



Titre: Quality Assurance in Deep Reinforcement Learning Applications
Title:

Auteur: Rached Bouchoucha
Author:

Date: 2023

Type: Mémoire ou thèse / Dissertation or Thesis

Référence: Bouchoucha, R. (2023). Quality Assurance in Deep Reinforcement Learning Applications [Master's thesis, Polytechnique Montréal]. PolyPublie.
Citation: <https://publications.polymtl.ca/55098/>

 **Document en libre accès dans PolyPublie**
Open Access document in PolyPublie

URL de PolyPublie: <https://publications.polymtl.ca/55098/>
PolyPublie URL:

Directeurs de recherche: Foutse Khomh
Advisors:

Programme: Génie informatique
Program:

POLYTECHNIQUE MONTRÉAL

affiliée à l'Université de Montréal

Quality Assurance in Deep Reinforcement Learning Applications

RACHED BOUCHOUCHA

Département de génie informatique et génie logiciel

Mémoire présenté en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
Génie informatique

Août 2023

POLYTECHNIQUE MONTRÉAL

affiliée à l'Université de Montréal

Ce mémoire intitulé :

Quality Assurance in Deep Reinforcement Learning Applications

présenté par **Rached BOUCHOUCHA**

en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*

a été dûment accepté par le jury d'examen constitué de :

Christopher J. PAL, président

Foutse KHOMH, membre et directeur de recherche

Heng LI, membre

DEDICATION

To my family . . . ,

ACKNOWLEDGEMENTS

First, I would like to express my profound gratitude to Pr. Foutse Khomh, who provided me with the opportunity to enter the world of research, for his guidance and patience throughout the Master's program, as well as for all of his feedbacks and support, which led to the realization of this thesis. Second, I want to thank to Dr. Housseem Ben Braiek, whose feedback, knowledge, and research experience have been valuable to me. Thirdly, I'd like to express my gratitude to the lab mates with whom I've shared many moments. I would like especially to recognize Ahmed Haj Yahmed, with whom I have collaborated and who has been a great friend. Also, I'd like to express my appreciation to Dr. Amin Nikanjam for his assistance and for all of his feedback, especially while writing my research papers. I would like also to acknowledge Pr. Christopher J. Pal and Pr. Heng Li for evaluating my thesis. Finally, I would like to thank my mother and all my family for their support throughout both good and difficult periods.

RÉSUMÉ

L'apprentissage par renforcement profond (DRL) a connu un intérêt croissant dans divers domaines, notamment la robotique, les jeux vidéos, et les systèmes de recommandation. Cet intérêt pour l'apprentissage par renforcement profond peut être attribué à ses capacités d'automatisation et son aptitude à traiter des tâches complexes de prise de décision séquentielle, ce qui s'alignent sur la tendance industrielle croissante à l'automatisation, à la personnalisation et à la robotisation. Récemment on a assisté à des réussites notables des applications de DRL, telles que la victoire d'AlphaGO contre le champion du monde du jeu de Go, la victoire d'AlphaStar contre des professionnels humains dans le jeu StarCraft II, et d'autres innovations importantes comme AlphaZero, OpenAI Five et AlphaFold. Ces réussites ont captivé à la fois les professionnels et les chercheurs, les poussant à développer davantage les techniques de DRL et à améliorer leur fiabilité afin d'accélérer leur adoption dans les industries. Par conséquent, la DRL est devenue un concept important dans le domaine de l'apprentissage automatique.

Malgré les nombreux avantages de l'apprentissage par renforcement profond et la pression croissante de l'industrie pour l'adopter, le DRL est toujours confronté à des défis lorsqu'il est déployé dans des environnements de production; soulignant la nécessité d'approches d'assurance qualité fiables pour les DRL. Ces défis découlent de la nature unique de DRL. Tout d'abord, la conception architecturale unique et le comportement d'apprentissage complexe de la DRL la distinguent des solutions d'apprentissage profond traditionnelles. Par conséquent, les approches d'assurance qualité développées pour l'apprentissage profond peuvent ne pas s'appliquer directement à la DRL. Deuxièmement, le réglage des hyperparamètres d'un modèle DRL est essentiel pour obtenir des performances optimales. Cependant, la détermination des valeurs appropriées pour ces paramètres n'est pas une tâche triviale, et de nombreux aspects du réglage des hyperparamètres restent des défis ouverts. Par exemple, trouver le bon équilibre entre l'exploration et l'exploitation ou déterminer le nombre optimal d'itérations pour atteindre l'efficacité de l'échantillon sont des domaines de recherche en cours. Troisièmement, la nature non stationnaire de l'environnement DRL introduit la possibilité d'une baisse significative des performances d'un système DRL pendant son exploitation.

Il devient donc impératif d'automatiser le processus d'assurance qualité et d'établir des outils d'assurance qualité spécifiques aux DRL. L'automatisation de la surveillance et du diagnostic des systèmes DRL joue un rôle essentiel dans la réduction de la charge de travail des développeurs et dans l'atténuation des comportements erronés inattendus avant qu'ils

n’entraînent des conséquences indésirables, améliorant ainsi la fiabilité globale des applications DRL.

Dans ce mémoire, nous proposons des approches pour détecter rapidement les comportements erronés et prévenir les comportements inattendus dans les applications DRL. Tout d’abord, nous nous attaquons au défi de la détection des erreurs et du débogage pendant la phase de développement. Un problème important pendant le développement est que les bogues peuvent se produire sans défaillance explicite, ce qui rend leur détection longue et difficile. A notre connaissance, il n’existe actuellement aucun outil dédié à la détection des erreurs, spécifiquement conçu pour les applications DRL. C’est pourquoi nous proposons une approche de détection automatique des erreurs pour identifier les comportements erronés de manière dynamique pendant l’entraînement en surveillant et en diagnostiquant de multiples traces collectées durant l’entraînement. Deuxièmement, nous abordons la question du changement d’environnement, qui peut entraîner une dégradation des performances par rapport à celles obtenues lors de l’entraînement. L’apprentissage continu (CL) est une approche qui vise à adapter les modèles DRL à ces changements. Cependant, des études récentes ont mis en évidence le fait que les changements survenant dans l’environnement de déploiement entraînent souvent des cycles d’adaptation prolongés ou infructueux dans l’apprentissage continu en raison d’inefficacités telles que l’oubli catastrophique, l’échec du démarrage à chaud et la lenteur de la convergence. Pour surmonter ces difficultés, nous proposons une approche efficace d’auto-guérison pour les systèmes DRL. Notre approche intègre un nouveau mécanisme d’oubli intentionnel dans la méthode classique de l’apprentissage continu, en donnant la priorité au processus d’adaptation sur les neurones les plus importants. Ce mécanisme nous permet de surmonter les principaux problèmes associés au CL et d’améliorer la capacité du système DRL à s’adapter aux changements de l’environnement.

ABSTRACT

Deep reinforcement learning (DRL) has seen growing interest in various fields, including robotics, video games and recommender systems. The recent surge of interest in DRL can be attributed to its capacity for automation and its ability to handle complex sequential decision-making tasks, which align with the growing industrial trend towards automation, personalization, and robotization. Recently, there have been notable successes of DRL applications, such as AlphaGO defeating the world champion in the game of Go, AlphaStar surpassing human grandmasters in playing StarCraft II, and other significant breakthroughs like AlphaZero, OpenAI Five, and AlphaFold. These accomplishments have captivated both practitioners and researchers, driving them to further develop DRL techniques and enhance their trustworthiness to accelerate its adoption in industries. Consequently, DRL has emerged as a prominent concept in the field of machine learning.

Despite the numerous advantages of deep reinforcement learning and its growing industry adoption, DRL still faces challenges when deployed in production environments; highlighting the need for reliable quality assurance approaches for DRL. These challenges stem from the unique nature of DRL. First, the unique architectural design and complex learning behavior of DRL distinguish it from traditional deep learning solutions. Second, tuning the hyperparameters of a DRL model is critical for achieving optimal performance. However, determining appropriate values for these parameters is not a trivial task, and many aspects of hyperparameter tuning remain open challenges. For instance, finding the right balance between exploration and exploitation or determining the optimal number of iterations for achieving sample efficiency are ongoing areas of research. Third, the non-stationary nature of the DRL environment introduces the possibility of drifts, wherein the environment’s parameters exhibit continuous variability. Severe drift can lead to a significant drop in the performance of a DRL system. This performance degradation can be particularly problematic in critical applications where maintaining a high level of performance is essential.

Thus, it becomes imperative to automate the quality assurance process and establish DRL-specific tools for quality assurance. Automating the monitoring and diagnosis of DRL systems plays a pivotal role in reducing the workload on developers and mitigating the occurrence of unexpected erroneous behavior before they lead to undesirable consequences, thereby enhancing the overall reliability of DRL applications.

In this thesis, we propose approaches to detect and prevent erroneous behaviors early in DRL applications. Firstly, we tackle the challenge of fault detection and debugging during

the development phase. A significant concern during training is that bugs can occur without explicit failures, making their detection time-consuming and challenging. To the best of our knowledge, there is currently no dedicated tool available for fault detection specifically developed for DRL applications. Therefore, we propose an automatic fault detection approach to identify erroneous behaviors dynamically during training by monitoring and diagnosing multiple dynamic training traces. Secondly, we address the issue of environment drift, which can lead to performance degradation compared to that achieved in training. Continual learning (CL) is one approach that aims to adapt DRL models to such drifts via fine-tuning. However, recent studies have highlighted that environmental drifts often result in prolonged or unsuccessful healing cycles in CL due to inefficiencies like catastrophic forgetting, warm-starting failure, and slow convergence. To overcome these challenges, we propose an effective self-healing approach for DRL systems. Our approach integrates a new mechanism of intentional forgetting into the classic CL method, prioritizing the adaptation process on the most important neurons. This mechanism enables us to overcome the main issues associated with CL and enhance the DRL system’s ability to adapt to environmental drifts.

TABLE OF CONTENTS

DEDICATION	iii
ACKNOWLEDGEMENTS	iv
RÉSUMÉ	v
ABSTRACT	vii
TABLE OF CONTENTS	ix
LIST OF TABLES	xii
LIST OF FIGURES	xiii
LIST OF SYMBOLS AND ACRONYMS	xiv
CHAPTER 1 INTRODUCTION	1
1.1 The application of Deep Reinforcement Learning	1
1.2 Challenges of Quality Assurance in DRL applications	3
1.3 Limitations of the Existing Quality Assurance Trends in DRL	4
1.4 Research Statement	5
1.5 Thesis Overview	6
1.6 Thesis Contribution	7
1.7 Organisation of the Thesis	8
CHAPTER 2 BACKGROUND	9
2.1 Deep Reinforcement Learning	9
2.1.1 Reinforcement Learning	9
2.1.2 Fundamentals of Deep Learning	11
2.1.3 Deep Reinforcement Learning: The Integration of Deep Learning and Reinforcement Learning	12
2.2 Quality Assurance in Deep Reinforcement Learning	13
2.2.1 Challenges of diagnosing DRL applications	13
2.2.2 Environment Drift and Continual Learning	14
2.3 Chapter Summary	15

CHAPTER 3	LITERATURE REVIEW	16
3.1	Testing in Deep Reinforcement Learning	16
3.2	Studies on Fault Detection in Machine Learning	18
3.2.1	Fault Detection in Deep Learning	18
3.2.2	Fault Detection in Deep Reinforcement Learning	19
3.3	Maintenance of Deep Reinforcement Learning Applications	19
3.3.1	Plasticity problem in Machine Learning	20
3.3.2	Continual Learning in Deep Reinforcement Learning applications . .	20
3.4	Chapter Summary	22
CHAPTER 4	TOWARD DEBUGGING DEEP REINFORCEMENT LEARNING PRO- GRAMS WITH RLEXPLOER	23
4.1	Approach	24
4.1.1	Overview	24
4.1.2	Failure symptoms and root causes	26
4.1.3	Detecting Failure Symptoms	29
4.1.4	Integration of custom fault detection diagnosis	31
4.2	Evaluation	32
4.2.1	Faulty Samples evaluation	32
4.2.2	Human evaluation	34
4.3	Discussion	38
4.3.1	Participants' Feedback	38
4.3.2	Limitations and Future Work	40
4.4	Threats to Validity	41
4.5	Chapter Summary	42
CHAPTER 5	DR.DRL : AN INTENTIONAL FORGETTING-DRIVEN SELF-HEALING METHOD FOR DEEP REINFORCEMENT LEARNING SYSTEMS	43
5.1	Motivating Example	44
5.2	Approach	45
5.2.1	Problem formulation	45
5.2.2	Intentional Forgetting Mechanism	46
5.2.3	Self-healing Workflow	49
5.3	Evaluation	51
5.3.1	Experimental Setup	51
5.3.2	Research Questions and Answers	55
5.4	Threats to Validity	60

5.5 Chapter Summary	61
CHAPTER 6 CONCLUSION	62
6.1 Summary	62
6.2 Discussion and Future Work	64
APPENDICES	66
REFERENCES	67

LIST OF TABLES

Table 4.1	Description of <i>RLExplorer</i> 's Diagnosis Approach for Detecting Fault Symptoms in RL Components (4.1.2)	30
Table 4.2	Results of <i>RLExplorer</i> on Stack Overflow Buggy Samples.	33
Table 4.3	Comparison between <i>RLExplorer</i> and manual debugging in terms of the number of diagnosed bugs and the average bug diagnosis time. . .	37
Table 5.1	Comparison of Healing Time and Fine-Tuning Episodes Decreases Using <i>Dr.DRL</i> over vanilla CL.	55
Table 5.2	Comparison of healed DRL Agent Reward Increases Using Dr.DRL over CL.	59

LIST OF FIGURES

Figure 2.1	Illustration of the agent environment interaction in reinforcement learning	9
Figure 4.1	Illustration the Three Steps of <i>RLExplorers</i> Workflow: Configuration, Execution, and Fault Logging.	25
Figure 4.2	Number of diagnosed faults by each participant using the two debugging approaches.	37
Figure 4.3	5-point Likert Scale Chart illustrating the feedback of participant on <i>RLExplorer</i>	39
Figure 4.4	5-point Likert Scale Chart illustrating the feedback of participants on the relevance <i>RLExplorer</i> 's Checks.	40
Figure 5.1	(A) The adaptation curves for the DQN agent's average reward per episode on MountainCar: (a) depicts the curve of the average reward when Dr. DRL is used, whereas (b) represents the curve of average reward when CL is used (B) Illustration of the MountainCar environment as described in [1]	44
Figure 5.2	Illustration of <i>Dr.DRL</i> Pipeline	49
Figure 5.3	Comparison of Healing Time and Fine-Tuning Episodes Box Plots Using <i>Dr.DRL</i> over vanilla CL.	56
Figure 5.4	Distribution of %AR over the 4 identified groups Per healing Method and RL Environment	57
Figure 5.5	Average reward Increase Ratio Comparison between Dr.DRL and Vanilla CL in Non-Adaptable Environments. The curves illustrate the improvement in predictive performance achieved by Dr.DRL over vanilla CL in jointly non-adaptable environments	59

LIST OF SYMBOLS AND ACRONYMS

AI	Artificial Intelligence
ML	Machine Learning
DL	Deep Learning
DNN	Deep Neural Network
RL	Reinforcement Learning
DRL	Deep Reinforcement Learning
CL	Continual Learning
SO	Stack Overflow
NN	Neural Network
DQN	Deep Q-Network
PPO	Proximal Policy Optimization
SAC	Soft Actor-Critic
A2C	Actor Critic
QA	Quality Assurance

CHAPTER 1 INTRODUCTION

Deep Reinforcement Learning (DRL) has significantly revolutionized the machine learning field. The combination of reinforcement learning and deep learning, allowed DRL to become a prominent approach in various important and critical fields, including robotics, software engineering, and finance. The main intuition behind the learning process of DRL is to acquire the ability to achieve a predefined goal in a complex environment. This objective is achieved using a reward mechanism, which grants rewards or penalties to the agent according to its proximity to the specified objective. This learning process has demonstrated notable performance in tackling complex sequential decision-making tasks that have traditionally caused difficulties for conventional deep learning techniques. In recent years, there has been a significant rise in the development of diverse applications using DRL techniques. However, despite its important achievements, the widespread adoption of DRL continues to lag behind in comparison to conventional deep learning techniques. The main factor contributing to this disparity in industrial adoption can be attributed to the complex architectural design of DRL. This complex architecture present various difficulties in terms of optimizing the monitoring processes and ensuring flexibility during the development and production of DRL-based programs [2–5]. Moreover, there exist numerous concerns regarding the reliability of DRL based software systems. In fact, the adoption of DRL necessitates an increasing focus on reliability challenges, especially for critical systems where even minor errors can result in severe consequences. Consequently, the quality assurance of DRL software systems represents a major obstacle to the widespread adoption of DRL technology. Hence, the studies on quality assurance in DRL are crucial in order to enhance the trustworthiness of DRL systems and encourage their broad adoption in various industries.

In the subsequent sections, we present an introductory overview of DRL, highlighting its notable achievements across various domains. Next, we discuss the challenges related to quality assurance in the context of DRL and delve into the limitations of the current approaches used for quality assurance purposes. Then, we present the research statement of the thesis, followed by an outline of its contents and the thesis contributions.

1.1 The application of Deep Reinforcement Learning

The unique architecture and learning behavior strategy of Deep Reinforcement Learning have enabled significant advancements in various fields, garnering considerable attention in recent years. DRL presents several advantages that position it as a viable alternative and solution

across different use cases.

Gaming, in particular, has proven to be a successful domain for DRL adoption, particularly in developing game-playing bots that imitate human behavior. The interactive nature of the agent-environment dynamics, prevalent in the gaming field, lends itself well to DRL applications. For instance, algorithms like Deep-Q-Network (DQN) have demonstrated exceptional performance in learning to play various Atari games [6]. AlphaGo [7] garnered significant attention when it defeated the world champion, while DeepMind’s AlphaStar [8] excelled in playing StarCraft, a complex and dynamic real-time strategy game, successfully surpassing master players. OpenAI’s OpenAI-Five [9], a DRL system, emerged as an excellent player in Dota2, a strategic multiplayer game.

DRL has also gained prominence in recommender systems, where it leverages consumer behavior to make personalized recommendations based on individual preferences. In commerce, offering a tailored experience enhances the consumers satisfaction, influences their behavior, and significantly impacts the business. Renowned platforms such as Netflix, Spotify, Facebook, YouTube, and various social networks employ DRL for service, information, and product recommendations [10, 11].

In the field of robotics, DRL has become a classical application, aiming to develop robots that exhibit correct behavior in the real world to accomplish specific tasks. Robotics has shown numerous challenges for traditional deep learning strategies that DRL has successfully addressed [12]. Traditional Deep Learning techniques often fall short due to the high dimensionality of the real world and the need for generalization with low uncertainty. By enabling efficient agent-environment interactions, DRL has propelled rapid progress in robotics. Notably, OpenAI’s Dactyl [13] stands as a successful DRL application, effectively controlling a robot hand to manipulate objects, including solving a Rubik’s Cube. Other studies have explored DRL’s effectiveness in controlling humanoid robots, such as [14] and [15], while several investigations have assessed DRL’s performance in tackling diverse robotics-related tasks [16].

The field of software engineering has also reaped the benefits of DRL advancements. DRL can be useful in various stages of software development and life cycle, including designing compilers [17], network planning [18], data augmentation [19], data management [20], testing [21, 22] and maintainance [23].

Moreover, numerous public service domains, such as transportation [24], finance [25, 26], healthcare [27, 28], and education [29, 30], have shown interest in harnessing the power of reinforcement learning. However, the contribution of DRL in these domains remains limited due to several factors, which we will discuss in the following section.

1.2 Challenges of Quality Assurance in DRL applications

Deep Reinforcement Learning holds immense potential as a versatile solution applicable across various fields. However, its development and production still encounter unique challenges. The gap between the theoretical aspects of DRL compared to its practical implementation and reliability in real-world applications is remarkable compared to classic deep learning applications (e.g. supervised and unsupervised learning) that have a more extensive production record.

While the theoretical promise of DRL applications extends to vital and critical domains such as education, transportation, and robotics, numerous studies [2–5] have highlighted the challenges faced in adopting DRL applications, revealing that these domains remain cautious due to the uncertainties surrounding the reliability and trustworthiness of DRL applications. The primary concern lies in effectively managing quality assurance to instill confidence in the practical applicability of DRL for critical use cases. In fact, at present, most successful DRL applications are limited to non-critical domains, such as gaming or recommender systems, where issues related to quality assurance can be tolerated.

The challenges pertaining to quality assurance in DRL stem from various theoretical complexities that render monitoring DRL applications difficult. These challenges encompass several aspects:

(1) DRLs, by employing both deep learning and reinforcement learning techniques, introduces heightened complexities compared to each one of them. Indeed, DRL presents novel challenges that further complicate its control and management. For instance, the exploration-exploitation trade-off dilemma becomes even more intricate in the context of DRL, as achieving the optimal balance between exploration and exploitation is crucial for attaining peak performance. Moreover, ensuring sample efficiency, which governs the number of experiences necessary for the agent to effectively learn and predict its policy, poses a formidable theoretical challenge.

(2) Users of DRL have limited control over the data used for training. Unlike in traditional Deep Learning, where users prepare and preprocess the data, DRL relies on the feedback loop between the agent and environment to control data generation. Moreover, monitoring the architectural design of the DRL application itself proves challenging. Users must manage multiple elements, such as selecting the appropriate DRL type (e.g., off/on policy), determining the optimal exploration-exploitation strategy, designing the suitable DNN for the agent, simulating the environment accurately, balancing exploration and exploitation, deciding the number of steps and episodes, tuning various hyperparameters that govern the

learning process, and appropriately designing memory. Accurately monitoring all these factors is necessary to achieve high-performance DRL applications, yet it remains a non-trivial endeavor.

(3) DRL environments exhibit non-stationarity, due to the dynamic physical and mathematical characteristics of the real-world environments. Adapting the agent’s behavior to these changes requires a high level of DRL system plasticity. Automating the process of adapting the agent’s behavior to environmental changes is crucial to prevent undesirable consequences and enhance practitioners’ confidence in DRL applications.

These challenges are non-trivial to address; resulting in developers investing significant time in diagnosing and fixing issues. Automating the detection and resolution of issues in DRL would improve the development experience of DRL applications and promote the production of high-quality DRL applications.

Another obstacle to quality assurance lies in the lack of specialized studies and tools developed explicitly for DRL, which could enhance the development experience and assist developers in quality assurance and DRL application monitoring. Although numerous tools have been created for traditional deep learning solutions, they have inherent limitations when applied in the DRL context. We will discuss these weaknesses more in the following section.

1.3 Limitations of the Existing Quality Assurance Trends in DRL

Quality assurance has emerged as a significant concern in the field of machine learning, particularly as deep reinforcement learning solutions become more prevalent in industrial settings and involve human interactions and natural resources. However, there are significant limitations and weaknesses in DRL’s current quality assurance practices.

A significant limitation in DRL is the scarcity of studies dedicated to quality assurance. Many specific problems related to DRL are still open issues and require further investigation and development. This knowledge gap not only restricts the options available to developers but also increases the likelihood of bias or incomplete understanding of some unexpected behavior, which adds more complexity to the developer’s decision-making when dealing with these unexpected behaviors.

Moreover, as discussed in Section 1.2, the complex architecture of DRL systems poses a considerable challenge for debugging its erroneous behaviors. To the best of our knowledge, no debugger is available to assist developers, during training, in interpreting complex erroneous behaviors and tracking their root causes interactively. Thus, facilitating the debugging process for DRL applications would greatly improve the overall quality assurance process

and accelerate development, given the considerable time and effort required for manual bug detection and resolution.

Attempting to employ deep learning quality assurance approaches as an alternative is not an efficient solution, also, due to the considerable behavioral and architectural differences between DRL and DL. Consequently, DL quality assurance techniques cannot be automatically adapted to DRL because they often neglect crucial characteristics specific to DRL, warranting a need for designing approaches to address DRL systems specifically.

Additionally, the lack of existing packages for DRL application debugging poses another limitation. Unlike in Deep Learning, where well-established and maintained tools like TFDBG [31], TFLint [32], and PyTorch Lightning’s Sanity Check [33] are available, DRL commonly relies on visualization tools such as GYM Monitor [34]. While these tools can help visualize unexpected behavior, users are left to manually interpret the visualizations when finding the root causes of errors for fixing.

1.4 Research Statement

After an investigation of quality assurance studies in DRL, and an analysis of the challenges inherent in DRL development, the following observations have been made:

- DRL exhibits a complex architecture and learning process, rendering it highly sensitive to design choices and hyperparameters. Consequently, monitoring DRL applications is a difficult task, and detecting bugs and errors poses a significant challenge for its users. Yet, the identification of faults in DRL can be accomplished by examining dynamic traces generated during the training process. This involves comparing the observed behavior of the DRL system against established DRL fundamentals, and best practices.
- The difficulty in implementing DRL hampers its adoption within the industry. This difficulty primarily arises from a lack of tools specifically designed to assist DRL developers in bug detection, particularly silent bugs, as well as identifying vulnerabilities that may impact the quality of DRL applications. Hence, there is a pressing need to develop integrated tools for DRL applications in order to simplify the process of DRL production.
- One weakness of DRL lies in its limited ability to generalize, particularly in adapting to the changes that may occur in the environment post-production. Given the dynamic

nature of environments, it is crucial to prevent a drop in DRL model performance after its deployment, especially in safety-critical systems where errors are intolerable.

- While numerous studies [35–39] have explored continual learning approaches to mitigate performance degradation in DRL applications post-production, it has been demonstrated that DRL exhibits limited plasticity [40] even using this continual learning approach. In fact, the existing approaches, such as fine-tuning and continual learning, can address a multitude of maintenance problems such as performance degradation; however, they may still fail to resolve certain drifted environments [40]. Moreover, current solutions for addressing performance drops are often costly and time-consuming. Hence, there is a need for novel and efficient approaches to enhance the efficacy of current methodologies in adapting Deep Reinforcement Learning agents to environmental changes.

In this thesis, we leverage concepts from software testing and maintenance to address the quality assurance challenges in DRL and improve their reliability and trustworthiness. Our objective is to develop novel techniques and tools that can assist DRL developers in building robust DRL applications by automatically detecting silent faults or overlooked bugs that may arise during the training of a DRL agent. Additionally, we address the critical concept of environmental drift by introducing an approach that enhances the healing process’s adaptability to drifting environments and improves its cost-efficiency through shorter healing episodes.

1.5 Thesis Overview

1. *A dynamic fault detection approach for DRL applications.* We introduce *RLExplorer*, the first fault diagnosis approach designed specifically for DRL-based software systems. *RLExplorer* automates the monitoring and diagnosis of DRL systems in order to alleviate the burden on DRL developers and improve systems’ reliability. *RLExplorer* automatically runs verification routines based on properties of the learning dynamics to detect the occurrence of DRL-specific faults. Our approach monitors training traces and conducts diagnoses that capture the learning dynamics. It then logs the results of these diagnoses as warnings that explain the potential root cause and solutions to the identified faults. The effectiveness of *RLExplorer* is evaluated using realistic faulty DRL samples extracted from Stack Overflow and results show that *RLExplorer* has an accuracy of 83% in diagnosing faulty behavior. We also evaluated *RLExplorer* via interviews with DRL experts and found that participants expressed high overall satis-

faction with *RLExplorer*, with 86% expressing a strong willingness in using the tool in the future.

2. *An automatic self-healing approach for DRL applications.* To improve the plasticity of DRL systems and avoid post-production performance degradation (caused by environmental drift), we introduce *Dr.DRL*. *Dr.DRL* is an efficient self-healing approach for DRL systems that integrates the mechanism of intentional forgetting into vanilla continual learning (CL) in order to enhance its plasticity and minimize the number of episodes required for the agent’s adaptation. *Dr.DRL* deliberately erases the DRL system’s minor behaviors to systematically prioritize the adaptation of the key problem-solving skills. More specifically, the concept of intentional forgetting involves resetting neurons associated with minor behaviors to a low-scaled value. This allows the healing process to prioritize the adaptation of the neurons responsible for the major behaviors and allows dual-speed continual learning. We evaluate our proposed approach on established DRL algorithms, specifically the Deep Q-Network (DQN) [6], Soft Actor-Critic (SAC) [41], and Proximal Policy Optimisation (PPO) [42] algorithms. We compare the performance of *Dr.DRL* against vanilla CL across multiple drifted environments. The comparison covers the healing time and the number of resolved drifted environments for each approach.

1.6 Thesis Contribution

- We propose *RLExplorer*, the first fault diagnosis approach for DRL systems. *RLExplorer* automatically runs diagnosis routines based on learning dynamics traces to detect the occurrence of RL and DNN faults. *RLExplorer* automatically records data and runs checks that cover learning dynamics. It then displays the checks’ results as warnings that combine theoretical concepts and correction strategies. In addition *RLExplorer* supports different types of RL algorithms. We assessed *RLExplorer* on the 11 StackOverflow faulty samples we collected. The result shows that *RLExplorer* can diagnose faults for 83% of cases. Second, we conducted a study with 15 DRL experts to assess *RLExplorer*’s effectiveness and usability. Participants using *RLExplorer* found 3.6 times more faults than manual debugging and found *RLExplorer* easy to use in DRL applications.
- We build a reproducible DRL bug benchmark with 11 faulty programs derived from StackOverflow. This benchmark acts as the ground truth for evaluating our approach and can be used in future research on DRL debugging and repair.

- We propose, *Dr.DRL*, a tool designed to facilitate the adaption of a DRL agent to drifted environments. *Dr.DRL* is the first application of the intentional forgetting concept for the DRL system’s self-healing problem. *Dr.DRL* exploits the intricacies of DNN structure to separate the DNN’s computation units responsible for major and minor behaviors as seen in the interaction with the original environment. It leverages the gradient-based optimization principles to smoothly achieve dual-speed self-healing, in which major behaviors are prioritized over minor behaviors. We conduct the evaluation of *Dr.DRL* using (i) three popular gym environments that we deliberately drift with varying parameters shifts, namely CartPole [43], MountainCar [1], and Acrobot [44]; (ii) three well-established DRL algorithms, namely Deep Q-Learning (DQN) [45], Soft Actor-Critic (SAC) [41], and Proximal Policy Optimization (PPO) [42].

1.7 Organisation of the Thesis

The following chapters of this thesis are organised as follows:

1. Chapter 2: Provides an overview of the preliminary concepts and knowledge required for comprehending the subsequent sections of the thesis. In this chapter, we will introduce the concept of deep reinforcement learning and examine several challenges related to its quality assurance.
2. Chapter 3: Reviews the relevant literature.
3. Chapter 4: Presents *RLExplorer*, an approach developed for the dynamic detection of faults during the training of a DRL application.
4. Chapter 5: Introduces *Dr.DRL*, a self-healing mechanism that employs the concept of intentional forgetting to enable the deployed DRL agent to adapt efficiently to a drifted environment.
5. Chapter 6: We present the conclusion of this thesis and discuss the future work.

CHAPTER 2 BACKGROUND

Chapter Overview In Section 2.1, we provide an introduction to the concept of Deep Reinforcement Learning. In Section 2.2, we report and discuss challenges associated with the debugging of DRL applications and the adaptation of agents to changing environments.

2.1 Deep Reinforcement Learning

Deep Reinforcement Learning has emerged as one of the most active areas of research in machine learning and neural networks. DRL employs a combination of advanced techniques, specifically Reinforcement Learning (RL) and Deep Learning (DL), to capture complex behaviors. In this section, we will start by explaining the concept of reinforcement learning in Section 2.1.1. Next, we proceed to explain the fundamentals of deep learning in Section 2.1.2. Finally, we discuss DRL algorithms, which combine both RL and DL and present various types of DRL in Section 2.1.3.

2.1.1 Reinforcement Learning

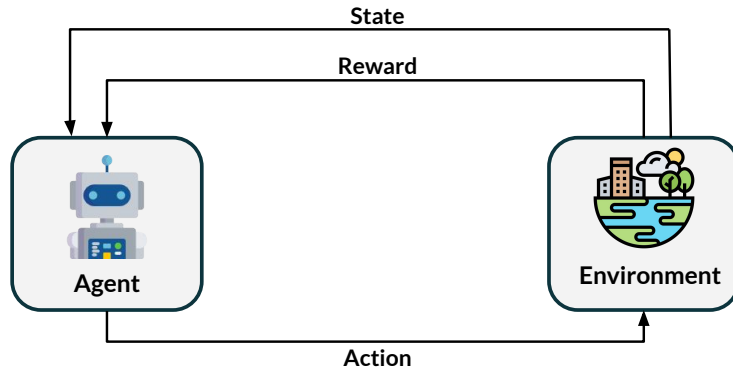


Figure 2.1 Illustration of the agent environment interaction in reinforcement learning

The Agent-Environment Interaction : As depicted in Figure 2.1, the Reinforcement Learning framework [46] consists of two main entities: the agent and the environment.

The agent, serving as the learning entity in RL, assumes the role of decision-maker. It engages with the environment by choosing an action based on its current state. Its goal is to learn how to choose the appropriate sequence of actions in order to reach a specific goal.

The environment represents either a real-world or simulated platform in which the agent is

deployed. It encompasses an interconnected collection of states and rewards. At each time step, the environment receives the agent’s action and provides a new state along with a reward. These outcomes (i.e., the state and reward) reflect the quality of the action and the agent’s proximity to achieving the desired objective.

The Markov Decision Process: Standard reinforcement learning [46] uses the framework of Markov Decision Processes (MDP) [47] to define the sequential decision-making problem. MDPs are commonly employed for modeling dynamic systems, due to their ability to provide a mathematical framework for representing stochastic decision-making processes.

In RL an MDP can be represented as a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma)$ where $\mathcal{S}$ is the set of states $s \in \mathcal{S}$, $\mathcal{A}$ is the set of possible actions $a \in \mathcal{A}$. $\mathcal{T}(s_{t+1}|s_t, a_t)$ represents the transition dynamics that describes the probability distribution over states at time $t + 1$ given a state-action pair at time t , $\mathcal{R} : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ is a reward function, and $\gamma \in [0, 1]$ is a discount factor where lower values denote a preference for more immediate rewards.

The final goal of RL is to acquire the ability to develop a policy that defines a mapping between states and actions, thereby controlling the decision-making behaviour of the agent. More specifically, a policy $\pi(a|s)$ in RL defines a distribution over actions given a state. In general, RL agents try to learn policies that maximize the discounted return at each step:

$$G_t = R_{t+1} + \gamma R_{t+1} + \gamma^2 R_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \quad (2.1)$$

Where G_t is the discounted return and R_t is the immediate reward at timestep t . The expected discounted return given action a is taken at state s and the agent is following policy π which is known as a Q-function:

$$Q_{\pi}(s, a) = \mathbb{E}_{\pi}[G_t | S_t = s, A_t = a]. \quad (2.2)$$

Most RL algorithms try to use function approximators to learn the optimal policy, π_* [48], or q_* , the Q-function [49] of the optimal policy. When using tabular function approximators, learning these functions becomes difficult or even impossible in domains with high dimensional or continuous observation or action spaces.

The Learning Process of Reinforcement Learning : The learning/training process in RL differs from that of traditional deep learning approaches. RL involves a dynamic feedback loop between the agent and the environment, and the data are generated based on the interaction between the agent and the environment. This means that in RL, the developer does not have direct control over the training data, unlike in DL, where developers

preprocess the data and then input it into the model for training.

During RL training, at each time step t , the agent receives a state s_t and produces an action a_t . The environment responds by providing a new state s_{t+1} and a corresponding reward r_{t+1} . This interaction between the agent and the environment forms a sequence of states known as episodes in RL. An episode begins in an initial state and ends when the current state s_t reaches a terminal state. Terminal states can be: reaching a goal, encountering failure, or reaching a predefined maximum number of states.

Other important aspects that play a crucial role in shaping the learning efficiency of RL agents are exploration and exploitation. Exploration involves dedicating episodes to discovering the environment. Its objective is to let the agent gather information about the environment by taking random actions, even if they may not be optimal, to have a better understanding of its environment. Exploitation, on the other hand, involves leveraging the knowledge acquired during exploration and continually fine-tuning the agent's policy in each episode to learn how to behave optimally in the given environment.

2.1.2 Fundamentals of Deep Learning

A deep neural network (DNN) is an artificial neural network with a stack of multiple computational layers, hence the adjective ‘deep’. A DNN encapsulates a mapping function $y = f(x)$ that maps the input x to its associated output y . As formulated below, the mapping function f is a composite function of all the subfunctions $f(f_{l-1}; W_l, b_l)$, where W_l and b_l represent the learnable parameters, respectively, weights and biases.

$$f(x; W, b) = f_L \left(f_{L-1} \left(\dots f_1 \left(x; W^{[1]}, b^{[1]} \right) \right) \dots; W^{[L]}, b^{[L]} \right) \quad (2.3)$$

Where $W = \{W^{[l]}, \forall l \in [1, L]\}$ and $b = \{b^{[l]}, \forall l \in [1, L]\}$.

DNNs learn their parameters using a backpropagation algorithm, which iteratively updates their parameters with the aim to reach an optimal set of parameters, producing a best-fitted DNN with low error rates. Backpropagation algorithm involves the following two alternative main steps, namely, forward pass and backward pass.

Forward Pass. A hidden layer l performs a linear transformation $z^{[l]}$ of their inputs, i.e., the activations computed by its previous layer, then applies a non-linear activation function, g . Thus, the forward pass can be formulated as follows:

$$\begin{aligned} z^{[l]} &= W^{[l]} a^{[l-1]} + b^{[l]} \\ a^{[l]} &= g(z^{[l]}) \end{aligned} \quad (2.4)$$

Backward Pass. As formulated in the following Equations, the backward pass involves

computing the gradients of the loss, $\mathcal{L}$, with respect to the weighted-sum outputs of each layer, $z^{[l]}$, then with respect to the layer parameters, $W^{[l]}$ and $b^{[l]}$. Using the derivative chain rule, the gradients computed for a layer are leveraged to compute gradients in adjacent layers, inversely (from the input layer to the output layer).

$$\frac{\partial \mathcal{L}}{\partial z^{[l]}} = [W^{[l+1]}]^T \cdot \frac{\partial \mathcal{L}}{\partial z^{[l+1]}} * g^{[l]'}(z^{[l]}) \quad (2.5)$$

$$\frac{\partial \mathcal{L}}{\partial W^{[l]}} = \frac{\partial \mathcal{L}}{\partial z^{[l]}} \cdot [a^{[l-1]}]^T \quad ; \quad \frac{\partial \mathcal{L}}{\partial b^{[l]}} = \frac{\partial \mathcal{L}}{\partial z^{[l]}} \quad (2.6)$$

Therefore, the parameters are updated in the opposite direction of their gradients w.r.t the loss $\mathcal{L}$ using a learning rate η , as formulated below.

$$W^{(i+1)} = W^{(i)} - \eta \frac{\partial \mathcal{L}}{\partial W^{[l]}} \quad ; \quad b^{(i+1)} = b^{(i)} - \eta \frac{\partial \mathcal{L}}{\partial b^{[l]}} \quad (2.7)$$

In several empirical studies [50, 51] aimed at demonstrating the interpretability of DNNs, it has been found that hyperactive neurons (i.e., neurons with high levels of activation) can provide beneficial learning patterns for DNNs. In their work, Ma et al. [50] identify hyperactive neurons and their combinations to depict the neurons responsible for major DNN behavior, while hypoactive neurons are attributed to minor DNN behavior.

2.1.3 Deep Reinforcement Learning: The Integration of Deep Learning and Reinforcement Learning

Deep Reinforcement Learning, a combination of deep learning and reinforcement learning, has effectively addressed the limitations of traditional RL and significantly enhanced its efficiency [48, 52]. By leveraging deep neural network-based function approximators, DRL enables agents to learn and handle high-dimensional state and action spaces, thus improving RL's scalability [48, 52].

A wide variety of agents types have been proposed by DRL researchers to deal with the numerous challenges and situations that arise while developing DRL applications.

DRL algorithms can be classified as either model-free or model-based [48]. In the model-free approach, agents directly learn a policy through interaction with the environment. The model-based approach involves learning a model of the RL environment and its state transitions. In this thesis, our focus is exclusively on studying model-free algorithms. Within the model-free paradigm, three categories of algorithms can be identified [48]:

Policy gradients algorithms [42]: These algorithms aim to represent the policy π_* using

a neural network. Typically, they employ on-policy training, where each update is based on data collected according to the current policy, and the goal is to keep on improving the policy until it has reached the expected maximum reward value. Several DRL algorithms fall into this category, such as A2C/A3C [53], TRPO [54], and PPO [42].

Value-based algorithms [49]: This category focuses on learning to predict the action-value function, denoted as Q . These algorithms often use the Bellman equation [55] as an objective function to guide model optimization. Unlike policy gradient algorithms, value-based algorithms perform off-policy optimization, allowing them to update the model weights using data collected from any policy during training. Multiple algorithms fall under this category, including C51 [56], Rainbow [57], and DQN [6].

Hybrid algorithms [41]: Hybrid algorithms combine both policy gradients and value-based methods. One popular algorithm in this category is SAC [41]. SAC consists of two networks: the Actor and the Critic network. The Actor network generates the policy and selects actions, while the Critic network evaluates the quality of actions by estimating the value of state-action pairs Q .

2.2 Quality Assurance in Deep Reinforcement Learning

Quality assurance in DRL poses significant difficulties. In fact, developing a DRL agent involves addressing various challenges, including the need for effective exploration of the environment [58], ensuring stable training of the agent’s function approximators [52, 59], and maintaining a high performance after the production [4]. In Section 2.2.1, we discuss the challenges associated with debugging in DRL. Subsequently, in Section 2.2.2, we delve into the concepts of environmental drift and continual learning.

2.2.1 Challenges of diagnosing DRL applications

While standard DL settings usually involve training a model on a static dataset, the data used to train the model in RL is generated as a result of an interaction between the agent and the dynamic environment. This has several implications:

1. In order to even access all the relevant experience that an environment provides, the agent must be able to explore well; otherwise, the agent will likely fail as it will encounter states it has not seen before.

2. As the agent learns, the experience it generates—and thus the data it is trained on—changes. For example, as an agent learns, it might be able to reach different parts of the environment which not only leads to new states the agent must learn but also potentially has to change its evaluation of previous states from earlier in the training process.
3. Unlike in other settings where the loss is expected to (generally) monotonically decrease, in RL, because of the nonstationary nature of the data, the trend of the loss does not necessarily correspond to the agent performance.
4. Due to the complex, dynamic nature of the system, there is often more noise in the results of RL experiments. Usually, most RL experiments need to be run for several seeds before any conclusions can be drawn.
5. Finally, because the system has several changing components that the developer must implement, it’s difficult to find faults.

In this context, applying fault detection techniques developed for traditional deep learning is inadequate to cope with the unique challenges posed by DRL/RL. The conceptual differences between DRL and DL, such as the learning paradigm, data format, learning objective, and architectural design, result in specificities that must be considered for effective fault detection in DRL. All these challenges need the creation of DRL-specific approaches to assist developers in the creation of DRL systems.

2.2.2 Environment Drift and Continual Learning

The emulation of real-world scenarios in reinforcement learning presents multiple difficulties. First, the environment exhibits a non-stationary behavior [3], which is reflected by the environment’s dynamic nature, continuous evolution, and susceptibility to temporal changes [60, 61]. For instance, in the case of a self-driving car, variations in the environment, such as updates to safety regulations or the emergence of new and previously unseen vehicles on the road, require adaptations in the agent’s behavior during execution. Additionally, there is always a gap between the simulated environment and the actual environment [4]. This difficulty in accurately replicating real-world settings, during the simulation, arises from the complexity and delicacy of mathematically representing the environments. These difficulties have led to the concept of environment drift in DRL, which can significantly impact the DRL applications’ performance in production [3, 62].

More specifically, the term, environment drift, refers to a substantial change in the parameters of the environment that may affect the performance of trained agents. It corresponds

to concept drift in supervised classification [63], which refers to a change in feature-based probabilities of data, such as variations in objects' appearance or surrounding illumination. Indeed, a drifted environment provides the learner with new, unexpected data, i.e., the rewards associated with a certain state-action pair (s, a) have deviated. Although a drifted environment yields different rewards for a subset of state-action pairs, it should not be considered a completely new environment. In fact, the original environment and its drifted counterpart share some regions of state-action space. However, in a completely new environment, the state-action space has no guaranteed connections with any previous environment.

Hence, an agent should begin the learning process from scratch in a completely new environment because it requires replacement learning, whereas learning in a drift environment requires only further fine-tuning [35], i.e., fusion of new information with the existing knowledge. This process of fine-tuning, known as the continual learning in deep learning [64], aims to improve the adaptability and plasticity of deep neural networks as they progressively adjust to evolving data and develop new behaviors. However, despite the remarkable achievements in developing continual learning techniques, it has been observed [65] that the integration of new knowledge can introduce interference with existing knowledge, potentially leading to the complete replacement of previously learned information. This phenomenon, known as catastrophic forgetting [66], highlights the challenge of maintaining a balance between acquiring new knowledge and preserving the previously acquired knowledge during the maintenance phase of a DRL-based software system. Thus, it becomes crucial to stabilize the adaptation process and ensure the harmonious coexistence of both new and old knowledge.

2.3 Chapter Summary

This chapter has provided an overview of deep reinforcement learning, covering essential aspects such as the definition of reinforcement learning, followed by an explanation of the fundamental principles of deep learning. Additionally, we delved into the various types of deep reinforcement learning techniques.

The second part of this chapter discusses debugging in DRL and the issue of environment drift. Moving forward, the next section chapter will present a comprehensive literature review, analyzing relevant works pertaining to the topic of quality assurance in DRL.

CHAPTER 3 LITERATURE REVIEW

In this chapter, we review the literature on the quality assurance of deep reinforcement learning applications.

Chapter Overview: Section 3.1 provides a summary of the current testing methodologies employed in deep reinforcement learning. In Section 3.2, we discuss studies on fault detection in reinforcement learning and deep learning. Finally, in Section 3.3 we outline the difficulties associated with the maintenance of deep reinforcement learning applications.

3.1 Testing in Deep Reinforcement Learning

Despite the growing interest in DRL development, there is a limited number of papers that specifically analyze testing and quality assurance in DRL compared to traditional DL. Several studies have highlighted the bugs and theoretical challenges in developing DRL-based software systems [4, 5, 67], which explains the need for more research related to the topic of quality assurance in DRL.

Trujillo et al. [68] conducted one of the first studies on testing in DRL. They specifically analyzed the significance of neuron coverage in the context of deep reinforcement learning. Their goal was to investigate whether increasing neuron coverage correlates with better rewards. The study concluded that neuron coverage does not have a direct correlation with rewards and does not provide substantial insights when applied to DRL. Additionally, they demonstrated that excessive exploration may lead to better coverage, but it does not necessarily maximize rewards. These findings underscore the need for developing more RL-specific testing approaches, rather than relying on the ones developed for conventional DL approaches.

Multiple studies have examined robustness and safety in DRL systems. The studies [69–72] focused on the robustness of reinforcement learning against adversarial attacks; exploring various attacking techniques on states, rewards, actions, and model spaces. These studies showed that RLs have robustness issues and are vulnerable to adversarial attacks. Besides they have been all able to decrease significantly the performance (i.e., the overall reward) of the DRL application through adversarial attacks. To mitigate these vulnerabilities, other studies proposed adversarial training techniques [73, 74, 74–80]. These techniques incorporate adversarial examples into the training data or introduce perturbations during training

episodes, aiming to improve the agent’s ability to handle adversarial scenarios and improve its robustness. However, it has been shown that existing approaches for improving robustness and preventing adversarial attacks in DRL are not as efficient and stable as in traditional deep learning [69, 72]. Moosavi-Dezfool et al. [72] showed that even after adding adversarial examples during the training they are still able to generate new adversarial examples. Ilahi et al. [69] argued that adversarial robustness is still an open issue as robustness in DRL is complex and the attacks can be at different aspects or stages of the DRL application.

In terms of safety and performance testing, Tappler et al. [81] used a search-based algorithm to assess the safety of RL agents. While effective for RL agents, the approach was not applicable to DRL agents. Another study by Zolfagharian et al. [82] proposed the STARLA approach, which employs genetic algorithms to search for failures in the decision-making policy of a DRL agent. The authors emphasized the importance of reducing the time and cost required for efficient testing in DRL due to its black-box nature, vast state and action spaces, and the challenges of exhaustive testing.

Besides addressing the non-functional requirements in DRL, one of the few studies that addressed fault detection in DRL is the work by Nikanjam et al. [83]. The goal of the study was to develop a taxonomy of DRL-specific bugs that arise during the development of DRL applications and eventually lead to erroneous learning behavior.

The authors manually analyzed faulty artifacts extracted from GitHub and Stack Overflow to identify 11 bug categories, which were further classified into four groups. They introduced DRLinter, a model-based fault detection tool that utilizes a meta-model consisting of a type graph and graph transformation rules to detect bugs. While DRLinter demonstrated impressive performance, it is limited to static analysis and cannot detect bugs that may occur during training. Additionally, the tool is not scalable as it only works with DQN-based algorithms.

Overall, the DRL research community has made significant efforts to address various quality assurance issues. However, there are still many pending issues specific to DRL that needs to be addressed; particularly in debugging and fault detection. Furthermore, the aforementioned studies mainly include adaptations of approaches that have already been developed for traditional deep learning.

3.2 Studies on Fault Detection in Machine Learning

This section discusses fault detection in deep learning and deep reinforcement learning. The reason for analyzing fault detection in deep learning is that there are insufficient studies on fault detection in DRL. Moreover, these studies serve as one of the main sources of inspiration for our work on identifying faults in DRL applications.

In Section 3.2.1 we discuss fault detection in DRL applications. In Section 3.2.2 we present the studies on fault detection in DRL.

3.2.1 Fault Detection in Deep Learning

Several approaches have recently been proposed to assist developers in debugging DL systems. These approaches can be divided into two categories:

The first category debugs DNN models to detect and fix faulty behaviors at the neural level (e.g., weights and biases of the model). For example, Ma et al. [84] proposed MODE, a model debugging approach that uses state differential analysis to identify the model’s internal features causing the fault and then correct it via training input selection. Similarly, Eniser et al. [85] proposed DeepFault, a fault localization approach that detects the suspicious neurons in DNNs and then fixes them using gradient ascent. While these studies focus on DNN models, they can not pinpoint the source of the fault in DNN programs.

The second category, closer to our approach for fault detection, debugs the whole DNN program to detect and fix faulty behaviors at the code level. NeuraLint [86] and DEBAR [87] are two static analysis approaches that detect faults in DNNs. Zhang et al. [88] proposed AutoTrainer to identify and fix DNN training faults at runtime. AutoTrainer detects five training faults: vanishing/exploding gradient, dying ReLU, oscillating loss, and slow convergence. Braiek and Khomh [89] proposed TheDeepChecker, an end-to-end property-based debugging approach for DNN training programs, and Wardat et al. [90] proposed DeepLocalize, a DNN fault localization approach that detects root causes by spotting numerical errors during training. In the same vein, UMLUAT [91] was proposed as a user interface tool to find and fix DL faults using heuristics. Amazon SageMaker Debugger [92] is another user interface tool to detect and alert developers about frequent training faults. Most recently, DeepDiagnosis [93] and DeepFD [94] were proposed. The first is a fault diagnosis approach that reports symptoms and proposes fixes, while the second is a learning-based fault localization approach that formulates fault localization as a learning task.

3.2.2 Fault Detection in Deep Reinforcement Learning

To the best of our knowledge, no study has yet proposed a fault diagnosis approach for DRL systems. However, there are approaches that indirectly contribute to fault detection in DRL by allowing visualization and explainability.

Several studies have explored feature visualization techniques to provide insights into the behavior of DRL agents. In their papers, Zahavy et al. [95] and Mnih et al. [96] employ the technique, t-SNE [97], to offer visualization dashboards that help developers understand the contribution of multiple features to the agent’s decision making and assist in debugging. Saliency methods have also been used to visualize the behaviors of value and advantage functions in Q-networks in [98]. Additionally, some studies employ generative models to synthesize visual inputs of interest for trained agents such as the study of Rupprecht et. al. [99]. Furthermore, Huang et. al. [100] used entropy to identify critical states that agents encounter in the environment. While visualization helps in understanding the behavior of the agent and provides additional clarity regarding its policy, discerning the underlying cause of a bug remains a challenging task. The developer is still required to invest significant effort in determining the source of unexpected behavior and devising an effective solution.

Explainability is another aspect that can help developers in debugging their DRL applications. Various researchers [101] have conducted surveys on explainability in DRL, which is an active research area in quality assurance in DRL. Many studies [102–104] focus on explaining the fundamentals of RL algorithms, contributing to a better understanding of their inner workings. However, we did not find any work proposing approaches for detecting faults in DRL applications dynamically during training. The only relevant paper we found in this context is the work of Nikanjam et al. [83] which we have already discussed in Section 3.1. The paper proposes a model-based tool to verify whether a DRL model adheres to predefined rules. This technique does not allow for dynamic bug detection during training and has not been tested extensively in complex DRL applications.

3.3 Maintenance of Deep Reinforcement Learning Applications

In this section, we delve into multiple aspects pertaining to adaptability and performance degradation in deep learning and deep reinforcement learning applications. Specifically, we focus on the plasticity and adaptability of models to changes occurring after their deployment in production. Furthermore, we explore existing research efforts aimed at mitigating performance degradation issues in production and ensuring sustained high performance despite alterations in data or the environment.

3.3.1 Plasticity problem in Machine Learning

One of the most important properties of an intelligent agent is its ability to constantly learn, adapt, and evolve [40]. A DRL system must be able to learn and adapt continuously throughout its lifetime to the changing environment in which it operates [105].

However, enabling continuous adaptability of a model is challenging because of the stability-plasticity dilemma [106]. Agents must find a balance between adapting to the new drifts and maintaining what is already learnt. This is important to avoid issues like catastrophic forgetting [65].

A whole academic subfield of lifelong learning [35, 36] is concerned with agents that never stop learning. In spite of the growing interest in this ability today, reinforcement learning has traditionally been used to acquire a new task by initializing a policy from scratch, collecting new data in a stationary environment (using either an online or offline strategy), and evaluating a static policy that was trained on this information. The static paradigm does not evaluate the ability of the RL agent to adapt. The continual learning of RL has been studied in a variety of settings, including multi-task settings [107, 108], where agents are taught one game and then challenged to learn another game sequentially without forgetting the first. Biagiola and Tonella [40] investigated the adaptability of RL-based systems on gradually shifted environments using continual learning and report that continual learning has a quantifiable adaptation frontier that defines the agent’s adaptability limits. To the best of our knowledge, no prior research has investigated the influence of intentional forgetting on the optimization of DRL agents’ adaption process.

3.3.2 Continual Learning in Deep Reinforcement Learning applications

In supervised learning, continual learning [109] is an effective method for coping with concept drifts since it allows the model to learn continuously from shifted datasets. In the following, we present main continual learning algorithms that have been developed for deep neural networks.

Rusu et. al. [110] propose freezing changes to pre-trained neural networks and adding new subnetworks with fixed capacities to be trained for new tasks. Kirkpatrick et al. [111] proposed the elastic weight consolidation (EWC) algorithm that identifies the weights that have been important in the past and penalizes their updates based on those weights that were less significant in the past. In [112] a generative model is trained to generate synthetic data that follows the distribution of the original data. Hence, original samples of data can be

interleaved with the novel shifted samples in order to preserve the acquired knowledge and learn new abilities. Zhizhong et al. [113] proposed the algorithm learning without forgetting (LwF) that constrains network predictions on the old tasks from changing significantly while optimizing the parameters of the new task at the same time. There are many efforts to repair neural networks [88, 114, 115] that experience unexpected behaviors by fine-tuning the model with additional data, but there is no guarantee that the fine-tuned model will eliminate unwanted behavior or learn novel required behaviors efficiently without catastrophic forgetting [66]. In response, advanced DNN repairing techniques have been developed to fix existing neural networks without retraining, which can be preferable in some circumstances. Zhang et al. [116] proposed Apricot, a weight adaptation approach, to evolve iteratively the DNN and fix its performance degradation. Sohn et al. [114] proposed Arachne, a search-based automated repair method for DNNs that updates weights directly instead of retraining the original DNN from the beginning.

Different from the existing research on continual learning and repairing in deep neural networks, our focus is on the self-healing of DNN-based DRL agents in newly-drifted environments.

Khetarpal et al. [39] provided a comprehensive taxonomy of existing approaches for continual reinforcement learning. The study defines three main types of continual reinforcement learning: (1) Explicit knowledge retention, which aims to stabilize learning to mitigate the catastrophic forgetting problem during model adaptation [117, 118]. (2) Leveraging Shared Structure, where agents learn to solve problems by identifying structures that can assist them in continual learning. These structures are used by the CL to fix the agent’s behavior through function composition mechanism [119, 120]. (3) Learning to learn, which encompasses meta-learning techniques [121]. Agarwal et al. [122] introduced their approach, Reincarnating Reinforcement Learning (RRL), with a primary emphasis on reducing the time and computational costs associated with tabula rasa RL training. This is achieved through the transfer and reuse of prior knowledge across different design iterations of an RL agent.

Other research focuses on developing strategies to optimize continuous learning in DRL [123]. Besides, several studies particularly address CL by focusing on the challenge of avoiding catastrophic forgetting by balancing the stability-plasticity dilemma. Elastic Weight Consolidation (EWC) is proposed by Kirkpatrick et al. [111] to weigh the importance of past tasks, in order to slow down their changes and preserve the agent’s ability to perform them. Kaplanis et al. [38] treat each neuron as a dynamical system with latent information, allowing control and protection of the neuron’s information.

Furthermore, numerous studies [112, 124, 125] concentrate on improving continual learning through experience replay buffers. These studies aim to incorporate past experiences when adapting the agent to maintain its ability to perform previously learned tasks. Mixing on-policy and off-policy updates in DRL is addressed in the literature [126, 127] to facilitate efficient learning of new tasks while retaining the ability to perform previously learned ones.

Despite these notable advancements, practical challenges hinder the widespread adoption of these techniques. Khetarpal et al. [39] and Biagiola et al. [40] argue that continual learning still exhibits limited plasticity, necessitating a better balance between stability and plasticity in the learned model. Additionally, accelerating the healing process is crucial, as performance degradation can be intolerable and potentially dangerous in many scenarios.

In this thesis, we develop an intentional forgetting mechanism that enables dual-speed fine-tuning of major and minor behaviors of DRL agents. This approach accelerates continual learning and enhances adaptability to newly drifted environments. Instead of introducing potential hidden defects through direct weight changes, our approach performs selective weight updates based on the principles of continual learning and gradient-based optimization.

3.4 Chapter Summary

In this chapter, we begin by providing an overview of the research efforts dedicated to testing and debugging. We delve into the approaches proposed for testing DRLs and emphasize the growing interest in this area. Subsequently, we explore the fault detection approaches proposed for both DL and DRL, wherein we highlight the disparity between the available tools for fault detection and underscore the limited studies on fault detection dedicated to DRLs. Then, we delve into the topic of maintenance. We start by presenting existing studies on plasticity and emphasize its significance in ensuring efficient maintenance. Additionally, we discuss the existing approaches to continual learning and review the state-of-the-art studies in this domain.

CHAPTER 4 TOWARD DEBUGGING DEEP REINFORCEMENT LEARNING PROGRAMS WITH RLEXPLOER

Deep reinforcement learning (DRL) has demonstrated promising accomplishments in a variety of disciplines in recent years such as robotics [128], computer games [129], recommendation systems [130], and computer vision [62]. In order to adapt effectively with the emergence of DRL, research is being conducted to aid practitioners in meeting the new challenges that DRL poses (e.g., the sim-to-real gap [131]). Developers are continuously releasing industrial-scale frameworks and libraries such as Stable Baselines [132], Keras-RL [133], TensorForce [134], and RL-Hive [135] to aid practitioners in the design of reliable DRL systems.

DRL systems, like other software systems, contain their own unique faults [82, 83], presenting developers/researchers with new challenges in debugging and diagnosing these systems. However, debugging DRL systems is particularly challenging [136–138] because, unlike conventional software systems, the decision logic of a DRL system is not explicitly encoded but rather derived from the interaction between the DRL agent and its environment [139]. In addition, Deep Neural Networks (DNNs), the backbone of any DL/DRL system, are “black box” entities that cannot be debugged using conventional methods such as breakpoints. To make matters worse, when faults are introduced, DRL systems often yield unexpected behavior without explicit error messages, or failures, making debugging tedious and time-consuming. Newcomers to RL, who frequently rely on high-level frameworks (like Stable Baselines [132]), have limited knowledge for debugging. RL experts, on the other hand, rely on their experience for debugging, often resorting to trial-and-error methods guided by intuition. Currently, existing DRL frameworks offer limited assistance for debugging and diagnosing faults during DRL system development. Tools like Tensorboard [140], Weights and Biases (Wandb) [141], and tfdbg [142] can aid developers in monitoring the training status. They are, however, incapable of analyzing the behavior of agents and providing meaningful insights. To address the aforementioned challenges, an approach that frees developers from manual monitoring and diagnosing DRL systems is paramount.

According to the DRL taxonomy of real faults proposed by Nikanjam et al. [83], faults in DRL systems are divided into 3 categories, and in this study, we focus on RL-specific faults (e.g., missing exploration), and DNN-specific faults (e.g., vanishing gradient). We propose *RLExplorer*, the first fault diagnosis approach for DRL systems. To design *RLExplorer*, we first rely on [89] to gather a list of diagnosis routines that detect DNN-related problems and can be applied to DRL systems. Second, we leverage Nikanjam et al.’s taxonomy [83] of DRL-

specific training faults. Then, we derived symptoms and root causes that are generated by these DRL training faults, in order to propose diagnosis routines that detect their occurrences during training. Attached to the DRL application runtime, *RLEplorer* automatically records data and runs diagnosis routines in the form of checks that encapsulate properties of the learning dynamics. *RLEplorer*, then, displays the results of these checks as warning messages that incorporate theoretical concepts, explain best practices, and suggest possible correction strategies to address the root causes. Finally, *RLEplorer* can handle a variety of model-free DRL algorithms, including on- and off-policy algorithms.

To evaluate *RLEplorer*, we built a benchmark comprising 11 faulty DRL samples from Stack Overflow (SO) that contain real faults, then assess *RLEplorer* on these samples. Results show that our approach was able to detect and diagnose faults in 83% of cases. We further conduct a human study with 15 DRL experts to assess *RLEplorer*’s effectiveness and usability in helping developers diagnose faults. Participants using *RLEplorer* were able to diagnose 3.6 times more faults compared to manual debugging. Participants also reported high satisfaction with the debugger and a high likelihood of leveraging *RLEplorer* in their development.

The remainder of this chapter is as follows: Section 4.1 covers our approach. Section 4.2 reports our findings. Section 4.3 discusses the results of our study. Section 4.4 discusses validity threats.

4.1 Approach

In this section, we explain our DRL-fault diagnosis approach, *RLEplorer*. We provide a description of the list of the DRL faults that *RLEplorer* monitor, along with their associated symptoms and root causes. Then, we detail our approach to detect these faults and explain the workflow of *RLEplorer*. To the best of our knowledge, *RLEplorer* is the first approach that thoroughly addresses the challenge of fault diagnosis specifically in DRL.

4.1.1 Overview

Training DRL applications is known to be time consuming and computationally expensive [143] [144]. *RLEplorer*’s main objective is to identify symptoms of faults as soon as possible during training. This can help DRL developers save time/effort and find faults without waiting until the end of the training. Figure 4.1 illustrates an overview of *RLEplorer*. *RLEplorer* begins by collecting various dynamic traces (e.g., reward, weights, and actions) from the DRL application during the training. It then analyzes these traces for

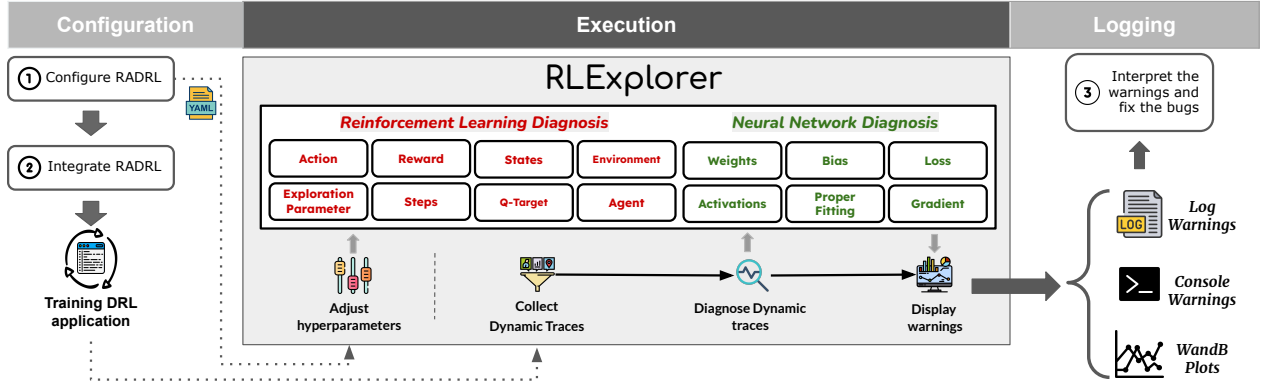


Figure 4.1 Illustration the Three Steps of *RLExplorers* Workflow: Configuration, Execution, and Fault Logging.

automatic and real-time identification of potential faults. Finally, if a symptom is detected, *RLExplorer* prompts warning messages that guide the user to address the root causes of the issue. *RLExplorer* has three phases: Configuration, Execution, and Logging.

Configuration *RLExplorer* offers a set of diagnostics to check the behavior of key components in a DRL application. Users specify which components to diagnose by selecting relevant diagnostics and their execution frequency. Each individual diagnostic can be further customized to meet the training environment’s specific characteristics. Besides, to enhance the *RLExplorer*’s usability, a default configuration is provided.

Execution *RLExplorer* monitors learning traces to perform the diagnosis. The *RLExplorer*’s dynamic trace collection phase employs two methods: (1) *RLExplorer* automatically detects standard DRL components’ traces (e.g., rewards and states from the environment) using the observer design pattern, and (2) for traces that cannot be detected automatically (e.g., exploration parameter), *RLExplorer* provides a predefined function that users can call to track these traces. Then, each chosen diagnostic periodically performs its verification routines looking for any symptom of a specific component.

Logging During training, *RLExplorer* operates as an event handler. When a symptom is detected by a diagnostic, *RLExplorer* is triggered to process the symptom and notify the user when a fault is detected. *RLExplorer* displays warning messages that provide explanations of the faulty behavior and suggest possible correction strategies to fix the fault. Finally, *RLExplorer* provides real-time plots of advanced features (e.g., epistemic uncertainty [145]) that conventional monitoring tools like Tensorboard [146] do not show. The event-driven nature of *RLExplorer* enables developers to resolve errors promptly rather than waiting until

the end of the training process. This is crucial in DRL applications given that training is a resource-intensive and time-consuming task.

4.1.2 Failure symptoms and root causes

To establish the diagnostic approach of *RLExplorer*, we leveraged prior research [83] to define symptoms of DRL faults and their root causes. Subsequently, we developed multiple diagnosis strategies to identify these symptoms and suggest possible mitigation actions. In the following subsections, we explain the DRL faults symptoms, their underlying causes, and the methods employed by *RLExplorer* to detect such symptoms. Notably, we classified DRL faults into two categories: RL-specific faults and NN-specific faults.

The Reinforcement Learning Symptoms

Nikanjam et al. [83], proposed a taxonomy of DRL faults based on an analysis of SO posts and GitHub issues to identify commonly reported faults among DRL developers. Based on this taxonomy, we designed diagnosis routines to address each fault category and detect their symptoms. Each fault may manifest in the behavior of various RL components. In the following, we present the RL-specific symptoms and root causes for 8 RL components.

Actions Symptoms.

Abnormal state entropy: Xin et al. [147] define State Entropy (SEN) as the RL agent’s uncertainty and randomness in selecting an action at each state. SEN tends to start high (during exploration) then decreases throughout learning and reaches its minimum when the learning process converges, providing the best-learned policy [136]. SEN’s erroneous behavior symptoms might include prolonged stagnation, sharp drops, unexpected increases, or severe fluctuations [136].

Root causes: Prolonged SEN stagnation can occur when the policy targets or gradient back-propagation are incorrectly computed [136]. Sharp drops and unexpected increases in SEN can occur when the agent collapses into a myopic policy (only considers the immediate reward [148]) and is not exploring anymore. Finally, severe fluctuations occur when the learning rate is too high [136].

Action stagnation: An agent repeating identically the same sequence of actions is a symptom of faulty behavior. Stagnation can occur inside an episode when the agent repeats the same action, or across successive episodes when the agent repeats the same sequence of actions.

Root causes: Action stagnation may be caused by performance plateau [149] when the agent gets stuck in a local optimum [150], or by the Noisy TV problem [151].

High epistemic uncertainty: Epistemic Uncertainty (EU) is a learner’s uncertainty caused by a lack of knowledge during a learning task [152, 153]. It provides insight into the confidence of the learner (i.e., agent) while making decisions. Typically, the EU value is initially high during the early stages of training and is expected to decline as iterations and data grow [152]. Any deviation from this expected behavior may indicate the presence of faulty behavior.

Root causes: High EU may be caused by a missing or suboptimal exploration mechanism or wrong computation of the agent’s gradient during backpropagation.

Agent Symptoms.

Wrong agent update policy: Leveraging DNNs as a function approximation in DRL makes the agent’s learning unstable [154]. This is due to the DNN’s nonlinearity. To that end, techniques have been introduced to stabilize the agent’s learning in DRL [154]. In this vein, target networks were proposed by Mnih et al. [45] as a stable reference for the main network to reduce the learning variance. Thus, it is recommended to use a target model to stabilize the learning process. The wrong agent update symptom arises when the synchronization between the main and target networks is disrupted, causing an unstable learning process.

Root causes: This symptom is due to the mix-up between the main and target networks, not updating the target network at the right time, or not updating it at all.

High Kullback-Leibler (KL) divergence in predictions: This symptom occurs when the KL divergence [155] of the main model predictions is high on a fixed batch of states over two consecutive model updates. The intuition is that the outputs of the model should remain consistent and not be changed drastically in few successive updates [136].

Root causes: This symptom may arise if the agent learns from new observations with a divergent distribution [136].

Environment Symptoms.

Wrong environment conception: When designing an environment it is crucial to ensure that the environment design conforms to the standards and commonalities of DRL [34]. Symptoms of inappropriate environment design occur when environmental functions (e.g. step and reset return) show numerical instabilities such as NaN or infinity values in their outputs.

Root causes: The root cause of these symptoms is usually a wrong implementation/design of the RL environment or missing the terminal state.

Exploration Parameter Symptoms.

Suboptimal exploration rate: A common learning strategy involves starting with a high exploration rate to encourage the agent to explore different actions and states, and gradually reducing the exploration rate to shift towards exploitation. In most of the exploration strategies, a parameter controls the ratio of exploration and exploitation (e.g., ϵ in epsilon-greedy

policy [156]). Faulty symptoms in this category include a suboptimal exploration rate manifesting in a sharp decrease in the exploration parameter value.

Root causes: These symptoms can be caused by improper exploration parameter setup, update, or high variation.

Reward symptoms.

Erroneous accumulated reward behavior: During training, the standard deviation (std) of the reward can be used to diagnose the behavior of a DRL agent. In the early learning steps, the agent's behavior is mostly random (the agent is exploring), leading to fluctuating and high reward std values. As learning progresses, the agent's behavior becomes more stable, and the std of the accumulated reward generally reduces over time. Any deviation from this normal behavior could potentially indicate an erroneous behavior.

Root causes: One potential factor contributing to this symptom is a suboptimal exploration rate.

Step Symptoms.

Premature episode termination: The number of steps executed by an agent while interacting with its environment is a critical parameter in DRL as it has a direct impact on the learning efficiency of the agent. Premature termination of an episode can hinder the learning efficiency of the agent, particularly during the exploitation phase when the agent is not actively learning.

Root causes: The root cause of these symptoms could be the low value set for the maximum number of steps per episode.

States Symptoms.

Repetitive states sequence: Faulty symptoms in this category manifest in the form of repetitive or stagnant sequence of states, which can hinder the agent from achieving its expected goal. Repetitive behavior in the sequence of states can occur in two forms: First, repeated sequence of states within one episode, which happens when the environment keeps returning to the same states in one episode. Second, repeated sequences of states across episodes occur when the agent repeatedly follows the same path in different episodes.

Root causes: Repetitive states may indicate that the agent is stuck in a local optimum and requires further exploration [150].

Q-Target Symptoms.

Wrong calculation of the $q_targets$: This symptom is particularly significant in DRL applications that rely on Q-value-based learning (please see Background). This symptom arises due to discrepancies between the $q_targets$ values, computed according to the original formula presented (2.2), and the one provided by the user.

Root causes: The main cause of this symptom is the wrong application of the `q_targets` calculation formula.

The Neural Network (NN) Diagnosis

The NN diagnostics involve checking various features of the NN such as weights, and biases during the model training. Several studies have proposed taxonomies of common faults in NNs [157, 158], along with approaches for detecting and fixing these faults [89, 90, 93, 94].

In this vein, Braiek and Khomh [89] proposed a collection of DNN training faults and their symptoms via mining posts from SO and GitHub. They also introduce TheDeepChecker a property-based debugging approach for DNN training programs. We relied on this study [89] and adapted multiple NN diagnoses when designing *RLExplorer*. However, due to the unique architecture and functionality of DRL applications, some diagnoses that require features such as accuracy and labels were excluded. The NN features that have been used in the *RLExplorer* are the *activation*, *bias*, *gradient*, *loss*, *weight*, and *proper fitting* (i.e., the NN’s ability to fit on a small batch of data). For more details on how each one of the NN diagnoses works, please refer to the paper [89].

4.1.3 Detecting Failure Symptoms

In Table 4.1, we report our approach for detecting the symptoms discussed in Section 4.1.2. Prior to running *RLExplorer*, users have the option to modify the checks’ configurations specified in Table 4.1. However, in our research, we determined default values for these configurations through trial-and-error experimentation, as well as validation with DRL domain experts. Besides, after various trials on different RL algorithms/environment, we believe that the given default configuration values have an acceptable level of generalizability in different DRL problems.

After configuring and integrating *RLExplorer* in the DRL system, *RLExplorer* subscribes to the user-activated components and listens to their required dynamic traces (Inputs in Table 4.1 and line 1 in algorithm 1). Then, *RLExplorer* is periodically triggered to perform three types of diagnostics: early-stage learning diagnostics, late-stage learning diagnostics, and diagnostics performed throughout the training process. The early-stage diagnostics (lines 2-7 in algo 1) investigates symptoms that may occur during exploration including ENV.d1, ENV.d2, ENV.d3, STT.d1, RWD.d1, ACN.d1, ACN.d2 (check Table 4.1); while the late-stage diagnostics (lines 8-13) handle the symptoms that may occur during exploitation in-

Table 4.1 Description of *RLExplorer*'s Diagnosis Approach for Detecting Fault Symptoms in RL Components (4.1.2)

Component	Input	Symptoms	Detection Method
Agent (AGT)	<i>MM</i> : main model; <i>TM</i> : target model; <i>AP</i> : actions probs; <i>TM_{per}</i> : TM update period;	Wrong agent update policy High KL divergence	If the current step has reached <i>TM_{per}</i> , we check if TM's params are updated with MM (d1). If not, we check if the two model's params are different (d2). We also check that APs are outputted by the correct model (MM) (d3). Finally, in two successive updates, we check if the agent's output KL divergence [155] on a given input batch is smaller than 0.1 (d4).
Environment (ENV)	<i>env</i> : environment;	Wrong environment conception	We analyze the environment's behavior against random actions. We check for numerical instabilities (NaN or infinity values) (d1), and unnormalized rewards as recommended by [136] (d2). We also checks if random episodes can easily reach the episode's maximum reward threshold (too-easy problem) (d3).
States (STT)	<i>env</i> : environment; <i>Max_r</i> : max reward;	Repetitive states sequence	We check for unnormalized observations outside [-10,10] range (recommended in [136]) (d1). Next, in the last 20% of episodes, we check for identical states within one episode (d2) or across successive episodes (d3).
Step (STP)	<i>env</i> : environment; <i>Max_r</i> : max reward; <i>Max_{steps}^{ep}</i> : max steps per episode;	Early episode termination	We check if episodes are prematurely ended due to $\text{Max}_{\text{steps}}^{\text{ep}}$ being reached whilst the agent fails to reach a reasonable average reward (e.g., $0.1 * \text{Max}_r$) (d1). d1 only runs in the last 20% episodes.
Exploration Parameter (EXP)	<i>EF</i> : exploration factor;	Suboptimal exploration rate	We check if the function of the Least Squares Solutions (<i>lstsq</i>) [159] of EF values is not strictly monotonous (d1). Next, we check if the second derivative of EF values (f''_{EF}) is too high (> 0.22), which indicates that the rate of change of EF values is fast (d2).
Reward (RWD)	<i>R_{std}</i> : reward std; <i>Max_r</i> : max reward;	Erroneous accumulated reward behavior	This check examines <i>R_{std}</i> in a window of successive episodes. In the first 20% episodes, we check if the reward per episode is stagnating. For that, we verify if the RMSE [160] between the Least Squares solutions (<i>lstsq</i>) of <i>R_{std}</i> and the actual <i>R_{std}</i> values is less than 0.1 (d1). In the last 20% episodes, we check if the slope of <i>lstsq</i> is fluctuating ($ lstsq > 0.25$) (d2) and if the agent is trapped at a low reward value (d3).
Actions (ACN)	<i>AP</i> : actions probs; <i>Max_r</i> : max reward;	Abnormal state entropy (SE)	First, we check the State Entropy (SE). In the first 20% episodes, we check if SE is increasing ($lstsq > 0.1$) (d1) or stagnating ($< 10^{-3}$) (d2). In the last 20% episodes, we check the rate of change of the SE is fast (method in EXP.d2)(d3). During the whole training, we check if SE is fluctuating using the same method as RWD.d1 (d4).
		Action stagnation	We check if the number of identical actions within a single episode (d5) or between two successive action sequences is (> 10) (d6).
		High epistemic uncertainty	Monte Carlo dropout quantifies uncertainty by adding multiple dropout layers to the network and generating different outputs for a given input. We check if the average std of model outputs is too high (> 0.5)(d7).
Q-Target (QTR)	<i>QTs</i> : q_targets; <i>QTs_{pred}</i> : pred QT	Wrong calculation of the q_targets	We check the equivalence of the q_targets obtained using the equation 2.2 with those provided by the user (d1).

cluding STP.d1, STT.d2, STT.d3, RWD.d2, RWD.d3, ACN.d3. The remaining diagnostics (AGT.d1, AGT.d2, AGT.d3, AGT.d4, EXP.d1, EXP.d2, ACN.d4, ACN.d5, ACN.d6, ACN.d5, QTR.d1) are performed during the whole training (lines 15-20). For the NN diagnostics, we suggest readers to look at Braiek and Khomh’s work [89] and our replication package [161] for more information on NN symptoms and methods for identifying them.

Finally, if a symptom is detected, *RLEplorer* logs and notifies the user with a warning message that highlights the fault explanation and potential debugging strategies (line 21). Additionally, *RLEplorer* provides real-time plots of the epistemic uncertainty, reward standard deviation, and KL divergence.

Algorithm 1: Debug function of *RLEplorer*

Require: Dynamic traces collected during the training
Ensure: Display warnings of potential faults in the code

```

1: Track_dynamic_traces()
2: if training_is_in_early_episodes() then
3:   for all check in early_stage_learning_diagnostics do
4:     if checks.period_reached() AND check.is_activated() then
5:        $w_{early} \leftarrow \text{check.run\_diagnosis}(\text{Input})$ 
6:     end if
7:   end for
8: else if training_is_in_late_episodes then
9:   for all checks in late_stage_learning_diagnostics do
10:    if checks.period_reached() AND check.is_activated() then
11:       $w_{late} \leftarrow \text{check.run\_diagnosis}(\text{Input})$ 
12:    end if
13:  end for
14: else
15:   for all checks in remaining_diagnostics do
16:    if checks.period_reached() AND check.is_activated() then
17:       $w_r \leftarrow \text{check.run\_diagnosis}(\text{Input})$ 
18:    end if
19:  end for
20: end if
21: display_warnings( $w_{early}, w_{late}, w_r$ )

```

4.1.4 Integration of custom fault detection diagnosis

To enhance the flexibility of our tool, we have focused on simplifying the process of integrating new checks, customized to the user’s specific needs and requirements. This integration can be carried out programmatically in two simple steps: firstly, by implementing the user’s

fault diagnosis strategy, and secondly, by activating it. The tool manages the rest of the process autonomously.

The reason for this seamless integration lies in our deliberate attention to the tool’s architectural design. We have employed several strategic design patterns, such as observer, factory, register and singleton, known for their ability to give applications greater flexibility and customizability. These models enable our tool to easily adapt to various fault diagnosis logics, increasing its overall usefulness and adaptability.

4.2 Evaluation

In this section, we aim to evaluate *RLExplorer* using (1) real faulty samples extracted from SO and (2) a study involving human participants. For the first evaluation, we used a benchmark of faulty samples gathered from SO to assess *RLExplorer*’s effectiveness in fault diagnosis and reported the number of accurately diagnosed faulty cases. We also evaluated the runtime overhead that *RLExplorer* adds to the collected SO faulty samples. For the second evaluation, we conducted a human study through a coding task and survey to solicit DRL experts’ feedback on *RLExplorer*.

4.2.1 Faulty Samples evaluation

Method

Using a mining strategy similar to prior works [88,90], we collected SO posts with completed and reproducible DRL buggy code and build the benchmark of faulty samples.

We started by searching for posts on SO with the tag “reinforcement learning.” We then retained posts with (i) accepted answers and (ii) code snippets in the post description. After removing duplicates and posts having the term “install” (to exclude installation-related posts), we obtained a total of 426 posts. In the second phase, two authors manually examined the retrieved posts and filtered out those that included incomplete code, non-reproducible statements, or crashing errors. In the final phase, we reviewed each post’s accepted answer, identified the faults, their symptoms, and root causes, and defined at least one expected fault diagnosis. The two authors performed cross-validation of the extracted posts to ensure agreement. Conflicts were handled and resolved through scheduled meetings.

In total, we collected 11 posts and 12 faults. The low number of posts is due to the lack of reproducible DRL faulty codes in SO. Furthermore, we note that there is no available dataset of reproducible DRL buggy programs. In addition in our research, we identified 5 additional

bugs on SO, distinct from those used by the study [83] when developing their DRL faults taxonomy.

Evaluation criteria

We use the following criteria to determine if a fault is effectively detected and diagnosed by *RLExplorer*: A fault is deemed correctly diagnosed if *RLExplorer*’s detection result corresponds to the post’s accepted answer. Additionally, we measured the time overhead introduced by *RLExplorer* when debugging DRL faulty samples using the following formula:

$$\text{Time overhead (\%)} = \frac{T_d - T_n}{T_n} \times 100 \quad (4.1)$$

where T_d is the time taken to execute the program with the debugger enabled and T_n is the time taken to execute the program without the debugger. Due to the stochasticity of DRL systems, we report the average time overhead of 5 runs.

Results

Table 4.2 Results of *RLExplorer* on Stack Overflow Buggy Samples.

Post #	Ref	Description	Runtime (s)	Time Overhead(%)	Fault Diagnosis
57106676	[162]	Algo: DQN; Env: CartPole-v0; Fault: Low update frequency of the target network.	135.17	3.34	✓
56964657	[163]	Algo: DQN; Env: CartPole-v0; Fault: Low update frequency of the target network.	2666.23	14.16	✓
47750291	[164]	Algo: DQN; Env: CartPole-v0; Fault: Missing exploration.	711.21	19.79	✓
54385568	[165]	Algo: DQN; Env: CartPole-v0; Fault: Wrong update of the exploration factor.	2.87	17.50	✓
74214034	[166]	Algo: REINFORCE; Env: CartPole-v0; Fault: wrong computation of the discounted reward.	169.58	18.17	X
45886398	[167]	Algo: DQN; Env: MountainCar-v0; Fault: Suboptimal exploration.	34981.4	11.37	✓
47643678	[168]	Algo: DQN; Env: CartPole-v0; Fault: Missing terminal state.	3215.47	11.98	✓
64690471	[169]	Algo: DQN/DDQN; Env: CartPole-v0; Fault: not adding the end of the episode to the replay buffer.	27.20	22.74	✓
67789148	[170]	Algo: DQN; Env: CartPole-v1; Fault: Wrong discount factor (=1)	59.83	11.97	✓
56816743	[171]	Algo: DQN; Env: CartPole-v1; Fault: noisy learning due to missing the replay buffer	11.34	22.33	✓
49035549	[172]	Algo: DQN; Env: Pong-v0; Fault 1: exploration factor decays very quickly	66.73	24.68	✓
		Fault 2: Missing ReLU Activation			X
Average			3822.45	16.18	10/12

Table 4.2 reports the results of debugging SO posts using *RLExplorer*. The first two columns display the post ID and its associated reference. The columns Description, Runtime, Time Overhead, and Fault Diagnosis show the fault’s description, the runtime of the code sample without integrating the debugger, the time overhead added by the debugger, and whether *RLExplorer* correctly diagnosed the fault, respectively. In the Fault Diagnosis columns, ✓ indicates that *RLExplorer* successfully diagnoses the fault, whereas X indicates that *RLExplorer* fails to diagnose it.

We compared our approach’s diagnosis results to the posts’ accepted answers. The results

demonstrate that *RLExplorer* was able to correctly diagnose 10 out of 12 faults (accuracy of 83%). This high accuracy indicates *RLExplorer*’s effectiveness to diagnose diverse types of faults (9 unique faults) across different DRL algorithms and environments (2 algorithms, 3 environments). In the remaining 2 faults, *RLExplorer* failed to find and diagnose the faulty behavior of the tested code. For example, in the sample [166], the wrong computation of the discounted reward was not detected by *RLExplorer*. Our approach does not explicitly check the discounted reward calculation since it might be computed using many valid formulas [173]. Nevertheless, *RLExplorer* warned that the uncertainty was not decreasing and the reward was too distant from the maximum reward in the final episodes, indicating a problem in the learning process. In the sample [172], the developer misses adding a ReLU activation function to the second and third Convolution layer of the model. *RLExplorer* does not cover these types of checks as they have already been treated by other static code analysis tools such as [86].

Finding 1: *RLExplorer* performed accurately in diagnosing faulty behaviors in DRL samples extracted from SO (83% accuracy).

Furthermore, *RLExplorer*’s time overhead was assessed when debugging DRL faulty samples. The efficiency of *RLExplorer* is a critical criterion for its adoption by developers. Table 4.2 reported the time overhead of running the benchmark with *RLExplorer* integrated. Running the DRL faulty samples without *RLExplorer* took an average of 3822.5 seconds whereas running these samples with *RLExplorer* integrated added 16.2 % of time overhead. Our approach needs to record and monitor numerous variables such as weight, activations, observations, and rewards to perform dynamic analysis. This results in an overhead if the DRL agent or the environment is large, however, based on Table 4.2, this overhead is acceptable and often negligible when compared to the amount of time required to manually debug DRL systems. Moreover, *RLExplorer* is configurable, allowing the user to adjust the diagnosis periodicity to further reduce the time overhead.

Finding 2: *RLExplorer* exhibits a low overhead and can be leveraged to alleviate manual debugging of DRL applications.

4.2.2 Human evaluation

We evaluate the usability and effectiveness of *RLExplorer* in assisting developers in finding faults in DRL code through a human study with 15 DRL experts. We injected faults into a DRL program and recorded how many faults participants could detect with and without the

RLExplorer.

Setup

We prepared a synthetic faulty example and conducted a one-hour individual session with each participant to assess their experience with *RLExplorer*. At the end of this session, each participant was given a survey to complete to assess *RLExplorer*’s (1) effectiveness, (2) usability, and (3) the relevance of its checks. We recruited 15 participants, from our network, working on DRL systems (12 male, 3 female; ages 18-30) to participate in our study. All participants were DRL researchers from one of the world’s top machine-learning institutes. 10 participants were Ph.D. students, 4 were Master’s students and 1 was a research assistant. DRL experience of participants ranged from less than a year to 5 years (median=2). The study was conducted remotely via Google Meets video conferencing software. Two authors were always present to moderate the meeting. The Python notebook for the debugging task was shared with the participant and run with Google Colaboratory on the participant’s local laptop.

For the debugging task, we followed the methodology of a previous study [91]. We adapted a simple DRL program from the Pytorch tutorial [174]. This program (denoted as program A) solves the Gym CartPole environment [43] using the Deep Q-Network (DQN) [6]. We compared debugging the programs using *RLExplorer* to a baseline strategy (i.e., manual debugging). Each time, we counted the number of faults diagnosed by the participant (i.e., the explicit root cause verbally indicated). We intended to keep these programs as simple as possible for two reasons. First, program simplicity facilitates the participant’s full comprehension. Second, a simple agent can be trained quickly.

We then injected five unique faults which we sampled from SO in program A to retain the realistic representation of the simulated erroneous behavior. Program A runs without any crashing error, but the faults have an influence on the agent’s performance (i.e., average cumulative reward). We further simplified the debugging task by highlighting the location of the injected faults to make the interviewee’s experience fair. This will reduce the debugging effort as the coding task is time-limited. Finally, the faults were chosen to cover common stages of the DRL development process (e.g., Agent Architecture, Parameter Tuning) and to be of different severity. The following faults were introduced into Program A:

- A1: Improper weight initialization for the main and target models (Low severity, model architecture)
- A2: Improper bias initialization for the main and target models (Low severity, model

architecture)

- A3: Using the target model to predict actions instead of the main model (High severity, agent workflow)
- A4: Not updating the target model’s parameters when reaching the update period (High severity, agent workflow)
- A5: The exploration factor (epsilon) value is decaying very fast (High severity, parameter tuning).

Procedure

Each participant was shown a presentation of our approach. This presentation included an overview of the checks performed by *RLExplorer*, an elaboration of the erroneous behaviors that the approach is capable of detecting, and an introductory example demonstrating the integration of *RLExplorer* into a DRL application. The structure of program A was explained to each participant. The participant was told that the program had numerous faults, and their task was to find them. The participant was also permitted to use any online resources, such as SO, or web search, but the moderators could not assist with debugging.

The participant begins with the baseline condition: debugging the program without the use of the *RLExplorer*. They were given 15 minutes for debugging (similar to previous studies [91]), and the number of faults found under the baseline conditions was counted. Next, the participant was allowed to use the *RLExplorer* for an additional 10 minutes. The number of newly found faults using the *RLExplorer* was counted at the end of the debugging time.

At the end of the meeting, each participant was asked about their experience with *RLExplorer* and given a survey to complete. The survey includes questions regarding the tool’s usability, effectiveness in assisting debugging and the relevance of the checks integrated into *RLExplorer*.

Results

Table 4.3 illustrates the consolidated results of the debugging task. Participants, when using *RLExplorer*, found and diagnose more faults ($\mu = 3.13, \sigma = 1.31$) compared to the baseline condition (i.e., manual debugging) ($\mu = 0.87, \sigma = 0.96$). This means that participants were able to diagnose 3.6 times more faults using *RLExplorer* compared to manual debugging. To determine the significance and effect size of this difference, we used the Wilcoxon statistical

Table 4.3 Comparison between *RLExplorer* and manual debugging in terms of the number of diagnosed bugs and the average bug diagnosis time.

	Manual Inspection		<i>RLExplorer</i>		Comparison	
	mean	std	mean	std	p-values	$\hat{A}_{12}$
# Diagnosed Bugs	0.87	0.96	3.13	1.31	0.002	0.91
Average BD time (sec)	426.7	203.1	216.3	125.8	0.03	0.18

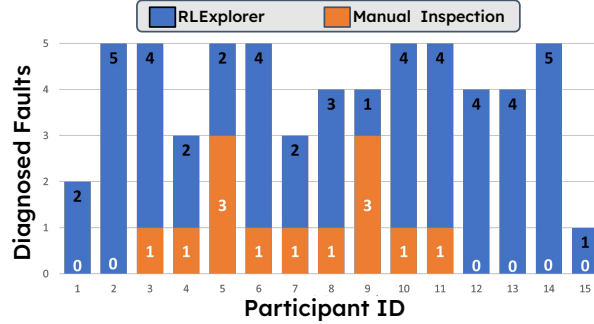


Figure 4.2 Number of diagnosed faults by each participant using the two debugging approaches.

test [175] and Vargha-Delaney ($\hat{A}_{12}$) [176]. The Wilcoxon statistical test is used to examine if a difference between two means is statistically significant (which corresponds to a p-value of less than 0.05). Vargha-Delaney $\hat{A}_{12}$ is frequently used to determine the magnitude of the difference between two groups. We used the Wilcoxon statistical test, a non-parametric test, to account for the likelihood that our participants' skill levels were not normally distributed due to differences in study level and background. Results show that this difference is statistically significant (p-value = 0.002) with a high effect size ($\hat{A}_{12} = 0.91$). Furthermore, participants, when using *RLExplorer*, were able to diagnose faults in less time ($\mu = 216.3 \text{ sec}, \sigma = 125.8$) compared to the baseline condition (i.e., manual debugging) ($\mu = 426.7 \text{ sec}, \sigma = 203.1$). This difference, once again, is statistically significant (p-value = 0.03) and has a large effect size ($\hat{A}_{12} = 0.18$).

Figure 4.2 shows the number of faults diagnosed by each participant using the two debugging approaches. Six individuals (40% of all participants) have failed to manually diagnose any faults. Also, no participant managed to find all faults using manual debugging alone. This demonstrates that manually debugging DRL code is very hard. On the other hand, all participants have managed to uncover at least one fault using *RLExplorer*. Nevertheless, only 47% of all participants were able to find all 5 faults by the end of the coding exercises.

This is mostly owing to the limited time given to debug the code similar to previous studies [91]. Figure 4.3 shows the participants’ replies to three 5-point Likert scale questions about the usability of *RLExplorer*. Participants were asked about their general impression of the tool, if they would consider including the *RLExplorer* in their development process, and if *RLExplorer* was easy to integrate. The replies from participants validate *RLExplorer*’s effectiveness and usefulness. Participants reported high satisfaction with the debugger and a high likelihood of leveraging *RLExplorer* in their development. Also, participants’ replies confirm the ease of integration of *RLExplorer* in DRL applications.

In addition, in the survey, we described the checks contained in *RLExplorer* and asked the participants to rate the relevance and correctness of the proposed check on a 5-point Likert scale (1= low relevance, 5= high relevance). Figure 4.4 summarises the replies of the participants. Overall, participants found all checks relevant ($rank > 1$) with an average rating of all checks equal to 4.4 and a standard deviation equal to 0.69. On average, all checks are considered highly relevant (rank=5) by 47% of participants, moderately relevant (rank=4) by 46%, neutral (rank=3) by 5%, and slightly relevant (rank=2) by 2%. Surprisingly, the Loss check was the check with the highest average rating ($\mu = 4.57$) and was highly relevant to 72% of participants. The common belief was that analyzing the loss in DRL is not that important compared to DL [177]. Yet, interviews with DRL experts and survey results demonstrate that checking the loss in DRL could be insightful. On the other hand, the Bias check was the check with the lowest average rating ($\mu = 4$) and was highly relevant to only 20% of participants. This relatively low rating can be explained by the relatively low impact of NN-bias problems on the learning performance of DRL systems. Nevertheless, multiple participants still find this check important and hard to manually detect. In fact, the correct fine-tuning and initialization of bias values could significantly improve the learning performance of DNNs.

Finding 3: All participants expressed high overall satisfaction with *RLExplorer*, 92% found it easy to use, and 86% expressed a strong willingness to use the tool in the future.

4.3 Discussion

4.3.1 Participants’ Feedback

During coding interviews, we asked participants to provide open-ended feedback on *RLExplorer*’s positive and negative aspects and what might be improved. We discuss below the key participant feedback received during the human evaluation.

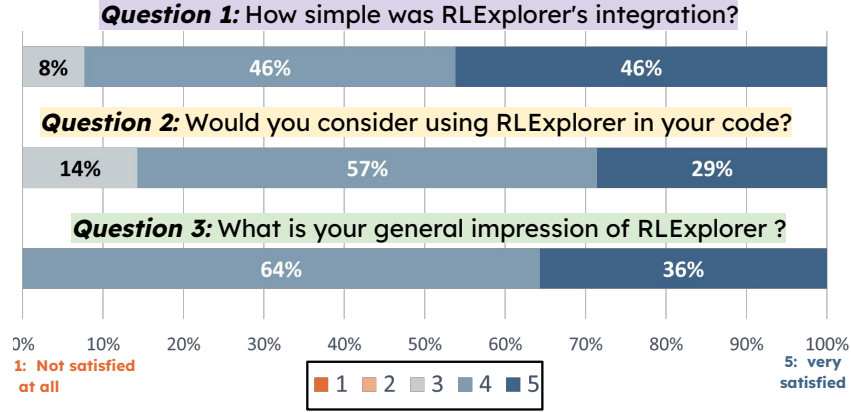


Figure 4.3 5-point Likert Scale Chart illustrating the feedback of participant on *RLE Explorer*.

***RLE Explorer* checks expose silent bugs and reduce debugging time.** Many participants in the study expressed their views on the challenges associated with debugging DRL code and commented on the difficulty of identifying DRL bugs due to their silent nature, lack of explicit code failure and the high complexity of DRL systems. Therefore, participants agreed that advanced approaches like *RLE Explorer* are essential to help developers save time and enhance the overall DRL development experience and highlighted how *RLE Explorer* can alleviate these challenges by simplifying and expediting the diagnosis process.

***RLE Explorer* provides insightful warnings.** Several participants emphasized the relevance of *RLE Explorer*'s warnings and how they can facilitate the identification and localization of faults. Participants also expressed their appreciation for *RLE Explorer*'s ability to monitor and display internal signals (e.g., state entropy) during the training process in real time. One participant stated that *RLE Explorer*'s plots are particularly advantageous because even in cases where warnings may produce false positives, monitoring can still provide insights into the overall learning behavior as it occurs. However, other participants requested modifications to the way the warnings were presented. These enhancements will be discussed in the next Section.

***RLE Explorer* was easily integrable into the DRL code.** *RLE Explorer*'s integration into a DRL application was meant to be easy and straightforward, requiring only a few steps and lines of code. This was validated by participants as they responded to a 5-point Likert scale question indicating high ease of integrating *RLE Explorer* into the DRL application ($\mu = 4.43$). Participants' feedback was similarly encouraging, with many expressing that the simplicity of integration of *RLE Explorer* would boost its chances of being embraced by the RL community.

***RLE Explorer* explicitly states that some checks might be inaccurate in some conditions.** DRL systems are black box and highly stochastic. Proposing a silver bullet check

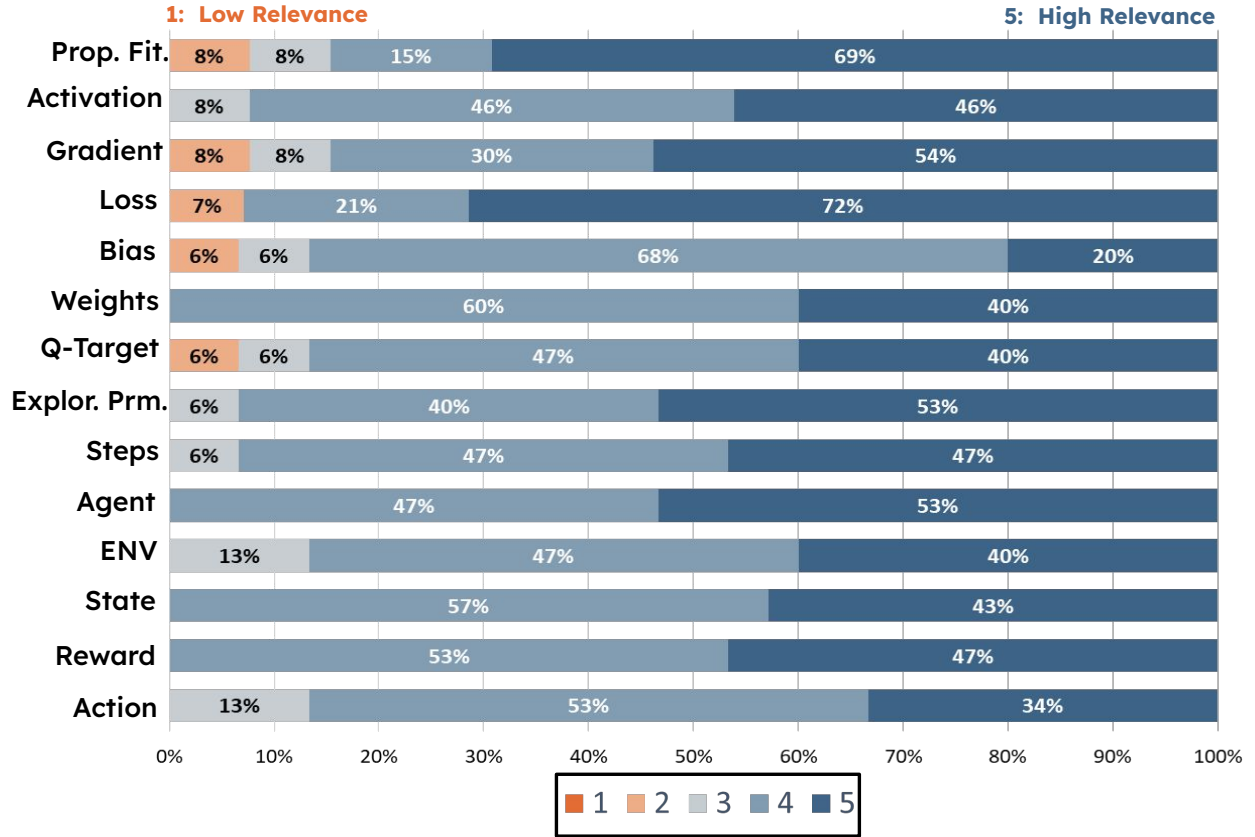


Figure 4.4 5-point Likert Scale Chart illustrating the feedback of participants on the relevance *RLExplorer*’s Checks.

that works in all circumstances is likely impossible. We clearly explain that *RLExplorer*’s check may fail in some particular cases and may produce inaccurate warnings. The participants were aware that *RLExplorer* might overlook faults or generate false positives. They still feel that *RLExplorer* is useful. To mitigate this problem, we offered, when writing warning messages, several indications in circumstances when there is no one sure root cause. Additionally, warnings like “high epistemic uncertainty” and “rise in the state entropy” can still be valuable even if the training process is normal and the agent was able to realize good average reward values.

4.3.2 Limitations and Future Work

In this section, we discuss the limitations of *RLExplorer* as well as directions for future work. *RLExplorer* may raise false warnings (false positives) and miss errors (false negatives). Due to the stochasticity of the DRL training, *RLExplorer* may raise false positive warnings. For example, if repeated episodes result in a low average reward value but the

global average reward is stable, the “fluctuating reward during exploitation” check might be raised. *RLExplorer* may also overlook errors (false negatives). Checks were designed to apply to general circumstances, however, these checks may not generalize to certain scenarios. Checking the Markov property of the RL environment [46, 178], for example, might be a beneficial verification for specific use cases. A deeper analysis can be used in future revisions of *RLExplorer* to improve and enhance some checks.

Mappings from warnings to root causes may not always hold. DRL systems are black box and highly stochastic and there are often several root causes that might result in a common error symptom. For example, a “fluctuation in the state entropy” indicates unstable learning, however, this unstable learning might be caused by a variety of factors.

***RLExplorer* may not generalize to new DRL algorithms.** Research is always producing new sorts of DRL algorithms and concepts, which may necessitate new debugging checks and strategies. While we rely on recent work [83] to grasp the taxonomy of common DRL faults, DRL is continuously evolving, and the error trends may shift over time. We mitigate this limitation by making *RLExplorer* support the creation of custom checks that can be implemented to handle a specific problem.

Participants suggested improvements to *RLExplorer*’s warnings presentation. Participants appreciated the explanations of the warning messages as they were detailed and provided actionable steps. Despite the relevance and usefulness of *RLExplorer*’s warning, participants suggested the inclusion of, summaries, HTML reports, or even web interfaces to make the debugging process more easier and appealing.

4.4 Threats to Validity

In the first part of the evaluation, we extracted real faulty samples from SO. In certain cases, the code was slightly adapted (e.g., moving from TensorFlow to PyTorch, updating obsolete packages), which may alter its logic. To mitigate this threat, two authors separately cross-validated each adapted code and reached an agreement. Another potential threat is the small sample size of participants in the human evaluation. However, the participants were researchers with good RL experience (3 years of experience on average), indicating their reliable evaluation abilities. Furthermore, our methodology matches previous studies [91] that interviewed a comparable number of participants. Finally, while developing *RLExplorer*, we used parameters specified by past works [89, 136] or thoroughly fine-tuned across multiple agents/environments. These selected values may not work in some circumstances. To mitigate this threat, we offered users to easily change these parameters for their specific usage.

4.5 Chapter Summary

Detecting faults during the training phase of DRLs is a challenging task due to the complex architecture and black box nature of DRL systems. Moreover, there is a lack of tools specifically designed for DRL to detect errors dynamically during the training of DRL applications. To address this issue, we propose *RLExplorer*, which is the first fault diagnosis approach specifically designed for DRLs to help developers detect symptoms and root causes of faults during training. *RLExplorer* performs multiple verification routines to identify faulty behaviors through analyzing the dynamic traces collected during training. When a fault is detected, *RLExplorer* provides an error message explaining the issue and proposes potential solutions to fix the problem. We conducted an evaluation of *RLExplorer* using erroneous DRL algorithms collected from Stack Overflow. The evaluation results indicate that our approach successfully diagnosed faults in 83% of the cases. Furthermore, we conducted a human study involving 15 participants to assess the effectiveness of *RLExplorer* in assisting developers with fault diagnosis. The study revealed that participants were able to diagnose 3.6 times more faults using *RLExplorer* compared to manual debugging. Additionally, participants expressed high satisfaction with the debugger and expressed a strong inclination to incorporate *RLExplorer* into their development processes.

CHAPTER 5 DR.DRL : AN INTENTIONAL FORGETTING-DRIVEN SELF-HEALING METHOD FOR DEEP REINFORCEMENT LEARNING SYSTEMS

DRL is currently gaining traction in comparison to other DL approaches owing to its appealing autonomy and reward-driven characteristics. Theoretically, the continuous exposure of the DRL agent to a newly-shifted environment for enough time leads to adapting its logic to cope with the induced behavioral changes. In practice, the capacity for continual learning in DRL, known as “plasticity” [40], is compromised by several emerging inefficiencies, such as catastrophic forgetting [66], failure of warm-starting [179], and slow adaptation. Furthermore, other studies have shown further weaknesses of CL in DRL, such as Primacy Bias [180], and Capacity Loss [181]. A recent study [40] examined the plasticity of DRL-based systems given different simulations of environmental shifts. Using an environmental shift simulation with varying changes in the environment’s parameters [182], the plasticity boundaries of DRL agents can be inferred and depicted. This environmental shift reflects the natural dynamic evolution of the environment, mirroring the DRL application in real-world production contexts. The capacity to constantly learn, adapt, and evolve is arguably one of the most important properties of DRL-based systems. One interesting venue of research is to investigate how to go beyond continual learning frontiers and provide an improved approach that can allow an agent to adapt more efficiently to changing environments. Due to the DL high capacity models and the RL exploration propensity, the use of a DRL agent trained in the original environment as a starting point for vanilla CL overestimates the transferability of previously-acquired knowledge and may mislead the fine-tuning steps towards unnecessary upgrade trials of minor behaviors strongly connected to the original environment [183]. As evidenced by its failure to handle the shifted environments, some of the agent’s acquired behaviors are closely related to the original parameters of the environment, and even though the environment task remains the same, such behaviors could not be transferred. In this context, one could think of the intentional forgetting concept [184, 185] that allows artificially-intelligent cognitive systems to deliberately discard irrelevant information with the aim of focusing attention on pertinent aspects of the solution.

In this chapter, we propose, *Dr.DRL*, a DRL self-healing approach leveraging intentional forgetting to reset hypoactive neurons with low-scaled weights in order to erase the agent’s minor behaviors before starting to fine-tune it in the drifted environment. One main advantage of selective weight forgetting is its ability to facilitate the process of unlearning minor behaviours that are linked to hypoactive neurons. In addition, the practise of under-scaling

initialization offers the advantage of continual learning at dual speeds. More specifically, major behaviour neurons, which remain unchanged, receive larger updates based on gradients compared to minor behaviour neurons, which are subjected to under-scaled updates.

In order to assess the efficacy of our proposed *Dr.DRL* approach, we conduct an evaluation using purposefully drifted gym environments [34] that exhibit varying degrees of drift intensity. The results indicate that the use of *Dr.DRL* effectively enhances the efficiency of the adaptation process, as evidenced by reduced runtime and a decreased number of fine-tuning episodes, in comparison to the conventional continual learning (CL) approach. *Dr.DRL* enhances, also, the plasticity frontiers available in CL by an average of 20%. Finally, *Dr.DRL* demonstrates the capability to efficiently maintain and, in certain instances, enhance the mean reward of CL by a notable margin of 45%.

The remainder of this chapter is structured as follows.

Section 5.1 provides a concrete example of the problem we are addressing and how we are tackling it. Section 5.2 describes each step of our approach. Section 5.3 reports the evaluation outcomes. Section 5.4 analyzes the threats to the validity of our approach.

5.1 Motivating Example

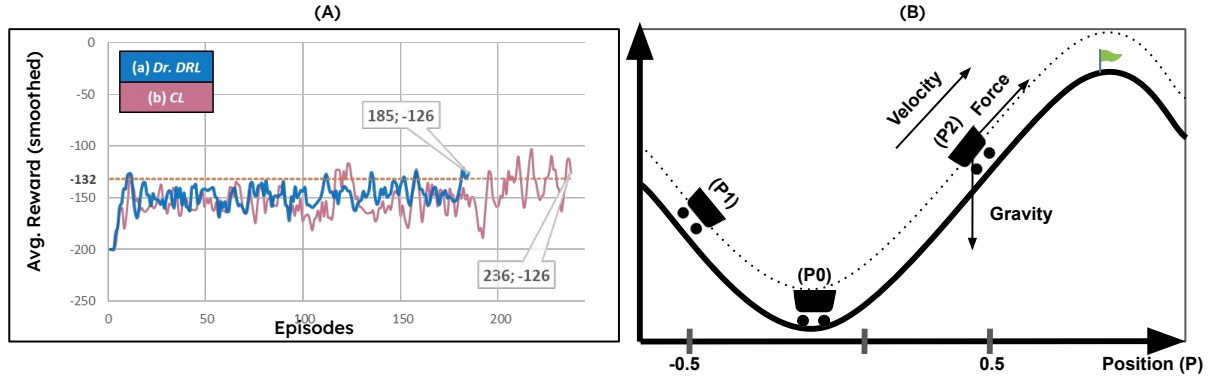


Figure 5.1 (A) The adaptation curves for the DQN agent’s average reward per episode on MountainCar: (a) depicts the curve of the average reward when *Dr. DRL* is used, whereas (b) represents the curve of average reward when CL is used (B) Illustration of the MountainCar environment as described in [1]

Figure 5.1 shows the MountainCar environment [1], where a car starts on a one-dimensional track situated between two mountains. The objective is to reach the goal position up the right mountain. The car starts at the bottom of the valley, and receives negative rewards, as

each timestep passes, to encourage it to reach the goal as soon as possible. MountainCar’s environment is episodic, where episodes end when 200 timesteps have passed or when the car reaches the flag. In order to solve the MountainCar, the agent must achieve an average reward of at least -110 over 100 consecutive evaluation episodes. After drift, we assign 20% of tolerance in the average reward. So the agent must achieve at least -132 .

Drifts in the MountainCar environment can affect three parameters, namely, force, gravity, and goal velocity. Originally, the initial settings of a MountainCar environment, E , are set as follows: force = $1e^{-3}$, gravity = $2.5e^{-3}$, and goal_velocity = 0. A drift d on E , for instance, can shift these parameters to force = $1.2e^{-3}$, gravity = $4e^{-3}$, and goal_velocity = 0, leaving the agent unable to solve the drifted environment E_d . CL would be used to fine-tune the trained agent on E_d for a set number of episodes. Nevertheless, CL is prone to inefficiencies such as catastrophic forgetting [66] and slow adaptation, resulting in inferior DRL system performance and higher repair costs. A trained DQN agent, for instance, took 236 training episodes and 436 seconds to adapt to E_d leveraging CL (Figure 5.1). Our proposed approach, *Dr.DRL*, optimizes cost-effectiveness, thereby reducing adaptation expenses. Considering the same E_d , *Dr.DRL* adapted the same DQN agent in 185 episodes and 273 seconds. The DQN agent also achieved an average reward of -122.3 when adapted with *Dr.DRL*, compared to -123.7 with CL. Finally, as evidenced by Figure 5.1, *Dr. DRL* improved the stability and monotonicity of the adaptation process by minimizing the fluctuation breadth of the average reward.

5.2 Approach

In this section, we describe the problem statement and the drifting phenomenon in RL environments. Then we present our healing tool, *Dr.DRL* which employs the intentional forgetting mechanism to improve adaptation efficiency. Finally, we describe *Dr.DRL*’s workflow.

5.2.1 Problem formulation

Let E represent an RL environment comprising a set of n parameters $\{p_1, \dots, p_n\}$, and A_E denote the RL agent trained on E until the environment is solved or reaches the desired performance level. Consider a new environment, E' , with parameters $\{p'_1, \dots, p'_n\}$. E' is the RL environment which will be drifting from E over time, due to its natural non-stationarity. While A_E is trained on E , it should ideally possess the ability to solve E' or maintain its performance level, as long as the parameters of E' have not significantly deviated from those

of E . However, in the event of a notable environmental drift, the performance of A_E declines in proportion to the deviations i.e. the higher the drift the lower the agent’s performance. Generally to adapt the agent to the drifts that occur in the environment, continual learning (CL) is employed. The CL will help derive, the new RL agent, A'_E , that is obtained by training A_E continually on E' for a certain number of episodes over a period of time T . When the agent A'_E actually solves the environment E' (or achieves a desired level of performance in E'), the process of healing is successful; otherwise, it is unsuccessful.

Assuming that a trained agent A_E in an environment E learns some major and minor behaviors that are both necessary to complete the task. The inefficiencies of vanilla CL, including catastrophic forgetting and slow adaptation, can be due to the equal importance accorded to learning behaviors of both types: majors and minors. In fact, major behaviors encompass crucial problem-solving abilities, whereas minor behaviors do not directly contribute to problem-solving but were necessary to achieve the desired performance during the training. In a drifted environment, continual learning of A_E can spend many episodes attempting to adapt minor behaviors that are not necessarily transferable given the environmental parameter shifts, whereas continual learning should prioritize the updates of major behaviors to adapt the agent’s problem-solving skills in light of the drifted environment’s specifics.

In our proposed self-healing approach, we improved continual learning approach through removing untransferable specificities of the initial environment and allowing the updating of major behaviors to take precedence over learning minor behaviors. In this way, it is possible to avoid, simultaneously, unnecessary adjustments to minor behaviors and slow-paced continuous updates to major behaviors. Importantly, an accelerated and focused self healing approach can boost the performance of adapted agents and help solve drifted episodic environments where the vanilla CL version cannot reach the solution before the allowed episodes run out.

5.2.2 Intentional Forgetting Mechanism

The phenomenon of winning lottery tickets has been shown [186, 187] to exist in deep neural networks, i.e., there are sub-networks that can be further fine-tuned to achieve as good performance as their source neural network. An example of its implications is the growing trend of reducing the size and resource consumption of DNNs by pruning irrelevant neurons after training [188]. In line with the lottery ticket hypothesis, the major DRL agent behaviors are likely to be yielded by a sub-network within the agent’s DNN. This sub-network encodes the critical problem-solving abilities responsible for achieving the major task. More precisely, this sub-network encapsulates the set of hyperactive neurons (i.e., major-behavior-

responsible units) that are expected to represent the essential agent skills to resolve most of the occurring environment states and predict the optimal action. We hypothesize that the DRL agent’s major behaviors (i.e., set of hyperactive neural network units) should be primarily adjusted by the DRL self-healing process against a drifted environment, while the minor behaviors are more likely to be non-transferable and should be learned again. Indeed, minor behaviors are likely to have no direct impact on problem-solving but are necessary to achieve the desired performance in a specific environment. Therefore, we adopt the concept of intentional forgetting [184,185] that removes irrelevant knowledge to improve the cognitive capabilities of intelligent systems by allowing them to focus solely on pertinent aspects of the given situation. In the following, we describe the two main steps of our intentional forgetting mechanism for the enhancement of the DRL agent’s healing ability and its associated continual learning stability.

Detection of Minor Behavior Regions

The first step of our intentional forgetting mechanism is to localize the regions of the DRL system responsible for the major behavior (i.e., contributing almost to all of the predictions) and minor behavior (i.e., contributing intermittently to the predictions). Structurally, the RL agent encapsulates a neural network, thus, we propose a neuronal categorization method to depict the hyperactive neurons (i.e., major-behavior-responsible units) and the hypoactive neurons (i.e., minor-behavior-responsible units). This neuronal categorization method provides a structural representation of the problem-solving skills acquired by the RL agent.

A neural network’s activations represent the intermediate computations that pass essential information between layers. The activations are computed using a nonlinear function attached to each neuron, which determines whether it should be activated (“fired”) or not, based on the relevance of neuron output for the final prediction. As illustrated in Equation 5.1 of the forward pass [189], all the activations contribute to the last layer activation, $a^{[L]}$.

$$a^{[L]} = \sigma \left(W^{[L]} g \left(\dots g \left(W^{[1]} a^{[0]} + b^{[1]} \right) \dots \right) + b^{[L]} \right) \quad (5.1)$$

Where $W = \{W^{[l]}, \forall l \in [1, L]\}$ and $b = \{b^{[l]}, \forall l \in [1, L]\}$ are the weights and biases of a DNN.

Hence, the neurons with higher activation levels, called hyperactive neurons, encode the representation of relevant features associated with major behaviors at that processing layer and thus play a significant role in the overall predictions. In contrast, hypoactive neurons, char-

acterized by lower activation levels, encode minor behavior features, but they still contribute to the model’s generalizability and confidence. Hyperactive neurons (i.e. neurons with high levels of activation) have been found to provide beneficial learning patterns for DNNs in several empirical studies [50, 190]. The concept of identifying hyper-hypoactive neurons was inspired by the Top-K-Neuron-Coverage (TKNC) technique [50]. The TKNC was developed to assess the diversity of activation patterns during DNN testing. When computing, TKNC, neurons with higher activation levels are deemed to be more significant in processing the input and are ranked higher, whereas neurons with lower activation are ranked lower.

Therefore in *RLExplorer*, to categorize the hypoactive and hyperactive neurons, we select the N_H hypoactive neurons per layer, i.e., those triggering low activation scores on a regular basis. According to the following formula, N_H is a relative number derived from a prefixed forget rate, F_r : $N_H^l = \text{Round}((F_r/100) * N_l)$, where N_H^l represents the number of hypoactive neurons in layer l and N_l is the number of neurons in layer l .

Forgetting & Dual Speed Gradient-based Healing

After detecting the major and minor behaviors of the DRL system, *RLExplorer* intentionally erases its minor behaviors and proceeds to dual-speed gradient-based healing. We assume that the inefficiency of CL may be partially due to the equal importance assigned to both major and minor behaviors, resulting in slow adaptation (all neurons are equally updated) and catastrophic forgetting (CL may reverse the neurons’ roles from major to minor, or vice versa). Hence, our approach offers two improvements to CL. As a first benefit, forgetting the minor behaviors that are likely to be non-transferable to the drifted environments, is expected to increase both the speed and effectiveness of major behavior adaptation. Second, the minor behavior forgetting mechanism enables self-healing at dual speeds, i.e., major behavior units would receive significant and more frequent updates than minor behavior units. Nevertheless, nullifying them would negatively affect the layer’s weight asymmetry, and the loss gradients w.r.t null weights would become zero as well; so their corresponding neurons freeze and no longer receive updates. Hence, our proposed method to properly forget the minor behaviors involves the re-initialization of their associated neurons using the original weight initializer. Indeed, random weights are sampled to maintain the asymmetry between neurons and calibrate the values distribution variance proportional to the size of the layer’s input. Furthermore, we under-scale these random weights generated for hypoactive neurons using a scale rate, $S_r < 1$.

$$\frac{\partial \mathcal{L}}{\partial z^{[l]}} = [W^{[2]}]^T \dots [W^{[L]}]^T \cdot \frac{\partial \mathcal{L}}{\partial z^{[L]}} * q^{[L-1]} * \dots * q^{[1]} \quad (5.2)$$

$$W^{(i+1)} = W^{(i)} - \eta \frac{\partial \mathcal{L}}{\partial W^{[l]}} \quad ; \quad b^{(i+1)} = b^{(i)} - \eta \frac{\partial \mathcal{L}}{\partial b^{[l]}} \quad (5.3)$$

Equations 5.2 and 5.3 show how the parameters of the model are updated at the iteration $i + 1$, where $\mathcal{L}$ is the loss function; $z^{[l]}$, $W^{[l]}$, $b^{[l]}$ are respectively the weighted-sum outputs, weights and biases of a layer l and $q^{[l]} = g^{[l]'}(z^{[l]})$ with $g^{[l]'}$ referring to the computed gradients in the layer l .

Equation 5.2 shows that the gradient-based updates include multiplication by weights, which makes their initial magnitude scale affect their growth and decay over iterations, thus controlling their learning speed. If S_r tends to 0, new weights would be very small, almost zeros, so their associated neurons would experience low update rates. If S_r tends to 1, minor behavior neurons will have original scale weights and learn at the same rate as major behavior neurons.

Hence, the reduction of weights magnitude affects directly the scale of gradient updates derived by backpropagation through layers. Lowering the magnitude of the reinitialized neurons' weights ensures the slow re-learning of the DRL agent's minor behavior, whereas maintaining the original magnitude of the retained neurons' weights guarantees substantial updates of the RL agent's major behaviors for stable and fast adaptation. Therefore, our proposed intentional forgetting mechanism leverages the weight-gradient scaling relationship to enable dual-speed gradient-based healing of the DRL agent's behaviors.

5.2.3 Self-healing Workflow

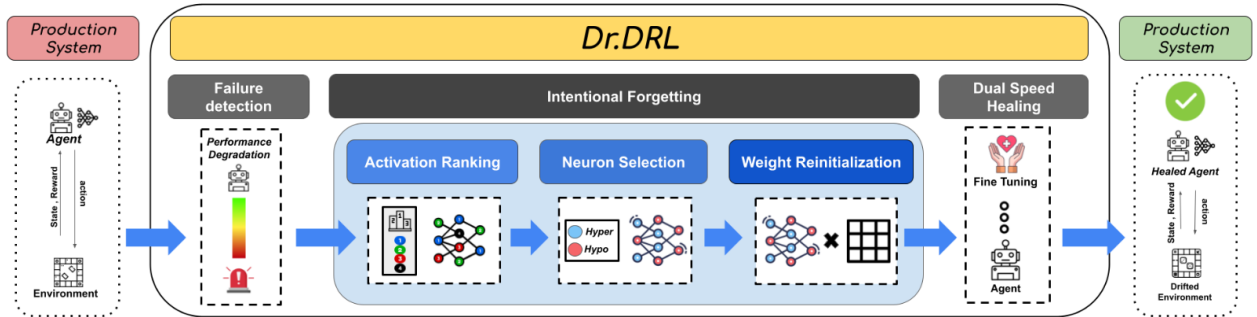


Figure 5.2 Illustration of *Dr.DRL* Pipeline

Figure 5.2 provides an overview of the healing workflow implemented in *Dr.DRL*. Algorithm

2 presents the step-by-step our approach.

In the following paragraph, we provide detailed explanations of the workflow steps and how developers can leverage *Dr.DRL* to mitigate performance drop issues:

During the training phase, the DRL agent learns to optimize its actions based on the states encountered in the original environment, aiming to maximize the cumulative rewards [191]. To be able to use *Dr.DRL* appropriately, it is crucial to collect activation traces during the training phase to later rank the importance of each neuron and assess its contribution to the decision-making process, as discussed in Section 5.2.2.

Once the DRL agent is trained, tested, and started operating in the real environment. It is expected that the environment parameters will be naturally shifting [60,61]. Upon a certain level of parameter shifts, we expect the trained agent to lose its ability to predict optimal actions, which means the agent cannot reach the required minimum reward. In that case, the agent logic needs to be adapted to the new execution context in order to solve the drifted environment, which can be solved using *Dr.DRL*.

In *Dr.DRL*, to automate the detection of performance degradation, we have incorporated a proactive event handler that continuously monitors the agent’s performance in the drifted environment. Once the performance degradation exceeds a user-configurable threshold, the event handler triggers the healing mechanism of *Dr.DRL*. This mechanism aims to adapt the agent as quickly as possible to the drifted environment, mitigating the performance drop.

The healing mechanism starts by categorizing neurons into minor (hypoactive) and major (hyperactive) neurons. This categorization relies on the activation traces collected during training in the original environment. Within each layer, neurons are ranked based on their activations. The hypoactive neurons are the ones that exhibit lower activations across many observations (lines 2 to 6 in the algorithm). More specifically, the hypoactive neurons within each layer constitute the minor behavior neurons, and their counts are determined according to the prefixed F_r value (check section 5.2.2). Subsequently, the weights associated with the minor behavior neurons are reinitialized using the same weight initializer used during the training, but with an underscaling factor of S_r . This scaling prioritizes the updates of major behavior neurons over the minor ones. Then, hyperparameters specific to the DRL algorithm training, are adjusted accordingly, particularly those controlling the exploration-exploitation trade-off. Additionally, in the case of the off-policy DRL algorithms like DQN and SAC, we restore the replay memory obtained from the previous training on the original environment and ingest them in the replay buffer to preserve the initial knowledge of the agent and avoid catastrophic forgetting issues. Once the setup is complete, the agent’s fine-tuning process in the drifted environment starts (lines 12 to 14). The goal is to enable the agent to adapt at dual speed to the environmental drifts encountered, ultimately achieving a comparable level

of performance as that in the original environment.

Algorithm 2: Dr. DRL’s healing mechanism

Input: $E' = \{p'_1, \dots, p'_n\}$ drifted environment, A_E : trained agent that solves $E = \{p_1, \dots, p_n\}$, n_t : training episodes, obs : set of training observations, F_r : Forget rate, s_r : Scale rate, $init$: weights initializer, hyp : hyperparameters of the agent.

Output: $A'_{E'}$: agent adapted to the drifted environment E'

```

1:  $mask := []$ ;
   /* Detection of Minor Behavior : */
2:  $acts := \text{compute\_activation\_trace}(A_E, obs)$ 
3: for  $i$  in  $A_E.hidden\_layers$  do
4:    $hypo\_n := \text{detect\_minor\_regions}(A_E, obs, F_r)$ ;
5:    $masks.append(\text{generate\_mask}(A_E, i, hyp, hypo\_n))$ ;
6: end for
   /* Intentional Forgetting of Minor Behavior : */
7: for  $i$  in  $A_E.hidden\_layers$  do
8:    $w := \text{apply\_mask}(weights\_mask, A_E)$ ;
9:    $new\_w := \text{reinitialize}(w, masks[i], s_r, init)$ ;
10:   $A_E.set\_weights(new\_w)$ 
11: end for
   /* Dual Speed Gradient-based Healing : */
12: for  $eps$  in  $n_t$  do
13:   $A'_{E'} := \text{dual\_speed\_cl}(A_E, E', hyp)$ ;
14: end for
15: return  $A'_{E'}$ 

```

5.3 Evaluation

In this section, we first introduce the DRL case studies (environments and algorithms), as well as our experimental settings, metrics, and procedure. Next, we evaluate *Dr.DRL* against vanilla CL in terms of cost reduction (RQ1), adaptability enhancement (RQ2), and reward improvement (RQ3).

5.3.1 Experimental Setup

Environments

The MountainCar environment [1] was described in the previous section (see Section 5.1), along with the parameters we vary to simulate environment drift.

The CartPole environment [43] is an inverted pendulum connected to a cart by an unactuated joint. The cart moves on a track and it’s a subject to a forces with equal magnitudes

that can either push it in the right or left direction. Keeping the pole upright is the goal of moving the cart; so the agent gets +1 every time the pole stays upright. During the training, an episode ends when the pole falls over 15° , the cart moves 2.4 units from the center, or 200 time steps pass. An agent is considered to have solved a task when their average reward over 100 testing episodes is greater than 195. The CartPole environment can be parameterized using four parameters: masspole, length pole, mass cart, and friction.

The Acrobot environment [44, 192] has two joints and two links where the joint between the two links can be actuated. Initial position of the system is with the links hanging downwards and the agent aims to raise the lower link to a given height. If the agent reaches its goal, it gets a reward of 0; otherwise, it gets -1 at every timestep. Episodes in Acrobot environment end when the goal or 500 time steps are reached. There is a reward threshold for the Acrobot environment that is -100 over 100 evaluation episodes. The environment parameters include 'link length 1' (length of the first link), 'link compos 1' (position of the center of mass of the first link), 'link mass 1' and 'link mass 2' (mass of both links).

DRL Algorithms

In the following, we present the selected DRL algorithms for our evaluation. They are widely used and most RL libraries provide a stable implementation of them.

Deep Q-Learning (DQN) [45] is a popular value-based method that has evolved over the years with a variety of optimizations [193]. DQN leverages neural network to approximate the action-value function, Q . Iterative learning of Q does not guarantee convergence, instabilities leading to divergence can be addressed by two techniques: using a buffer to replay older experiences and using two DNNs to stabilize the learning process. DQN uses an ϵ -greedy policy, where the agent selects the optimal action with probability $1 - \epsilon$, or a random action with probability ϵ . Furthermore, ϵ is linearly annealed from 1.0 to 0.1 for K steps. Hence, the agent explores much more in the early training phase, while exploitation occurs more at the end.

Proximal Policy Optimization (PPO) [42] is a state-of-the-art algorithm whose goal is to improve the policy as much as possible without compromising performance. For stable policy updates, PPO uses clipping mechanism to prevent too much change to the policy and a hyperparameter, ϵ , that controls how much it can diverge from the last policy. Instead of learning from current experience and discarding past experiences, PPO is designed to reuse even old experiences, which makes it more sample efficient. PPO trains a stochastic policy that explores the environment in its initial phases of training, but gradually it becomes more deterministic and encourages the agent to exploit rather than explore.

Soft Actor Critic (SAC) [41] focuses on maximizing returns and policy entropy (i.e. de-

gree of its "randomness"), simultaneously. SAC also performs its updates off-policy, similar to DQN. More precisely, to encourage exploration, SAC regulates the entropy of the policy through a coefficient α , which can be fixed manually or adjusted automatically over the training. In our tests, we use the auto-adjusted version of the entropy regularization coefficient α .

Software and Hardware Configuration

We implemented our approach as an open-source tool using Python 3.7 [194] and it supports Tensorflow (version 2.4.4) [195]. We evaluated environments from the Gym library (version 0.23.1) [34] and DRL algorithms adapted from popular GitHub repositories [196, 197]. We used two GPU-enabled servers. The first server runs Ubuntu 20.04 and features a 24-core AMD EPYC 7413 CPU, an ASPEED AST2500 GPU, and 500 GB of RAM. The second runs CentOS 7 and has a 12-core 1.70 GHz Intel Xeon Bronze CPU, an NVIDIA GeForce RTX 2080 Ti GPU.

Evaluation Procedure and Metrics

Procedure: The DRL algorithms' hyperparameters were tuned using the open-source repository rl-baselines-zoo [198]. As minor implementation changes may substantially affect a DRL algorithm performance [199], we conducted ten distinct training runs with various random seeds. The final hyperparameters for each DRL algorithm were chosen based on the highest average reward. For each DRL algorithm, we sampled six realistic environmental drift settings. To simulate different shifts in production, we intentionally induce drifts of varying magnitudes. For distinguishing between intensities of simulated drifts, we rely on the environment's parameter ranges described in [40]. Then, we use either our approach, *Dr.DRL*, or vanilla CL to heal the DRL agent to the six newly-drifted environments. To have a fair comparison, we used the same hyperparameters and max number of episodes of the DRL agent beforehand. Nevertheless, we tuned the specific parameters, F_r and S_r , of *Dr.DRL* for each environment using, respectively, the ranges of [50, 40, 30, 20, 10]% and [0.00001, 0.0001, 0.001, 0.01, 0.1]. We found that the best parameters are $F_r=50\%$ and $S_r=0.1$ for the CartPole environment. For both Acrobot and MountainCar environments, $F_r=10\%$ and $S_r=0.0001$ work better. The difference in the parameters of *Dr.DRL* can be explained by the level of difficulty. The easiest environment here is CartPole. MountainCar and Acrobot pose more challenges due to their deceptive natures. In fact, these deceptive reward environments discourage agents from exploring (by offering a negative reward), causing them to fall into

local optimality. Following the guidance of the prior study [40], we assign an adaptability tolerance ratio of 20% to these two environments. As a result, we accept a performance degradation of 20% during the adaptation. For instance, in the MountainCar environment, the agent adapts to the drifted environment if it gets an average reward of -132 , corresponding to -110 (solved environments) minus 20% (see Figure 5.1(A)).

Finally, all experiments are run 10 times to overcome the stochastic nature of these DRL algorithms. We use statistical significance testing, i.e., the non-parametric Wilcoxon [175] test and the Vargha-Delaney A12 effect size [176], to compare the results obtained by *Dr.DRL* and vanilla CL.

Environmental Drifts: On the basis of the study [40] on the plasticity of DRL agents, two types of drifted environments are identified: 1) an adaptable environment to which the agent will be able to adapt, and 2) a non-adaptable environment to which the agent cannot adapt. Hence, we consider both types of drifted environments. For the adaptable type, our goal is to evaluate the stability of our self-healing approach, as well as its ability to heal the behavior of the RL agent faster and more economically. Regarding the non-adaptable type, we aim to investigate how well our self-healing approach improves the DRL agent’s adaptability (healing ability) and enables its adaptation despite the failure of CL.

Metrics: Below, we introduce the different evaluation metrics that have been used in the empirical evaluations.

Increase Ratio ($\%IR(X)$) / *Decrease Ratio* ($\%DR(X)$). It consists of the percentage increase/decrease of the quantity X from applying *Dr.DRL* rather than CL, as formulated in Eq. 5.4/Eq. 5.5. Thereby, positive values indicate the on-watch quantity, X , has known an improvement, whereas negative values indicate a degradation.

$$\%IR(X) = \frac{Dr.DRL_X - CL_X}{CL_X} \times 100 \quad (5.4)$$

$$\%DR(X) = \frac{CL_X - Dr.DRL_X}{CL_X} \times 100 \quad (5.5)$$

Adaptability Ratio ($\%AR$). It consists of the percentage of pairs (a trained DRL agent, a drifted environment) that a given approach Y is able to adapt, as formulated in Eq. 5.6.

$$\%AR(Y) = \frac{(Env, Agent)_{adpt(Y)}}{(Env, Agent)_{All}} \times 100 \quad (5.6)$$

5.3.2 Research Questions and Answers

RQ1 (Cost-effectiveness): How cost-effective *Dr.DRL* is in adapting DRL systems compared to continual learning?

Motivation: The purpose of this RQ is to compare *Dr.DRL* with the vanilla CL technique in terms of the cost-effectiveness of healing the DRL systems that failed to solve drifted environments. As the healing strategies operate on the DRL agent at runtime, delays in healing cycles can be resource-intensive and costly, and consequently, such operations are constrained by a maximum number of steps allowed in solving episodic environments. Thus, the more effective a healing strategy is, the shorter the time it takes.

Method: We measure the time and episode count necessary to heal a DRL system in response to a drifted environment. Both metrics are important to compare the cost-effectiveness of *Dr.DRL* and CL because one healing strategy can solve the drifted environment in fewer episodes while it reaches the maximum number of steps allowed in each episode. Hence, the time reflects the total number of steps done by the healing strategy. To consider a healed DRL agent as having successfully solved a drifted environment, it must achieve an average reward over 100 consecutive episodes greater or equal to the defined threshold (i.e., same as the original environment). To estimate the magnitude of the speed-up obtained when using *Dr.DRL* versus vanilla CL, we compute the decrease ratio, %DR (Eq. 5.5), of the fine-tuning time and episodes reached by *Dr.DRL* over the ones obtained by vanilla CL.

Results: Table 5.1 shows that *Dr.DRL* often requires fewer healing cycles as measured by both fine-tuning time and episode count than its conventional alternative, vanilla CL. Indeed, the bold values indicate that the difference between the two compared healing approaches is statistically significant (i.e., the p-value < 0.05). The underlined values indicate that their associated magnitude of the effect size A12 is medium or large. Furthermore, Figure

Table 5.1 Comparison of Healing Time and Fine-Tuning Episodes Decreases Using *Dr.DRL* over vanilla CL.

		DQN	PPO	SAC
Episodes DR (%)	CartePole	21.2	<u>45.9</u>	0.9
	MountainCar	17.7	2.9	20.1
	Acrobot	3.2	9.3	<u>38.3</u>
Time DR (%)	CartePole	<u>30.3</u>	-20.5	1
	MountainCar	20.3	<u>24.4</u>	<u>18.7</u>
	Acrobot	5.3	16.2	<u>73</u>

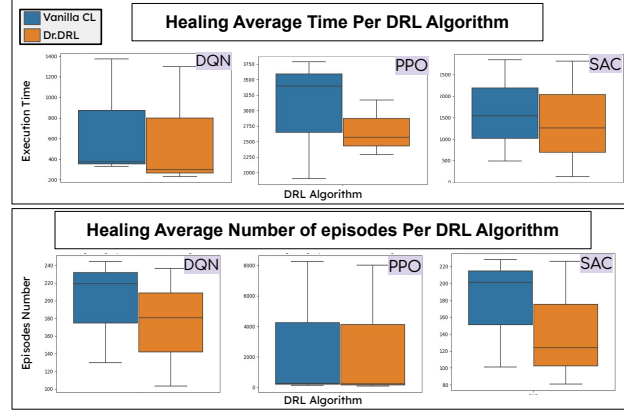


Figure 5.3 Comparison of Healing Time and Fine-Tuning Episodes Box Plots Using *Dr.DRL* over vanilla CL.

5.3 demonstrates the juxtaposition of box plots drawn from the distributions of time and episode count obtained by *Dr.DRL* versus vanilla CL when tried with different pairs of DRL algorithms and environments. In line with statistical tests, the median values of both time and episode count achieved by *Dr.DRL* are consistently lower than their counterparts yielded by vanilla CL.

To be more specific, *Dr.DRL* outperforms vanilla CL in terms of fine-tuning episode count across all the pairs of agents and environments, where the gap between them were statistically significant in 44% of experiments with either a medium or large effect size. In regards to healing time, *Dr.DRL* outperforms vanilla CL across all the pairs of agents and environments, except for the pair of "PPO" algorithm and "CartePole" environment. In fact, *Dr.DRL* succeeds in healing PPO-based agents against drifted CartePole environments using longer steps by episode than its conventional alternative, vanilla CL, which delayed the healing time by *Dr.DRL* even with very few fine-tuning episodes.

Finding 1. *Dr.DRL* enables a cost-efficient healing of DRL systems in drifted environments, through fewer fine-tuning episodes and shorter healing time.

RQ2 (adaptability frontiers): Can *Dr.DRL* enhance the adaptability of DRL systems to drifted environments?

Motivation: [40] demonstrated that vanilla CL has an environmental drift boundary beyond which it fails to heal the DRL system. *Dr.DRL* is designed with the objective of expanding the constrained scope of adaptable drifted environments. Hence, we assess the ratio of drifted

environments that cannot be resolved through CL, but are instead resolved by *Dr.DRL*, and vice versa.

Method: We compute the adaptability ratios (%AR) (Eq. 5.6) obtained by the two compared healing approaches, *Dr.DRL* and vanilla CL, for all the studied agent-environment pairs. We expect these ratios can be divided into four major groups, representing four possible scenarios: (1) agent-environment pairs fitted by both healing approaches (i.e., *Dr.DRL* and vanilla CL); (2) pairs fitted exclusively by *Dr.DRL*; (3) pairs fitted exclusively by vanilla CL; and (4) pairs not fitted by both approaches.

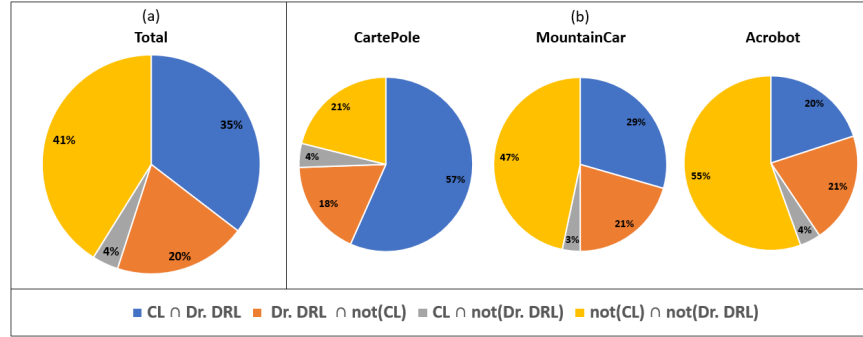


Figure 5.4 Distribution of %AR over the 4 identified groups Per healing Method and RL Environment

Results: Figure 5.4 shows (a) the Pie chart demonstrating the global adaptability ratios (%AR) achieved by either *Dr.DRL*, vanilla CL, both, or none of them; and (b) Pie charts depicting the same results of adaptability ratios (%AR) broken down by environments.

As can be seen in Figure 5.4-a, our approach, *Dr.DRL*, succeeds in extending the boundary of adaptable environment ratio obtained by vanilla CL. More specifically, *Dr.DRL* was able to heal about 20% of agent-environment pairs on which vanilla CL fails. This shows the effectiveness of intentional forgetting of minor behaviors in the enhancement of the DRL system self-healing ability, leading to a reduced number of non-adaptable environmental drifts. Nevertheless, this expansion in the adaptability frontiers of vanilla CL using *Dr.DRL* was achieved at the cost of a few drifted environments on which *Dr.DRL* fails despite they can be resolved by vanilla CL. The reason is that any intentional forgetting, no matter how carefully screened or guided, induces a loss of information (i.e., the reset hypoactive neurons). The latter may cause inefficiencies in the DRL self-healing ability. However, the results of RQ1 shows the cost-effectiveness resulting from our intentional forgetting mechanism in terms of the sample complexity. Hence, these 4% of non-adaptable drifted environments can be down to zero by reducing the ratio of erased minor behavior neurons over the healing trials

by *Dr.DRL* in its future versions (i.e., the less minor behavior neurons are reset, the more *Dr.DRL* behaves as a vanilla CL).

In line with the previous study on DRL system plasticity [40], there are environmental drift frontiers of the DRL system adaptation in response to drifted environments that are beyond the capabilities of both vanilla CL and our novel healing approach, *Dr.DRL*. This reinforces the assertion that the continuous self-healing of in-production DRL agents to constantly-shifting environments has limits. There are still instances of drifted environmental that require the re-training of the DRL agent from scratch or the redesign of the DRL algorithm beforehand in order to be able to resolve them.

According to Figure 5.4-b, the proportions of agent-environment pairs that are healed exclusively by one of the two healing approaches remain almost constant across all the studied RL environments. Meanwhile, the proportions of pairs that are either adapted or not equally by both approaches differ substantially depending on the studied environment. This can be explained by the differences in the parameterization complexity of these environments, i.e., involving high dimensional and sensitive parameters. Drifted instances derived from complex parametric environments are more challenging for self-healing approaches.

Research in healing software systems is very active and continues to advance towards equipping modern systems with self-healing capabilities in order to cope with ever-changing requirements throughout operation [200]. For DRL systems, our proposed approach, *Dr.DRL*, is in line with these current efforts, and by increasing the healing success ratio from 39% (i.e., 35% + 4%) to 55% (i.e., 35% + 20%), it represents a step further in assuring the DRL system’s self-healing ability in response to environmental drifts often encountered in production settings.

Finding 2. *Dr.DRL* outperforms the vanilla CL in healing a higher proportion of DRL systems to drifted environments, expanding the vanilla CL adaptability frontiers.

RQ3 (Performance): Are *Dr.DRL*’s healed systems more performant in terms of average reward than their vanilla CL counterparts?

Motivation: One of the primary factor for evaluating a DRL system’s self-healing approach is its ability to restore or surpass the original performance in a drifted environment. Predictive performance serves as the main criterion for evaluating such criterion. Therefore, to compare the effectiveness of *Dr.DRL* and vanilla CL in terms of the accumulated reward, we conduct

a comparison based on the predictive performance of the healed DRL system in drifted environments.

Method: We evaluate the performance of the healed agent by calculating the average rewards obtained over 100 consecutive episodes using the two self-healing approaches, namely *Dr.DRL* and vanilla CL. We then determine the average reward increase ratio (%IR, Eq. 5.4) of *Dr.DRL* compared to vanilla CL. Our analysis focuses exclusively on two scenarios: (i) jointly adaptable environments, where both vanilla CL and *Dr.DRL* succeeded in healing the agent-environment pairs, and (ii) jointly non-adaptable environments, where both CL and *Dr.DRL* failed. The remaining scenarios, where one approach succeeds while the other fails, are not relevant for predictive performance comparison due to the significant performance gap resulting from the failure of one approach to heal the DRL system.

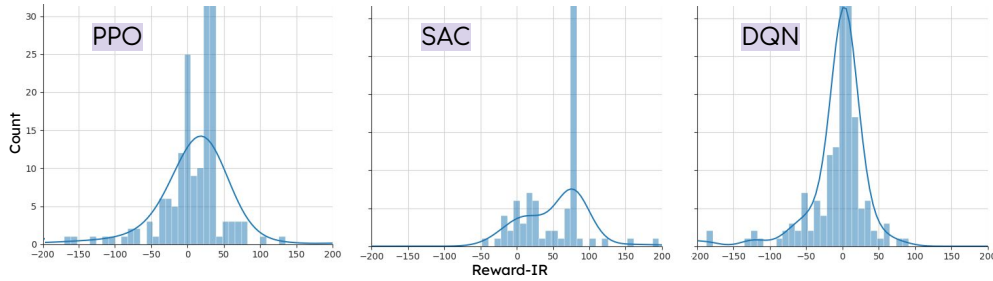


Figure 5.5 Average reward Increase Ratio Comparison between Dr.DRL and Vanilla CL in Non-Adaptable Environments. The curves illustrate the improvement in predictive performance achieved by Dr.DRL over vanilla CL in jointly non-adaptable environments

Results: The reward increase ratio of *Dr.DRL* compared to vanilla CL in jointly adaptable environments is presented in the first row of Table 5.2. In 22% of the experiments, *Dr.DRL* achieved a minimum 10% increase in average reward, while maintaining a nearly equal average reward (no less than -1.2%) in the remaining 78% of the experiments. The second row of

Table 5.2 Comparison of healed DRL Agent Reward Increases Using Dr.DRL over CL.

		DQN	PPO	SAC
Reward IR (%)	CartePole	-0.1	-0.06	<u>-0.2</u>
	MountainCar	0.8	-0.5	<u>10.4</u>
	Acrobot	-0.5	-1.2	<u>13.1</u>
Reward IR (%)	CartePole	<u>-34.9</u>	45.7	2.2
	MountainCar	<u>3.5</u>	<u>25.8</u>	-15.4
	Acrobot	<u>14.2</u>	13.5	<u>21.2</u>

Table 5.2 displays the reward increase ratio of *Dr.DRL* compared to vanilla CL in jointly non-adaptable environments. *Dr.DRL* improved the average reward in 78% of the configurations, with an increase ratio of up to 45% compared to vanilla CL’s average reward. The results in Table 5.2 demonstrate that *Dr.DRL* successfully maintains or improves upon the baseline average reward achieved by vanilla CL in jointly adaptable environments. Furthermore, *Dr.DRL* consistently achieves better average rewards than vanilla CL in drifted environments where both self-healing approaches fail.

For a more detailed analysis, the distributions of *Dr.DRL*’s reward increase ratio (averaged by DRL algorithms) over vanilla CL in jointly non-adaptable environments are visualized in Figure 5.5. Overall, the distributions follow a curve that tends to the right direction, indicating an increase ratio of ≥ 0 . These right-leaning curves demonstrate that *Dr.DRL* surpasses vanilla CL’s predictive performance in non-adaptable environments, as reflected by the improved reward increase ratios.

Finding 3: *Dr.DRL* was capable of achieving similar or even higher performances in adaptable environments, and it also earned higher performances in non-adaptable ones, showing how much closer *Dr.DRL* was to meet the requirements than vanilla CL.

5.4 Threats to Validity

In the following, we discuss the threats to validity.

Threats to internal validity. They may result from how the empirical study was conducted. To mitigate these issues, we have configured both *Dr.DRL* and CL with identical parameters (e.g., number of training episodes, maximum allowed steps per episode, etc.) across all studied pairs of RL environments and DRL algorithms.

Threats to conclusion validity. They are often related to random variations and inappropriate use of statistical tests. To mitigate these threats, we ran each experiment using either *Dr.DRL* and CL, 6×10 times (6 environments with varying parameter shifts, and 10 different optimized DRL agents). Specifically, the training of 10 agents of each DRL algorithm using different random seeds, was intended to overcome the DRL algorithm’s randomness. To confirm the statistical significance of our findings, we performed statistical hypothesis testing and effect size assessments using the Vargha–Delaney effect size [176] and the non-parametric Wilcoxon test [175].

Threats to external validity. They may limit the generalizability of our findings to other RL-based systems. Although more subjects would be needed to fully assess the generalizability of our results, we have chosen 3 classic and diverse environments that are widely used

in the DRL community, as well as, their implementations are included in gym library [34]. In addition, we have assessed our method on three distinct DRL algorithms from different types and having distinct learning strategies.

5.5 Chapter Summary

One of the challenges encountered in the adoption of Deep Reinforcement Learning is the performance degradation caused by drifts in the agent’s performance. These performance drops are typically attributed to environmental drifts, where the parameters of the environment deviate, thus the agent necessitate an adaptation. Although existing approaches like continual learning have been proposed to address this issue, they suffer from limitations such as plasticity issues and the time required for agent adaptation. In response to these challenges, we have introduced *Dr.DRL*, a self-healing approach that leverages the intentional forgetting mechanism to enhance the agent’s adaptability to drifted environments and improve its plasticity. The intentional forgetting mechanism prioritizes the adaptation of neurons responsible for major behavior by reinitializing minor behavior neurons with low scaled values, allowing for larger updates in major neurons. To evaluate the effectiveness of *Dr.DRL*, we compared it to vanilla continual learning using three different environments and agents. The results demonstrate that *Dr.DRL* achieves faster healing, greater adaptability to the drifted environments, and higher average rewards.

CHAPTER 6 CONCLUSION

In this chapter, we summarize our findings and outline some directions for future work.

6.1 Summary

We are witnessing a rapid transition towards an era of automation and robotization. Deep Reinforcement Learning emerges as a powerful technology to tackle the unique challenges posed by this automation wave, surpassing the capabilities of classical deep learning applications. However, the reliability of DRL remains a concern due to the lack of quality assurance tools designed for DRL applications. The complex and opaque nature of DRL architectures introduces difficulties in monitoring and detecting erroneous behavior, making the development and control process challenging.

This thesis addresses the critical issue of quality assurance in deep reinforcement learning by introducing novel approaches that facilitate the automatic monitoring of DRL applications. The main goal of these approaches is to detect faulty behavior early on in the DRL application lifecycle, enabling developers to address and rectify issues before they manifest into undesirable consequences. Additionally, the proposed approaches help mitigate performance degradation caused by environmental drifts. This thesis made the following two key contributions:

***RLExplorer* : A fault Detection approach for Deep Reinforcement Learning**

RLExplorer is the first fault diagnosis approach designed specifically for DRL-based systems. Attached to the DRL application at runtime, *RLExplorer* automatically runs verification routines based on properties of the learning dynamics to detect the occurrence of DRL-specific faults. These verification routines encompass multiple diagnosis strategies, aiming to detect either the symptoms or the root cause of a bug. The approach builds on multiple checks that can be triggered periodically during the training in order to detect any indicative signals of anomalies in the training. These checks fall into two categories: RL-specific faults, which encompass symptoms associated with the reinforcement learning entities (i.e., Action, Agent, Environment, Exploration Parameter, Reward, States, and Q-Target), and NN-specific faults, which relate to symptoms pertaining to deep learning concepts (i.e., Activation, Weights, Bias, Gradient, and Loss). The developer can integrate our approach into his code, once the training starts *RLExplorer*, displays the results of these checks as warning messages that incorporate theoretical concepts, recommended practices, and potential solutions. The approach also provide visualization, such as the entropy, KL divergence and the

standard deviation of the reward, that can help the developer understand and monitor the behaviours of his agent during the training. The integration process of *RLExplorer* into the developer’s code is straightforward. Once training starts, *RLExplorer* displays the results of these checks as warning messages that incorporate theoretical concepts, recommended practices, and potential solutions. Furthermore, *RLExplorer* allows visualization of different learning traces, such as entropy, KL divergence, and standard deviation of the reward, empowering the developer to comprehend and monitor their agent’s behavior throughout the training process. To evaluate *RLExplorer*, we collected 11 faulty DRL samples from SO and assessed *RLExplorer* on these samples. Results show that *RLExplorer* can detect faults in 83% of cases. We further conduct a human study with 15 participants to assess *RLExplorer*’s effectiveness in helping developers diagnose faults. Results show that participants were able to diagnose 3.6 times more faults using *RLExplorer* compared to manual debugging. Participants also reported high satisfaction with the debugger and a high likelihood of leveraging *RLExplorer* in their development.

***Dr.DRL* : A Self-Healing Method For Deep Reinforcement Learning Systems**

Dr.DRL, is a novel self-healing approach for DRL systems, that can be used to adapt efficiently the DRL agent to the drifts that may occur in the environment. Unlike traditional continual learning strategies, *Dr.DRL* incorporates an intentional forgetting mechanism to systematically prioritize the adaptation of the DRL agent’s skills when faced with environmental changes. The intentional forgetting mechanism identifies the agent’s minor behaviors by monitoring neurons with consistently low-ranked activations throughout training episodes, known as hypoactive neurons. During the healing process, these hypoactive neurons are reset, allowing the agent to accelerate the adaptation process and focus more on adapting the neurons responsible for the major behavior, called hyperactive neurons. When the DRL agent encounters a drifted environment resulting in performance degradation, *Dr.DRL* automatically triggers the healing process. This process involves two key steps: (i) re-initializing the weights associated with hypoactive neurons using low-scaled values, and (ii) performing continual learning updates on the patched DRL agent using the newly-drifted environment. By doing that, *Dr.DRL* enables the adaptation of DRL agent behaviors at dual speeds. Specifically, untouched major behavior neurons maintain larger updates compared to under-scaled minor behavior neurons.

We have evaluated *Dr.DRL* by conducting self-healing experiments on different DRL agents using *Dr.DRL* and compared them to vanilla continual learning (CL) approaches. The experiments involved drifted environments with varying parameter shifts. The results demonstrate that *Dr.DRL* outperforms vanilla CL in several aspects: (i) it achieves faster healing of the DRL agent, (ii) it adapts to a greater number of environments, and (iii) it achieves equal or

higher average rewards compared to vanilla CL. These findings validate the effectiveness and superiority of *Dr.DRL* in addressing the challenges posed by environmental drifts in DRL systems.

The results of this thesis show that our proposed approaches significantly contribute to enhancing quality assurance in DRL applications. By giving developers more control over potential faults that may arise during training, our approach *RLExplorer* helps mitigate the occurrence of silent errors and defects that could otherwise be challenging to detect manually or could be transmitted inadvertently to the deployed version. This heightened level of control instills higher confidence in the reliability of DRL applications. Moreover, to avoid intolerable performance degradation and ensure sustained high-quality performance of the DRL agent post-production, we have introduced *Dr.DRL*, an efficient and automated self-healing approach. This approach effectively adapts the agent’s policy when a performance degradation occurs, thereby reinforcing the plasticity of the DRL application.

6.2 Discussion and Future Work

The main goal of *RLExplorer* is to automatically detect faults and alert users whenever erroneous behavior arises during the training of DRL applications. By facilitating the identification of bugs and pinpointing their potential root causes, *RLExplorer* effectively empowers developers to exercise more control over the training process and streamline the monitoring of DRL applications. This approach enhances the overall development experience and promotes more efficient bug resolution during the training phase.

However, *RLExplorer*’s proposed solutions for the detected bugs are static and may not be applicable in all cases. To enhance the capabilities of *RLExplorer*, we aim to incorporate automatic repair functionalities that can automatically fix bugs with clear root causes. The ultimate goal is to empower *RLExplorer* to either resolve bugs automatically or provide comprehensive reports on how and where the faults can be corrected.

In the case of *Dr.DRL*, it facilitates maintenance of the DRL models and prevents performance degradation due to environment drift. However, activation-based ranking is employed to categorize neurons as hyperactive or hypoactive. As part of our future work, we intend to explore alternative neuron categorization strategies beyond activation-based ranking. By doing so, we hope to identify the most significant neurons (i.e., hyperactive neurons) more accurately, particularly in the context of deep reinforcement learning. Moreover, while multi-agent DRL systems are widely used by many developers, it is important to note that neither *RLExplorer* nor *Dr.DRL* currently encompass this aspect in their conception and experiments. Therefore, our plan includes expanding the functionality of both *RLExplorer* and

Dr.DRL to support multi-agent DRL applications. This expansion will involve incorporating specific diagnosis techniques tailored to multi-agent systems, as they present some unique challenges. In addition to automating the process of detecting performance degradation and implementing optimal healing strategies for the different agents in the multi-agent DRL application. Furthermore, we aim to broaden the scope of our evaluation by incorporating a wider range of environments for both *RLExplorer* and *Dr.DRL*. This expansion will encompass more complex environments, including real-world challenges such as self-driving cars and robotics. Finally, during the evaluation of *RLExplorer*, multiple participants to our users' study reported using distributed training for training their agents. As a result, we plan to address bugs related to distributed and parallelized agent training. By doing so, we can further enhance the capabilities of *RLExplorer* and expand its coverage of errors that may arise during different training processes.

APPENDIX A CO-AUTHORSHIP

Earlier studies in the thesis were published/submitted as follows:

- Toward Debugging Deep Reinforcement Learning Programs with RLEplorer, **Rached Bouchoucha***, Ahmed Haj Yahmed*, Darshan Patil, Janarthanan Rajendran, Amin Nikanjam, Sarath Chandar, Foutse Khomh, in *ACM Transactions on Software Engineering and Methodology (TOSEM)*
- An Intentional Forgetting-Driven Self-Healing Method For Deep Reinforcement Learning Systems, Ahmed Haj Yahmed*, **Rached Bouchoucha***, Housseem Ben Braiek, Foutse Khomh, in *38th IEEE/ACM International Conference on Automated Software Engineering (ASE 2023)*

The following publications were produced during my master's degree studies but is not directly related to the content of the thesis:

- Quality issues in Machine Learning Software Systems, Pierre-Olivier Côté, Amin Nikanjam, **Rached Bouchoucha**, Ilan Basta, Mouna Abidi, Foutse Khomh, in *EMSE 2023 manuscript (2023)*
- Robustness assessment of hyperspectral image CNNs using metamorphic testing, **Rached Bouchoucha**, Housseem Ben Braiek, Foutse Khomh, Sonia Bouzidi, Rania Zaatour, in *Information and Software Technology*, Volume 162, 2023, 107281, ISSN 0950-5849, <https://doi.org/10.1016/j.infsof.2023.107281>.
- Can Ensembling Preprocessing Algorithms Lead to Better Machine Learning Fairness?, Khaled Badran, Pierre-Olivier Côté, Amanda Kolopanis, **Rached Bouchoucha**, Antonio Collante, Diego Elias Costa, Emad Shihab, Foutse Khomh, in *Computer*, vol. 56, no. 4, pp. 71-79, April 2023, doi: 10.1109/MC.2022.3220707.

REFERENCES

- [1] A. W. Moore, “Efficient memory-based learning for robot control,” *Citeseer*, 1990.
- [2] G. Dulac-Arnold, N. Levine, D. J. Mankowitz, J. Li, C. Paduraru, S. Goyal, and T. Hester, “Challenges of real-world reinforcement learning: definitions, benchmarks and analysis,” *Machine Learning*, vol. 110, no. 9, pp. 2419–2468, 2021.
- [3] G. Dulac-Arnold, D. Mankowitz, and T. Hester, “Challenges of real-world reinforcement learning,” *arXiv preprint arXiv:1904.12901*, 2019.
- [4] Y. Li, “Reinforcement learning in practice: Opportunities and challenges,” *arXiv preprint arXiv:2202.11296*, 2022.
- [5] Z. Ding and H. Dong, “Challenges of reinforcement learning,” *Deep Reinforcement Learning: Fundamentals, Research and Applications*, pp. 249–272, 2020.
- [6] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, “Playing atari with deep reinforcement learning,” *arXiv preprint arXiv:1312.5602*, 2013.
- [7] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, “Mastering the game of go with deep neural networks and tree search,” *nature*, vol. 529, no. 7587, pp. 484–489, 2016.
- [8] O. Vinyals, I. Babuschkin, W. M. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. H. Choi, R. Powell, T. Ewalds, P. Georgiev *et al.*, “Grandmaster level in starcraft ii using multi-agent reinforcement learning,” *Nature*, vol. 575, no. 7782, pp. 350–354, 2019.
- [9] C. Berner, G. Brockman, B. Chan, V. Cheung, P. Dębiak, C. Dennison, D. Farhi, Q. Fischer, S. Hashme, C. Hesse *et al.*, “Dota 2 with large scale deep reinforcement learning,” *arXiv preprint arXiv:1912.06680*, 2019.
- [10] E. Akanksha, N. Sharma, K. Gulati *et al.*, “Review on reinforcement learning, research evolution and scope of application,” in *2021 5th international conference on computing methodologies and communication (ICCMC)*. IEEE, 2021, pp. 1416–1423.
- [11] K. Roose, “The making of a youtube radical,” *The New York Times*, vol. 8, 2019.

- [12] A. R. Mahmood, D. Korenkevych, G. Vasan, W. Ma, and J. Bergstra, “Benchmarking reinforcement learning algorithms on real-world robots,” in *Conference on robot learning*. PMLR, 2018, pp. 561–591.
- [13] O. M. Andrychowicz, B. Baker, M. Chociej, R. Jozefowicz, B. McGrew, J. Pachocki, A. Petron, M. Plappert, G. Powell, A. Ray *et al.*, “Learning dexterous in-hand manipulation,” *The International Journal of Robotics Research*, vol. 39, no. 1, pp. 3–20, 2020.
- [14] X. B. Peng, P. Abbeel, S. Levine, and M. Van de Panne, “Deepmimic: Example-guided deep reinforcement learning of physics-based character skills,” *ACM Transactions On Graphics (TOG)*, vol. 37, no. 4, pp. 1–14, 2018.
- [15] X. B. Peng, Z. Ma, P. Abbeel, S. Levine, and A. Kanazawa, “Amp: Adversarial motion priors for stylized physics-based character control,” *ACM Transactions on Graphics (ToG)*, vol. 40, no. 4, pp. 1–20, 2021.
- [16] J. Ibarz, J. Tan, C. Finn, M. Kalakrishnan, P. Pastor, and S. Levine, “How to train your robot with deep reinforcement learning: lessons we have learned,” *The International Journal of Robotics Research*, vol. 40, no. 4-5, pp. 698–721, 2021.
- [17] C. Cummins, B. Wasti, J. Guo, B. Cui, J. Ansel, S. Gomez, S. Jain, J. Liu, O. Teytaud, B. Steiner *et al.*, “Compilergym: Robust, performant compiler optimization environments for ai research,” in *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2022, pp. 92–105.
- [18] H. Zhu, V. Gupta, S. S. Ahuja, Y. Tian, Y. Zhang, and X. Jin, “Network planning with deep reinforcement learning,” in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, 2021, pp. 258–271.
- [19] J. Kim, M. Kwon, and S. Yoo, “Generating test input with deep reinforcement learning,” in *Proceedings of the 11th International Workshop on Search-Based Software Testing*, 2018, pp. 51–58.
- [20] J. Zhang, Y. Liu, K. Zhou, G. Li, Z. Xiao, B. Cheng, J. Xing, Y. Wang, T. Cheng, L. Liu *et al.*, “An end-to-end automatic cloud database tuning system using deep reinforcement learning,” in *Proceedings of the 2019 International Conference on Management of Data*, 2019, pp. 415–432.

- [21] L. Harries, R. S. Clarke, T. Chapman, S. V. Nallamalli, L. Ozgur, S. Jain, A. Leung, S. Lim, A. Dietrich, J. M. Hernández-Lobato *et al.*, “Drift: Deep reinforcement learning for functional software testing,” *arXiv preprint arXiv:2007.08220*, 2020.
- [22] Y. Zheng, X. Xie, T. Su, L. Ma, J. Hao, Z. Meng, Y. Liu, R. Shen, Y. Chen, and C. Fan, “Wuji: Automatic online combat game testing using evolutionary deep reinforcement learning,” in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 772–784.
- [23] Y. Liu, X. Qi, J. Zhang, H. Li, X. Ge, and J. Ai, “Automatic bug triaging via deep reinforcement learning,” *Applied Sciences*, vol. 12, no. 7, p. 3565, 2022.
- [24] A. Haydari and Y. Yilmaz, “Deep reinforcement learning for intelligent transportation systems: A survey,” *IEEE Transactions on Intelligent Transportation Systems*, vol. 23, no. 1, pp. 11–32, 2020.
- [25] X.-Y. Liu, H. Yang, Q. Chen, R. Zhang, L. Yang, B. Xiao, and C. D. Wang, “Finrl: A deep reinforcement learning library for automated stock trading in quantitative finance,” *arXiv preprint arXiv:2011.09607*, 2020.
- [26] Z. Jiang, D. Xu, and J. Liang, “A deep reinforcement learning framework for the financial portfolio management problem,” *arXiv preprint arXiv:1706.10059*, 2017.
- [27] C. Yu, J. Liu, S. Nemati, and G. Yin, “Reinforcement learning in healthcare: A survey,” *ACM Computing Surveys (CSUR)*, vol. 55, no. 1, pp. 1–36, 2021.
- [28] O. Gottesman, F. Johansson, M. Komorowski, A. Faisal, D. Sontag, F. Doshi-Velez, and L. A. Celi, “Guidelines for reinforcement learning in healthcare,” *Nature medicine*, vol. 25, no. 1, pp. 16–18, 2019.
- [29] A. Singla, A. N. Rafferty, G. Radanovic, and N. T. Heffernan, “Reinforcement learning for education: Opportunities and challenges,” *arXiv preprint arXiv:2107.08828*, 2021.
- [30] W. Cai, J. Grossman, Z. J. Lin, H. Sheng, J. T.-Z. Wei, J. J. Williams, and S. Goel, “Bandit algorithms to personalize educational chatbots,” *Machine Learning*, vol. 110, no. 9, pp. 2389–2418, 2021.
- [31] S. Cai, E. Breck, E. Nielsen, M. Salib, and D. Sculley, “Tensorflow debugger: Debugging dataflow graphs for machine learning,” 2016.
- [32] (2015) Tflint. <https://github.com/terraform-linters/tflint>.

- [33] W. A. Falcon, “Pytorch lightning,” *GitHub*, vol. 3, 2019.
- [34] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, “Openai gym,” *arXiv preprint arXiv:1606.01540*, 2016.
- [35] M. B. Ring *et al.*, “Continual learning in reinforcement environments,” *Citeseer*, 1994.
- [36] S. Powers, E. Xing, E. Kolve, R. Mottaghi, and A. Gupta, “Cora: Benchmarks, baselines, and metrics as a platform for continual reinforcement learning agents,” *arXiv preprint arXiv:2110.10067*, 2021.
- [37] C. Zhao, O. Sigaud, F. Stulp, and T. M. Hospedales, “Investigating generalisation in continuous deep reinforcement learning,” *arXiv preprint arXiv:1902.07015*, 2019.
- [38] C. Kaplanis, M. Shanahan, and C. Clopath, “Continual reinforcement learning with complex synapses,” in *International Conference on Machine Learning*. PMLR, 2018, pp. 2497–2506.
- [39] 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, 2022.
- [40] M. Biagiola and P. Tonella, “Testing the plasticity of reinforcement learning based systems,” *ACM Transactions on Software Engineering and Methodology*, 2022.
- [41] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, “Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor,” in *International conference on machine learning*. PMLR, 2018, pp. 1861–1870.
- [42] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal policy optimization algorithms,” *arXiv preprint arXiv:1707.06347*, 2017.
- [43] 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*, no. 5, pp. 834–846, 1983.
- [44] A. Geramifard, C. Dann, R. H. Klein, W. Dabney, and J. P. How, “Rlpy: a value-function-based reinforcement learning framework for education and research.” *J. Mach. Learn. Res.*, vol. 16, no. 1, pp. 1573–1578, 2015.
- [45] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, “Human-level control through deep reinforcement learning,” *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

- [46] L. P. Kaelbling, M. L. Littman, and A. W. Moore, “Reinforcement learning: A survey,” *Journal of artificial intelligence research*, vol. 4, pp. 237–285, 1996.
- [47] D. J. White, “A survey of applications of markov decision processes,” *Journal of the operational research society*, vol. 44, no. 11, pp. 1073–1096, 1993.
- [48] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, “Deep reinforcement learning: A brief survey,” *IEEE Signal Processing Magazine*, vol. 34, no. 6, pp. 26–38, 2017.
- [49] C. J. Watkins and P. Dayan, “Q-learning,” *Machine learning*, vol. 8, pp. 279–292, 1992.
- [50] L. Ma, F. Juefei-Xu, F. Zhang, J. Sun, M. Xue, B. Li, C. Chen, T. Su, L. Li, Y. Liu *et al.*, “Deepgauge: Multi-granularity testing criteria for deep learning systems,” in *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*, 2018, pp. 120–131.
- [51] H. B. Braiek and F. Khomh, “On testing machine learning programs,” *Journal of Systems and Software*, vol. 164, p. 110542, 2020.
- [52] Y. Li, “Deep reinforcement learning: An overview,” *arXiv preprint arXiv:1701.07274*, 2017.
- [53] V. Konda and J. Tsitsiklis, “Actor-critic algorithms,” *Advances in neural information processing systems*, vol. 12, 1999.
- [54] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, “Trust region policy optimization,” in *International conference on machine learning*. PMLR, 2015, pp. 1889–1897.
- [55] R. Bellman, “On the theory of dynamic programming,” *Proceedings of the national Academy of Sciences*, vol. 38, no. 8, pp. 716–719, 1952.
- [56] M. G. Bellemare, W. Dabney, and R. Munos, “A distributional perspective on reinforcement learning,” in *International conference on machine learning*. PMLR, 2017, pp. 449–458.
- [57] M. Hessel, J. Modayil, H. Van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver, “Rainbow: Combining improvements in deep reinforcement learning,” in *Proceedings of the AAAI conference on artificial intelligence*, vol. 32, no. 1, 2018.

- [58] P. Ladosz, L. Weng, M. Kim, and H. Oh, “Exploration in deep reinforcement learning: A survey,” *Information Fusion*, 2022.
- [59] C. Packer, K. Gao, J. Kos, P. Krähenbühl, V. Koltun, and D. Song, “Assessing generalization in deep reinforcement learning,” *arXiv preprint arXiv:1810.12282*, 2018.
- [60] Z. Wang, C. Chen, H.-X. Li, D. Dong, and T.-J. Tarn, “A novel incremental learning scheme for reinforcement learning in dynamic environments,” in *2016 12th World Congress on Intelligent Control and Automation (WCICA)*. IEEE, 2016, pp. 2426–2431.
- [61] —, “Incremental reinforcement learning with prioritized sweeping for dynamic environments,” *IEEE/ASME Transactions on Mechatronics*, vol. 24, no. 2, pp. 621–632, 2019.
- [62] M. Panzer and B. Bender, “Deep reinforcement learning in production systems: a systematic literature review,” *International Journal of Production Research*, vol. 60, no. 13, pp. 4316–4341, 2022.
- [63] G. I. Webb, R. Hyde, H. Cao, H. L. Nguyen, and F. Petitjean, “Characterizing concept drift,” *Data Mining and Knowledge Discovery*, vol. 30, no. 4, pp. 964–994, 2016.
- [64] R. Hadsell, D. Rao, A. A. Rusu, and R. Pascanu, “Embracing change: Continual learning in deep neural networks,” *Trends in cognitive sciences*, vol. 24, no. 12, pp. 1028–1040, 2020.
- [65] A. Cahill, “Catastrophic forgetting in reinforcement-learning environments,” Ph.D. dissertation, University of Otago, 2011.
- [66] R. M. French, “Catastrophic forgetting in connectionist networks,” *Trends in cognitive sciences*, vol. 3, no. 4, pp. 128–135, 1999.
- [67] A. Paleyes, R.-G. Urma, and N. D. Lawrence, “Challenges in deploying machine learning: a survey of case studies,” *ACM Computing Surveys*, vol. 55, no. 6, pp. 1–29, 2022.
- [68] M. Trujillo, M. Linares-Vásquez, C. Escobar-Velásquez, I. Dusparic, and N. Cardozo, “Does neuron coverage matter for deep reinforcement learning? a preliminary study,” in *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, 2020, pp. 215–220.

- [69] I. Ilahi, M. Usama, J. Qadir, M. U. Janjua, A. Al-Fuqaha, D. T. Hoang, and D. Niyato, “Challenges and countermeasures for adversarial attacks on deep reinforcement learning,” *IEEE Transactions on Artificial Intelligence*, vol. 3, no. 2, pp. 90–109, 2021.
- [70] S. Huang, N. Papernot, I. Goodfellow, Y. Duan, and P. Abbeel, “Adversarial attacks on neural network policies,” *arXiv preprint arXiv:1702.02284*, 2017.
- [71] A. Pan, Y. Lee, H. Zhang, Y. Chen, and Y. Shi, “Improving robustness of reinforcement learning for power system control with adversarial training,” *arXiv preprint arXiv:2110.08956*, 2021.
- [72] S.-M. Moosavi-Dezfooli, A. Fawzi, O. Fawzi, and P. Frossard, “Universal adversarial perturbations,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 1765–1773.
- [73] A. Pattanaik, Z. Tang, S. Liu, G. Bommannan, and G. Chowdhary, “Robust deep reinforcement learning with adversarial attacks,” *arXiv preprint arXiv:1712.03632*, 2017.
- [74] K. L. Tan, Y. Esfandiari, X. Y. Lee, S. Sarkar *et al.*, “Robustifying reinforcement learning agents via action space adversarial training,” in *2020 American control conference (ACC)*. IEEE, 2020, pp. 3959–3964.
- [75] J. Kos and D. Song, “Delving into adversarial attacks on deep policies,” *arXiv preprint arXiv:1705.06452*, 2017.
- [76] X. Y. Lee, Y. Esfandiari, K. L. Tan, and S. Sarkar, “Query-based targeted action-space adversarial policies on deep reinforcement learning agents,” in *Proceedings of the ACM/IEEE 12th International Conference on Cyber-Physical Systems*, 2021, pp. 87–97.
- [77] E. Vinitzky, Y. Du, K. Parvate, K. Jang, P. Abbeel, and A. Bayen, “Robust reinforcement learning using adversarial populations,” *arXiv preprint arXiv:2008.01825*, 2020.
- [78] V. Behzadan and A. Munir, “Mitigation of policy manipulation attacks on deep q-networks with parameter-space noise,” in *Computer Safety, Reliability, and Security: SAFECOMP 2018 Workshops, ASSURE, DECSoS, SASSUR, STRIVE, and WAISE, Västerås, Sweden, September 18, 2018, Proceedings 37*. Springer, 2018, pp. 406–417.
- [79] —, “Whatever does not kill deep reinforcement learning, makes it stronger,” *arXiv preprint arXiv:1712.09344*, 2017.

- [80] Y. Han, B. I. Rubinstein, T. Abraham, T. Alpcan, O. De Vel, S. Erfani, D. Hubczenko, C. Leckie, and P. Montague, “Reinforcement learning for autonomous defence in software-defined networking,” in *Decision and Game Theory for Security: 9th International Conference, GameSec 2018, Seattle, WA, USA, October 29–31, 2018, Proceedings 9*. Springer, 2018, pp. 145–165.
- [81] M. Tappler, F. C. Córdoba, B. K. Aichernig, and B. Könighofer, “Search-based testing of reinforcement learning,” *arXiv preprint arXiv:2205.04887*, 2022.
- [82] A. Zolfagharian, M. Abdellatif, L. C. Briand, M. Bagherzadeh, and S. Ramesh, “A search-based testing approach for deep reinforcement learning agents,” *IEEE Transactions on Software Engineering*, 2023.
- [83] A. Nikanjam, M. M. Morovati, F. Khomh, and H. Ben Braiek, “Faults in deep reinforcement learning programs: a taxonomy and a detection approach,” *Automated Software Engineering*, vol. 29, no. 1, p. 8, 2022.
- [84] S. Ma, Y. Liu, W.-C. Lee, X. Zhang, and A. Grama, “Mode: automated neural network model debugging via state differential analysis and input selection,” in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 175–186.
- [85] H. F. Eniser, S. Gerasimou, and A. Sen, “Deepfault: Fault localization for deep neural networks,” in *Fundamental Approaches to Software Engineering: 22nd International Conference, FASE 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6–11, 2019, Proceedings*. Springer, 2019, pp. 171–191.
- [86] A. Nikanjam, H. B. Braiek, M. M. Morovati, and F. Khomh, “Automatic fault detection for deep learning programs using graph transformations,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 1, pp. 1–27, 2021.
- [87] Y. Zhang, L. Ren, L. Chen, Y. Xiong, S.-C. Cheung, and T. Xie, “Detecting numerical bugs in neural network architectures,” in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 826–837.
- [88] X. Zhang, J. Zhai, S. Ma, and C. Shen, “Autotrainer: An automatic dnn training problem detection and repair system,” in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 359–371.

- [89] H. B. Braiek and F. Khomh, “Testing feedforward neural networks training programs,” *ACM Transactions on Software Engineering and Methodology*, 2022.
- [90] M. Wardat, W. Le, and H. Rajan, “Deeplocalize: Fault localization for deep neural networks,” in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 251–262.
- [91] E. Schoop, F. Huang, and B. Hartmann, “Umlaut: Debugging deep learning programs using program structure and model behavior,” in *Proceedings of the 2021 CHI conference on human factors in computing systems*, 2021, pp. 1–16.
- [92] N. Rauschmayr, V. Kumar, R. Huilgol, A. Olgiati, S. Bhattacharjee, N. Harish, V. Kannan, A. Lele, A. Acharya, J. Nielsen *et al.*, “Amazon sagemaker debugger: a system for real-time insights into machine learning model training,” *Proceedings of Machine Learning and Systems*, vol. 3, pp. 770–782, 2021.
- [93] M. Wardat, B. D. Cruz, W. Le, and H. Rajan, “Deepdiagnosis: automatically diagnosing faults and recommending actionable fixes in deep learning programs,” in *Proceedings of the 44th international conference on software engineering*, 2022, pp. 561–572.
- [94] J. Cao, M. Li, X. Chen, M. Wen, Y. Tian, B. Wu, and S.-C. Cheung, “Deepfd: automated fault diagnosis and localization for deep learning programs,” in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 573–585.
- [95] T. Zahavy, N. Ben-Zrihem, and S. Mannor, “Graying the black box: Understanding dqns,” in *International conference on machine learning*. PMLR, 2016, pp. 1899–1908.
- [96] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, “Asynchronous methods for deep reinforcement learning,” in *International conference on machine learning*. PMLR, 2016, pp. 1928–1937.
- [97] L. Van der Maaten and G. Hinton, “Visualizing data using t-sne.” *Journal of machine learning research*, vol. 9, no. 11, 2008.
- [98] S. Greydanus, A. Koul, J. Dodge, and A. Fern, “Visualizing and understanding atari agents,” in *International conference on machine learning*. PMLR, 2018, pp. 1792–1801.
- [99] C. Rupprecht, C. Ibrahim, and C. J. Pal, “Finding and visualizing weaknesses of deep reinforcement learning agents,” *arXiv preprint arXiv:1904.01318*, 2019.

- [100] S. H. Huang, K. Bhatia, P. Abbeel, and A. D. Dragan, “Establishing appropriate trust via critical states,” in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2018, pp. 3929–3936.
- [101] A. Heuillet, F. Couthouis, and N. Díaz-Rodríguez, “Explainability in deep reinforcement learning,” *Knowledge-Based Systems*, vol. 214, p. 106685, 2021.
- [102] X. Li, H. Xiong, X. Li, X. Wu, X. Zhang, J. Liu, J. Bian, and D. Dou, “Interpretable deep learning: Interpretation, interpretability, trustworthiness, and beyond,” *Knowledge and Information Systems*, vol. 64, no. 12, pp. 3197–3234, 2022.
- [103] L. Wells and T. Bednarz, “Explainable ai and reinforcement learning—a systematic review of current approaches and trends,” *Frontiers in artificial intelligence*, vol. 4, p. 550030, 2021.
- [104] V. Zambaldi, D. Raposo, A. Santoro, V. Bapst, Y. Li, I. Babuschkin, K. Tuyls, D. Reichert, T. Lillicrap, E. Lockhart *et al.*, “Relational deep reinforcement learning,” *arXiv preprint arXiv:1806.01830*, 2018.
- [105] A. Nagabandi, I. Clavera, S. Liu, R. S. Fearing, P. Abbeel, S. Levine, and C. Finn, “Learning to adapt in dynamic, real-world environments through meta-reinforcement learning,” *arXiv preprint arXiv:1803.11347*, 2018.
- [106] G. A. Carpenter and S. Grossberg, “A massively parallel architecture for a self-organizing neural pattern recognition machine,” *Computer vision, graphics, and image processing*, vol. 37, no. 1, pp. 54–115, 1987.
- [107] N. Duminy, S. M. Nguyen, J. Zhu, D. Duhaut, and J. Kerdreux, “Intrinsically motivated open-ended multi-task learning using transfer learning to discover task hierarchy,” *Applied Sciences*, vol. 11, no. 3, p. 975, 2021.
- [108] S. Pateria, B. Subagdja, A.-h. Tan, and C. Quek, “Hierarchical reinforcement learning: A comprehensive survey,” *ACM Computing Surveys (CSUR)*, vol. 54, no. 5, pp. 1–35, 2021.
- [109] M. De Lange, R. Aljundi, M. Masana, S. Parisot, X. Jia, A. Leonardis, G. Slabaugh, and T. Tuytelaars, “A continual learning survey: Defying forgetting in classification tasks,” *IEEE transactions on pattern analysis and machine intelligence*, vol. 44, no. 7, pp. 3366–3385, 2021.

- [110] A. A. Rusu, N. C. Rabinowitz, G. Desjardins, H. Soyer, J. Kirkpatrick, K. Kavukcuoglu, R. Pascanu, and R. Hadsell, “Progressive neural networks,” *arXiv preprint arXiv:1606.04671*, 2016.
- [111] J. Kirkpatrick, R. Pascanu, N. Rabinowitz, J. Veness, G. Desjardins, A. A. Rusu, K. Milan, J. Quan, T. Ramalho, A. Grabska-Barwinska *et al.*, “Overcoming catastrophic forgetting in neural networks,” *Proceedings of the national academy of sciences*, vol. 114, no. 13, pp. 3521–3526, 2017.
- [112] H. Shin, J. K. Lee, J. Kim, and J. Kim, “Continual learning with deep generative replay,” *Advances in neural information processing systems*, vol. 30, 2017.
- [113] Z. Li and D. Hoiem, “Learning without forgetting,” *IEEE transactions on pattern analysis and machine intelligence*, vol. 40, no. 12, pp. 2935–2947, 2017.
- [114] J. Sohn, S. Kang, and S. Yoo, “Search based repair of deep neural networks,” *arXiv preprint arXiv:1912.12463*, 2019.
- [115] T. Xie, N. Jiang, H. Wang, C. Xiong, and Y. Bai, “Policy finetuning: Bridging sample-efficient offline and online reinforcement learning,” *Advances in neural information processing systems*, vol. 34, pp. 27 395–27 407, 2021.
- [116] H. Zhang and W. Chan, “Apricot: A weight-adaptation approach to fixing deep learning models,” in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 376–387.
- [117] D. Borsa, T. Graepel, and J. Shawe-Taylor, “Learning shared representations in multi-task reinforcement learning,” *arXiv preprint arXiv:1603.02041*, 2016.
- [118] C. D’Eramo, D. Tateo, A. Bonarini, M. Restelli, and J. Peters, “Sharing knowledge in multi-task deep reinforcement learning,” in *International Conference on Learning Representations*, 2019.
- [119] S. Thrun and J. O’Sullivan, “Discovering structure in multiple learning tasks: The tc algorithm,” in *ICML*, vol. 96. Citeseer, 1996, pp. 489–497.
- [120] T. L. Griffiths, F. Callaway, M. B. Chang, E. Grant, P. M. Krueger, and F. Lieder, “Doing more with less: meta-reasoning and meta-learning in humans and machines,” *Current Opinion in Behavioral Sciences*, vol. 29, pp. 24–30, 2019.

- [121] A. Xie, J. Harrison, and C. Finn, “Deep reinforcement learning amidst continual structured non-stationarity,” in *International Conference on Machine Learning*. PMLR, 2021, pp. 11 393–11 403.
- [122] R. Agarwal, M. Schwarzer, P. S. Castro, A. Courville, and M. G. Bellemare, “Beyond tabula rasa: Reincarnating reinforcement learning,” *arXiv preprint arXiv:2206.01626*, 2022.
- [123] A. Chen, A. Sharma, S. Levine, and C. Finn, “You only live once: Single-life reinforcement learning,” *Advances in Neural Information Processing Systems*, vol. 35, pp. 14 784–14 797, 2022.
- [124] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, “Prioritized experience replay,” *arXiv preprint arXiv:1511.05952*, 2015.
- [125] T. Hester, M. Vecerik, O. Pietquin, M. Lanctot, T. Schaul, B. Piot, D. Horgan, J. Quan, A. Sendonaris, I. Osband *et al.*, “Deep q-learning from demonstrations,” in *Proceedings of the AAAI conference on artificial intelligence*, vol. 32, no. 1, 2018.
- [126] B. O’Donoghue, R. Munos, K. Kavukcuoglu, and V. Mnih, “Combining policy gradient and q-learning,” *arXiv preprint arXiv:1611.01626*, 2016.
- [127] Z. Wang, V. Bapst, N. Heess, V. Mnih, R. Munos, K. Kavukcuoglu, and N. de Freitas, “Sample efficient actor-critic with experience replay,” *arXiv preprint arXiv:1611.01224*, 2016.
- [128] H. Nguyen and H. La, “Review of deep reinforcement learning for robot manipulation,” in *2019 Third IEEE International Conference on Robotic Computing (IRC)*, 2019, pp. 590–595.
- [129] K. Shao, Z. Tang, Y. Zhu, N. Li, and D. Zhao, “A survey of deep reinforcement learning in video games,” *arXiv preprint arXiv:1912.10944*, 2019.
- [130] X. Chen, L. Yao, J. McAuley, G. Zhou, and X. Wang, “A survey of deep reinforcement learning in recommender systems: A systematic review and future directions,” *arXiv preprint arXiv:2109.03540*, 2021.
- [131] W. Zhao, J. P. Queralta, and T. Westerlund, “Sim-to-real transfer in deep reinforcement learning for robotics: a survey,” in *2020 IEEE symposium series on computational intelligence (SSCI)*. IEEE, 2020, pp. 737–744.

- [132] “Stable baselines3,” Dec. 2022, original-date: 2020-05-05T05:52:26Z. [Online]. Available: <https://github.com/DLR-RM/stable-baselines3>
- [133] M. Plappert, “keras-rl,” <https://github.com/keras-rl/keras-rl>, 2016.
- [134] A. Kuhnle, M. Schaarschmidt, and K. Fricke, “Tensorforce: a tensorflow library for applied reinforcement learning,” Web page, 2017. [Online]. Available: <https://github.com/tensorforce/tensorforce>
- [135] “Quickstart — RLHive 1.0.1 documentation.” [Online]. Available: <https://rlhive.readthedocs.io/en/stable/quickstart.html>
- [136] A. Jones, “Debugging reinforcement learning systems.” [Online]. Available: <https://andyljones.com/posts/rl-debugging.html>
- [137] S. Deshpande, B. Eysenbach, and J. Schneider, “Interactive visualization for debugging rl,” *arXiv preprint arXiv:2008.07331*, 2020.
- [138] R. Rajan, J. L. B. Diaz, S. Guttikonda, F. Ferreira, A. Biedenkapp, J. O. von Hartz, and F. Hutter, “Mdp playground: A design and debug testbed for reinforcement learning,” *arXiv preprint arXiv:1909.07750*, 2019.
- [139] D. Ghosh, J. Rahme, A. Kumar, A. Zhang, R. P. Adams, and S. Levine, “Why generalization in rl is difficult: Epistemic pomdps and implicit partial observability,” *Advances in Neural Information Processing Systems*, vol. 34, pp. 25 502–25 515, 2021.
- [140] D. Mané *et al.*, “Tensorboard: Tensorflow’s visualization toolkit,” *Retrieved October*, vol. 8, p. 2021, 2015.
- [141] “Weights & biases – developer tools for ml.” [Online]. Available: <https://wandb.ai/site/>, <http://wandb.ai/site>
- [142] S. Cai, “Debug tensorflow models with tfdbg,” 2017.
- [143] Y. Duan, X. Chen, R. Houthoofd, J. Schulman, and P. Abbeel, “Benchmarking deep reinforcement learning for continuous control,” in *International conference on machine learning*. PMLR, 2016, pp. 1329–1338.
- [144] A. V. Clemente, H. N. Castejón, and A. Chandra, “Efficient parallel methods for deep reinforcement learning,” *arXiv preprint arXiv:1705.04862*, 2017.

- [145] F. L. Da Silva, P. Hernandez-Leal, B. Kartal, and M. E. Taylor, “Uncertainty-aware action advising for deep reinforcement learning agents,” in *Proceedings of the AAAI conference on artificial intelligence*, vol. 34, no. 04, 2020, pp. 5792–5799.
- [146] “Tensorboard: Tensorflow’s visualization toolkit.” [Online]. Available: <https://www.tensorflow.org/tensorboard>
- [147] B. Xin, H. Yu, Y. Qin, Q. Tang, Z. Zhu *et al.*, “Exploration entropy for reinforcement learning,” *Mathematical Problems in Engineering*, vol. 2020, 2020.
- [148] S. Wang, H. Liu, P. H. Gomes, and B. Krishnamachari, “Deep reinforcement learning for dynamic multichannel access in wireless networks,” *IEEE Transactions on Cognitive Communications and Networking*, vol. 4, no. 2, pp. 257–265, 2018.
- [149] T. Schaul, D. Borsa, J. Modayil, and R. Pascanu, “Ray interference: a source of plateaus in deep reinforcement learning,” *arXiv preprint arXiv:1904.11455*, 2019.
- [150] Z.-W. Hong, T.-Y. Shann, S.-Y. Su, Y.-H. Chang, T.-J. Fu, and C.-Y. Lee, “Diversity-driven exploration strategy for deep reinforcement learning,” *Advances in neural information processing systems*, vol. 31, 2018.
- [151] Y. Burda, H. Edwards, A. Storkey, and O. Klimov, “Exploration by random network distillation,” *arXiv preprint arXiv:1810.12894*, 2018.
- [152] O. Lockwood and M. Si, “A review of uncertainty for deep reinforcement learning,” in *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 18, no. 1, 2022, pp. 155–162.
- [153] W. R. Clements, B. Van Delft, B.-M. Robaglia, R. B. Slaoui, and S. Toth, “Estimating risk and uncertainty in deep reinforcement learning,” *arXiv preprint arXiv:1905.09638*, 2019.
- [154] T. Kobayashi and W. E. L. Ilboudo, “T-soft update of target network for deep reinforcement learning,” *Neural Networks*, vol. 136, pp. 63–71, 2021.
- [155] S. Kullback, “Letter to the editor: The kullback-leibler distance,” 1987.
- [156] M. Rawson and R. Balan, “Convergence guarantees for deep epsilon greedy policy learning,” *arXiv preprint arXiv:2112.03376*, 2021.
- [157] N. Humbatova, G. Jahangirova, G. Bavota, V. Riccio, A. Stocco, and P. Tonella, “Taxonomy of real faults in deep learning systems,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 1110–1121.

- [158] Z. Chen, H. Yao, Y. Lou, Y. Cao, Y. Liu, H. Wang, and X. Liu, “An empirical study on deployment faults of deep learning based mobile applications,” in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 674–685.
- [159] H. Abdi *et al.*, “The method of least squares,” *Encyclopedia of measurement and statistics*, vol. 1, pp. 530–532, 2007.
- [160] T. Chai and R. R. Draxler, “Root mean square error (rmse) or mean absolute error (mae),” *Geoscientific model development discussions*, vol. 7, no. 1, pp. 1525–1534, 2014.
- [161] “Toward debugging deep reinforcement learning programs with rlexplorer,” May 2023. [Online]. Available: <https://zenodo.org/record/7902443>
- [162] Alex, “Weird results when playing with DQN with targets,” Jul. 2019. [Online]. Available: <https://stackoverflow.com/q/57106676>
- [163] —, “Cartpole-v0 loss increasing using DQN,” Jul. 2019. [Online]. Available: <https://stackoverflow.com/q/56964657>
- [164] D. Kalra, “Deep Q score stuck at 9 for CartPole,” Dec. 2017. [Online]. Available: <https://stackoverflow.com/q/47750291>
- [165] S. Peterson, “Tensorflow DQN can’t solve OpenAI Cartpole,” Jan. 2019. [Online]. Available: <https://stackoverflow.com/q/54385568>
- [166] desert_ranger, “Why is my REINFORCE algorithm not learning?” Oct. 2022. [Online]. Available: <https://stackoverflow.com/q/74214034>
- [167] mrmjauh, “Function approximator and q-learning,” Aug. 2017. [Online]. Available: <https://stackoverflow.com/q/45886398>
- [168] H. U. Sheikh, “DQN not working Properly,” Dec. 2017. [Online]. Available: <https://stackoverflow.com/q/47643678>
- [169] Y. L, “Pytorch DQN, DDQN using .detach() caused very wild loss (Increases exponentially) and do not learn at all,” Nov. 2020. [Online]. Available: <https://stackoverflow.com/q/64690471>
- [170] Virus, “DQN Pytorch Loss keeps increasing,” Jun. 2021. [Online]. Available: <https://stackoverflow.com/q/67789148>

- [171] toenails_sauce, “Keras Q-learning model performance doesn’t improve when playing CartPole,” Jul. 2019. [Online]. Available: <https://stackoverflow.com/q/56816743>
- [172] L. Barazza, “How to implement DQN algorithm correctly,” Feb. 2018. [Online]. Available: <https://stackoverflow.com/q/49035549>
- [173] P. W. Glimcher, “Understanding dopamine and reinforcement learning: the dopamine reward prediction error hypothesis,” *Proceedings of the National Academy of Sciences*, vol. 108, no. supplement_3, pp. 15 647–15 654, 2011.
- [174] “Reinforcement Learning (Dqn) Tutorial — PyTorch Tutorials 2.0.0+cu117 documentation.” [Online]. Available: https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html
- [175] F. Wilcoxon, “Individual comparisons by ranking methods,” *Biometrics Bulletin*, vol. 1, no. 6, p. 80, Dec. 1945. [Online]. Available: <https://www.jstor.org/stable/10.2307/3001968?origin=crossref>
- [176] A. Vargha and H. D. Delaney, “A critique and improvement of the cl common language effect size statistics of mcgraw and wong,” *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000.
- [177] niko, “Loss not decreasing but performance is improving,” Nov. 2017. [Online]. Available: <https://stats.stackexchange.com/q/313876>
- [178] M. Mutti, R. De Santi, and M. Restelli, “The importance of non-markovianity in maximum state entropy exploration,” in *International Conference on Machine Learning*. PMLR, 2022, pp. 16 223–16 239.
- [179] J. Ash and R. P. Adams, “On warm-starting neural network training,” *Advances in Neural Information Processing Systems*, vol. 33, pp. 3884–3894, 2020.
- [180] E. Nikishin, M. Schwarzer, P. D’Oro, P.-L. Bacon, and A. Courville, “The primacy bias in deep reinforcement learning,” in *International Conference on Machine Learning*. PMLR, 2022, pp. 16 828–16 847.
- [181] C. Lyle, M. Rowland, and W. Dabney, “Understanding and preventing capacity loss in reinforcement learning,” *arXiv preprint arXiv:2204.09560*, 2022.
- [182] M. Biagiola and P. Tonella, “Testing the Plasticity of Reinforcement Learning-based Systems,” *ACM Transactions on Software Engineering and Methodology*, vol. 31, no. 4, pp. 80:1–80:46, Jul. 2022. [Online]. Available: <https://dl.acm.org/doi/10.1145/3511701>

- [183] Z. Zhu, K. Lin, and J. Zhou, “Transfer learning in deep reinforcement learning: A survey,” *arXiv preprint arXiv:2009.07888*, 2020.
- [184] I. J. Timm, S. Staab, M. Siebers, C. Schon, U. Schmid, K. Sauerwald, L. Reuter, M. Ragni, C. Niederée, H. Maus *et al.*, “Intentional forgetting in artificial intelligence systems: Perspectives and challenges,” in *Joint German/Austrian Conference on Artificial Intelligence (Künstliche Intelligenz)*. Springer, 2018, pp. 357–365.
- [185] C. Beierle and I. J. Timm, “Intentional forgetting: An emerging field in ai and beyond,” pp. 5–8, 2019.
- [186] J. Frankle and M. Carbin, “The lottery ticket hypothesis: Finding sparse, trainable neural networks,” *arXiv preprint arXiv:1803.03635*, 2018.
- [187] E. Malach, G. Yehudai, S. Shalev-Schwartz, and O. Shamir, “Proving the lottery ticket hypothesis: Pruning is all you need,” in *International Conference on Machine Learning*. PMLR, 2020, pp. 6682–6691.
- [188] A. E. Siswanto, “Block sparsity and weight initialization in neural network pruning,” Ph.D. dissertation, Massachusetts Institute of Technology, 2021.
- [189] C. M. Bishop, “Neural networks and their applications,” *Review of scientific instruments*, vol. 65, no. 6, pp. 1803–1832, 1994.
- [190] X. Xie, L. Ma, F. Juefei-Xu, M. Xue, H. Chen, Y. Liu, J. Zhao, B. Li, J. Yin, and S. See, “Deephunter: a coverage-guided fuzz testing framework for deep neural networks,” in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 146–157.
- [191] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.
- [192] R. S. Sutton, “Generalization in reinforcement learning: Successful examples using sparse coarse coding,” *Advances in neural information processing systems*, vol. 8, 1995.
- [193] E. Urtans and A. Nikitenko, “Survey of deep q-network variants in pygame learning environment,” in *Proceedings of the 2018 2nd International Conference on Deep Learning Technologies*, 2018, pp. 27–36.
- [194] G. Van Rossum and F. L. Drake, *Python 3 reference manual*. CreateSpace, 2009.

- [195] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, “{TensorFlow}: a system for {Large-Scale} machine learning,” in *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, 2016, pp. 265–283.
- [196] “Welcome to RLHive’s documentation! — RLHive 1.0.1 documentation.” [Online]. Available: <https://rlhive.readthedocs.io/en/stable/>
- [197] A. Hu, “Tf2-rl: Reinforcement learning algorithms implemented for tensorflow 2.0+,” 2020. [Online]. Available: <https://github.com/anita-hu/TF2-RL>
- [198] “Welcome to stable baselines docs! - rl baselines made easy — stable baselines 2.10.2 documentation.” [Online]. Available: <https://stable-baselines.readthedocs.io/en/master/>
- [199] L. Engstrom, A. Ilyas, S. Santurkar, D. Tsipras, F. Janoos, L. Rudolph, and A. Madry, “Implementation matters in deep policy gradients: A case study on ppo and trpo,” *arXiv preprint arXiv:2005.12729*, 2020.
- [200] N. Perino, “A framework for self-healing software systems,” in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 1397–1400.