K-percent Evaluation for Lifelong Reinforcement Learning

by

Golnaz Mesbahi

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

Master of Science

Department of Computing Science

University of Alberta

Abstract

If we aspire to design algorithms that can run for long periods, 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 the deployment environment for the full lifetime of the agent. In this thesis, we propose a new approach for evaluating lifelong RL agents where only k percent of the experiment data can be used for hyperparameter tuning. We then conduct an empirical study of DQN and SAC across a variety of continuing and non-stationary domains. We find agents generally perform poorly when restricted to k-percent tuning, whereas several algorithmic mitigations designed to maintain network plasticity help with the performance. In addition, we explore the impact of the tuning budget (k) on algorithm performance and hyperparameter selection, and assess various mitigation strategies' network properties to analyze their behavior.

Preface

This thesis is an extension of the "K-percent Evaluation for Lifelong RL" submission to the Conference on Neural Information Processing Systems (NeurIPS), 2024. It is a joint work with Parham Mohammad Panahi, Olya Mastikhina, Martha White, and Adam White. Parham and I are responsible for the main experiments. Olya helped with the SAC experiments, and Adam and Martha wrote the main body of the paper, including the introduction, methodology, and conclusion. I wrote the experiment section and appendix with the help of Parham and Olya.

To my parents for their constant support
and to my sister, who brings brightness to my life

In the darkest of nights
at the end of a blind alley
on the top of a mud wall
jasmine buds burst into bloom...

– Abbas Kiarostami

Acknowledgements

This thesis is the result of the support and encouragement of many amazing people I have met throughout my studies, from the first day of school to my undergraduate and graduate days. While I cannot name everyone, I would like to express my deep gratitude to those who have made significant contributions and have been inspiring teachers to me.

In particular, I am grateful to Peyman Setoodeh, who first introduced me to reinforcement learning in his course. His engaging teaching sparked my interest in the topic and set me on this path.

I would like to genuinely thank my supervisors, Adam and Martha White. They have provided consistent support throughout my research journey, helping me navigate the ups and downs of exploratory research. Their guidance has also helped me improve my writing and research skills.

My gratitude also extends to Matthew Guzdial for his valuable feedback as a committee member.

I am deeply appreciative of Andrew Patterson and Matthew Schlegel for teaching me how to run better experiments when I first joined RLAI lab and for their insightful advice on improving my research.

Special thanks to Olya Mastikhina and Parham Mohammad Panahi for their assistance with the paper submission.

Finally, I want to express my heartfelt appreciation to my friends, especially Mahdieh Mallahnezhad, who supported me through some of my challenging days, and to my family, whose warmth has been a comforting presence.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Continual or lifelong reinforcement learning (RL) arises in many applications. 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 the length of the deployment [26]. Similar situations arise in data-center cooling [23], water treatment [17], and many other industrial control settings. These examples illustrate the widespread relevance and importance of continual RL in real-world applications.

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 complete each game (similar to the Switching ALE benchmark [1]). Mujoco tasks are naturally continuing, but common practice is to truncate experiments after a fixed number of interactions, resetting to some initial configuration. These benchmarks highlight the need for suitable methodologies to design and evaluate agents in such tasks. In lifelong learning tasks, we should design and evaluate 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.

Despite its importance, 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 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 evaluate our agents, given access to their full lifetime.

Given these challenges, we suggest deploying the agents after a limited number of interactions with the environment and without further tuning of hyperparameters for their lifetime. Once deployed, the agent should ideally adapt and perform well over time. This approach of evaluating agents in a continual setting motivates the design of algorithms that are robust to environmental nonstationarities and can adapt over time.

In this introduction, we first briefly define continual RL and discuss common challenges faced by deep RL algorithms in a continual setting. We then highlight the critical role of hyperparameter tuning and demonstrate that current selection methods do not consider the constraints of continual learning systems, as they tune for the agent's full lifetime.

To address the issues in continual RL, we introduce solutions and evaluation methods proposed in the literature and motivate further research into evaluation methods for continual learning. Finally, we outline the contributions made in this thesis.

1.1 A Brief Definition of Continual RL

This thesis focuses on the evaluation strategies in continual reinforcement learning (CRL). Although continual RL is a broad term with ongoing research trying to formalize the term, we try to specify the framework in the following paragraphs.

Continual reinforcement learning is usually defined as a framework where the agent-environment interaction is never-ending and the agent has to learn to adapt to new information endlessly, as opposed to finding a fixed policy [2]. In reality, the agent-environment interaction eventually ends. However, the agent is not aware of the exact time it will end as it is also unaware of when or how the changes happen in the environment. This continual need for

adaptation necessitates unique approaches to the design and evaluation of the algorithms.

In this framework, we typically look for particular properties [40]: The environment is more complex than the agent. Moreover, the agent operates in a continual setting with a single uninterrupted life, without resets. Additionally, the dynamics or the reward function change over time due to the non-stationary nature of the environment. This non-stationarity can be either sudden or gradual. Note that there is already an inherent non-stationarity built into the RL framework because of the improving policy [16]. Understanding these properties is crucial for developing effective evaluation strategies.

Enforcing these constraints onto the agent is particularly interesting because they can help us develop algorithms that are better suited for real-world applications [18]. This thesis explores an evaluation method that better follows these constraints.

1.2 Challenges of Continual RL

A growing body of literature demonstrates that within the framework of continual reinforcement learning, particularly when using deep RL methods, neural networks show poor performance [4].

The following is a list of these closely related phenomena in CRL:

1. **Loss of Plasticity** The neural network's capacity to learn new policies is diminished over time as a result of changes in data distribution [11].

2. **Primacy Bias**: Deep RL methods tend to overfit early interactions and ignore the later experience, which is detrimental to the learning process [35].

3. **Catastrophic Forgetting**: Neural networks are prone to gradually forgetting previously known information [12].

4. **Implicit Underparametrization**: The combination of bootstrapping and gradient descent in deep RL methods can lead to implicit underparametrization, resulting in deteriorating performance over time [20].

There have been several investigations on the potential causes of this phenomena. Some of the potential reasons include an increase in the number of dead neurons [1] , weight and gradient norm growth [29], [35], and pathologies in the loss landscape of the optimization problem [25], [28]. Each potential cause can be assessed using specific metrics to gauge its impact.

1.3 Role of Hyperparameters and Design Choices in Continual RL

The choice of hyperparameters can directly affect the phenomena mentioned above in algorithms. These choices include the optimizer, step size, activation functions, warmup steps, width, and depth of the networks, batch size, and many other agent-specific hyperparameters. Selecting the right combination of these hyperparameters is crucial for performance, and poor choices can lead to suboptimal performance or even performance collapse.

For example, architectural choices play an important role in influencing the performance of the algorithms. Hyperparameters such as width, depth, use of batch normalization, skip connections, and pooling layers can directly affect the performance. In some cases, even simply modifying the architecture can achieve a similar or better performance compared to specially designed continual learning algorithms [31]. Therefore, carefully considering architectural elements is essential for improving algorithm performance

Hyperparameter tuning can significantly affect the stability and performance of algorithms. For instance, wider networks have been shown to mitigate catastrophic forgetting [30]. Furthermore, factors such as dropout, learning rate decay, and batch size can also affect the loss landscape and catastrophic forgetting. Additionally, it is shown that hyperparameters cannot be transferred across different data regimes [37]. Therefore, proper hyperparameter tuning is crucial for the success of continual learning methods.

The choice of activation function also influences loss of plasticity. Activation functions such as leaky ReLU [10] and Concatenated ReLU [1] can help maintain plasticity. Furthermore, adjusting the hyperparameters of the

Adam optimizer can directly impact algorithm stability during task switches [27]. Consequently, selecting appropriate activation functions and optimizer settings is vital for maintaining learning efficiency.

Despite the importance of hyperparameters in the behavior of algorithms, current practices often fall short. The widely used method for choosing hyperparameters for an algorithm in deep RL is either to tune for the lifetime of the agent or to use previously selected hyperparameters. These traditional approaches take no notice of the direct effect of the hyperparameters on the learning difficulties of algorithms in a continual setting. They also do not take into account the assumptions in continual learning that agent-environment interaction will be for a very long and undefined time, and the agent is not aware of when or how the non-stationarities will happen in the environment. This oversight highlights the need for more dynamic and context-aware hyperparameter tuning and evaluation methods.

1.4 Solutions and Methods in Continual RL

There has been increased focus on extending or modifying existing deep RL agents for continual RL, with limited success. These approaches can be roughly categorized into three groups: resetting, regularization, and normalization.

1) Resetting: This approach periodically resets parts of the agent's network to random initial values, initially causing large performance drops but eventually leading to improved final performance [8], [34], [35], [46].

2) Regularization: Regularization techniques introduce additional information by either modifying the loss function of the agent or modifying the weight updates, in order to prevent overfitting [24]. Examples of this method are dropout [32], L2 regularization [11], elastic weight consolidation [19], and regenerative regularization [21]. In particular, we further explore regenerative regularization in this thesis, which balances error reduction with keeping the agent's network parameters close to initialization [21]; this helps because the random initial parameters help the network learn quickly.

3) Normalization: Recent studies have found that layer normalization can

help maintain the ability to learn [29].

All these approaches are mitigations: algorithmic fixes applied to a base agent that is not designed for lifelong learning.

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 [8], [34].

In contrast, there are several algorithms designed from the first principles for continual RL. Continual backpropagation [11], for example, was designed for and evaluated in never-ending regression and RL control tasks. This algorithm randomly re-initializes connections in the network to promote continual adaptation in the face of non-stationarity. Similarly inspired, Permanent-transient networks [3] 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.

However, more is needed than to evaluate these proposed mitigations in conventional ways. The average or final performance will not give us enough information about these mitigation methods' reliability for a continual setting. [18], [24]. New metrics such as backward transfer, forward knowledge transfer, and average forgetting over time are proposed to better evaluate the continual learners [9]. However, hyperparameter selection should be an integral part of the evaluation process because the choice of hyperparameters can directly impact the performance and stability of these mitigation strategies. Still, hyperparameter selection is a critical yet overlooked aspect in this area.

In conclusion, while these solutions show promise in mitigating the challenges of continual RL, there is a need for improved evaluation methodologies that account for the constraints of continual learning scenarios

1.5 Thesis Contributions

This thesis explores the notion that progress in continual RL research has been held back by inappropriate empirical methodologies. We propose a new methodology for tuning and evaluating continual RL agents inspired by the constraints of real-world applications of RL. Our proposal is based on a simple idea: continual 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: a small percent of the total lifetime. Only k-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 is inspired by 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. The goal of our proposed evaluation methodology is to encourage the development of agents that are more suitable for continual RL and perhaps deployment in the real world, not introduce a way to tune hyperparameters for real-world deployment.

In our first set of experiments, we verify that a popular and performant deep RL agent, DQN, performs poorly across a suite of continual RL tasks irrespective of what metric is used to select the best hyperparameters under k-percent evaluation. We additionally test Soft Actor-Critic, to see the impact of k-percent evaluation on a different algorithm in the continuous action setting, finding similar outcomes.

Moreover, we investigate the effect of k on the performance of the algorithms, and the hyperparameters chosen. We show that the minimum value of k, the interaction budget for tuning required for good performance, is agent-environment dependent, and the value of the hyperparameters including step size and warmup step is meaningfully related to the amount of data used for tuning.

We then investigate several mitigation strategies, including regularizing to

the initial weights, Concatenated ReLU, and layer normalization, under k-percent evaluation finding most actually improve performance compared to the base algorithms. In many cases, however, 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 [20], dormant [43] or inactive neurons [1], [11], [27], and weight norms [35]. Under k-percent evaluation, we see that some of these metrics actually correlate with performance, suggesting explanations for the agents' behavior.

Finally, we show that mitigation methods that are more robust under k-percent evaluation, including Permanent-transient networks (PT-DQN), and layer normalization, are more desirable.

Chapter 2

Background

In this chapter, we present the problem formulation and the notations used throughout this thesis. We begin by describing the framework and then introduce the baseline algorithms used in our study.

2.1 Markov Decision Process

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 transitions 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 is 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 ends.

2.2 Value Functions

The agent's task is to find a policy π that maximizes the expected discounted sum of rewards. To achieve this, the agent needs an estimate of how good a state is, provided by the state value function $v(s)$. The state value function is

defined as:

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t \mid S_t = s]$$

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 [45] for further details).

Additionally, there is the state-action value (Q-value) function, $q_\pi(s, a)$, which considers both the state and the action, unlike the value function which only considers the state. The state-action value function provides an estimate of how good an action is, given the current state, and is defined as:

$$q_\pi(s, a) \doteq \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a]$$

Respectively, the optimal state-action value function is defined as:

$$q_*(s, a) = \max_\pi q_\pi(s, a)$$

The goal is to find a policy that maximizes the state-value function, which we refer to as the optimal policy, π_*.

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 chapter, is that in lifelong RL the agent designer does not exploit knowledge of T in the design or evaluation of the agent.

2.3 Agents Considered in This Thesis

In most interesting tasks, the agent cannot directly observe the underlying state; instead, only an observation 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: $\hat{q}_w(S_t, A_t) \approx \mathbb{E}_\pi[G_t|S_t = s, A_t = a]$, where w are the learnable parameters of a neural network. We use the DQN algorithm [33] to learn $\hat{q}_w$ and select actions. In the case of continuous actions, we learn a parameterized policy π_w where w are the parameters of a network with Soft Actor-critic (SAC) [14]. We will provide a brief definition of these two algorithms in the next two sections.

2.3.1 Deep Q-Network (DQN)

As mentioned, we use the Deep Q-Network agent (DQN), where the future reward, called the Q-value function, is estimated with a neural network as:

$$\hat{q}_w(S_t, A_t) \approx \mathbb{E}_\pi[G_t|S_t = s, A_t = a]$$

where w are the parameters of the Q-network. DQN utilizes bootstrapping and gradient descent for training the network to minimize the mean-squared temporal difference error. The mean-squared temporal difference error is defined as:

$$L(w) = \sum_{s\in\mathcal{S}, a\in\mathcal{A}} \left(R_{t+1} + \gamma \max_{a\in\mathcal{A}} \hat{q}_{\overline{w}}(S_{t+1}, a_t) - \hat{q}_w(S_t, A_t) \right)^2$$

where $\overline{w}$ is the delayed copy of w, which belongs to a second network referred to as the target network. The target network is utilized to calculate target Q-values. Target networks are usually used to stabilize learning. As shown in sections 6.1.1, and 6.1.2, the loss function $L(w)$ can then be modified to add regularization in order to help the network maintain learning stability.

The Q-network consists of multiple layers, each containing a number of hidden units. We refer to the weights of each layer l as w_l, the gradients of the weights as ∇w_l, and the output of each hidden layer l at as h_l. We use this notation to later define the network properties in chapter 7.

2.3.2 Soft Actor-Critic (SAC)

The Soft Actor Critic consists of two components. The actor is a parametric policy that is responsible for choosing actions, and the critic is a value function learned by a temporal difference learning algorithm. The goal of SAC is to optimize the continuing entropy-regularized objective, defined as:

$$\mathcal{J}_\tau(\pi) = \mathbb{E}_{\{S_{t+k}\} \text{ in trajectory of } \pi} \left[G_t + \tau \sum_{k=0}^{T} \mathbb{H}(\pi(\cdot|S_{t+k})) \right]$$

where $\mathcal{J}_\tau(\pi)$ is the entropy-regularized objective with entropy scale parameter $\tau \in \mathbb{R}^+$, π represents the policy, G_t denotes the discounted sum of rewards, and $\mathbb{H}(\pi(\cdot|S_{t+k}))$ is the entropy of the policy π given state S_{t+k}.

The optimization problem associated with SAC is to find the policy π^*_τ that maximizes $\mathcal{J}_\tau(\pi)$. This objective forces the algorithm to find policies that not only maximize the expected reward but also the entropy. This encourages exploration and improves the robustness of the algorithm.

2.4 Summary

In this chapter, we formulated the problem addressed in this thesis. We began by introducing Markov Decision Processes (MDPs), and episodic and continuing problems, focusing on continuing problems, which are particularly relevant in lifelong learning. The focus is on continuing problems in this thesis, as two out of the three environments used for testing DQN are continuing, and SAC experiments are also performed in continuing settings. A continuing problem is characterized by a never-ending agent-environment interaction, where the agent has a single life. Next, we discussed value functions and extended this discussion to value functions with function approximation, highlighting Deep Q-Networks (DQN) as the primary algorithm used in our experiments. Finally, we introduced Soft Actor-Critic (SAC) as the second algorithm evaluated in our experiments.

Chapter 3

k-percent Tuning

The common agent development-evaluation loop in RL is artificial and not particularly reflective of biological systems or 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 want to report the performance of an agent that is well-engineered for the task. 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 the lifetime and record the performance. The process is iterated until 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. A DQN is one of the simplest such agents, and it contains over 14 hyperparameters controlling the 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 allow us to instantiate DQN variants 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 allows 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 [29].

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 a 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 hyperparameters. 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 k-percent tuning methodology mechanizes these goals. The name describes the relatively simple idea: we propose to tune the agent only for $k\%$ of its lifetime. Though the agent cannot know its 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 $\lfloor 0.01kn \rfloor$ steps. We multiply by 0.01 to convert the percent from the range of 0 to 100 into a percentage from 0 to 1. In other words, for every hyperparameter setting, we run the agent for $\lfloor 0.01kn \rfloor$ steps to obtain the performance metric after this short learning time. We then choose 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.

Chapter 4

Failure of Standard Algorithms Under k-Percent Evaluation

In this chapter, we evaluate our proposed methodology for tuning lifelong RL agents. In our experiments, we use k equal to one and twenty percent, for different environment settings. We contrast the k-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 no obvious default hyperparameters. We perform the experiments with DQN in three discrete action environments and SAC in one continuous control environment.

4.1 Failure of DQN Under k-Percent Evaluation

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 k-percent-tuning. These metrics include area under the learning curve (AUC) which corresponds to overall performance in the tuning phase, the best performance in the final 10% of the tuning phase, and finally the best worst-case

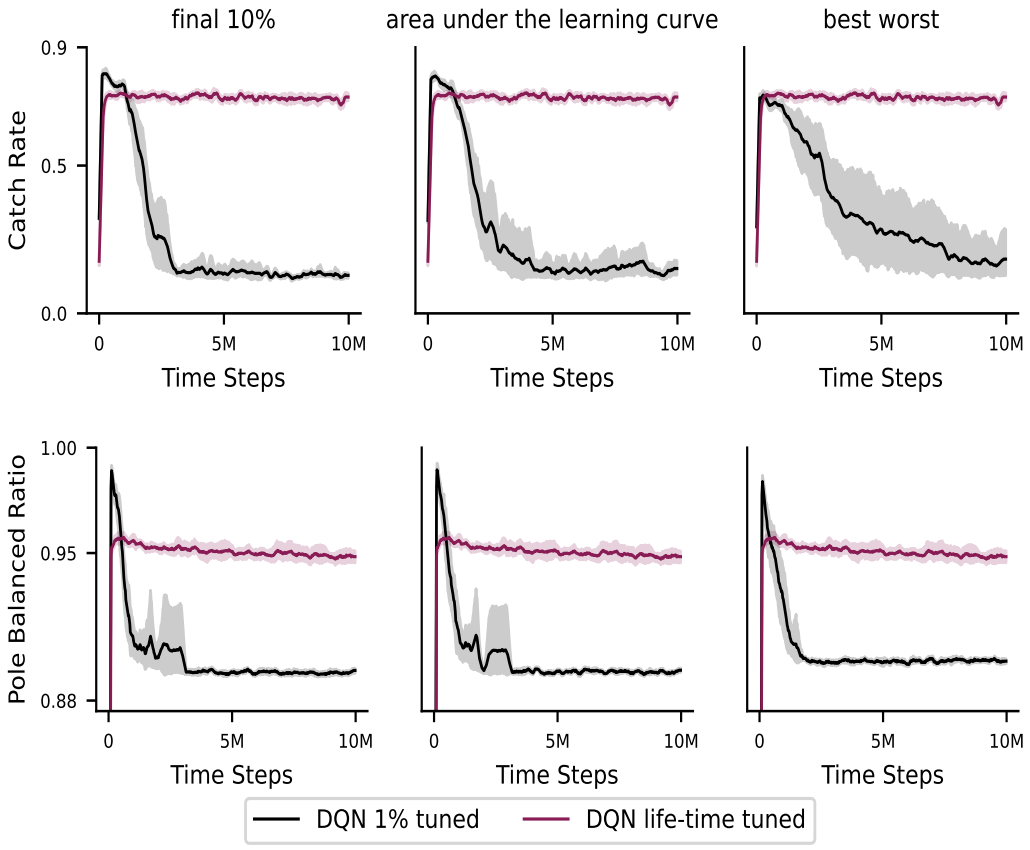

Figure 4.1: Tuning on one-percent of a lifetime leads to poor performance for DQN in Non-stationary Catch and Continuing Cart-pole. Each row of plots corresponds to a different environment, and each column corresponds to a different hyperparameter selection strategy. Lines are averaged over ten seeds and the shaded regions are 95% bootstrap confidence interval.

performance across seeds, to select hyperparameters that are robust across seeds, which we call best-worst.

We test DQN in two environments: Non-stationary Catch and Continuing Cart-pole. Non-stationary Catch [13] 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 a probability of 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 agents are run for 10 million steps, with $100,000$ steps for the one-percent-tuning. 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 [6] 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 the left or right with the 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 teleported to the position, but the agent is not reset. The agents are run for 10 million steps, with $100,000$ steps for the one-percent-tuning. 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 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 4.1 and are as expected after the first

100, 000 steps. None of the three criteria prevent this collapse and result in relatively similar performance. Best-Worst is more effective than Final 10% and AUC in Non-stationary Catch, and all three are similar in Continuing Cart-pole.

4.2 Failure of SAC in Continuous Control

We ran a similar experiment with SAC in a modified environment from the DeepMind Control Suite [44]. The DeepMind Control Suite environments are large-scale continuous control environments commonly used in deep RL research. The environments are physical simulations, making them useful for investigating tuning in semi-real-world settings.

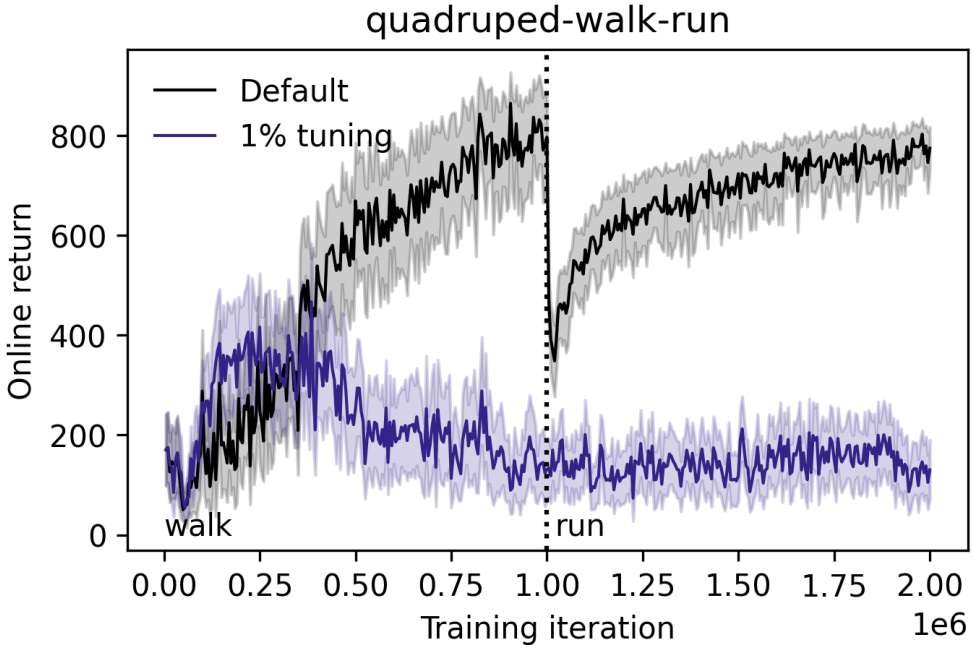

Figure 4.2: Tuning on one-percent of a run similarly leads to poor performance for SAC in a task-switching setting. The results are averaged over ten runs with standard error.

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 [14]. The agents are run for 1 million steps, with $10,000$ exploration steps followed by training over $10,000$ steps.

We 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. We call this designed environment the switching Quadruped-walk-run. The agent is tuned for one-percent of the experiment in quadruped-walk. In Figure 4.2, we see a more noticeable improvement over SAC with default hyperparameters in early learning for quadruped-walk, but we see a performance drop and then almost no learning in quadruped-run.

4.3 Scaling Up: Jelly Bean World

We also performed larger-scale experiments with DQN in an environment called Jelly Bean World. Jelly Bean World is a testbed for developing never-ending learning algorithms [38]. This environment is an infinite two-dimensional grid world that is filled with different items, each with its corresponding reward, and the agent can move through the procedurally generated environment, constantly trying to adapt.

We follow the modified version of this environment, where the reward function is swapped every 150k steps to add reward non-stationarity [3]. In this configuration, the observation is an 11*11 RGB array representing an egocentric 360-degree view of the agent. The agent can take the four actions of up, down, left, and right, and each action takes the agent to the next square of the grid world in that direction. The items in the environment are represented with colors, and each color has its corresponding rewards. The reward is $+0.1$ for some items, and other items' rewards alternate between -1 and $+2$ every 150k steps. You can find a visualization of the agent's view of the environment in Figure 4.3. We report the average reward over a 1000-sized window as the performance measure. We ran DQN in this environment for 1.5 million steps,

where the agent sees 10 swaps. We tuned the agents for twenty percent of their lifetimes ($k = 20$), to allow them to see both sides of the game only once. This provides enough time for the agent to see part of the non-stationary in this complex environment and reach a fairly good performance in this duration, but not be aware of later non-stationarities.

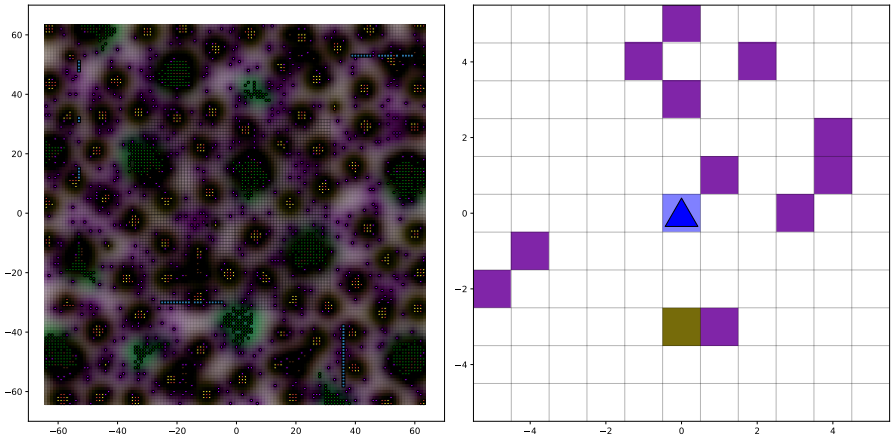

Figure 4.3: Agent World view in Jelly Bean World

We can see in Figure 4.4 that the same pattern holds, with the twenty-percent-tuned agent performing initially better but worse over the lifetime. Note that for this experiment, this plot only shows the agents with hyper-parameters selected based on the best-worst performance of the final 10%, which is a combination of two of the hyperparameter selection strategies. This method tends to be more robust because it takes into account both the worst seed and the most recent information.

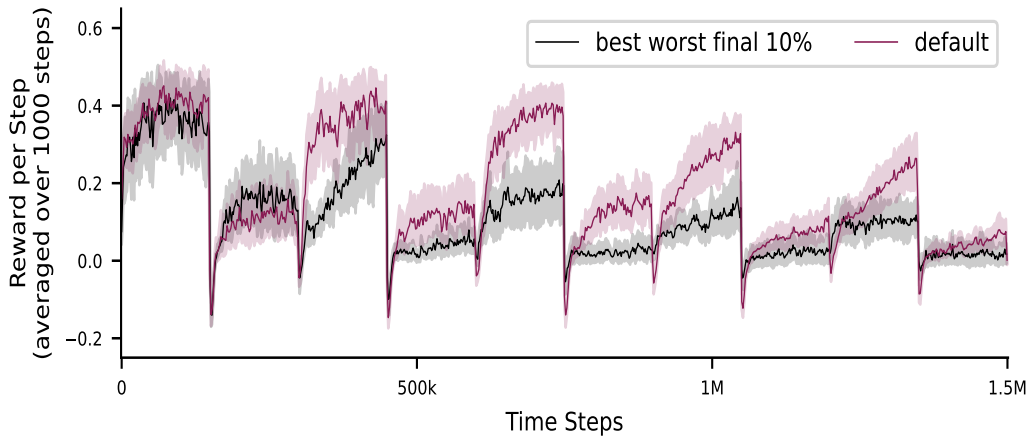

Figure 4.4: Tuning for twenty-percent of the lifetime leads to poor performance for DQN in Jelly Bean World. Lines are averaged over ten seeds with 95% bootstrap confidence intervals.

4.4 Summary

In this chapter, we tested DQN and SAC under the k-percent evaluation methodology. The experiments show that irrespective of the hyperparameter selection strategy (final ten percent, best-worst performance, and AUC), agents fail to perform well continually when exposed to only a fraction of their lifetime as the tuning phase. DQN fails under one-percent evaluation in Non-stationary Catch, and Continuing Cart-pole, and under twenty-percent tuning in Jelly Bean World, a computationally scaled-up environment. Moreover, SAC fails under one-percent tuning in Quadraped walk-run. These experiments emphasize the vulnerability of the standard algorithms under k-percent evaluation. While agents exhibit competitive performance during the tuning phase, they struggle to maintain this performance throughout their entire lifetime.

Chapter 5

The Impact of k

In this chapter, we intend to answer this empirical question: what is the impact of k on the performance and behavior of the agent? Furthermore, how do hyperparameters change with different values of k?

The value of k is important as it indicates how much of the agent's lifetime the agent has access for tuning. The choice of k is problem and research-question dependent. This choice depends on the goals of the experimenter. k can be chosen in a way that gives the agent enough time to reach a fairly good performance and visit some of the non-stationaries (for instance, visiting a limited number of task switches, season changes, etc.), but not all of them. This is to simulate a condition where we have limited knowledge about the agent's lifetime and non-stationarity. Tuning under increasing k can also give researchers better insight into their algorithm.

5.1 Case Study: DQN in Non-stationary Catch and Continuing Cartpole

We evaluate the DQN agent in Non-stationary Catch and Cotinuing Cartpole for values of k= 1, 5, 10, 20, 30, 50, 70, 100 percent. We tune the DQN for the mentioned durations, select the best-performing hyperparameters based on four hyperparameter selection strategies, and report the mean performance of those hyperparameters in a full-length experiment (10M steps for 10 seeds). In Figure 5.1, as we expand the tuning window, the performance starts to improve. More demonstrations are further discussed in section 6.4.

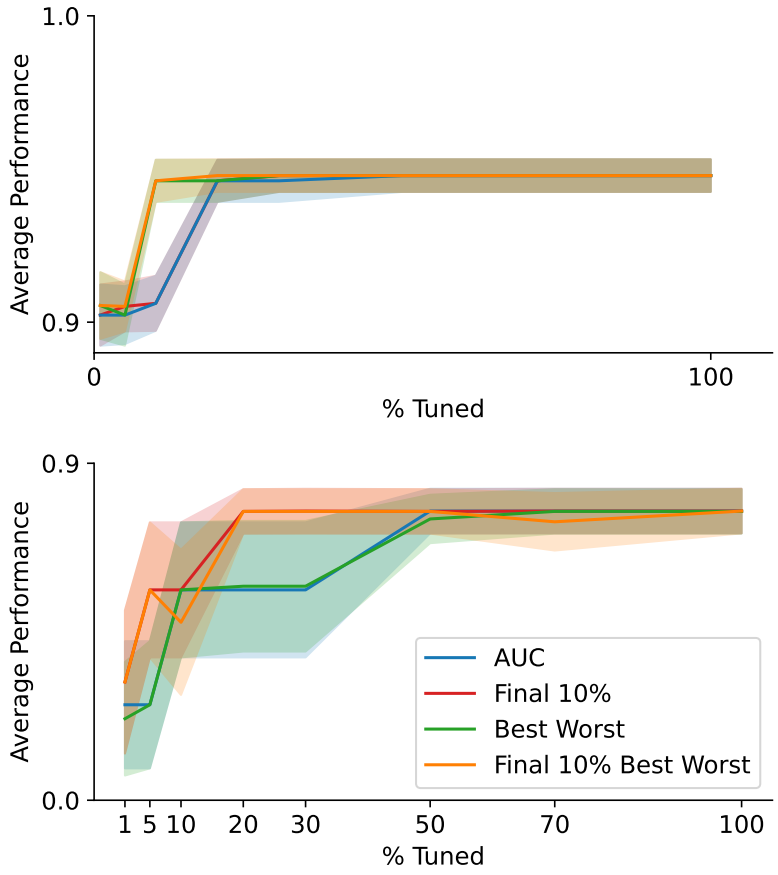

Figure 5.1: Effect of k on the performance of DQN in Non-stationary Catch (down) and continuing Cart-pole (up), over its entire lifetime. Results are averaged over 30 seeds with shaded regions being 95% student-T confidence intervals.

5.2 Effect of k-Percent Tuning on Hyperparameter Selection

We can also look at the difference in hyperparameters under different k:

Learning Rate: We observe that the tuning procedure mostly chooses larger learning rates for smaller k values. For instance, for DQN in Non-stationary Catch, learning rates of 0.001 and 0.0001 are chosen respectively for k values of one and a hundred. We also found that DQN in Jelly Bean chose 0.001 for twenty percent tuning, in contrast to 0.0001 as the default.

Exploration Factor: We also found that smaller values for the exploration factor were chosen for smaller k values. For instance, in Continuing Cartpole,

one-percent tuned DQN has an exploration factor of 0.01 whereas the value is 0.1 in the lifetime-tuned agent.

Warmup Steps: Finally, the number of chosen warmup steps was generally smaller for smaller k. For instance, in the Non-stationary Catch, the one-percent tuned agent chooses a warmup value of 0 under two out of the three hyper-selection strategies, compared to a value of 1000 under full-lifetime tuning.

It is also valuable to compare different hyperparameter selection methods and their effects on the hyperparameters. Our experiments show that the best-worst metric (and also a combination of final 10% and best-worst) tends to choose more robust hyperparameters: smaller learning rates, larger exploration factors, and bigger warmup values.

5.3 Summary

We investigated the effect of k, demonstrating that researchers can gain valuable insights into their algorithms by observing their behavior under different levels of data exposure. Through a series of experiments with DQN in Non-stationary Catch and Continuing Cartpole, we evaluated performance across k values ranging from 1 to 100. The results indicated that performance generally improves as k increases. Additionally, we observed that smaller k values tend to result in higher learning rates, lower exploration factors, and fewer warmup steps. This forces the agent to learn faster initially but does not necessarily lead to good performance and stability in the long run. We also compared different hyperparameter selection methods, finding that the final 10 percent best-worst metric tends to select more robust hyperparameters. In this chapter, we focused on the effect of k on standard DQN. In the next chapter, we will investigate the effect of mitigations on performance collapse under the k-percent evaluation regime and revisit the effect of k on the proposed mitigations compared to the standard DQN in Section 6.4.

Chapter 6

Mitigations Help Under k-percent Evaluation

In this chapter, we investigate if mitigation strategies designed for lifelong learning improve performance under our k-percent evaluation methodology. We revisit the same environments and base algorithms as in chapter 4 but now include new algorithms using several mitigation strategies layered on top of the base learner.

6.1 A Brief Overview of Mitigation Strategies

In this section, we introduce the mitigation strategies we chose to test k-percent evaluation on for this thesis and expand on the algorithmic details and mechanisms these mitigations use to suit continual learning. In this thesis, we attempt to consider the various ways algorithms mitigate the issues in continual learning such as loss of plasticity.

We consider the following mitigations, where most are used for both DQN and SAC and otherwise are used only for one. They do not perfectly share the same mitigations, because for example, the PT-DQN algorithm [3] is designed only for action-values methods, so we included an additional different mitigation for SAC.

6.1.1 W0Regularization

There is previous research showing that resetting the parameters of the network to their initial distribution helps with the loss of plasticity issue in continual learning. This family of algorithms intuitively mitigates the issue by reactivating the dead or dormant neurons and using a set of freshly initialized parameters for learning new tasks [11], [35], [43]. Keeping the parameters close to initialization will also maintain the rank of the weights, a property shown to sometimes correlate with plasticity [20].

Inspired by this idea, W0Regularization [21] implicitly resets the low-utility weights by adding a new term to the loss function. In this method, the ℓ_2 norm of the difference between the weights and the initial weights is added to the loss function to encourage the weights to stay near the initialization:

$$L_{\text{w0reg}}(w) = L(w) + \lambda \|w - w^0\|_2^2$$

where w^0 are the initial weights and λ is a regularization parameter that controls the strength of the regularization. We sweep over values of 0.001, 0.0001, and 0.01 for the λ in our experiments.

6.1.2 L2Regularization

Previous research shows that parameter norm growth is one of the potential properties that contribute to the loss of plasticity [28], [35].

In the L2Regularization [10], [22], 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 magnitudes smaller in the network. The new loss function is then defined as:

$$L_{\text{l2reg}}(w) = L(w) + \lambda \|w\|_2^2$$

where $\|w\|_2^2$ is the ℓ_2 norm of the weights, and λ is the regularization parameter. We sweep over the same values as for the W0Regularization method for the λ hyperparameter.

6.1.3 CReLU

The concatenated ReLU (CReLU) activation function was first proposed as a tool for improving the performance of Convolutional Neural Network architectures [41]. The concatenated ReLU activation function concatenates the output of ReLU(x) with ReLU($-x$), meaning the output for one CReLU unit is as follows:

$$\text{CReLU}(x) = [\text{ReLU}(x), \text{ReLU}(-x)]$$

This has been shown to help the loss of plasticity issue since it limits the number of inactive units and reduces the percentage of inactive neurons since CReLU maintains 50% of the neurons in an active state [1].

6.1.4 PT-DQN

Permanent-Transient DQN [3] is an algorithm inspired by complementary learning systems (CLS) theory in the brain [36]. This theory explains that learning is done by two systems, one that slowly obtains the structured and permanent knowledge of the environment and another that adapts fast and learns new information in the non-stationary world rapidly.

Inspired by this idea, PT-DQN consists of two networks, each of which is responsible for one type of learning. The value function is decomposed into two separate networks: permanent and transient. In the case of control using function approximation, as DQN is, the overall action-value function is computed as the sum of the two value functions, as shown below:

$$Q(s, a) = Q_p(s, a; w_p) + Q_t(s, a; w_t)$$

where $Q_p(s, a; w_p)$ is the value function learned by the permanent network with parameters w_p, and $Q_t(s, a; w_t)$ is the value function learned by the transient network with parameters w_t.

The permanent network stores the permanent action-value function, and similar to the role of the neocortex, it is responsible for acquiring the general structure of the knowledge in the environment. It is updated less regularly and is usually updated with a smaller step size. The transactions are stored

in a buffer corresponding to the permanent network, and with an update frequency, the permanent network is updated, similar to a supervised learning update with the permanent value function as the target, and the combined value function as the predicted value. In other words, the permanent buffer is updated by distilling the transient network's predictions. Afterwards, the permanent buffer is reset.

On the other hand, the transient network, similar to the role of the hippocampus, is responsible for learning new information rapidly. It is updated toward the residual error from combining both networks' predictions. This network's parameters are decayed or reset after incorporating new information into the permanent network. Resetting the network to its initial parameter distribution prepares the algorithm for learning new information more plastically compared to a pre-trained network.

6.1.5 Weight Normalization

In this method, weight matrices are split into the weight magnitudes and weight directions, with separate gradients for each [39]. This decoupling of the weight magnitudes from their directions will help accelerate and stabilize the training process. Weight normalization has also been shown to push activation outputs to a homeostatic state, where all neurons in the neural network are equally activated. Hidden units in weight-normalized networks have more similar activation probabilities and may promote plasticity [42].

6.1.6 Layer Normalization

This method applies normalization to pre-activations of the neural network by using the statistics from all of the summed inputs to the neurons within one layer [5]. Layer normalization is shown to make networks more robust to optimizer choices, and induce weaker gradient covariance which intuitively corresponds with less interference [29]. It also helps with the pre-activation distribution shift, which is shown to be one of the properties that correlate with loss of plasticity [28].

6.2 k-percent-tuning for DQN with Mitigations

Figure 6.1 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, except LayerNormalization which fails under final 10% and AUC tuning and is slightly less effective than other mitigations, although better than the baseline, in best-wort tuning.

In Continuing Cart-pole, performance is much more mixed. 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. PT-DQN performs more steadily in AUC and best-worst tuning and has a higher final performance compared to other mitigations except LayerNormalization which consistently performs well under all the tuning strategies.

Figure 6.2 shows the performance of DQN with mitigations under twenty-percent tuning in Jelly Bean World. DQN performs poorly under twenty-percent tuning, but adding mitigations including l2Regularization, PT-DQN, and W0Regularization helps with performance, with W0Regularization being the most effective. CReLU initially has a good performance but is then followed by a collapse after the third swap. LayerNormalization fails under twenty-percent tuning, performing worse than other mitigations, and partially worse than the baseline. We also measured several properties of these agents, to give more insight beyond the performance analysis. These properties include stable rank [20], dormant [43] or inactive neurons [1], [11], [27], and weight norms [35]. These results are given in Chapter 7.

6.3 k-percent-tuning for SAC with Mitigations

Figure 6.3 shows the performance of SAC with different mitigations under one-percent tuning in the switching Quadruped-walk-run environment. Most

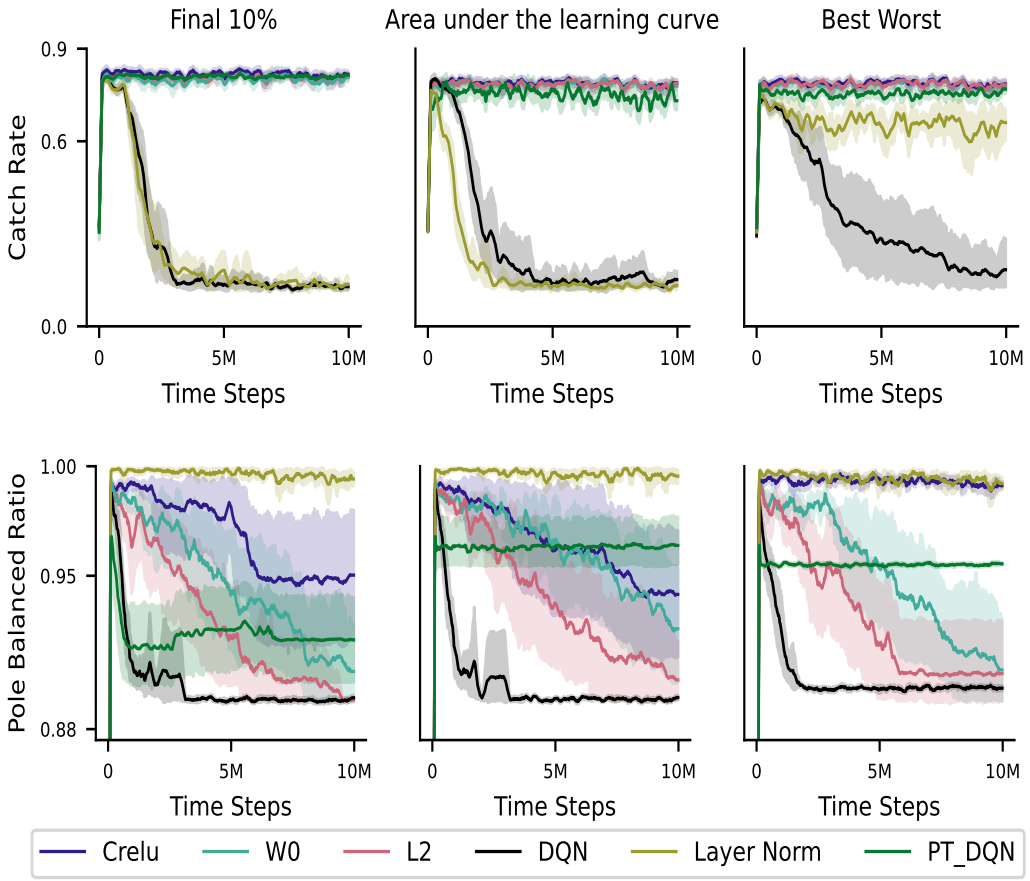

Figure 6.1: The effect of incorporating mitigations into DQN under one-percent tuning in Non-stationary Catch and Continuing Cart-pole. Each of the plots shows a different approach for choosing the hyper-parameters during one-percent tuning. Results are averaged over ten seeds and shaded regions reflect the 95% bootstrap confidence intervals.

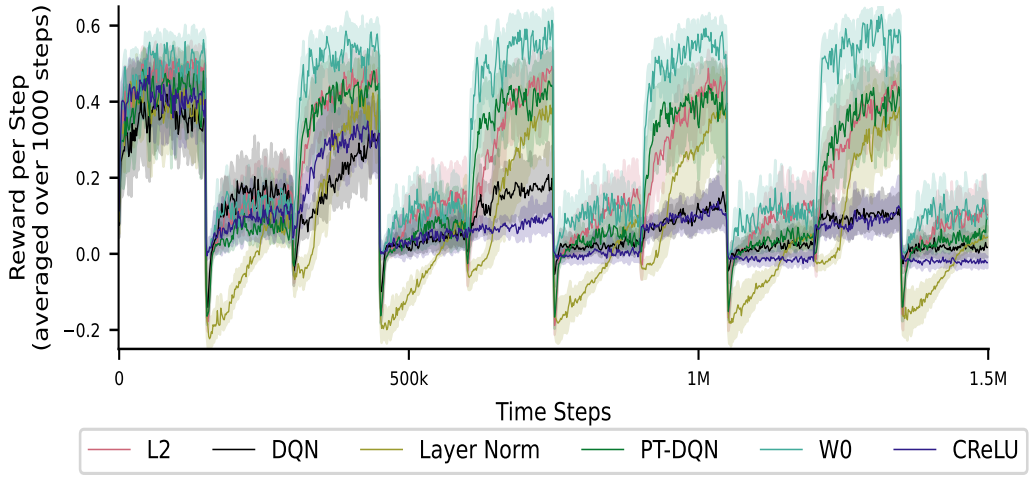

Figure 6.2: The effect of incorporating mitigations into DQN under twenty-percent tuning in Jelly Bean World. Results are averaged over ten seeds and shaded regions reflect the 95% bootstrap confidence intervals.

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. Of note, the learning rate chosen by one-percent tuning in quadruped-walk-run is $1 \cdot 10^{-3}$ which is higher than the default value of $3 \cdot 10^{-4}$. As normalization has been shown to allow for the use of larger learning rates [7], [39], that may be why weight normalization leads to effective mitigation for Quadruped-walk-run. Although L2Regularization has previously been shown to increase the effective learning rate [22], it does not appear to be sufficient here.

6.4 Impact of k on Mitigations

In chapter 5, we analyzed the effect of different values of k in k-percent evaluation of DQN. In this section, we further analyze how k can affect the performance for the mitigations in$k$-percent evaluation. In Figures 6.4, 6.5, 6.6, 6.7,

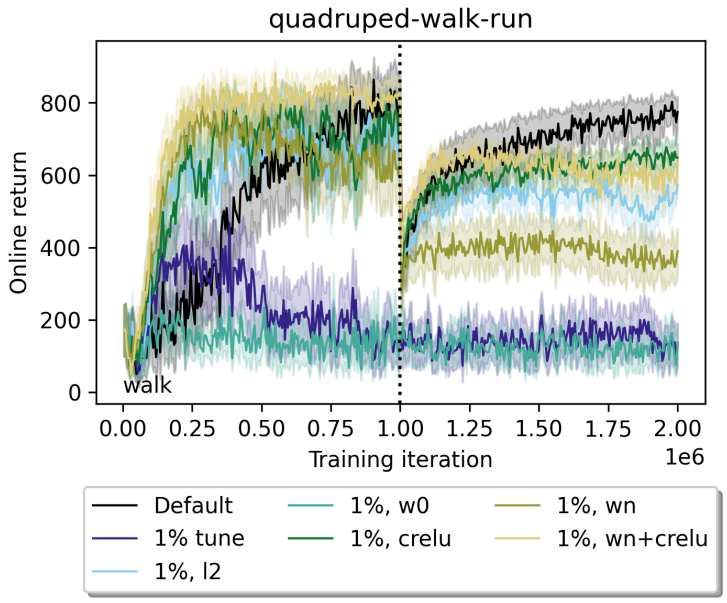

Figure 6.3: Multiple mitigation strategies do improve the performance of quadruped-walk-to-run with the sub-optimal hyperparameters obtained from tuning on one-percent of quadruped-walk. $l2$ is weight decay $= 1 \cdot 10^{-5}$, $w0$ is with penalization of weights moving away from their initialization values, and wn is weight normalization. The results are averaged over 10 seeds, and the shading is the standard error.

and 6.8 you can find the effect of k on performance in mitigations in Continuing Cartpole. Layer norm and PT-DQN are more robust to the value of k, consistently having the same performance with exposure to different percentages of data. On the other hand, l2Regularization and W0Regularization have more inconsistent performances. CReLU does poorly for smaller k values but eventually reaches a good performance for larger k.

!

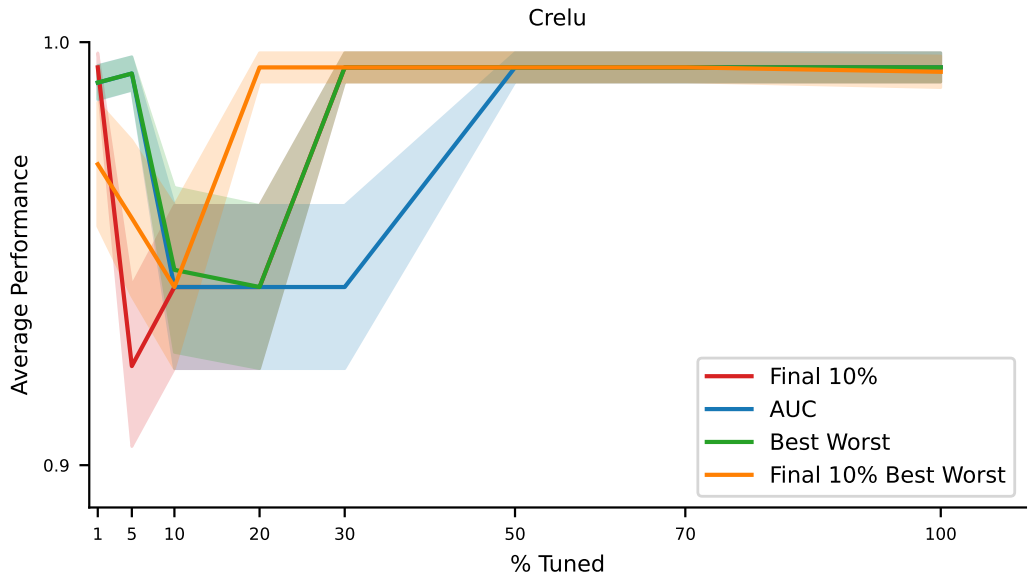

Figure 6.4: Effect of k on performance of Crelu in Continuing Cartpole, over its entire lifetime. Results are averaged over 30 seeds with shaded regions being 95% student-T confidence intervals.

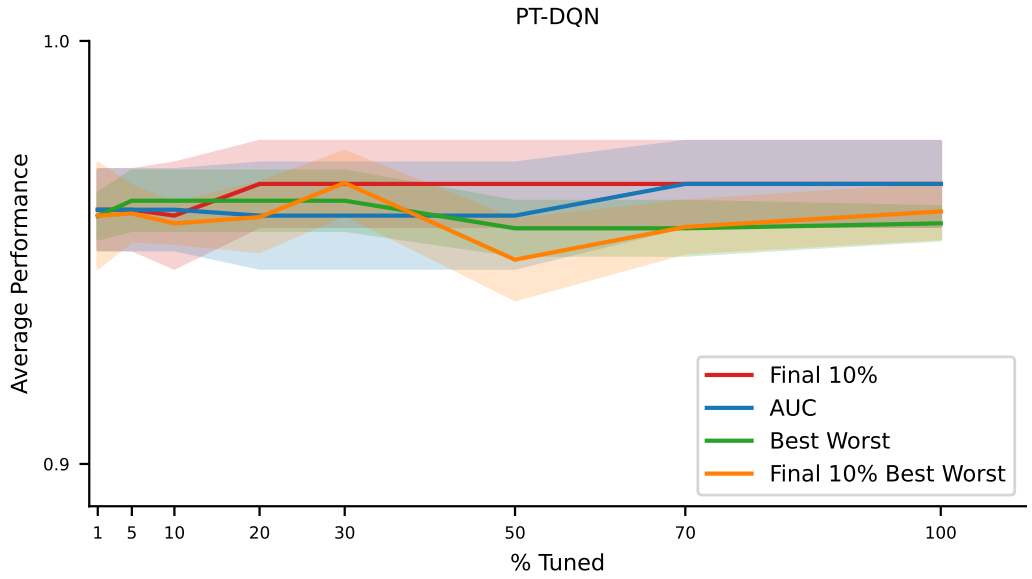

Figure 6.5: Effect of k on performance of PT DQN in Continuing Cartpole, over its entire lifetime. Results are averaged over 30 seeds with shaded regions being 95% student-T confidence intervals.

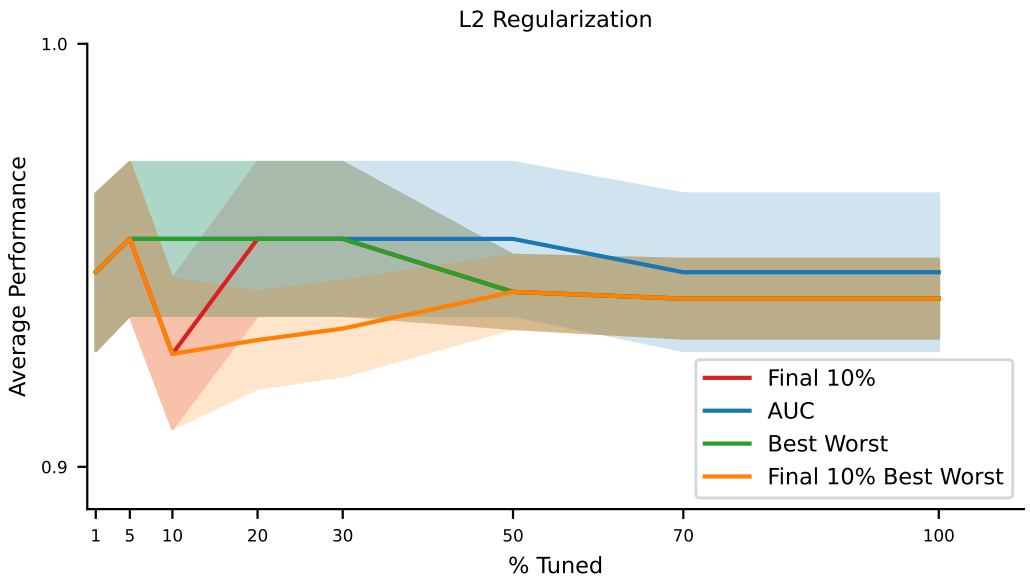

Figure 6.6: Effect of k on performance of L2 Regularization in Continuing Cartpole, over its entire lifetime. Results are averaged over 30 seeds with shaded regions being 95% student-T confidence intervals.

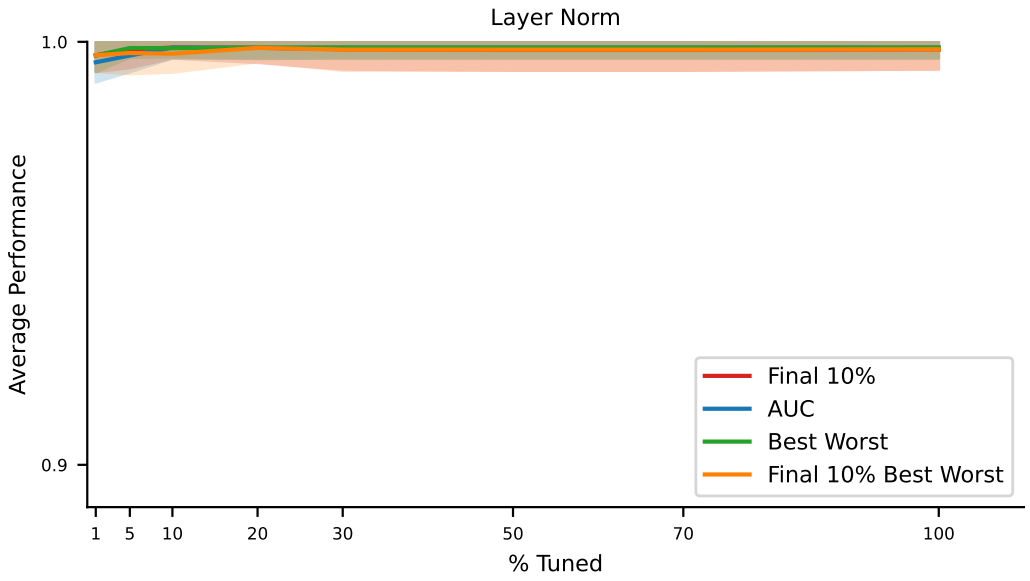

Figure 6.7: Effect of k on performance of Layer Norm in Continuing Cartpole, over its entire lifetime. Results are averaged over 30 seeds with shaded regions being 95% student-T confidence intervals.

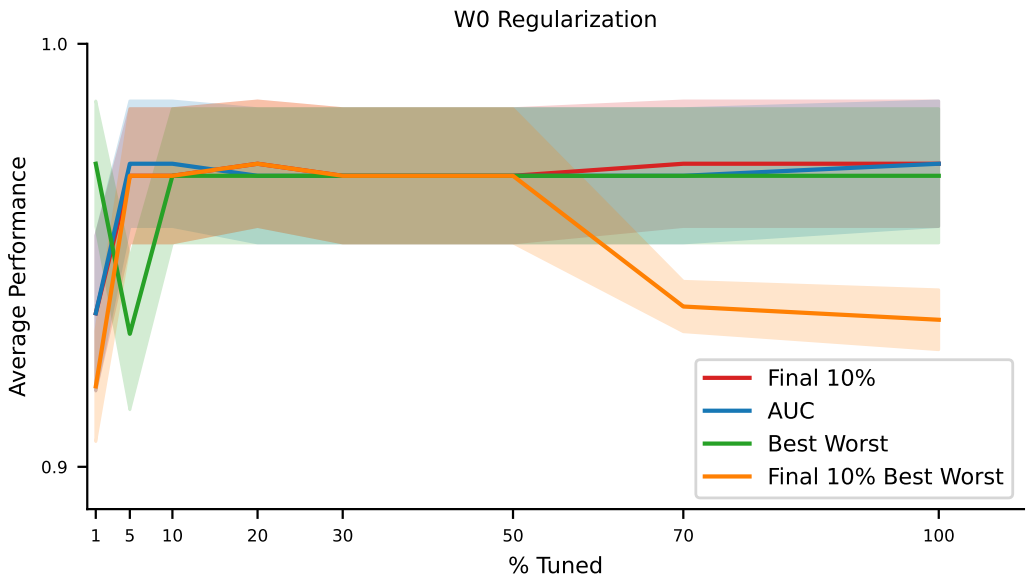

Figure 6.8: Effect of k on performance of W0 Regularization in Continuing Cartpole, over its entire lifetime. Results are averaged over 30 seeds with shaded regions being 95% student-T confidence intervals.

6.4.1 Jelly Bean World

To further analyze the effect of k in another environment, we focus on the performance plots under different values of k for Jelly Bean World. The plots illustrate the varying behaviors of DQN and its mitigations when exposed to different percentages of data. Figures 6.9, 6.10, 6.11, and 6.2 show the performance for k values of 1, 5, 10, and 20, respectively. CReLU starts with poor performance under one-percent tuning but improves with more data exposure. On the other hand, Layernorm begins with fairly good performance but deteriorates in twenty-percent tuning. Other agents exhibit mixed patterns.

As shown in Jelly Bean World for LayerNorm, and also in Figures 6.4 and 6.8, performance can sometimes deteriorate with larger k values. In Continuing Cartpole, CReLU performs better with $k = 5$ compared to $k = 10$ when using the best worst selection method. Additionally, W0Regularization exhibits a decline in performance at larger k values in the Cartpole environment, particularly when the hyperparameter selection method is based on the final 10 percent of best worst performances.

Upon closer analysis, it becomes clear that the performance differences between agents with different k values are exceedingly close in the tuning phase, which can lead to suboptimal selections. Although the agent selected during tuning shows higher performance within the tuning window, it does not necessarily maintain that performance outside of it. For example, the CReLU agent tuned with $k = 10$ achieves an average performance of 0.9942 within the 10% tuning window, compared to 0.9926 for the $k = 5$ agent, a difference of just 0.0016. However, when evaluated for 5 percent of the lifetime, the $k = 5$ agent scores 0.9921, outperforming the $k = 10$ agent, which scores 0.9911. This indicates that while the performance of the agents with $k = 5$ and $k = 10$ is very close during their respective tuning phases, they show noticeable differences in average full-length performance.

This suggests that small differences in tuning phase performance can significantly affect the selection process. To address this, we hypothesize that increasing the number of seeds before tuning could enhance statistical robustness, or that using a more reliable method for hyperparameter selection might yield more consistent results. In future work, it would be beneficial to explore hyperparameter selection methods that take into account both the variance between different configurations and their average performances, to ensure more robust and generalizable results.

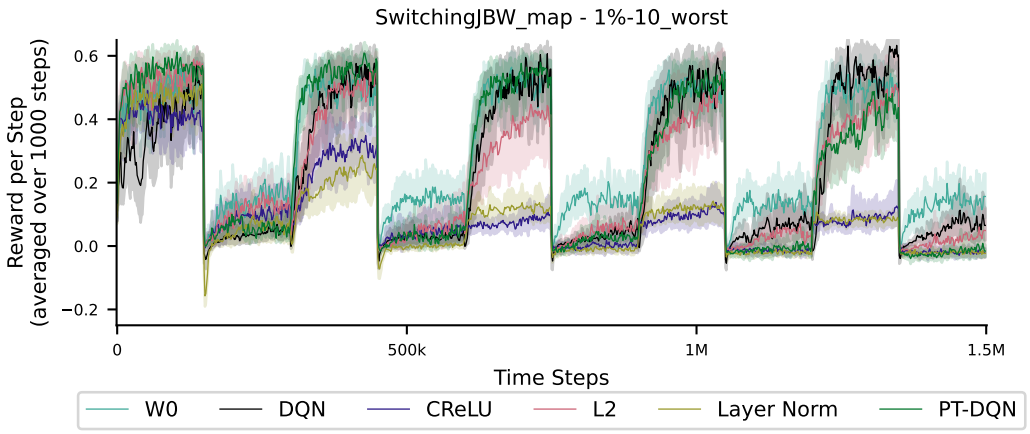

Figure 6.9: DQN and mitigations under one-percent tuning in Jelly Bean World. Results are averaged over ten seeds and shaded regions reflect the 95% bootstrap confidence intervals.

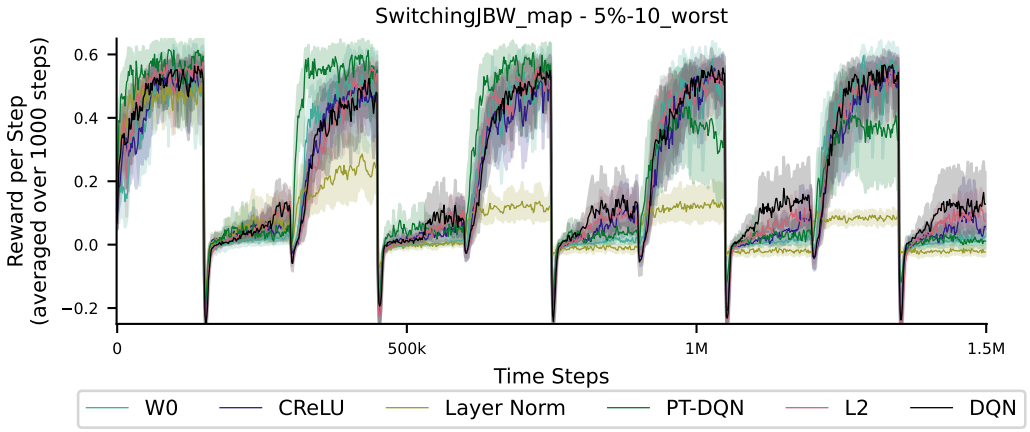

Figure 6.10: DQN and mitigations under five-percent tuning in Jelly Bean World. Results are averaged over ten seeds and shaded regions reflect the 95% bootstrap confidence intervals.

6.5 Summary

In this chapter, we showed that the performance collapse in the k-percent evaluation setting is improved significantly by using mitigation techniques. However, different tuning strategies and environmental factors determine how beneficial they can be. Moreover, by evaluating the effect of different values of k on the performance, in DQN in Continuing Cartpole, and Jelly Bean World, we compared the behavior and robustness of different mitigations under various

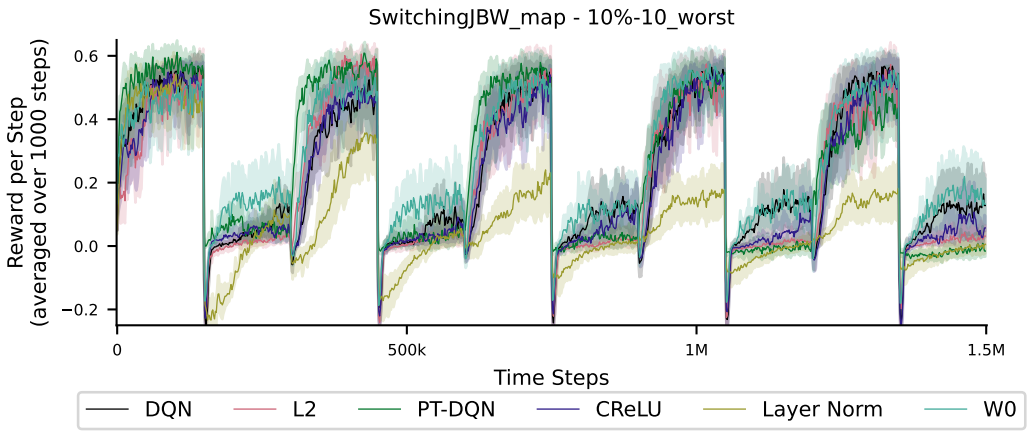

Figure 6.11: DQN and mitigations under ten-percent tuning in Jelly Bean World. Results are averaged over ten seeds and shaded regions reflect the 95% bootstrap confidence intervals.

conditions. In particular, mitigation methods that are more robust under k-percent evaluation are more desirable.

Chapter 7

Revisiting Network Properties

In this chapter, we measure the properties of the k-percent-tuned agents during learning to examine if they correlate with performance. We investigate six properties and measure them for DQN in the three environments. We measure these properties in the Q-network, rather than the target network. For PT-DQN agents, we both measure the properties of the transient network and the permanent network.

7.1 Properties

Percentage of Inactive Neurons [1]: A hidden unit with an output of zero is an inactive neuron. A higher percentage of inactive neurons is shown to affect the agents negatively in a continual setting [1]. The percentage of inactive neurons is measured online through the experiments as follows:

$$\text{Percentage of Inactive Neurons} = \frac{\sum_l \sum_i \mathbf{1}(h_{l,i} = 0)}{\sum_l N_l \cdot L} \times 100\%$$

where $h_{l,i}$ is the output of the hidden unit in layer l at position i, $\mathbf{1}(\cdot)$ is the indicator function that equals 1 if the condition is true and 0 otherwise, N_l is the total number of hidden units in layer l, and L is the total number of layers.

Normalized Stable Rank of the Weights [20]: A higher value of stable rank means that the layer's weight matrix carries more information [15]. It is shown that regularization methods that can help keep the weight rank higher are helpful in mitigating the implicit underparametrization phenomenon that

causes the agents to perform poorly in deep RL settings [20]. 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. The stable rank is normalized to be between 0 and 1.

The ℓ_0 Norm of the Gradient: This metric corresponds to the number of non-zero values in the gradients, and is defined as:

$$\|\nabla w\|_0 = \frac{1}{L}\sum_{l=1}^{L}\sum_{i}\mathbf{1}(\nabla w_{l,i} \neq 0)$$

where $\nabla w_{l,i}$ is the gradient of the weight at layer l and position i, $\mathbf{1}(\cdot)$ is the indicator function that equals 1 if the condition is true and 0 otherwise, and L is the total number of layers.

The ℓ_2 Norm of the Gradient: This metric reflects the magnitude of the gradient not just the active elements. It is defined as:

$$\|\nabla w\|_2 = \sqrt{\frac{1}{L}\sum_{l=1}^{L}\sum_{i}(\nabla w_{l,i})^2}$$

where $\nabla w_{l,i}$ is the gradient of the weight at layer l and position i, and L is the total number of layers.

The ℓ_2 Norm of the Weight Matrices: This metric is also averaged across layers. Growing parameter norms have been shown to be problematic in neural networks, causing training instabilities [28]. In this thesis, we specifically consider the Frobenius norm. We measure the Frobenius norm of the weight matrices, defined as:

$$\|w\|_2 = \sqrt{\frac{1}{L}\sum_{l=1}^{L}\sum_{i} w_{l,i}^2}$$

where $w_{l,i}$ is the weight at layer l and position i, and L is the total number of layers.

The Distance From Initialization: Calculated as the l2 norm of the difference between current and initial weights, averaged over layers:

$$\|\Delta w\|_2 = \sqrt{\frac{1}{L}\sum_{l=1}^{L}\sum_{i}(w_{l,i} - w_{l,i}^0)^2}$$

where $w_{l,i}$ is the current weight and $w_{l,i}^0$ is the initial weight at layer l and position i, and L is the total number of layers.

7.2 Observations

We examine the DQN agents with mitigations, and omit DQN under k-percent evaluation which largely fails in the environments. Note that for the percentage of inactive neurons, CReLU always has exactly 50% active neurons by design. All properties are measured during the experiment length, and then averaged over the lifetime, so that they can be summarized in a dot. Figure 7.1, 7.2, and 7.3 show correlations of mitigations in Continuing Cartpole, Non-stationary Catch, and Jelly Bean World. These suggest that the properties are agent-and-environment dependent, with some properties being more meaningfully correlated than others. For instance, there is a negative correlation with the distance from initialization and l2 norm of the weights. There is also mostly a positive correlation with the stable rank, and l0 norm of gradients. Note that the PT-DQN algorithm consists of two networks. In the figures, above are the properties of the permanent network, and below are the properties of the transient network.

In Non-stationary Catch, clear correlations are observed in Figure 7.1. There is a negative correlation with the distance from initialization and the l_2 norm of the weights. Conversely, there is a positive correlation with the l_0 and l_2 norms of the gradients and the stable rank. No clear correlation is seen with the inactive units. Notably, there is significant variability among different variants of the Layernorm method. Each dot in the figure represents

a different way to select the hyperparameters during one-percent tuning of a different seed (4 selection methods times 10 seeds, totaling 40 dots). Most mitigations perform well and form clear groupings in the correlation plots. However, the Layernorm method forms clusters of well-performing and failed groups with different properties. The failed clusters have clearly lower stable ranks, lower l_0 and l_2 norms of the gradients, higher weight magnitudes, and higher distances from initialization.

We can also compare the permanent and transient network properties to understand their learning dynamics better. The permanent network has a higher stable rank, which is expected as it is updated less frequently.

In Continuing Cart-pole in Figure 7.2 the mitigations were less effective, and the correlations are different from Non-stationary Catch in some cases due to this. In particular, Layernorm which was less performant in Nonstationary Catch shows better performance in this environment. Furthermore, for example, there is a positive correlation with the percentage of inactive 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 large gradient norms. There is a similar correlation to stable rank and a negative correlation between the l2 norm of the weights and performance, and also a negative correlation for distance from initialization. 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. The patterns for stable rank comparison in permanent and transient network is the same.

In Jelly Bean World in Figure 7.3, we can see the same pattern of negative correlation for distance from initialization and l2 norm of the weights and positive correlation of stable rank as the two previous environments. The

correlation of inactive units with performance is negative and has also generally a lower value compared to the previous environments. Layernorm again has some clusters of agents that fail in performance, and generally have higher l2 norm of weights, and lower stable ranks. The permanent network has a higher stable rank and lower number of inactive units compared to the transient network.

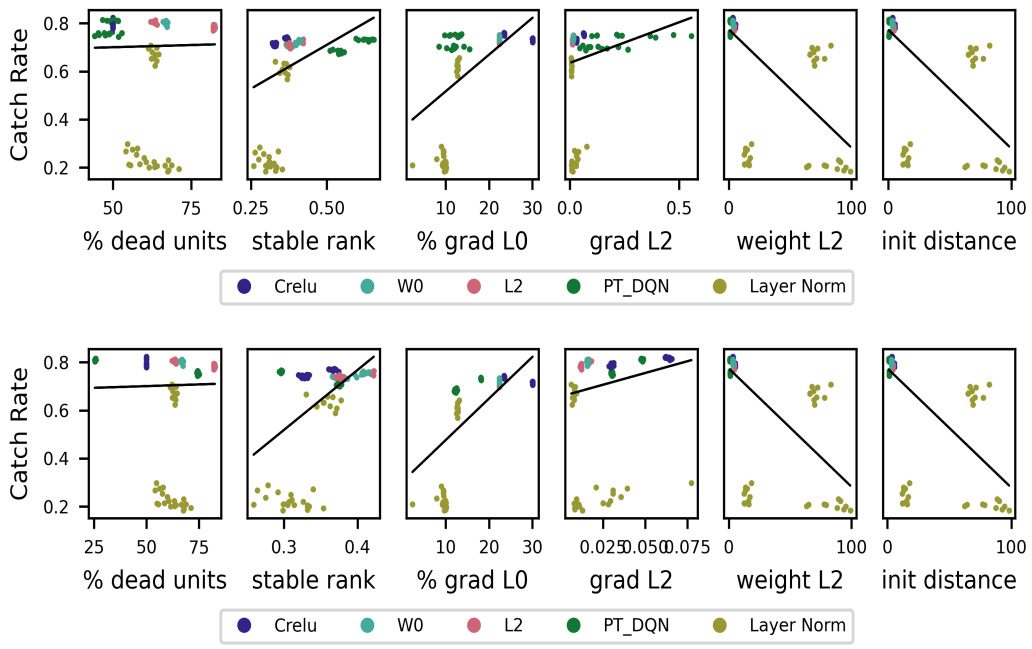

Figure 7.1: The correlations between properties for DQN with mitigations under one-percent tuning and final returns in Non-stationary Catch. The top row is for the properties of the permanent network, and the bottom row is for the properties of the transient network. Each color represents one mitigation combination, and there are 40 dots per color corresponding to the four ways to select hyperparameters during one-percent tuning and the ten seeds used per selected hyperparameter

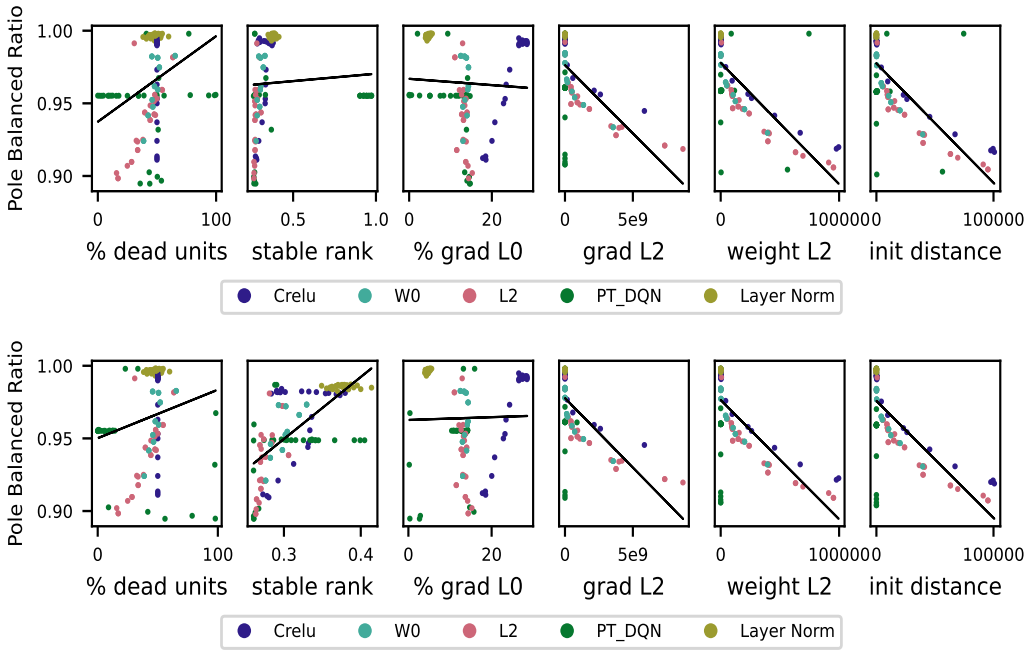

Figure 7.2: The correlations between properties for DQN with mitigations under one-percent tuning and final returns in Continuing Cart-pole. The top row is for the properties of the permanent network, and the bottom row is for the properties of the transient network. Each color represents one mitigation combination, and there are 40 dots per color corresponding to the four ways to select hyperparameters during one-percent tuning and the ten seeds used per selected hyperparameter.

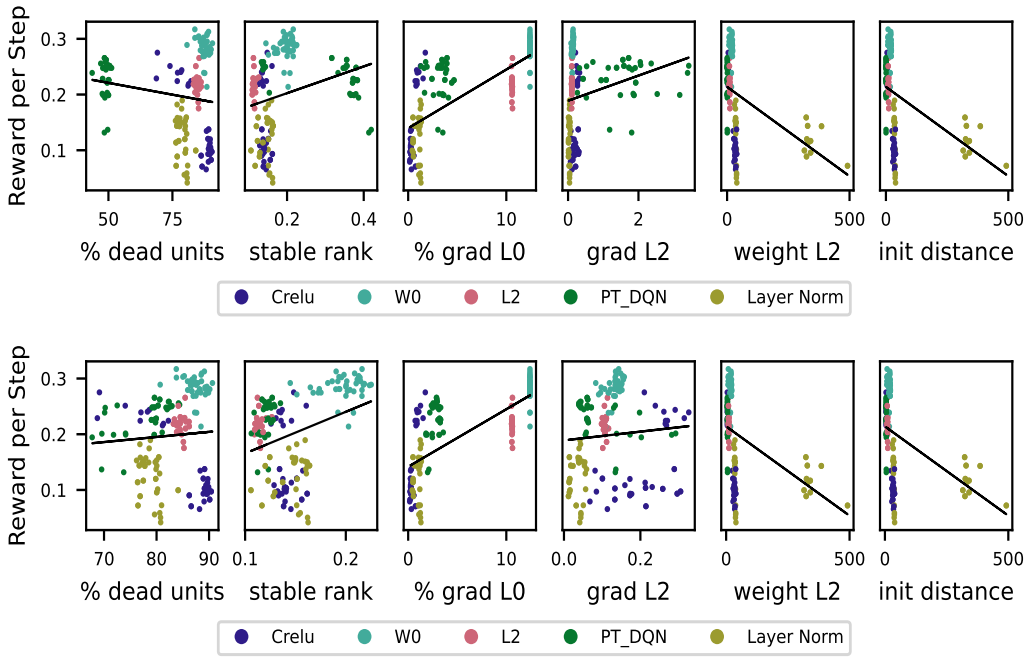

Figure 7.3: The correlations between properties for DQN with mitigations under twenty-percent tuning in Jelly Bean World. The top row is for the properties of the permanent network, and the bottom row is for the properties of the transient network. Each color represents one mitigation combination, and there are 40 dots per color corresponding to the four ways to select hyperparameters during one-percent tuning and the ten seeds used per selected hyperparameter.

7.3 Summary

In this chapter, we revisited the properties of the DQN agents chosen under the k percent evaluation, in three environments. These properties suggest the potential reasons for the performance collapse. By filtering out the non-performant agents using k-percent evaluation, and keeping agents that initially performed well in the tuning phase, we can have a clearer view of the potential causes. It should be noted that the results indicate that the correlation between properties and performance is ultimately agent- and environment-dependent. However, in general, agents that leverage higher stable rank, lower zero gradients, and lower L2 norm of the weights and gradients are more desirable and robust to collapses.

Chapter 8

Conclusion and Future Work

In this thesis, we introduced the k-percent evaluation methodology to better evaluate lifelong reinforcement learning agents. Agents that perform well under k-percent evaluation should be better suited to continual adaption without assuming access to the deployment setting which is neither realistic nor lifelong. Our methodology should allow researchers to better assess existing agents and provide direction for developing new lifelong learning agents.

We showed that agents tuned for the first k-percent of interaction can learn faster than agents 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 k-percent evaluation can be a useful methodology for identifying good and bad continual learning algorithms.

Moreover, by exploring a range of k values, we gained a deeper understanding of algorithm performance and behavior under different constraints. Smaller k values encourage larger learning rates, and smaller epsilon and warmup values, which can initially help the agent to learn faster but will lead to performance degradation in a continual setting. Although initially expensive to try out k values, it leads to a better understanding of our algorithms and advocates for algorithms that eventually need less computation for tuning and online adaptation.

We also found that the separation between good and bad learners given by k-percent tuning also led to more meaningful correlations to properties than reported in previous work.

There are multiple directions for future work on this thesis:

1. Improved Hyperparameter Selection Strategies: Although we tried various hyperparameter selection strategies, there is still room for improvement. We chose hyperparameter configurations based on different metrics, including the best area under the curve, the best performance in the final 10 percent of the tuning phase, the best performance of the worst agent, and a combination of the last two for more robust results. However, these metrics do not account for the variance of performance across multiple seeds of one hyperparameter configuration. Developing a hyperparameter selection strategy that considers performance variance would help in choosing more robust hyperparameters under the k-percent evaluation method.

2. Adaptive and Online Hyperparameter Tuning: k-percent evaluation exposed the vulnerability of current algorithms in a continual setting. It demonstrated that hyperparameters ideal for a fraction of the agent's lifetime are not necessarily good in the long run. For instance, a large learning rate at the beginning of tuning might be beneficial but detrimental to performance as the agent is exposed to more data. Future work could explore better ways to keep hyperparameters updated throughout the agent's lifetime.

k-percent evaluation will force us to develop agents with important properties that the usual RL evaluation scheme does not address. Specifically, it will encourage the creation of agents with minimal hyperparameters that are not sensitive, agents capable of automatically adapting hyperparameters during deployment, and agents that are robust to novel, unknown non-stationarities, meaning they can effectively track changes. Furthermore, this methodology will drive the development of agents that do not merely converge but continue to learn and explore. As a community, we still lack effective lifelong learning algorithms, and the full impact of the k-percent evaluation remains to be seen.

References

[1] Z. Abbas, R. Zhao, J. Modayil, A. White, and M. C. Machado, "Loss of plasticity in continual deep reinforcement learning," *arXiv preprint arXiv:2303.07507v1*, Mar. 2023. 1, 4, 8, 28, 30, 40, 56

[2] D. Abel, A. Barreto, B. Van Roy, D. Precup, H. van Hasselt, and S. Singh, "A definition of continual reinforcement learning," no. arXiv:2307.11046, Dec. 2023, arXiv:2307.11046 [cs]. 2

[3] N. Anand and D. Precup, "Prediction and control in continual reinforcement learning," *arXiv preprint arXiv:2312.11669*, 2023. 6, 20, 26, 28, 56

[4] J. T. Ash and R. P. Adams, "On warm-starting neural network training," *Advances in Neural Information Processing Systems*, vol. 2020-December, Oct. 2019, ISSN: 10495258. 3

[5] J. L. Ba, J. R. Kiros, and G. E. Hinton, *Layer Normalization*, Jul. 2016. 29

[6] A. G. Barto, R. S. Sutton, and C. W. Anderson, "Neuronlike adaptive elements that can solve difficult learning control problems," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC-13, no. 5, pp. 834–846, 1983. 18

[7] N. Bjorck, C. P. Gomes, B. Selman, and K. Q. Weinberger, "Understanding Batch Normalization," in *Advances in Neural Information Processing Systems*, vol. 31, Curran Associates, Inc., 2018. (visited on 02/01/2024). 32

[8] P. D'Oro, M. Schwarzer, E. Nikishin, P.-L. Bacon, M. G. Bellemare, and A. Courville, "Sample-efficient reinforcement learning by breaking the replay ratio barrier," in *Deep Reinforcement Learning Workshop NeurIPS 2022*, 2022. 5, 6

[9] N. Díaz-Rodríguez, V. Lomonaco, D. Filliat, and D. Maltoni, "Don't forget, there is more than forgetting: New metrics for continual learning," *arXiv preprint arXiv:1810.13166*, 2018. 6

[10] S. Dohare, J. F. Hernandez-Garcia, P. Rahman, R. S. Sutton, and A. R. Mahmood, "Loss of plasticity in deep continual learning," *arXiv preprint arXiv:2306.13812v2*, Jun. 2023. 4, 27

[11] S. Dohare, R. S. Sutton, and A. R. Mahmood, "Continual backprop: Stochastic gradient descent with persistent randomness," *arXiv preprint arXiv:2108.06325v3*, Aug. 2021. 3, 5, 6, 8, 27, 30

[12] R. M. French, "Catastrophic forgetting in connectionist networks," *Trends in Cognitive Sciences*, vol. 3, pp. 128–135, 4 Apr. 1999, ISSN: 1364-6613. 3

[13] Google-Deepmind, *GitHub - google-deepmind/csuite*, 2022. 18

[14] T. Haarnoja, A. Zhou, K. Hartikainen, *et al.*, "Soft actor-critic algorithms and applications," *arXiv preprint arXiv:1812.05905*, 2018. 11, 20, 56

[15] M. S. Hosseini, M. Tuli, and K. N. Plataniotis, "Exploiting explainable metrics for augmented sgd," *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, vol. 2022-June, pp. 10 286–10 296, Mar. 2022, ISSN: 10636919. 40

[16] M. Igl, G. Farquhar, J. Luketina, W. Boehmer, and S. Whiteson, "Transient non-stationarity and generalisation in deep reinforcement learning," *arXiv preprint arXiv:2006.05826v4*, Jun. 2020. 3

[17] M. K. Janjua, H. Shah, M. White, E. Miahi, M. C. Machado, and A. White, "Gvfs in the real world: Making predictions online for water treatment," *arXiv preprint arXiv:2312.01624v1*, Dec. 2023. 1

[18] K. Khetarpal, M. Riemer, I. Rish, and D. Precup, "Towards continual reinforcement learning: A review and perspectives," *Journal of Artificial Intelligence Research*, vol. 75, pp. 1401–1476, Dec. 2020, ISSN: 10769757. 3, 6

[19] J. Kirkpatrick, R. Pascanu, N. Rabinowitz, *et al.*, "Overcoming catastrophic forgetting in neural networks," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 114, pp. 3521–3526, 13 Mar. 2017, ISSN: 10916490. 5

[20] A. Kumar, R. Agarwal, D. Ghosh, and S. Levine, "Implicit under-parameterization inhibits data-efficient deep reinforcement learning," *arXiv preprint arXiv:2010.14498v2*, Oct. 2020. 3, 8, 27, 30, 40, 41

[21] S. Kumar, H. Marklund, and B. Van Roy, "Maintaining plasticity via regenerative regularization," *arXiv preprint arXiv:2308.11958*, 2023. 5, 27

[22] T. van Laarhoven, *L2 Regularization versus Batch and Weight Normalization*, Jun. 2017. (visited on 02/01/2024). 27, 32

[23] N. Lazic, T. Lu, C. Boutilier, *et al.*, "Data center cooling using model-predictive control," *Advances in Neural Information Processing Systems*, vol. 31, 2018. 1

[24] T. Lesort, V. Lomonaco, A. Stoian, D. Maltoni, D. Filliat, and N. Díaz-Rodríguez, "Continual learning for robotics: Definition, framework, learning strategies, opportunities and challenges," *arXiv preprint arXiv:1907.00182*, 2019. 5, 6

[25] A. Lewandowski, H. Tanaka, D. Schuurmans, and M. C. Machado, "Directions of curvature as an explanation for loss of plasticity," *arXiv preprint arXiv:2312.00246*, 2024. 4

[26] J. Luo, C. Paduraru, O. Voicu, *et al.*, "Controlling commercial cooling systems using reinforcement learning," *arXiv preprint arXiv:2211.07357*, 2022. 1

[27] C. Lyle, M. Rowland, and W. Dabney, "Understanding and preventing capacity loss in reinforcement learning," in *International Conference on Learning Representations*, 2022. 5, 8, 30

[28] C. Lyle, Z. Zheng, K. Khetarpal, *et al.*, "Disentangling the causes of plasticity loss in neural networks," Feb. 2024. 4, 27, 29, 41

[29] C. Lyle, Z. Zheng, E. Nikishin, B. A. Pires, R. Pascanu, and W. Dabney, "Understanding plasticity in neural networks," *Proceedings of Machine Learning Research*, vol. 202, pp. 23 190–23 211, Mar. 2023, ISSN: 26403498. 4, 6, 14, 29

[30] S. I. Mirzadeh, A. Chaudhry, D. Yin, *et al.*, "Wide neural networks forget less catastrophically," *Proceedings of Machine Learning Research*, vol. 162, pp. 15 699–15 717, Oct. 2021, ISSN: 26403498. 4

[31] S. I. Mirzadeh, A. Chaudhry, D. Yin, *et al.*, "Architecture matters in continual learning," Feb. 2022. 4

[32] S. I. Mirzadeh, M. Farajtabar, R. Pascanu, and H. Ghasemzadeh, "Understanding the role of training regimes in continual learning," *Advances in Neural Information Processing Systems*, vol. 2020-December, Jun. 2020, ISSN: 10495258. 5

[33] V. Mnih, K. Kavukcuoglu, D. Silver, *et al.*, "Human-level control through deep reinforcement learning," *Nature 2015 518:7540*, vol. 518, pp. 529–533, 7540 Feb. 2015, ISSN: 1476-4687. 11

[34] E. Nikishin, J. Oh, G. Ostrovski, *et al.*, "Deep reinforcement learning with plasticity injection," *arXiv preprint arXiv:2305.15555*, 2023. 5, 6

[35] E. Nikishin, M. Schwarzer, P. D'Oro, P.-L. Bacon, and A. Courville, "The primacy bias in deep reinforcement learning," *arXiv preprint arXiv:2205.07802v1*, May 2022. 3–5, 8, 27, 30

[36] R. C. O'Reilly, R. Bhattacharyya, M. D. Howard, and N. Ketz, "Complementary learning systems," *Cognitive science*, vol. 38, pp. 1229–1248, 6 2014, ISSN: 1551-6709. 28

[37] J. Obando-Ceron, J. G. M. Araújo, A. Courville, and P. S. Castro, *On the consistency of hyper-parameter selection in value-based deep reinforcement learning*, 2024. 4

[38] E. A. Platanios, A. Saparov, and T. Mitchell, "Jelly bean world: A testbed for never-ending learning," in *International Conference on Learning Representations*, 2020. 20

[39] T. Salimans and D. P. Kingma, "Weight Normalization: A Simple Reparameterization to Accelerate Training of Deep Neural Networks," in *Advances in Neural Information Processing Systems*, vol. 29, Curran Associates, Inc., 2016. 29, 32

[40] T. Schaul, H. van Hasselt, J. Modayil, *et al.*, "The barbados 2018 list of open issues in continual learning," *arXiv preprint arXiv:1811.07004v1*, Nov. 2018. 3

[41] W. Shang, K. Sohn, D. Almeida, and H. Lee, "Understanding and improving convolutional neural networks via concatenated rectified linear units," *33rd International Conference on Machine Learning*, vol. 5, pp. 3276–3284, Mar. 2016. 28

[42] Y. Shen, J. Wang, and S. Navlakha, "A correspondence between normalization strategies in artificial and biological neural networks," *Neural Computation*, vol. 33, 12 2021. 29

[43] G. Sokar, R. Agarwal, P. S. Castro, and U. Evci, "The dormant neuron phenomenon in deep reinforcement learning," *Proceedings of Machine Learning Research*, vol. 202, pp. 32 145–32 168, Feb. 2023, ISSN: 26403498. 8, 27, 30

[44] Y. Tassa, Y. Doron, A. Muldal, *et al.*, *DeepMind Control Suite*, arXiv:1801.00690 [cs], Jan. 2018. 19

[45] M. White, "Unifying task specification in reinforcement learning," in *International Conference on Machine Learning*, PMLR, 2017, pp. 3742–3750. 10

[46] S. Zaidi, T. Berariu, H. Kim, *et al.*, "When does re-initialization work?," no. arxiv.2206.10011, 2023. 5

Appendix A

Tuning Details

A.1 DQN tuning

For tuning the DQN agent, we sweep over the hyperparameters mentioned in table A.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 A.2, and the lifelong tuned agent's hyperparameters are shown in table A.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 A.1: Default hyperparameters values for DQN on dancing catch

For the Switching-JellyBeanWorld experiments we first sweep over a range of hyper-parameters for 20% of the total experiment length (300k steps) for 5 seeds. We then select hyperparameters that achieve best worst final 10% performance among seeds and run the full length experiment (1.5M steps)

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

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

for 10 seeds. We sweep over learning rate $\alpha \in \{10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}\}$, exploration factor $\epsilon \in \{0.1, 0.01\}$, buffer size $\{1000, 8000, 100000\}$, and adam optimizer's secondary parameter $\beta 2 \in \{0.9, 0.999\}$. In addition for agents that perform regularization (W0, L2) we sweep over the regularization param-

	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 A.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 $100,000$

eter $\lambda \in \{0.0001, 0.001, 0.01\}$. We fix the batch size to 64 and target refresh rate to 200.

For PT DQN hyperparameter sweeps, adding to ϵ, $\beta 2$, and buffer size mentioned above, are both transient net step sizes $\{10^{-5}, 10^{-4}, 10^{-3}\}$ and permanent net step sizes $\{10^{-4}, 10^{-3}, 10^{-2}\}$. Finally we sweep over the following range of time steps between decaying the transient network's weights $\{1000, 10000, 50000, 150000\}$. The batch size is 256 and target refresh rate is 128.

We use the same network architecture as Anand and Precup [3] by employing a 3 layer neural network with ReLU activation of sizes 512, 256, 128 respectively. For the PT DQN agent we halve the size of all layers to compensate for having two networks. For the Crelu agent we use a Crelu activation [1] in the final layer.

A.2 SAC tuning

The architecture as well as the default hyperparameter values are as previously described for the DeepMind Control Suite [14], and we use orthogonal initialization. We use 3 random seeds for tuning SAC agents. The hyperparameter tuning ranges can be seen in Table A.6, and the default hyperparameters and the tuning results in A.7.

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

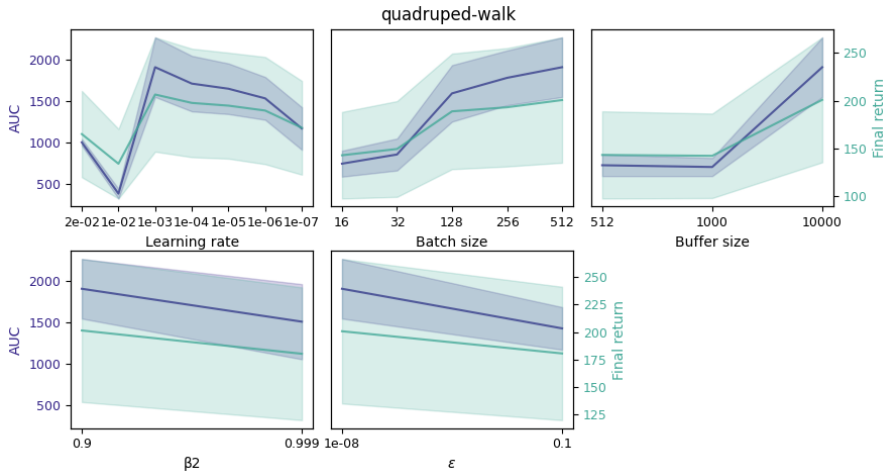

Figure A.1: Hyperparameter values for one-percent tuning of SAC on quadruped-walk. There are three seeds per point. The shading is the standard deviation.

	default	quadruped-walk
Learning rate	$3 \cdot 10^{-4}$	$1 \cdot 10^{-3}$
Batch size	256	512
Buffer size	$1,000,000$	$10,000$
Adam optimizer $\beta 2$	0.999	0.9
Adam optimizer ϵ	$1 \cdot 10^{-8}$	$1 \cdot 10^{-8}$

Table A.7: Default hyperparameter values and values selected from 1% tuning for SAC for the DeepMind Control Suite environment in this paper. Tuning was done with three seeds. The values were the same for selection via AUC as for final 10% return