How Reinforcement Learning Systems Fail and What to do About It

Pouya Hamadanian[1], Malte Schwarzkopf[2], Siddartha Sen[3], Mohammad Alizadeh[1]
[1]MIT CSAIL [2]Brown University [3]Microsoft Research
{pouyah,alizadeh}@csail.mit.edu,malte@cs.brown.edu,sidsen@microsoft.com

Abstract

Recent research has turned to Reinforcement Learning (RL) to solve challenging decision problems, as an alternative to hand-tuned heuristics. RL can learn good policies without the need for modeling the environment's dynamics. Despite this promise, RL remains an impractical solution for many real-world systems problems. A particularly challenging case occurs when the environment changes over time, i.e. it exhibits *non-stationarity*. In this work, we characterize the challenges introduced by non-stationarity and develop a framework for addressing them to train RL agents in live systems. Such agents must explore and learn new environments, without hurting the system's performance, and remember them over time. To this end, our framework (1) identifies different environments encountered by the live system, (2) explores and trains a separate expert policy for each environment, and (3) employs safeguards to protect the system's performance. We apply our framework to straggler mitigation, and evaluate it against a variety of alternative approaches using real-world. We show that each component of our framework is necessary to cope with non-stationarity.

CCS Concepts: • **Networks** → *Network algorithms*; • **Computing methodologies** → **Reinforcement learning**.

Keywords: Reinforcement Learning, Non-stationary, System Optimization, Load Balancing, Machine Learning for Systems, Multiple experts, Safety, Context detection

1 Introduction

— The only constant is change. (Heraclitus 500 B.C.)

Deep Reinforcement Learning (RL) has been proposed as a powerful solution to complex decision making problems [37, 49]. In systems, it has recently been applied to a wide variety of tasks, such as adaptive video streaming [31], congestion control [24], query optimization [28, 34], scheduling [33], resource management [30], device placement [17], and others. Reinforcement learning is particularly well-suited to these problems due to the abundance of data and its ability to automatically learn a good policy in the face of complex dynamics and objectives.

Fundamentally, reinforcement learning trains an agent by giving it feedback for decisions it makes while interacting with an environment. This interaction can occur in a controlled environment, such as a simulation or testbed, or in a real environment, such as a live deployment. While using a controlled environment seems like an attractive choice—e.g., it is data-efficient and less invasive—policies trained in this manner do not fare well in the real world [54]. This is not surprising, because creating a controlled environment for a complex, evolving system can be as large an undertaking as building the system itself [7, 16], making this approach prone to modeling mismatches that bias the final policy [16].

A mismatch can occur when the live system encounters an environment that is previously unseen in the controlled setting or in prior data. This is common in real-world systems, which are time-varying, *non-stationary* environments subject to considerable changes: for example, shifting read/write patterns in databases, fluctuating bandwidth in video streaming, resizes or migrations in cloud resources, churn in competing flows in datacenter networks, and so on[1]. To circumvent this mismatch, therefore, it is appealing to carry out reinforcement learning *in-situ*, i.e., by interacting with the live system. However, training an agent on a live system introduces several challenges.

First, an agent interacting with a new environment will inevitably incur suboptimal performance, as it has not trained on the environment before. This could endanger the system and lead to catastrophic performance loss. Second, adapting to new environments requires continual reinforcement learning, which is non-trivial [26]—simply retraining the agent on new data is not sufficient. Reinforcement learning follows a two stage training procedure: during *exploration*, different actions are evaluated to learn their associated rewards, and during *exploitation*, this evaluation is used to make optimal decisions. If the environment is constantly changing, then an RL agent must also continually explore. But the agent is only beneficial in exploitation. Furthermore, training in a new environment may lead to forgetting older ones, also known as *catastrophic forgetting* [35]. Left unattended, this would necessitate retraining the agent on every change in the system, even on environments that have been observed and trained on numerous times before.

An ideal reinforcement learning agent would explore exactly as much as necessary, and exploit thereafter, until the environment shifts to a new dynamic. It would retain all of its knowledge forever, and never need to retrain on the same environment twice. We aim to mimic this ideal agent as closely

[1]There is another form of non-stationarity in multi-agent RL settings, where training one agent changes the environment from the perspective of other agents. However, in our problem, training can not affect non-stationarity.

as possible. Concretely we aim to minimize online exploration and transient and/or disastrous performance loss, all while maximizing long-term performance. This is a highly restrictive ideal, but it imbues the flexible online learning process with properties (safety, robustness, efficiency) that make it competitive against traditional approaches.

We propose a framework to realize this goal (§5). Our framework includes an environment detector that identifies new environments, and triggers exploration when necessary. To retain knowledge about all environments, the framework trains an ensemble of expert policies, each tailored to a specific environment. To protect the system during exploration, the framework uses a safety monitor to check for a given unsafe condition, and reverts to a default policy (that is known to be safe) when this condition arises. These techniques have appeared individually in prior work (§2). However, our goal is to synthesize a complete framework for online reinforcement learning and understand how well it performs in practical system optimization problems.

We evaluate our framework on a challenging system problem, using real-world data: straggler mitigation in job scheduling (§6). We find that our framework captures and avoids notable failure modes, resulting in a robust decision learning paradigm. We believe a successful application of reinforcement learning in the wild must address the challenges discussed in this paper.

2 Related Work

Non-stationarity in RL: Non-stationarity has been explored in RL in various contexts with varying assumptions, but a general solution has not been proposed [26]. One class of methods train meta models prior to deployment, and use few-shot learning to adapt after deployment [3, 38, 39]. However, such methods require access to the environments before deployment, which we do not assume. Some methods have been proposed for detecting environment changes in discrete state spaces [11], piece-wise stationary environments with Gaussian transition dynamics [4], assigning responsibility signals through a fixed number of learned models [14] and identifying separate environments over time by assuming a stationary latent Markovian context and learning a model for the generalized MDP [42]. For a comprehensive review of non-stationary RL, refer to [26].

Safety in RL: Safe RL takes on many definitions [18]. Our focus is on avoiding disastrous performance outcomes in a live system while exploring or exploiting. [32] use a safeguard policy when safety conditions are violated, while imposing little to no bias on the agent's training. [44] use several forecast signals and revert to a default policy whenever the agent is prone to mistakes. [12] use logged data to learn when a set of constraints can be violated, and disallow the agent from taking such actions in deployment. Several approaches attempt to keep safety by assuming regularity [1] or smoothness [8] in the system. Another model-based RL approach uses active learning and Gaussian Processes for exploration and avoiding safety violations [10]. For a full review of safety in RL, refer to [18]. Using logged interaction data from a deployed policy, one could bootstrap a safe policy for deployment [18, 52]. However, even assuming logged data provides full coverage over the state space, such methods are prone to distributional shifts caused by train/test mismatch in environments [29].

Catastrophic forgetting (CF) in RL: Learned models are prone to catastrophically forgetting their previous knowledge when training sequentially on new information [35, 40]. Recently, interest has piqued concerning CF in RL problems. Three general approaches exist for mitigation [40]: (1) regularizing model parameters so that sequential training does not cause memory loss [25, 27]; (2) selectively training parameters for each task and expanding when necessary [46]; (3) using experience replay or rehearsal mechanisms to refresh previously attained knowledge [5, 23, 43]; or combinations of these techniques [48]. For a full review of these approaches, refer to [40].

3 Preliminaries

Markov Decision Process (MDP): An RL problem consists of an *environment*, which is a dynamic control system modeled as an MDP [50], and an *agent*, which is the entity affecting the environment through a sequence of decisions. The agent observes the environment's *state*, and decides on an *action* that is suitable to take in that state. The environment responds to the action with a *reward* and then transitions to a new state. Formally, at time step t the environment has state $s_t \in \mathcal{S}$, where $\mathcal{S}$ is the space of possible states. The agent takes action $a_t \in \mathcal{A}$ from the possible space of actions $\mathcal{A}$, and receives feedback in the form of a scalar reward $r_t(s_t,a_t) : \mathcal{S}\times\mathcal{A} \rightarrow \mathbb{R}$. The environment's state and the agent's action determine the next state, s_{t+1}, according to a transition kernel, $T(s_{t+1}|s_t,a_t)$ Finally, d_0 defines the initial state (s_0) distribution. An MDP is defined by the tuple $\mathcal{M} = (\mathcal{S},\mathcal{A},T,d_0,r)$[2]. The goal of the agent is to optimize the *return*, i.e. a discounted sum of rewards $R_0 = \sum_{t=0}^{\infty}\gamma^t r_t$.

RL algorithm: In this paper we focus on two model-free RL algorithms [50]. Model-free algorithms are the most common approach in RL-based systems, since they avoid the need to model the environment's dynamics, which is difficult (and often infeasible) in real-world systems [30]. Here we focus on two classes of model-free RL algorithms: *on-policy* and *off-policy* algorithms.

To train an RL agent with an on-policy method, we start by deploying an initial policy to interact with the system. Each interaction creates a sample *experience* (a tuple comprising of a state, action, reward, and next state), and once enough samples are collected, the agent is trained for a single step. Importantly, the collected samples are discarded after each policy

[2] In many problems, the agent cannot observes the full state s_t of the environment and observes a limited observation o_t instead. These problems can be modeled as Partially Observable Markov Decision Process (POMDP)s.

update, which is the main limitation of on-policy methods: a policy can only be trained on samples created by the same policy. Operationally, such methods require a substantial amount of interaction with the live system for training, and they are inherently prone to forgetting past behaviors because they discard old experience data. However, they are known to be less prone to unstable training, suboptimal results, and hyperparameter sensitivity than off-policy methods [15, 19, 20], which can train on stale interactions as well. In this paper, we consider Advantage Actor Critic (A2C), a prominent on-policy algorithm [36] based on policy gradients.

Unlike on-policy methods, off-policy approaches can use samples from a policy different from the one being trained. Thus, we can use historical data to train, despite this data coming from a different policy (e.g., an earlier policy used during training). Off-policy methods maintain a record of previous interactions (also called an experience replay buffer) and use them for training. The most popular off-policy deep RL method is the Deep Q Network (DQN) algorithm [37]; we consider a variant of it in this paper. We refer the reader to the technical report [21] for a brief explanation of these approaches.

4 Challenges of Non-stationary RL

This section explores the key challenges that arise when using RL to train an agent in a time-varying, or non-stationary, environment. As a motivating example, we consider the problem of mitigating stragglers in an online service by "hedging" requests—i.e., replicating a request when a response doesn't arrive within a timeout, covered in depth in §6. For the sake of this section, the important fact is that the system is subject to time-varying *workloads*, as illustrated in Figure 1a. These workloads determine how the agent's decisions affect the system: e.g., a certain timeout threshold for request hedging may be beneficial for one workload, but lead to congestion in another. The active workload in each period of time is indicated at the bottom of Figure 1a. The curves in the figure show the tail of job latencies over time (note the logarithmic scale), with the objective of minimizing this latency.

Offline training is insufficient: Suppose we obtain a faithful simulator for the system, use it to train a policy, and deploy the policy in the live system. We can compare such a pre-trained offline policy to a hypothetical "oracle" RL policy, which knows the workload that will appear in each interval, and uses a policy trained specifically for that workload ahead of time. Figure 1a demonstrates that the pre-trained offline agent performs significantly worse than this oracle: 188% higher tail latency in one workload and 38% in another. This is due to differences between the training and deployment environments [9, 47]; in this example, pre-trained offline was trained on a different workload (*OneStore*, see the technical report [21] for details). Although one can try to create representative training datasets, it is difficult, if not impossible, to anticipate and capture every behavior that can occur in a live system [54].

Online RL requires safeguards: In principle, one can avoid these issues by training an RL agent online in the live system, adapting it on the go. But this introduces other problems. As discussed earlier, RL training involves two phases: exploration and exploitation. In exploration, the agent aims to test and evaluate a wide range of actions, including both good and bad actions. It then exploits what it has learned to select favorable actions.

Online exploration in a live system can degrade performance, and possibly drive the system to disastrous states. This is shown by the "online training without safeguards" curve in Figure 1a. Without a safety mechanism to keep things in check, the tail latency can shoot up and even grow without bound. Recent work has proposed using safeguards to mitigate the damage of online exploration and the instabilities that can occur during RL training [32]. A simple safeguard in Figure 1a is to disable hedging when tail latency exceeds a certain threshold (see §6 for details). Using this method, "online training with safeguards" attains stable exploration: the tail latency remains bounded throughout.

Learning new behaviors without forgetting the past: To learn new behaviors when the environment (workload) changes, we need to explore again. However, perpetually exploring on every change is not desirable, because even with safeguards, exploration incurs a performance cost. Ideally, we should explore sparingly, i.e., once for each new workload. In Figure 1a, "online training with safeguards" tracks environment changes and initiates an exploration phase that lasts for one T_c (period of convergence, see Figure 1a) the first time it encounters a new workload. Accordingly, the tail latency initially degrades when a new workload begins, and then improves as the agent shifts from exploration to exploitation.

Figure 1a shows another challenge for online RL, however. Ideally, when workload B appears a second time, the agent should be able to immediately exploit its past knowledge. But the results show that "online training with safeguards" fails to remember what it had learnt. This is rooted in the fact that neural network-based policies forget the past when learning sequentially [5, 35, 40]. Specifically, when a neural network is updated based on experiences derived from one workload for a long time, it tends to overfit to the data distribution of that workload, and forgets behaviors learned from earlier data. This problem is called catastrophic forgetting (CF).

5 Framework Overview

Successful deployment of RL in non-stationary systems is challenging, as evident in §4. In this section, we outline a framework, visualized in Figure 1b, that serves as a blueprint for training RL agents in a live system. Our framework consists of three key modules: a safety monitor, a default policy, and an environment detector, all defined by the system designer. The *safety monitor* checks for violation of a safety condition as the

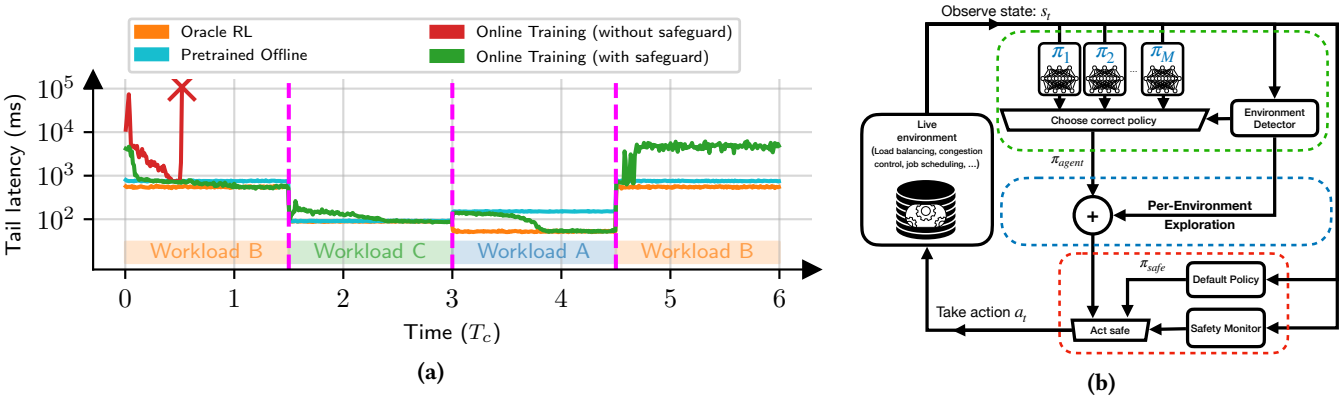

Figure 1. (a) An example demonstrating the challenges of learning a straggler mitigation policy in an environment with time-varying workloads. Curves denote the 95$^{\text{th}}$ percentile job latency over 5 minutes windows. T_C denotes the convergence time when training on a stationary workload. Active workloads are marked and colored below the curves and vertical dashed lines denote workload changes. **(b)** Framework overview for non-stationary RL. Green components mitigate catastrophic forgetting, Blue components concern per-environment exploration, and Red components deal with safety.

agent interacts with the environment. The *default policy* is an existing scheme that, when activated in an unsafe regime, can return the system to safe operation. The *environment detector* aims to detect which environment is active at any given time. It typically uses a small set of features (selected by the system designer) to detect changes in the environment. For instance, to detect changes in the workload, we might use features such as job arrival rates and job types. We now discuss how our framework addresses the three challenges described in §4.

Safety: Safety is handled in a straightforward way in our framework. As shown in Figure 1b, the safety monitor controls whether the agent or the default policy decides the next action. When in an unsafe state, the default policy is activated, which drives the system to a safe region before control is given back to the RL agent to resume training. This limits the performance impact of exploration or other misbehaviors by the RL agent (e.g., transient policy problems during training).

Per-Environment Exploration: As each environment has its own dynamics, knowledge attained by exploring in one environment may not generalize to others. Thus, for each observed environment, we initiate a one-time exploration. The choice to explore can be driven by the environment detection signal, as shown in Figure 1b. We believe that in many systems, lifelong performance improvements justify the cost of per-environment exploration (with safeguards). As the agent collects more experience, we expect it to encounter new environments less frequently, and thus mostly exploit past knowledge. We considered an alternative strawman approach that employed a small level of exploration all the time. However, prior work [2] and our initial experiments showed that this leads to poor policies and is difficult to tune appropriately.

Catastrophic forgetting: Forgetting past knowledge stems from training a single policy with recent data that is devoid of old experiences. We adopt a simple solution that avoids this problem altogether by training different policies for each environment, also referred to as multiple "experts" [45]. We select the appropriate expert to use and train in each environment using the environment detector. Our results suggest that this approach is quite effective, even when the environment detector isn't perfect and makes mistakes. We also investigate an alternate approach that uses off-policy RL to train a single model based on an experience replay buffer. While this approach can also mitigate CF (since the buffer allows the agent to replay past experiences), we found it to be less robust than the multi-expert approach.

6 Case Study: Straggler Mitigation

We consider a simulated request proxy with hedging (Figure 2). In this environment, a proxy receives and forwards requests to one of $n = 10$ servers. The servers process requests in their queues one by one. To load balance, the proxy sends the request to the server with the shortest queue. To respond to a request, the server launches a job that requires a nominal processing time, which we will refer to as its size. The size of a job is not known prior to it being processed. Henceforth we will use the terms "request" and "job" interchangeably.

In a real system, some jobs may incur a slow down and take longer than the nominal time, e.g., due to unmet dependencies, IO failure, periodic events such as garbage collection [13], noisy neighbours [41], etc. These slowdowns also affect other jobs further in the queue. The effect is especially pronounced at the tail of job latency, which is the time between the job's arrival and its completion. To simulate such behaviors in our environment, we inflate the processing time of a job relative to the nominal value by a factor of k with a small probability p. We use $k = 10$ and $p = 0.1$ in our experiments.

To mitigate the effect of slowdowns, if a job has not completed by a timeout, the proxy "hedges" it by sending a duplicate request to another server. This duplication only happens

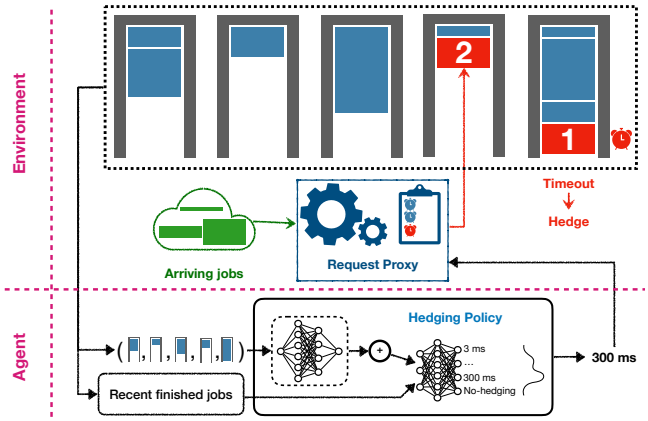

Figure 2. Illustration of a request proxy with hedging.

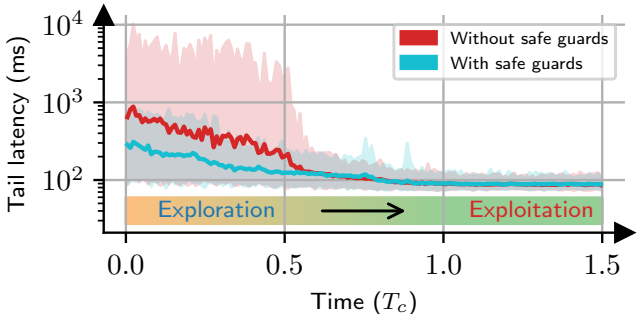

Figure 3. Tail latency vs. time in workload C, when a safe guard reduced performance loss in exploration, and when not. Shades denote min-to-max tail latency in 3 random seeds.

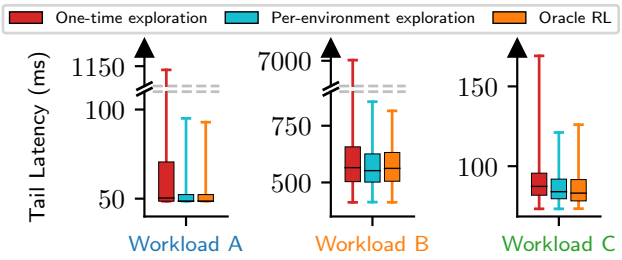

Figure 4. Tail latency distribution after convergence; Each box represents one workload. Whiskers show 99 and 1 percentiles. Upper, middle and lower edge of boxes show 75, 50, and 25 percentiles. Each workload is given enough time to converge ($T_{sw} = 2.25T_c$).

once per job. The job completes when either request (the original or its duplicate) finishes. If the hedged request finishes faster than the original one, the job's latency is reduced. Our goal is to minimize the 95$^{\text{th}}$ percentile latency, and our control decision is the hedging timeout. The hedging policy gets to pick among a set of six timeout values, ranging from 3^{ms} to 300^{ms}, or alternatively to do no hedging.

There is an inherent trade off in selecting a hedging timeout. A low timeout leads to more jobs being hedged, possibly reducing their latency, but it also creates more load on the system, resulting in bigger queues. An RL agent can learn the optimal threshold, which depends on the workload (e.g., the job arrival rate and job sizes) and the amount of congestion (e.g., queue sizes) in the system. Since these change over time, this environment is non-stationary.

As nothing is known about jobs prior to processing, individual decisions on a per-job basis do not help. Instead, the agent chooses one hedging timeout for all jobs that arrive within a time window (500 ms in our experiments). It makes these decisions using the following observations: **(a)** instantaneous and time-averaged server queue sizes, **(b)** average and max of job processing times and arrival rate within $m = 4$ time windows, **(c)** average load in the last window, and **(d)** whether a safeguard is active (more details below). The reward is the negated 95$^{\text{th}}$ percentile latency of jobs in that window. The order in which queue sizes (**a**) appear in the observation vector does not matter. We exploit this and reduce the sample complexity of training, with a neural network architecture invariant to permutations. [55]. The technical report [21] has a full description of our training setup, neural networks and environment.

Safeguard: We use a safeguard [32] to improve transient performance and stabilize training. The safeguard overrides the agent when at least one queue builds up past a threshold (50) and disables hedging thereafter. It relinquishes control back to the agent once all queue sizes are below a safe level (3).

Workloads: We use traces from a production web framework cluster at AnonCo, collected from a single day in February 2018. The framework services high-level web requests to different websites and storefront properties and routes them to various backend services (e.g., product catalogs, billing, etc.). The traces are noisy, heavily temporally-correlated, and change considerably over time. See the technical report [21] for more details about these workloads.

For experiments in non-stationary settings, we consider a scenario where several workloads (*Workload A*, *B* and *C* from the technical report [21]) change according to a schedule. Specifically, workloads change in a periodic and cyclic manner, each active for a period of T_{sw} at a time. In all experiments, we run this scenario with different permutations of workloads (e.g., *ABC*, *BCA*, *CBA*, etc.), as the order of workloads can affect RL training schemes. The period between switches, T_{sw} is relative to convergence time T_c of an RL agent trained for only one workload. While not particularly realistic, periodic and abrupt workload changes provide a simple setting to understand the strengths and weakness of different online RL strategies.

Evaluation metric: For evaluation, we calculate the 95$^{\text{th}}$ percentile latency in 5 minute windows. We use this tail latency metric in all experiments, by either plotting it as a timeseries, or by visualizing its distribution with boxplots.

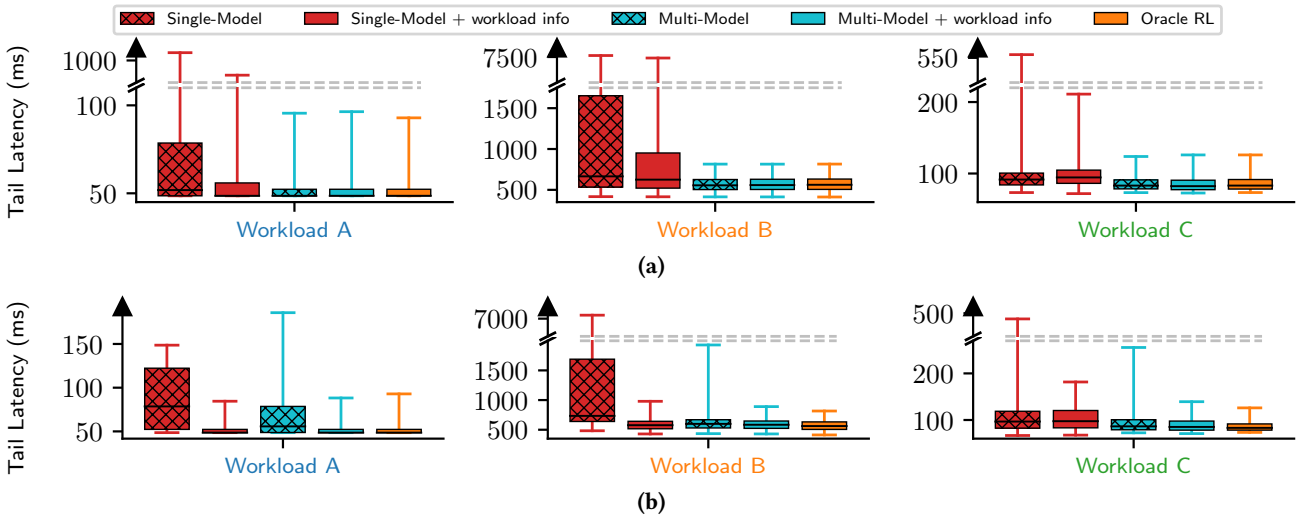

Figure 5. Tail latency distribution after convergence, when either a single model is used or multiple, and when workload statistics are observed or not. **(Top)** $T_{sw} = T_c$ (Slow workload switching), **(Bottom)** $T_{sw} = 0.001\ T_c$ (Fast workload switching).

6.1 Experiments

Safeguards bound performance degradation during exploration: As discussed before, training online in a live system requires online exploration, which leads to transient performance loss. While our main priority is strong eventual performance, bounding transient degradation while exploring is a secondary but important goal. We achieve this with safeguard policies, as explained in §5. Figure 3 demonstrates the effect. Without safeguards, latencies can be an order of magnitude higher than the worst case performance with safeguards.

We observed that occasionally training sessions without safeguards failed to learn at all and queues grew beyond tens of thousands of jobs. Figure 1a showed an example of such a failure mode. This occurs because when queues exceed a certain bound, all jobs will take longer than the maximum hedging timeout (300 ms) and get duplicated. In such a situation, the only way to drain the queues is to disable hedging entirely. But while the agent tries to learn this behavior, the queues continue to grow and the system repeatedly reaches states that the agent has never seen before. Hence, the agent is not able to spend enough time exploring the same region of the state space to learn a stabilizing policy and a vicious cycle forms.

Exploration is critical with new workloads: In §5 we stated that every time the workload changes, we need to explore again. This requires designing an environment detector and switching to the exploration phase whenever a change is observed. In this environment, detecting changes is relatively simple. We use two features: the job arrival rate and the job sizes to characterize the workload (as shown in the technical report [21]). By tracking these features, we can cluster our workloads and train a classifier for them. Of course, the classifier may not be perfect, e.g., the clusters may overlap and not be completely separable. But our experiments show that this simple approach is adequate. It is possible to design more complex features and clustering schemes, but the exact approach to designing an environment detector is not our focus in this work.

Figure 4 demonstrates the effect of exploring once for each workload. Each boxplot shows tail latency distributions for one of three workloads (across different experiments in which we permute the workload order). The boxes show the 25^{th}, median and 75^{th} percentiles of the distributions and whiskers denote 1^{st} and 99^{th}. While one-time exploration (for T_c time) fails to handle new workloads, per-environment exploration converges to a good policy for each workload.

Catastrophic Forgetting: The largest obstacle to online learning is CF. As observed in Figure 1a, sequential training will cause a single model to forget its previous training. We evaluate two techniques to mitigate CF: (1) Providing features as part of the agent's observation that enable it to distinguish between different workloads. By adding these features, the agent can potentially use a single model while learning different behaviors for different workloads. For this scheme, we use the same features used by our workload classifier. (2) Employing multiple experts, each trained and used for a unique workload, as explained in §5. Ideally the expert for workload A will never be used for and trained in workload B, thus never forgetting the policy it learned for workload A. In practice, the environment detector will however make mistakes, but we found that even a 10% error rate will not degrade performance.

In Figure 5, we evaluate multiple experts vs. a single one, and the impact of providing workload features in the observation. We consider two different switching periods between workloads: $T_{sw} = T_c$ (slow switching) and $T_{sw} = 0.001\ T_c$ (fast switching). As a baseline, we also include results for Oracle RL that uses trained policies specific to each workload.

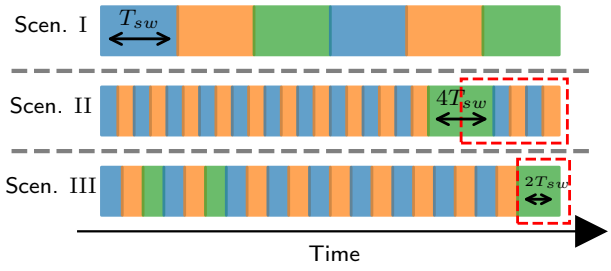

Figure 6. We evaluate three non-stationary scenarios. Scenario I: the system cycles through three different workloads, each active for T_{sw} at a time. Scenario II: two workloads occur periodically for a long time, then a new workload shows up. Scenario III: the system cycles through three workloads, one of which becomes inactive for a long time, then reoccurs.

Figure 5a concerns situations where workload changes occur at long periods. Here, multiple experts significantly outperform a single expert, even when workload information is included in the observations. Workload features don't help in this case since they remain roughly constant throughout the convergence interval of the RL algorithm. On the other hand in Figure 5b where workloads change rapidly, the single model with workload features matches multiple experts. When workloads change at a fast pace, the agent gets to observe samples from all environments (with different workload features) as the RL algorithm converges. Note that in this case the multiple-expert scheme's performance is worse without workload information; this is due to environment classification errors caused by stale information. However, multiple experts with workload information manages to perform well despite these errors since each expert learns to handle observations from non-matching workloads as well. Overall, multiple experts with workload information is robust in all cases.

There are a rich variety of solutions to CF [6, 27, 46]. Such methods might enable sharing knowledge learned from one workload for another. However, multiple experts has the advantage of being robust and simple to design and interpret.

Can we avoid catastrophic forgetting with a single model with off-policy methods? As explained in §3, off-policy methods such as DQN can train on historical samples using an experience replay buffer. Therefore perhaps an off-policy approach can avoid CF even with a single model. Using a single model simplifies the agent and could accelerate learning by enabling shared learning across workloads.[3]

While appealing, off-policy schemes have their own challenges. In particular, their performance is sensitive to the samples saved in the experience buffer, which must be selected carefully to match the mixture of workloads experienced by the agent over time. Further, using a single model (the Q-network in DQN) across different workloads requires reward scaling to ensure some workloads with large rewards do not drown out others.

[3]Note that we still need a workload detection scheme for per-environment exploration.

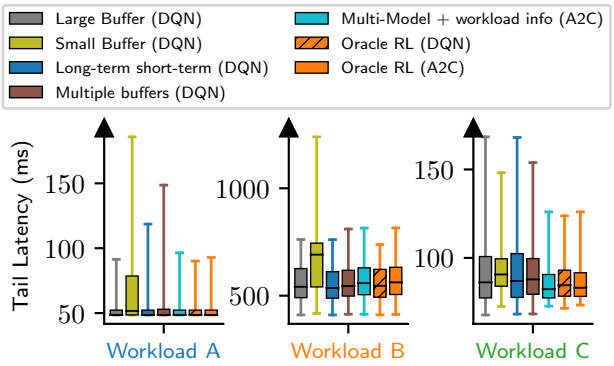

Figure 7. Tail latency distribution after convergence in Scenario I, when $T_{sw} = T_c$ (similar to Figure 5). DQN with a small buffer forgets past workloads.

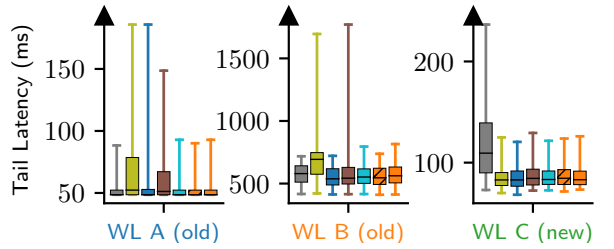

Figure 8. Tail latency distribution after convergence in Scenario II, when $T_{sw} = 0.5T_c$. Left and middle plot concern remembering common workloads after training on a new one. Right plot shows performance on a new workload, after convergence time (T_c). Large and small buffers fail while long-term short-term uses the best of both to fare best.

In the following experiments, we examine DQN with several buffering strategies and compare them to oracle baselines and the on-policy multiple-expert approach: a **(1) Large Buffer** that is akin to saving every sample, a **(2) Small Buffer**, which inherently prioritizes recent samples, **(3) Long-term short-term** that saves experiences in two buffers, one large and one small [23], and samples from them equally during training to combine data from the entire history with recent samples, and **(4) Multiple buffers** that keeps a separate buffer for each workload and samples them equally during training.

Our evaluations use three workload schedules shown in Figure 6. In the on-policy experiments we focused on scenario I where workloads change in a cyclic manner, but here we also consider cases where some workloads might be encountered rarely. Notably, we will find that unlike the on-policy method, the off-policy approach can be sensitive to the schedule.

Figure 7 demonstrates the results for scenario I. Unsurprisingly, a small buffer does not fare well; it loses experience samples of previous workloads and the agent forgets them,

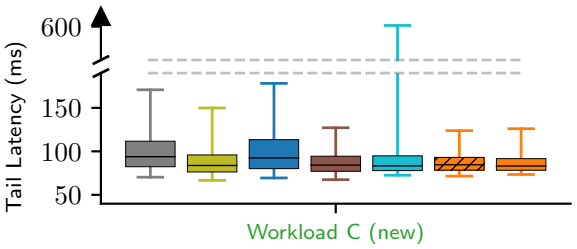

Figure 9. Tail latency distribution after convergence in Scenario III, when $T_{sw} = 0.5T_c$. This plot concerns how well an agent remembers a rarely occurring workload. Long-term short-term is outperformed by multiple buffers.

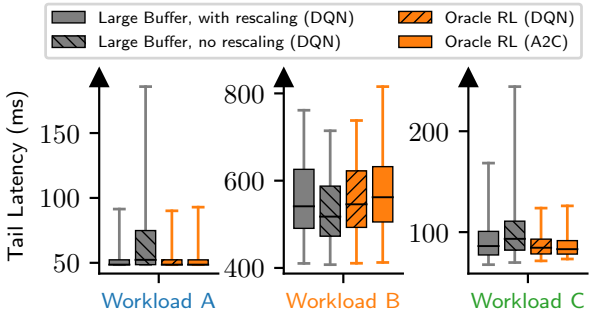

Figure 10. Tail latency distribution after convergence in Scenario I, when $T_{sw} = T_c$ (similar to Figure 5). Without considering rewards scales, DQN favors workloads with high magnitude rewards at the expense of others.

leading to CF. Also, in workload C, the DQN-based oracle fares slightly better than all other DQN methods that train simultaneously on multiple workloads. A similar observation arises in Figure 5, when using a single model with workload info at a fast workload switching setting. We believe this slight loss is due to shared learning, which can sometimes diminish performance instead of improving it when using a single neural network model for diverse tasks [51].

Figure 8 shows the same set of schemes in scenario II, where a rare workload does not come up until well into training. We use workload C as this rare workload and A and B as the common ones. The evaluations in this plot show performance after workload C converges (a region shown by the red box in scenario II in Figure 6). In chronological order, the results for workload C show the large buffer struggling; a large buffer will amass a high volume of samples after a long time and a new workload will have a minuscule share of buffer and training samples. The results for workloads A and B on the other hand exhibit the forgetfulness of a small buffer, similar to the previous experiment. Contrary to both methods, long-term short-term performs well.

Finally, Figure 9 shows results for scenario III, which demonstrates how well an approach can remember a rarely occurring workload (shown by the red box in Figure 6). Here long-term short-term does not cope well; the small buffer forgets workload C and the large one favors more common ones. Multiple buffers performed well here, but it is worse than long-term short-term in scenario II (Figure 8). Overall, these results show that none of these schemes can be a universal strategy that performs well in all circumstances. By contrast, the on-policy multiple-expert approach is more robust.

A further nuance with the off-policy approach, is reward scaling. Since the Q-value network will train on samples from multiple workloads with different magnitudes (latency in one workload can be 10 *ms* while 500 *ms* in another), workloads with larger rewards will be overemphasized in the L2 loss in training. Therefore, the rewards must be normalized but done so consistently. Figure 10 shows the effect; If scaling is not performed, workloads with smaller rewards (latencies), such as workloads A and C are sacrificed for superior performance in workloads with large rewards. Besides manual normalization, methods exist for automatic adaptation to reward scales [22, 53].

7 Conclusion

We investigated the challenges of online RL in non-stationary systems environments, and proposed a framework to address these challenges. Our work shows that an RL agent must be augmented with several components in order to learn robustly while minimally impacting system performance. In particular, we propose an environment detector to control exploration and select an appropriate model for each environment; and we also propose a safety monitor and default policy that protect the system when it enters an unsafe condition. Our evaluation on a straggler mitigation problem shows that applying our framework leads to policies that perform well in each environment and remember what they have learned.

References

[1] Joshua Achiam, David Held, Aviv Tamar, and Pieter Abbeel. 2017. Constrained Policy Optimization. arXiv:1705.10528 [cs.LG]

[2] Zafarali Ahmed, Nicolas Le Roux, Mohammad Norouzi, and Dale Schuurmans. 2019. Understanding the impact of entropy on policy optimization. arXiv:1811.11214 [cs.LG]

[3] Maruan Al-Shedivat, Trapit Bansal, Yuri Burda, Ilya Sutskever, Igor Mordatch, and Pieter Abbeel. 2018. Continuous Adaptation via Meta-Learning in Nonstationary and Competitive Environments. arXiv:1710.03641 [cs.LG]

[4] Lucas N Alegre, Ana LC Bazzan, and Bruno C da Silva. 2021. Minimum-Delay Adaptation in Non-Stationary Reinforcement Learning via Online High-Confidence Change-Point Detection. In *Proceedings of the 20th International Conference on Autonomous Agents and MultiAgent Systems*. 97–105.

[5] Craig Atkinson, Brendan McCane, Lech Szymanski, and Anthony Robins. 2021. Pseudo-rehearsal: Achieving deep reinforcement learning without catastrophic forgetting. *Neurocomputing* 428 (Mar 2021), 291–307. https://doi.org/10.1016/j.neucom.2020.11.050

[6] Craig Atkinson, Brendan McCane, Lech Szymanski, and Anthony Robins. 2021. Pseudo-rehearsal: Achieving deep reinforcement learning without catastrophic forgetting. *Neurocomputing* 428 (2021), 291–307.

[7] Mihovil Bartulovic, Junchen Jiang, Sivaraman Balakrishnan, Vyas Sekar, and Bruno Sinopoli. 2017. Biases in Data-Driven Networking, and What to Do About Them. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*. ACM, 192–198.

[8] Yash Chandak, Scott M. Jordan, Georgios Theocharous, Martha White, and Philip S. Thomas. 2020. Towards Safe Policy Improvement for Non-Stationary MDPs. arXiv:2010.12645 [cs.LG]

[9] Paul Christiano, Zain Shah, Igor Mordatch, Jonas Schneider, Trevor Blackwell, Joshua Tobin, Pieter Abbeel, and Wojciech Zaremba. 2016. Transfer from Simulation to Real World through Learning Deep Inverse Dynamics Model. arXiv:1610.03518 [cs.RO]

[10] Alexander Cowen-Rivers, Daniel Palenicek, Vincent Moens, Mohammed Abdullah, Aivar Sootla, Jun Wang, and Haitham Bou Ammar. 2022. SAMBA: safe model-based active reinforcement learning. *Machine Learning* 111 (01 2022), 1–31. https://doi.org/10.1007/s10994-021-06103-6

[11] Bruno C. da Silva, Eduardo W. Basso, Ana L. C. Bazzan, and Paulo M. Engel. 2006. Dealing with Non-Stationary Environments Using Context Detection. In *Proceedings of the 23rd International Conference on Machine Learning* (Pittsburgh, Pennsylvania, USA) *(ICML '06)*. Association for Computing Machinery, New York, NY, USA, 217–224. https://doi.org/10.1145/1143844.1143872

[12] Gal Dalal, Krishnamurthy Dvijotham, Matej Vecerik, Todd Hester, Cosmin Paduraru, and Yuval Tassa. 2018. Safe exploration in continuous action spaces. *arXiv preprint arXiv:1801.08757* (2018).

[13] Jeffrey Dean and Luiz André Barroso. 2013. The Tail at Scale. *Commun. ACM* 56 (2013), 74–80. http://cacm.acm.org/magazines/2013/2/160173-the-tail-at-scale/fulltext

[14] Kenji Doya, Kazuyuki Samejima, Ken-ichi Katagiri, and Mitsuo Kawato. 2002. Multiple model-based reinforcement learning. *Neural computation* 14, 6 (2002), 1347–1369.

[15] Yan Duan, Xi Chen, Rein Houthooft, John Schulman, and Pieter Abbeel. 2016. Benchmarking deep reinforcement learning for continuous control. In *International conference on machine learning*. PMLR, 1329–1338.

[16] Sally Floyd and Vern Paxson. 2001. Difficulties in simulating the Internet. *IEEE/ACM Transactions on Networking (ToN)* 9, 4 (2001), 392–403.

[17] Yuanxiang Gao, Li Chen, and Baochun Li. 2018. Spotlight: Optimizing Device Placement for Training Deep Neural Networks. In *Proceedings of the 35th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 80)*, Jennifer Dy and Andreas Krause (Eds.). PMLR, 1676–1684. https://proceedings.mlr.press/v80/gao18a.html

[18] Javier Garcıa and Fernando Fernández. 2015. A comprehensive survey on safe reinforcement learning. *Journal of Machine Learning Research* 16, 1 (2015), 1437–1480.

[19] Shixiang Gu, Timothy Lillicrap, Zoubin Ghahramani, Richard E Turner, and Sergey Levine. 2016. Q-prop: Sample-efficient policy gradient with an off-policy critic. *arXiv preprint arXiv:1611.02247* (2016).

[20] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. 2018. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*. PMLR, 1861–1870.

[21] Pouya Hamadanian, Malte Schwarzkopf, Siddartha Sen, and Mohammad Alizadeh. 2022. Reinforcement Learning in Time-Varying Systems: an Empirical Study. arXiv:2201.05560 [cs.LG]

[22] Matteo Hessel, Hubert Soyer, Lasse Espeholt, Wojciech Czarnecki, Simon Schmitt, and Hado van Hasselt. 2018. Multi-task Deep Reinforcement Learning with PopArt. arXiv:1809.04474 [cs.LG]

[23] David Isele and Akansel Cosgun. 2018. Selective Experience Replay for Lifelong Learning. arXiv:1802.10269 [cs.AI]

[24] Nathan Jay, Noga H. Rotman, P. Brighten Godfrey, Michael Schapira, and Aviv Tamar. 2019. Internet Congestion Control via Deep Reinforcement Learning. arXiv:1810.03259 [cs.NI]

[25] Christos Kaplanis, Murray Shanahan, and Claudia Clopath. 2018. Continual Reinforcement Learning with Complex Synapses. arXiv:1802.07239 [cs.AI]

[26] Khimya Khetarpal, Matthew Riemer, Irina Rish, and Doina Precup. 2020. Towards continual reinforcement learning: A review and perspectives. *arXiv preprint arXiv:2012.13490* (2020).

[27] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, et al. 2017. Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences* 114, 13 (2017), 3521–3526.

[28] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph Hellerstein, and Ion Stoica. 2019. Learning to Optimize Join Queries With Deep Reinforcement Learning. arXiv:1808.03196 [cs.DB]

[29] Sergey Levine, Aviral Kumar, George Tucker, and Justin Fu. 2020. Offline Reinforcement Learning: Tutorial, Review, and Perspectives on Open Problems. arXiv:2005.01643 [cs.LG]

[30] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. 2016. Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM workshop on hot topics in networks*. 50–56.

[31] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. 2017. Neural adaptive video streaming with pensieve. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. 197–210.

[32] Hongzi Mao, Malte Schwarzkopf, Hao He, and Mohammad Alizadeh. 2019. Towards safe online reinforcement learning in computer systems.

[33] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. 2019. Learning Scheduling Algorithms for Data Processing Clusters. In *Proceedings of the ACM Special Interest Group on Data Communication* (Beijing, China) *(SIGCOMM '19)*. Association for Computing Machinery, New York, NY, USA, 270–288. https://doi.org/10.1145/3341302.3342080

[34] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A learned query optimizer. *arXiv preprint arXiv:1904.03711* (2019).

[35] Michael McCloskey and Neal J. Cohen. 1989. Catastrophic Interference in Connectionist Networks: The Sequential Learning Problem. *Psychology of Learning and Motivation* 24 (1989), 109–165. https://doi.org/10.1016/S0079-7421(08)60536-8

[36] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. 2016. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*. PMLR, 1928–1937.

[37] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).

[38] Anusha Nagabandi, Ignasi Clavera, Simin Liu, Ronald S. Fearing, Pieter Abbeel, Sergey Levine, and Chelsea Finn. 2019. Learning to Adapt in Dynamic, Real-World Environments Through Meta-Reinforcement Learning. arXiv:1803.11347 [cs.LG]

[39] Anusha Nagabandi, Chelsea Finn, and Sergey Levine. 2019. Deep Online Learning via Meta-Learning: Continual Adaptation for Model-Based RL. arXiv:1812.07671 [cs.LG]

[40] German I. Parisi, Ronald Kemker, Jose L. Part, Christopher Kanan, and Stefan Wermter. 2019. Continual lifelong learning with neural networks: A review. *Neural Networks* 113 (2019), 54–71. https://doi.org/10.1016/j.neunet.2019.01.012

[41] Xing Pu, Ling Liu, Yiduo Mei, Sankaran Sivathanu, Younggyun Koh, and Calton Pu. 2010. Understanding Performance Interference of I/O Workload in Virtualized Cloud Environments. *2010 IEEE 3rd International Conference on Cloud Computing* (2010), 51–58.

[42] Hang Ren, Aivar Sootla, Taher Jafferjee, Junxiao Shen, Jun Wang, and Haitham Bou Ammar. 2022. Reinforcement Learning in Presence of

Discrete Markovian Context Evolution.

[43] David Rolnick, Arun Ahuja, Jonathan Schwarz, Timothy P. Lillicrap, and Greg Wayne. 2019. Experience Replay for Continual Learning. arXiv:1811.11682 [cs.LG]

[44] Noga H Rotman, Michael Schapira, and Aviv Tamar. 2020. Online Safety Assurance for Learning-Augmented Systems. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*. 88–95.

[45] Andrei A. Rusu, Sergio Gomez Colmenarejo, Caglar Gulcehre, Guillaume Desjardins, James Kirkpatrick, Razvan Pascanu, Volodymyr Mnih, Koray Kavukcuoglu, and Raia Hadsell. 2016. Policy Distillation. arXiv:1511.06295 [cs.LG]

[46] Andrei A Rusu, Neil C Rabinowitz, Guillaume Desjardins, Hubert Soyer, James Kirkpatrick, Koray Kavukcuoglu, Razvan Pascanu, and Raia Hadsell. 2016. Progressive neural networks. *arXiv preprint arXiv:1606.04671* (2016).

[47] Andrei A. Rusu, Mel Vecerik, Thomas Rothörl, Nicolas Heess, Razvan Pascanu, and Raia Hadsell. 2018. Sim-to-Real Robot Learning from Pixels with Progressive Nets. arXiv:1610.04286 [cs.RO]

[48] Jonathan Schwarz, Jelena Luketina, Wojciech M. Czarnecki, Agnieszka Grabska-Barwinska, Yee Whye Teh, Razvan Pascanu, and Raia Hadsell. 2018. Progress & Compress: A scalable framework for continual learning. arXiv:1805.06370 [stat.ML]

[49] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. 2018. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science* 362, 6419 (2018), 1140–1144. https://doi.org/10.1126/science.aar6404 arXiv:https://www.science.org/doi/pdf/10.1126/science.aar6404

[50] Richard S. Sutton and Andrew G. Barto. 2018. *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA.

[51] Matthew E. Taylor and Peter Stone. 2011. An Introduction to Intertask Transfer for Reinforcement Learning. *AI Magazine* 32, 1 (Mar. 2011), 15. https://doi.org/10.1609/aimag.v32i1.2329

[52] Philip S Thomas. 2015. *Safe reinforcement learning*. Ph. D. Dissertation. University of Massachusetts Libraries.

[53] Hado van Hasselt, Arthur Guez, Matteo Hessel, Volodymyr Mnih, and David Silver. 2016. Learning values across many orders of magnitude. arXiv:1602.07714 [cs.LG]

[54] Francis Y Yan, Hudson Ayers, Chenzhi Zhu, Sadjad Fouladi, James Hong, Keyi Zhang, Philip Levis, and Keith Winstein. 2020. Learning in situ: a randomized experiment in video streaming. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*. 495–511.

[55] Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Ruslan Salakhutdinov, and Alexander Smola. 2018. Deep Sets. arXiv:1703.06114 [cs.LG]