic (SAC) which is widely used in complex, continuous action environments—see Haarnoja et al. (2018) for details.

3. One-percent Tuning

The common agent development-evaluation loop in RL is artificial and not particularly reflective of biological systems nor applications. In RL research, we conduct experiments on computer simulations or robots, running for a predetermined number of steps. Naturally, as agent designers we want our agents to perform well and we want to report the performance of an agent that is well engineered for the task at hand. The typical process is to fix the total budget of experience or lifetime of the agent and then begin design and tuning iterations: tweak the algorithm and the hyperparameter settings (e.g., step-size, exploration rate, replay parameters, etc.) and run the agent for lifetime and record the performance. The process is iterated until their performance plateaus or the designer is happy with the outcome.

Hyperparameters have a dramatic impact on both the performance and learning dynamics of deep RL agents. DQN is one of the simplest such agents and it contains over 14 hyperparameters controlling size of the replay buffer, target network updated rate, averaging constants in the Adam optimizer and exploration over time, to name a few. These hyperparameters allows us to instantiate variants of DQN that learn incredibly slowly to mitigate noise and off-policy instability, to fast online learners that can track stationary targets. The proliferation of hyperparameters in modern Deep RL agents effectively allow the agent designer to select which algorithm they want to use ahead of time for a given task. This is even more important in lifelong RL, as recent work has shown that the default hyperparameter settings of popular agents must be significantly adjusted to deal with long-running non-stationary learning tasks (Lyle et al., 2023).

The design iteration described above seems at odds with the goals of lifelong learning. In lifelong RL, we aspire to build agents that will run for long-periods of time, continually adapting to unpredictable changes in the environment and continually revealing new regions of the state space. Using hyperparameters to effectively select the algorithm that works best over the entire lifetime of the agent is only possible in simulators. If your MDP is basically stationary you can set the hyperparameters to exploit this knowledge. Imagine deploying our agents to control a water treatment plant or to interact with customers on the internet. It is totally unclear how these imagined deployment settings even match the standard agent development-evaluation loop described above. In these examples, it is much more natural to imagine that the designer has access to the deployment scenario for limited amount of time. During this time she can try out different hyperparameters and agent designs, but eventually deployment time beckons. This empirical setup would not only be a better match for many applications, but also motivate the development of algorithms with fewer critically sensitive hyperparamters. In other words, agents capable of adapting their learning online, forever plastic, adapting to the nature of task non-stationarities—a lifelong learning agent.

Our proposed one-percent tuning methodology mechanizes these goals. The name describes the relatively simple idea: we propose to tune the agent only for 1% of its lifetime. Though the agent cannot know it's lifetime, as experimenters, we know how long we will run our experiment and can constrain ourselves to tune only over a small window. If we know the agent will run for n steps, then we tune the agent for $\lfloor 0.01n \rfloor$ steps. In other words, for every hyperparameter setting, we run the agent for $\lfloor 0.01n \rfloor$ steps to obtain the performance metric after this short learning time. We then chose the best hyperparameter configuration, for example according to the best performance in the final 10% of the tuning phase. The agent is then deployed with these hyperparameters for the full n steps, for multiple runs, to get the performance of that lifelong learning agent.

4. Failure of standard algorithms under one-percent tuning

In this section, we evaluate our proposed methodology for tuning lifelong RL agents. We contrast the one-percent-tuned agent with an agent with either default hyperparameters from the literature, or hyperparameters chosen based on tuning for the whole lifetime in environments for which there are obvjous default hyperparameters. We perform the experiments with DQN in two discrete action environments and SAC in five different continuous control environments.

Failures of DQN under one-percent tuning: We consider a large set of hyperparameters for DQN, each over a wide range, including exploration (epsilon), learning rate, batch size, buffer size, minimum number of steps before the first update, and the values of β_2 and ϵ in the Adam optimizer. The ranges and chosen hyperparameters are outlined in Appendix A.1. We test three different criteria to choose the best hyperparameter configuration, primarily to see if any allow for DQN to perform well under one-percent-tuning. These metrics include area under the learning curve (AUC) which corresponds to overall performance in this one-percent tun-

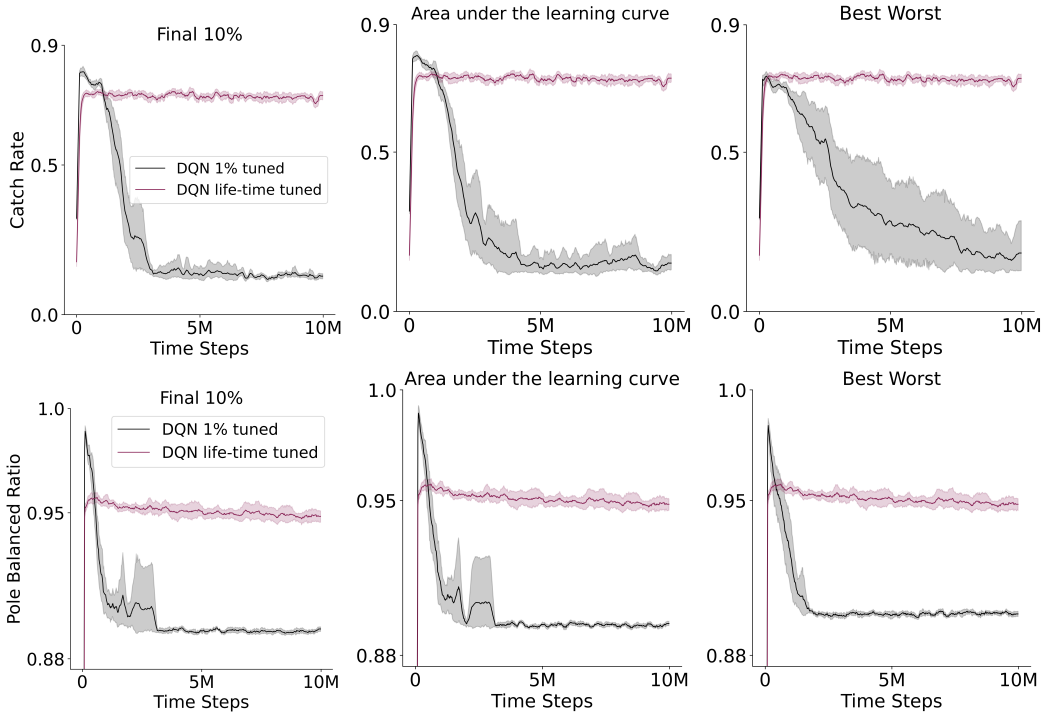

Figure 1. Tuning on one-percent of a lifetime leads to poor performance for DQN in Non-stationary Catch and Continuing Cart-pole. Each row of plots corresponds to a different environment, and each column corresponds to a different hyperparameter selection strategy. Lines are averaged over ten seeds and the shaded regions are 95% bootstrap confidence interval.

ing phase, best performance in the final 10% of the tuning phase, and finally the best worst-case performance across seeds, to select hyperparameters that are robust across seeds, which we call best-worst. The agents are run for 10 million steps, with $100,000$ steps for the one-percent-tuning.

We test DQN in two environments: Non-stationary Catch and Continuing Cart-pole. Non-stationary Catch (Google-Deepmind, 2022) is a visual control domain from the DeepMind C-suite library of continuing environments. The agent controls a paddle on the bottom of a 10 by 5 board, and the goal is to collect as many falling objects as the agent can, with new objects spawned with probability 0.1—making this a continuing MDP. There are three actions, $\{$left, right, stay-still$\}$. If the paddle successfully catches a ball, a reward of $+1$ is received. If it fails to catch a ball, a reward of -1 is received. Otherwise, a reward of 0 is given. The non-stationarity is induced by randomly swapping two entries in the observation every $10,000$ steps. The agent goes through 10 non-stationary transitions during tuning for the 100,000 steps. The performance measure is catch rate which is defined as the moving average of the ratio of the balls caught. An optimal agent (without exploration) would achieve a catch rate of 1 while a random agent would get 0.2.

Continuing Cart-pole (Barto et al., 1983) is a simple classic control task, with completely stationary dynamics. The agent's observations are the position and velocity of the cart and its pole. At each step the agent takes one of two actions: push the cart toward left or right with goal of keeping the pole balanced on top of the cart. The reward is $+1$ for every step that the pole is balanced. Once the pole falls more than 24 degrees from its upright position, the agent receives a reward of 0 and the pole is teleport to the position, but the agent is not reset. The agent's performance is measured as an exponential moving average (0.99 averaging constant) of the ratio of recent time steps that the pole was successfully balanced. Under this performance measure a perfect agent that keeps the pole balanced indefinitely would attain a score of 1. This environment provides a non-stable equilibrium requiring constant learning and adjustment.

The results are shown in Figure 1, and are as expected. The lifetime-tuned DQN is stable across its lifetime. The one-percent-tuned DQN, on the other hand, performs equally well or better in the beginning of learning, but then its performance collapses soon after the first $100,000$ steps. None of the three criteria prevent this collapse, and result in relatively similar performance. Best-Worst is less effective than Final 10% and AUC in Non-stationary Catch, and all three are similar in Continuing Cart-pole.

Scaling up: continuing continuous control: We ran a similar experiment with SAC in several environments from the DeepMind Control Suite (Tassa et al., 2018). The DeepMind Control Suite environments are large-scale continuous

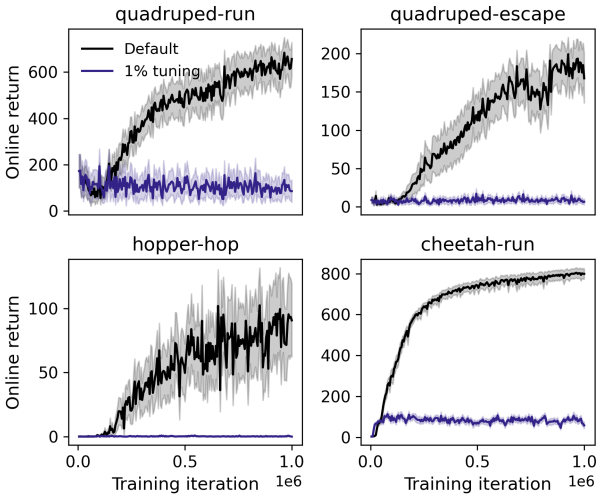

Figure 2. One-percent tuning performance of SAC in four DeepMind Control Suite environments compared with SAC using default hyperparameters. The results are averaged over ten runs and plotted with standard error shading.

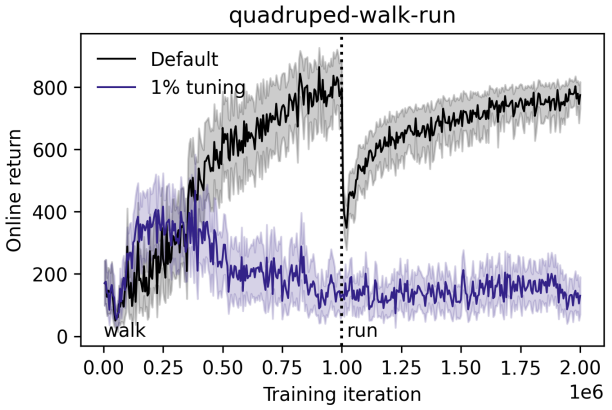

Figure 3. Tuning on one-percent of a run similarly leads to poor performance for SAC in a task-switching setting. The results are averaged over ten runs with standard error.

control environments commonly used in deep RL research. The environments are physical simulations, making them useful for investigating tuning on semi real-world settings. The quadruped-escape environment is also partially non-stationary; the agent encounters a different bumpy terrain that it must escape in each episode.

We again consider a large set of hyperparameters for SAC, including the learning rate, batch size, buffer size and the values of β_2 and ϵ in the Adam optimizer. The ranges and chosen hyperparameters are outlined in Appendix A.2. We compare the one-percent-tuned values with the default hyperparameters previously reported for the DeepMind Control Suite (Haarnoja et al., 2018). For this setting, the agents are run for 1 million steps, with $10,000$ exploration steps followed by training over $10,000$ steps.

In Figure 2, we see similar results to our previous experiment, although with even worse performance for the agents using one-percent-tuning. The same values were picked for AUC as for final 10% performance for one-percent-tuning, so we report only one set of value in these plots. For cheetah-run, the one-percent-tuning agent does outperform SAC with default hyperparameters in early learning but then quickly plateaus. In the other environments, the one-percent-tuning agents exhibit almost no learning, with a nearly flat return across the steps.

We also investigated how SAC performs with one-percent-tuning in a lifelong learning setting where the environment switches from quadruped-walk to quadruped-run halfway through the experiment. The agent is tuned for one-percent of the experiment in quadruped-walk. In Figure 3, we see a more noticeable improvement over SAC with default hyperparameters in early learning for quadruped-walk, but again we see a performance drop and then almost no learning in quadruped-run.

5. Mitigations help under one-percent tuning

In this section we investigate if mitigation strategies designed for lifelong learning improve performance under our one-percent tuning methodology. We revisit the same environments and base algorithms as in the last section, but now include new algorithms using several mitigation strategies layered on-top of the base learner.

We consider the following mitigations, where most are used for both DQN and SAC and otherwise is used only for one. They do not perfectly share the same mitigations, because for example the PT-DQN algorithm (Anand & Precup, 2023) is designed only for action-values methods, so we included an additional different mitigation for SAC.

W0Regularization (Kumar et al., 2023): The ℓ_2 loss between the weights and the initial weights is added to the loss function, to encourage the weights to stay near the initialization.

L2Regularization (Dohare et al., 2023; van Laarhoven, 2017): In this method, a term proportional to the ℓ_2 norm of the weights of the network is added to the loss function. This will result in keeping the weight magnitude smaller in the network.

CReLU (Abbas et al., 2023): The concatenated ReLU activation function limits the number of inactive units by concatenating the output of $\text{ReLU}(x)$ with $\text{ReLU}(-x)$. This mitigation should reduce the percentage of dead neurons since CReLU maintains 50% of the neurons in an active state.

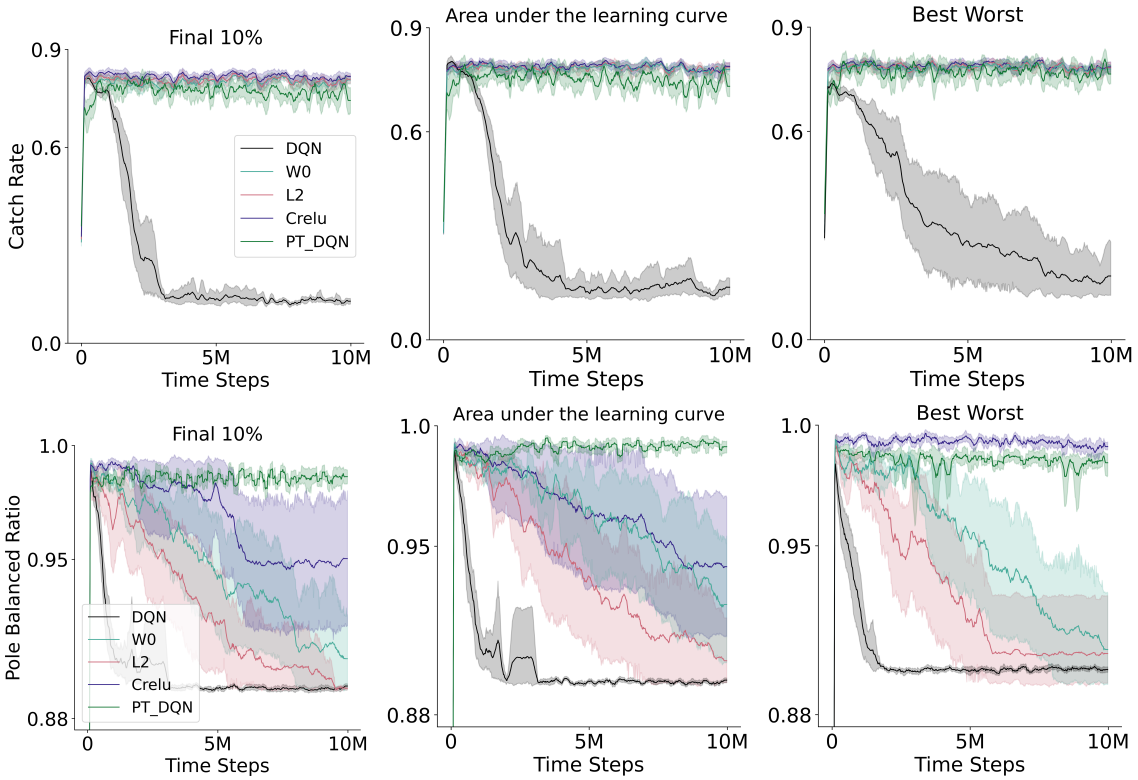

Figure 4. The effect of incorporating mitigations into DQN under one-percent tuning in Non-stationary Catch and Continuing Cart-pole. Each of the plots shows a different approach for choosing the hyper-parameters during one-percent tuning. Results are averaged over ten seeds and shaded regions reflects the 95% bootstrap confidence intervals.

PT-DQN (Anand & Precup, 2023): The value function is decomposed into two separate networks: permanent, and transient. The transient is updated toward the residue error from combining both networks predictions and is reset periodically. The permanent network is only updated by distilling the transient network's predictions.

Weight normalization (Salimans & Kingma, 2016): weights matrices are split into the weight magnitudes and weight directions, with separate gradients for each.

One-percent-tuning for DQN with mitigations: Figure 4 summarizes the performance of DQN with mitigation under one-percent tuning in Non-stationary Catch and Continuing Cart-pole. All mitigations perform well in Non-stationary Catch, although W0Regularization is slightly less effective under AUC and Final 10% tuning. In Continuing Cart-pole performance is much more mixed. PT-DQN performs well under all the tuning strategies. CReLU performs well when the hyperparameters are chosen according to the best-worst performance, and otherwise performs poorly, though it does degrade less quickly than other mitigations. L2Regularization and W0Regularization help reduce the performance collapse, but steadily degrade over time.

One-percent-tuning for SAC with mitigations: Figure 5 shows the performance of SAC with different mitigations under one-percent tuning in the switching Quadruped-walk-run environment. Most mitigation strategies improve performance over SAC with one-percent tuning, except for W0regularization which further decreases performance. CReLU improves performance the most on its own, and combining CReLU with weight normalization has the strongest effect. Interestingly, weight normalization on its own is the least effective when moving from walk to run.

In contrast, none of the mitigation strategies help with one-percent-tuning for quadruped-escape in Figure 6. Of note, the learning rate chosen by one-percent tuning in quadruped-escape is $1 \cdot 10^{-5}$. This is below the default learning rate of $3 \cdot 10^{-4}$, and above in one-percent-tuned quadruped-walk the learning rate is higher at $1 \cdot 10^{-2}$. As normalization has been shown to allow for the use of larger learning rates (Bjorck et al., 2018; Salimans & Kingma, 2016; Ba et al., 2016), that may be why weight normalization leads to effective mitigation for Quadruped-walk-run but not Quadruped-escape. Although l2 regularization has previously been shown to increase the effective learning rate (van Laarhoven, 2017), it does not appear to be sufficient here.

In summary, the performance collapse in the one-percent tuning setting is improved significantly by using mitigation techniques. However, different tuning strategies and envi-

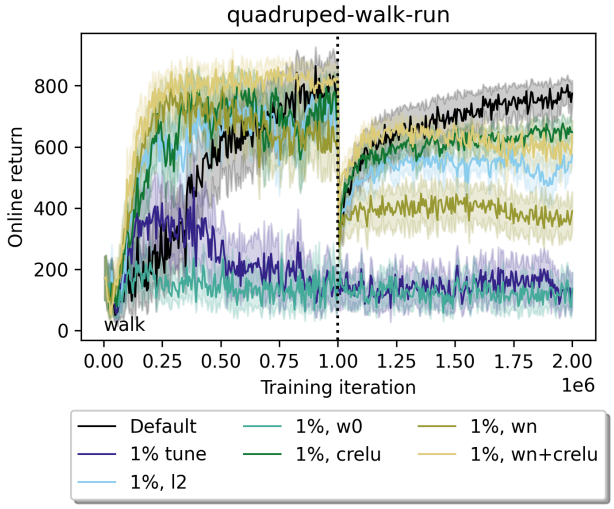

Figure 5. Multiple mitigation strategies do improve the performance of quadruped-walk-to-run with the sub-optimal hyperparameters obtained from tuning on one-percent of quadruped-walk. $l2$ is weight decay $= 1 \cdot 10^{-5}$, $w0$ is with penalization of weights moving away from their initialization values, and wn is weight normalization. There are ten seeds per run, and the shading is the standard error.

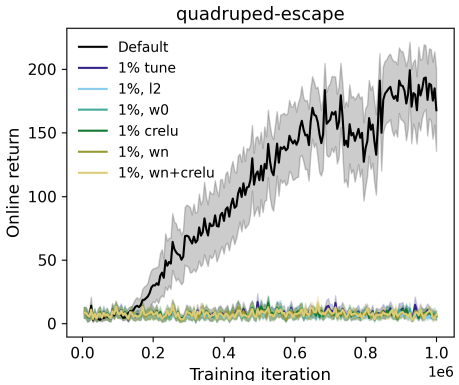

Figure 6. Multiple mitigation strategies do not improve the performance of quadruped-escape with the sub-optimal hyperparameters obtained from tuning on one-percent of a run. Results are averaged over ten runs and shaded regions depict standard error.

ronmental factors determine how beneficial they can be. In particular, mitigation methods that are more robust under one-percent tuning are more desirable. PT-DQN shows consistently strong performance under different tuning strategies in the two environments it was tested.

6. Revisiting network properties

In this section we measure properties of the one-percent tuned agents during learning, to examine if they correlate with performance. Previous works have advocated measuring different properties as a strategy for diagnosing and rectifying loss of plasticity and failures in lifelong learning (Kumar et al., 2020; Sokar et al., 2023; Dohare et al., 2021; Abbas et al., 2023; Lyle et al., 2022; Nikishin et al., 2022). Interestingly, a recent empirical study found that many of these properties where not correlated with good or bad performance (Lyle et al., 2023). In the one-percent tuning setting, however, we are more faithfully evaluating lifelong learning agents. Those that succeed under one-percent tuning are likely better lifelong learners, whereas those that fail are likely able to learn in early learning—they are in at least one sense an effective agent—but are not effective lifelong learners. Under lifetime tuning, an agent that fails is potentially simply a bad learner and its properties are largely meaningless, polluting the correlation measures. In this section, we investigate whether properties are more meaningfully correlated with performance in the one-percent tuning setting.

We investigate five properties, and measure properties for DQN in the two environments. We measure these properties in the Q-network, rather than the target network.

1. Percentage of dead neurons (Abbas et al., 2023). A hidden unit with an output of zero is a dead neuron. The percentage of dead neurons is measured online through the experiments.
2. Normalized stable rank of the weights (Kumar et al., 2020). A higher value of stable rank means that the layer's weight matrix carries more information (Hosseini et al., 2022). See Appendix B for details. The stable rank is normalized to be between 0 and 1.
3. The l0 norm of the gradient, which corresponds to the number of non-zero values in the gradient.
4. The l2 norm of the gradient, which reflects the magnitude of the gradient not just the active elements.
5. The l2 norm of the weight matrices, averaged across layers. Keeping the spectral norm of weight matrices closer to one reduces vanishing and exploding gradients, leading to more training stability, additionally allowing for better generalization (Yoshida & Miyato, 2017; Lin et al., 2021).

We examine the DQN agents with mitigations, and omit DQN under one-percent tuning which largely fails in both environments. Note that for the percentage of dead neurons, CReLU always has exactly 50% active neurons by design. We omit PT-DQN because it is not clear how to appropriately measure properties for a constantly changing fast network.

In Non-stationary Catch we can see some clear correlations in Figure 7. There is a negative correlation with the percentage of dead neurons, a negative correlation with the l0 norm of the gradient, a positive correlation with the l2 norm of the gradient and a negative correlation with the l2 norm

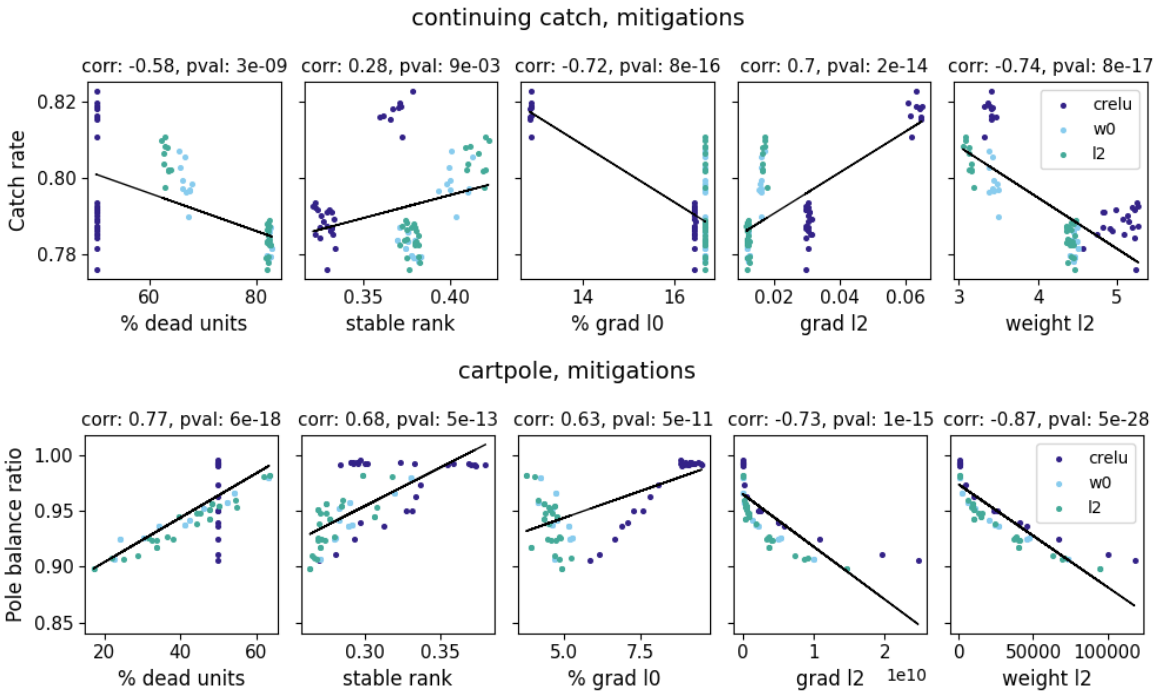

Figure 7. The correlations between properties for DQN with mitigations under one-percent tuning and final returns in Non-stationary Catch and Continuing Cart-pole. Each color represents one mitigation combination, and there are 30 dots per color corresponding to the three ways to select hyperparameters during one-percent tuning and the ten seeds used per selected hyperparameter.

of the weight matrices. There is no clear correlation with stable rank. Particularly interesting is how much variability there is amongst different variants of CReLU. Each dot corresponds to a different way to select the hyperparameters during one-percent tuning a different seed (3 selection methods times 10 seeds for a total of 30 dots). The behavior of CReLU provides some of the clearest correlations, where groupings of CReLU behave well and have a very different property measure from the other the grouping of less performant CReLU.

In Continuing Cart-pole the mitigations were less effective, and in our correlation plots only some of the CReLU groupings correspond to good performance with the remaining dots for all agents generally being relatively poor performance. The correlations are different from Non-stationary Catch in some cases due to this. For example, there is a positive correlation with percentage of dead neurons, but that is likely because even at its highest level it is still lower than the best performing agents in Non-stationary Catch. The correlation is also opposite for the l2 norm of the gradient, but that is because the smallest values in Cartpole—where performance is good—match the magnitudes of good performance in Catch. But the poor performing agents have very small magnitude l2 gradient norms in Catch, whereas the poor performing ones in Cart-pole have very big gradient norms. There is similar minimal correlation to stable rank and a negative correlation between the l2 norm of the weights and performance. This consistency in the l2 norm of the weights across environments makes sense, as we typically want the weights to stay smaller in magnitude; keeping the weights closer to 1, should promote stable (non-vanishing and non-exploding) gradients.

7. Conclusion

In this paper we introduced the one-percent tuning methodology to better evaluate lifelong reinforcement learning agents. This setting better matches realistic restrictions on lifelong learning agents and can help us appropriately assess the true lifelong learning capabilities of an algorithm. We showed that agents tuned for the first one-percent of interaction can learn faster than an agent tuned for the entire lifetime, but that these agents quickly degrade as learning progresses. Such a strict tuning setting may seem challenging, making it seem potentially obvious that these learners should fail, but we found that several simple mitigations introduced for lifelong learning were actually able to perform well in this regime. Our results highlight that one-percent-tuning can be a powerful methodology for identifying good and bad continual learning algorithms. We found that the separation between good and bad learners given by one-percent tuning also led to more meaningful correlations to properties than reported in previous work, specifically the ℓ_2 norm

commonly used to assess agents.

Impact Statement

This paper presents work whose goal is to advance the field of Machine Learning. There are many potential societal consequences of our work, none which we feel must be specifically highlighted here.

References

Abbas, Z., Zhao, R., Modayil, J., White, A., and Machado, M. C. Loss of plasticity in continual deep reinforcement learning. *arXiv preprint arXiv:2303.07507v1*, 3 2023.

Anand, N. and Precup, D. Prediction and control in continual reinforcement learning. *arXiv preprint arXiv:2312.11669*, 2023.

Ba, J. L., Kiros, J. R., and Hinton, G. E. Layer Normalization, July 2016.

Barto, A. G., Sutton, R. S., and Anderson, C. W. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13(5):834–846, 1983.

Bjorck, N., Gomes, C. P., Selman, B., and Weinberger, K. Q. Understanding Batch Normalization. In *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

Dohare, S., Sutton, R. S., and Mahmood, A. R. Continual backprop: Stochastic gradient descent with persistent randomness. *arXiv preprint arXiv:2108.06325v3*, 8 2021.

Dohare, S., Hernandez-Garcia, J. F., Rahman, P., Sutton, R. S., and Mahmood, A. R. Loss of plasticity in deep continual learning. *arXiv preprint arXiv:2306.13812v2*, 6 2023.

D'Oro, P., Schwarzer, M., Nikishin, E., Bacon, P.-L., Bellemare, M. G., and Courville, A. Sample-efficient reinforcement learning by breaking the replay ratio barrier. In *Deep Reinforcement Learning Workshop NeurIPS 2022*, 2022.

Google-Deepmind. GitHub - google-deepmind/csuite, 2022.

Haarnoja, T., Zhou, A., Hartikainen, K., Tucker, G., Ha, S., Tan, J., Kumar, V., Zhu, H., Gupta, A., Abbeel, P., et al. Soft actor-critic algorithms and applications. *arXiv preprint arXiv:1812.05905*, 2018.

Hosseini, M. S., Tuli, M., and Plataniotis, K. N. Exploiting explainable metrics for augmented sgd. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2022-June:10286–10296, 3 2022. ISSN 10636919.

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. *arXiv preprint arXiv:2312.01624v1*, 12 2023.

Kumar, A., Agarwal, R., Ghosh, D., and Levine, S. Implicit under-parameterization inhibits data-efficient deep reinforcement learning. *arXiv preprint arXiv:2010.14498v2*, 10 2020.

Kumar, S., Marklund, H., and Van Roy, B. Maintaining plasticity via regenerative regularization. *arXiv preprint arXiv:2308.11958*, 2023.

Lazic, N., Lu, T., Boutilier, C., Research, R. G., Wong, E., Roy, B., Imwalle, G., and Cloud, G. Data center cooling using model-predictive control. *Advances in Neural Information Processing Systems*, 31, 2018.

Lin, Z., Sekar, V., and Fanti, G. Why Spectral Normalization Stabilizes GANs: Analysis and Improvements. In *Advances in Neural Information Processing Systems*, volume 34, pp. 9625–9638. Curran Associates, Inc., 2021.

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

Lyle, C., Rowland, M., and Dabney, W. Understanding and preventing capacity loss in reinforcement learning. In *International Conference on Learning Representations*, 2022.

Lyle, C., Zheng, Z., Nikishin, E., Pires, B. A., Pascanu, R., and Dabney, W. Understanding plasticity in neural networks. *Proceedings of Machine Learning Research*, 202:23190–23211, 3 2023. ISSN 26403498.

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 2015 518:7540*, 518: 529–533, 2 2015. ISSN 1476-4687.

Nikishin, E., Schwarzer, M., D'Oro, P., Bacon, P.-L., and Courville, A. The primacy bias in deep reinforcement learning. *arXiv preprint arXiv:2205.07802v1*, 5 2022.

Nikishin, E., Oh, J., Ostrovski, G., Lyle, C., Pascanu, R., Dabney, W., and Barreto, A. Deep reinforcement learning with plasticity injection. *arXiv preprint arXiv:2305.15555*, 2023.

Salimans, T. and Kingma, D. P. Weight Normalization: A Simple Reparameterization to Accelerate Training of

Deep Neural Networks. In *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc., 2016.

Sokar, G., Agarwal, R., Castro, P. S., and Evci, U. The dormant neuron phenomenon in deep reinforcement learning. *Proceedings of Machine Learning Research*, 202:32145–32168, 2 2023. ISSN 26403498.

Sutton, R. S., McAllester, D., Singh, S., and Mansour, Y. Policy gradient methods for reinforcement learning with function approximation. *Advances in neural information processing systems*, 12, 1999.

Tassa, Y., Doron, Y., Muldal, A., Erez, T., Li, Y., Casas, D. d. L., Budden, D., Abdolmaleki, A., Merel, J., Lefrancq, A., Lillicrap, T., and Riedmiller, M. DeepMind Control Suite, January 2018. arXiv:1801.00690 [cs].

van Laarhoven, T. L2 Regularization versus Batch and Weight Normalization, June 2017.

White, M. Unifying task specification in reinforcement learning. In *International Conference on Machine Learning*, pp. 3742–3750. PMLR, 2017.

Yoshida, Y. and Miyato, T. Spectral Norm Regularization for Improving the Generalizability of Deep Learning, May 2017.

A. Appendix: Tuning Details

A.1. DQN tuning

For tuning the DQN agent, we sweep over the hyperparameters mentioned in table 3. The DQN agent's q-network and target network consist of a two-layer network with ReLU activations, each layer with 32 hidden units. We use orthogonal initialization, and we use 10 seeds for each hyperparameter setting for tuning. The hyperparameters chosen for one-percent tuning is shown in table 2, and the lifelong tuned agent's hyperparameters are shown in table 1. (The same process of hyperparameter selection was done for continuing cartpole.)

	Default DQN values on dancing catch
Learning rate	$1 \cdot 10^{-4}$
Batch size	256
Buffer size	$10,000$
Initial buffer fill	1000
Exploration ϵ	0.1
Adam optimizer $\beta 2$	0.999
Adam optimizer ϵ	$1 \cdot 10^{-8}$

Table 1. Default hyperparameters values for DQN on dancing catch

	DQN		
	AUC	10%	Best Worst
LR	10^{-3}	10^{-3}	10^{-3}
batch	256	256	256
buffer	$10,000$	$10,000$	$10,000$
warmup	256	1000	256
ϵ	0.01	0.01	0.1
$\beta 2$	0.999	0.999	0.9
ϵ	10^{-8}	10^{-8}	10^{-8}

Table 2. Values for DQN on dancing catch from 1% tuning, selected by AUC and by final 10% performance and best worst performance

	1%-tuning values for DQN and mitigations on dancing catch
Learning rate	$1 \cdot 10^{-1},\ 1 \cdot 10^{-2},\ 1 \cdot 10^{-3},\ 1 \cdot 10^{-4},\ 1 \cdot 10^{-5}$
Batch size	1, 4, 32, 256
Buffer size	$1000,\ 10,000,\ 100,000$
Initial buffer fill	batch size, 1000
Exploration ϵ	$0.01,\ 0.1$
Adam optimizer $\beta 2$	$0.9,\ 0.999$
Adam optimizer ϵ	$1 \cdot 10^{-8},\ 0.1$

Table 3. Hyperparameter ranges for one-percent-tuning on DQN and mitigations on dancing catch

A.2. SAC tuning

The architecture as well as the default hyperparameter values are as previously described for the DeepMind Control Suite (Haarnoja et al., 2018), and we use orthogonal initialization. We use 3 random seeds for tuning SAC agents. The hyperparameter tuning ranges can be seen in Table 6, and the default hyperparameters and the tuning results in 8. The tuning curves can be seen in Figure 8 to 12.

For one-percent-tuning, the agent performs random exploration for $10,000$ iterations, followed by training for $10,000$ iterations. The top hyperparameters are picked based on the biggest Area Under the curve (AUC) for the $10,000$ training iterations, or for the 10% final return for those iterations.

For final training, we use 10 random seeds. The online return is used in all cases to simulate an agent learning while performing real-world tasks.

	1%-tuning values for PT DQN on dancing catch
Learning rate θ	$3 \cdot 10^{-2},\ 2 \cdot 10^{-2},\ 1 \cdot 10^{-2},\ 1 \cdot 10^{-3},\ 1 \cdot 10^{-4},\ 1 \cdot 10^{-5},\ 1 \cdot 10^{-6}$
Learning rate w	$2 \cdot 10^{-2},\ 1 \cdot 10^{-2},\ 1 \cdot 10^{-3},\ 1 \cdot 10^{-4}$
Batch size	64
Buffer size	$100,000$
Initial buffer fill	$64,\ 1000$
Exploration ϵ	$0.01,\ 0.1$
Adam optimizer $\beta 2$	$0.9,\ 0.999$
Adam optimizer ϵ	$1 \cdot 10^{-8},\ 0.1$

Table 4. Hyperparameter ranges for one-percent-tuning on PT-DQN on dancing catch

	AUC	final 10%	best-worst
Learning rate θ	$1 \cdot 10^{-3}$	$2 \cdot 10^{-2}$	$2 \cdot 10^{-2}$
Learning rate w	$1 \cdot 10^{-2}$	$1 \cdot 10^{-2}$	$1 \cdot 10^{-2}$
Batch size	64	64	64
Buffer size	$100,000$	$100,000$	$100,000$
Initial buffer fill	64	1000	64
Exploration ϵ	0.01	0.01	0.01
Adam optimizer $\beta 2$	0.9	0.999	0.999
Adam optimizer ϵ	$1 \cdot 10^{-8}$	$1 \cdot 10^{-8}$	$1 \cdot 10^{-8}$

Table 5. PT-DQN values on dancing catch from one-percent-tuning, selected by AUC, by final 10% performance, and by best-worst performance. Tuning was done with 3 seeds. Batch size is at a default value of 64, and buffer size at a default value of $100,000$

	1%-tuning SAC parameter values
Learning rate	$2 \cdot 10^{-2},\ 1 \cdot 10^{-2},\ 1 \cdot 10^{-3},\ 1 \cdot 10^{-4},\ 1 \cdot 10^{-5},\ 1 \cdot 10^{-6},\ 1 \cdot 10^{-7}$
Batch size	$16,\ 32,\ 128,\ 256,\ 512$
Buffer size	$512,\ 1000,\ 10000$
Adam optimizer $\beta 2$	$0.9,\ 0.999$
Adam optimizer ϵ	$1 \cdot 10^{-8},\ 0.1$

Table 6. Hyperparameter ranges for one-percent-tuning on SAC on DeepMind Control Suite environments

	default	quadruped-run	quadruped-walk	quadruped-escape	cheetah-run	hopper-hop
Learning rate	$3 \cdot 10^{-4}$	$1 \cdot 10^{-6}$	$1 \cdot 10^{-3}$	$1 \cdot 10^{-5}$	$1 \cdot 10^{-2}$	$1 \cdot 10^{-2}$
Batch size	256	32	512	128	128	512
Buffer size	$1,000,000$	512	$10,000$	$10,000$	$1,000$	512
Adam optimizer $\beta 2$	0.999	0.9	0.9	0.999	0.999	0.999
Adam optimizer ϵ	$1 \cdot 10^{-8}$	0.1	$1 \cdot 10^{-8}$	0.1	$1 \cdot 10^{-8}$	$1 \cdot 10^{-8}$

Table 7. Default hyperparameter values and values selected from one-percent-tuning for SAC for each of the DeepMind Control Suite environments in this paper. Tuning was done with three seeds. The values were the same for selection via AUC as for final 10% return

B. Definition of Stable Rank

The normalized stable rank for a layer's weight matrix, w_l with dimensions $n * m$ is defined as

$$R(w_l) \ = \ \frac{1}{n} \frac{\|w_l\|_*}{\|w_l\|_2} \ = \ \frac{1}{n \, \sigma_1^2(w_l)} \sum_{i \, = \, 1}^{n'} \sigma_i^2(w_l)$$

where, $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_{n'}$ are the singular values in descending order and $\| \cdot \|_*$ stands for nuclear norm.

To get the stable rank for an entire network, we use the average of the normalized stable ranks for all weights in the network.

	default	quadruped-run	quadruped-walk	quadruped-escape	cheetah-run	hopper-hop
Learning rate	$3 \cdot 10^{-4}$	$1 \cdot 10^{-6}$	$1 \cdot 10^{-3}$	$1 \cdot 10^{-5}$	$1 \cdot 10^{-2}$	$1 \cdot 10^{-2}$
Batch size	256	32	512	128	128	512
Buffer size	$1,000,000$	512	$10,000$	$10,000$	$1,000$	512
Adam optimizer $\beta 2$	0.999	0.9	0.9	0.999	0.999	0.999
Adam optimizer ϵ	$1 \cdot 10^{-8}$	0.1	$1 \cdot 10^{-8}$	0.1	$1 \cdot 10^{-8}$	$1 \cdot 10^{-8}$

Table 8. Default hyperparameter values and values selected from 1% tuning for SAC for each of the DeepMind Control Suite environments in this paper. Tuning was done with three seeds. The values were the same for selection via AUC as for final 10% return

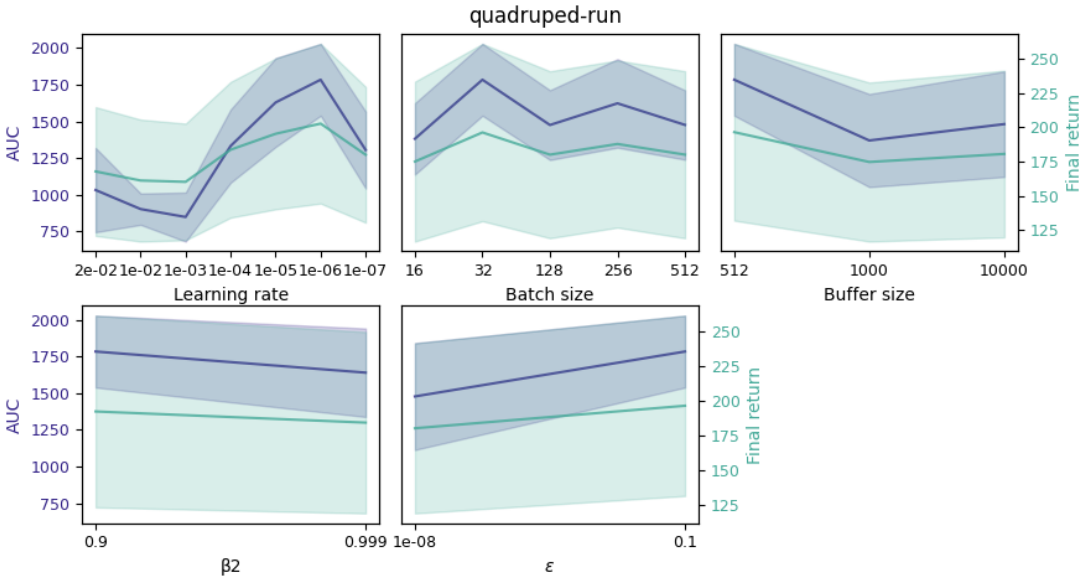

Figure 8. Hyperparameter values for one-percent tuning of SAC on quadruped-run. There are three seeds per point. The shading is the standard deviation.

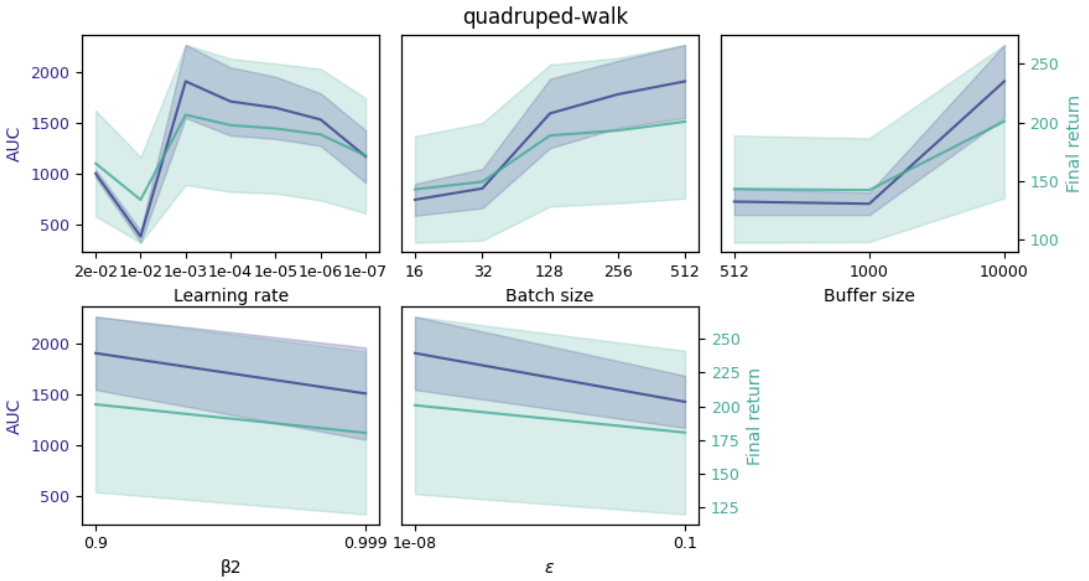

Figure 9. Hyperparameter values for one-percent tuning of SAC on quadruped-walk. There are three seeds per point. The shading is the standard deviation.

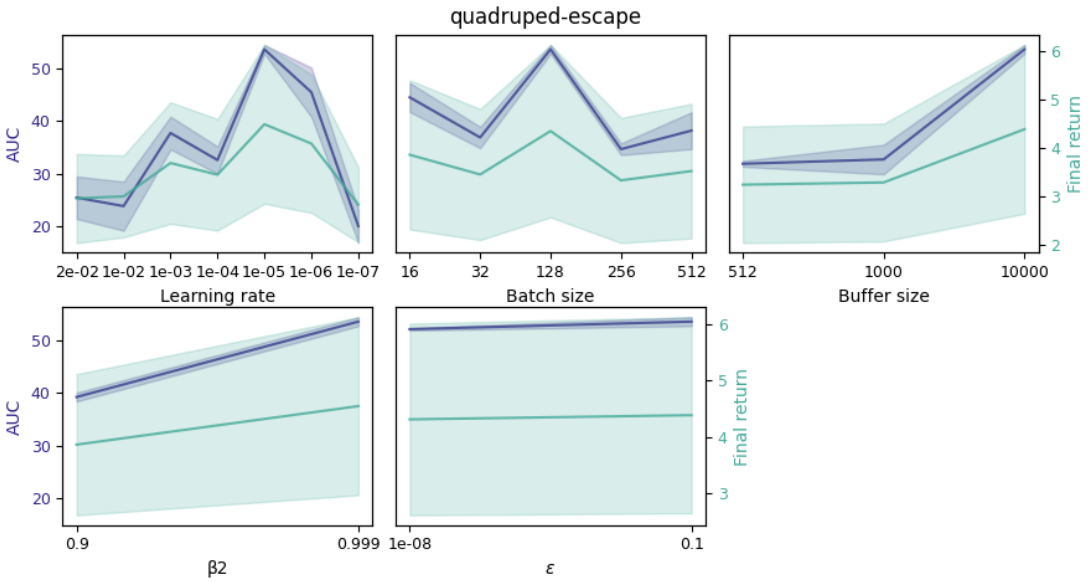

Figure 10. Hyperparameter values for one-percent tuning of SAC on quadruped-escape. There are three seeds per point. The shading is the standard deviation.

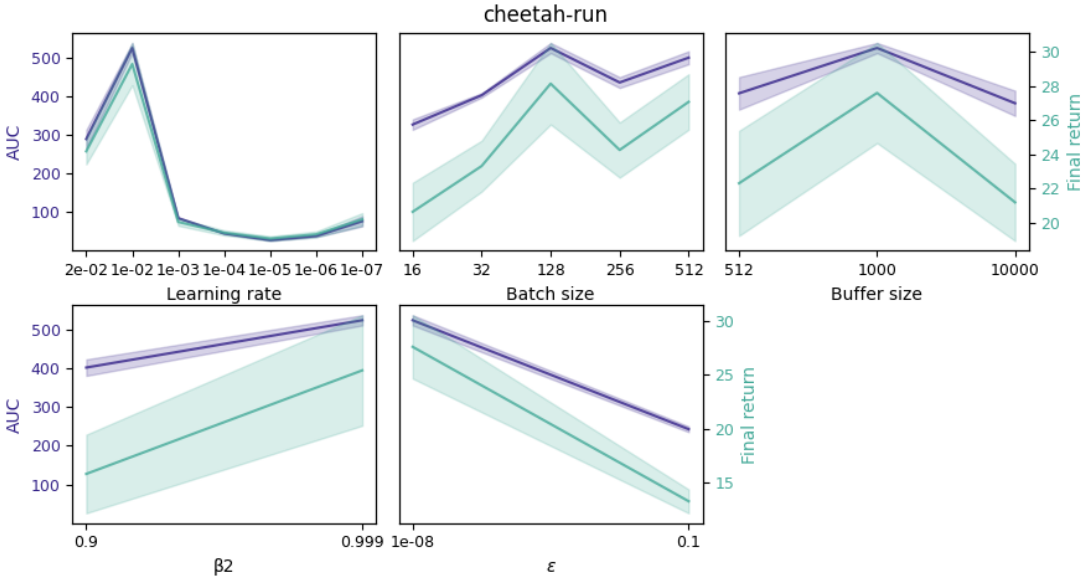

Figure 11. Hyperparameter values for one-percent tuning of SAC on cheetah-run. There are three seeds per point. The shading is the standard deviation.

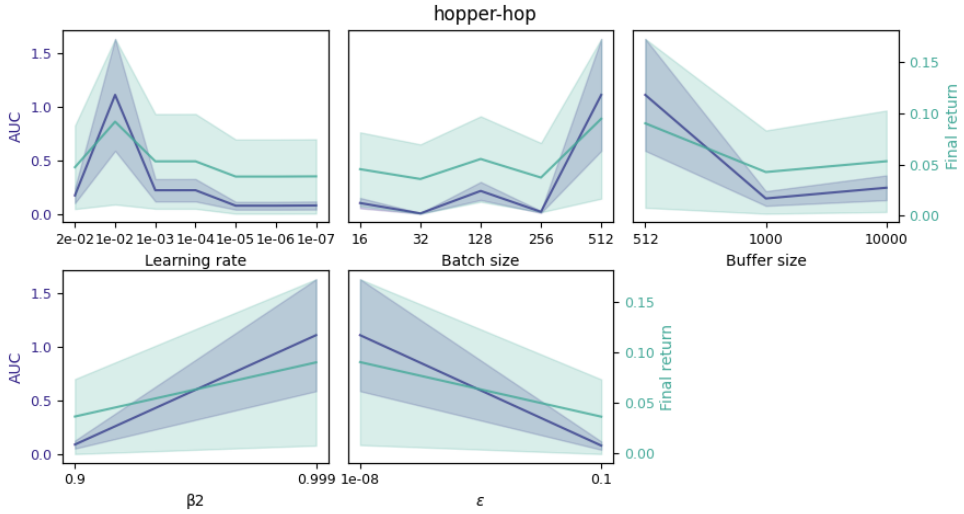

Figure 12. Hyperparameter values for one-percent tuning of SAC on hopper-hop. There are three seeds per point. The shading is the standard deviation.