POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Deep Reinforcement Learning in Software Engineering: Empirical Insights and Practical Guidelines

PAULINA STEVIA NOUWOU MINDOM
Département de génie informatique et génie logiciel

Thèse présentée en vue de l'obtention du diplôme de Philosophiæ Doctor
Génie informatique

Janvier 2025

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Cette thèse intitulée :

Deep Reinforcement Learning in Software Engineering: Empirical Insights and Practical Guidelines

présentée par **Paulina Stevia NOUWOU MINDOM**
en vue de l'obtention du diplôme de Philosophiæ Doctor
a été dûment acceptée par le jury d'examen constitué de :

DEDICATION

To my family.

ACKNOWLEDGEMENTS

First, I want to express my gratitude to both my supervisors, Pr. Foutse Khomh and Pr. John Mullins, for giving me the opportunity to do this PhD, for their advice which contributed to enhancing the quality of all my works and for their emotional and economic support and guidance throughout this winding road of research. Secondly, I would like to thank Dr. Amin Nikanjam for sharing an invaluable amount of time to revise all my works for the duration of my thesis. Furthermore, I also would like to thank my committee members Pr Michel Desmarais, Pr Heng Li and Pr Tse-Hsun (Peter) Chen for evaluating this thesis. Finally, I thank my dearest friends and family for their support during this adventure.

RÉSUMÉ

Le génie logiciel vise à développer des approches structurées et économiques pour garantir des logiciels de haute qualité et fiables, à travers les processus de conception, développement, assurance qualité et maintenance. Choisir la bonne approche est essentielle, car la technique inadéquates peut compromettre la qualité du produit final et retarder sa livraison.

Les revues de code et les tests manuels contribuent à améliorer la qualité des logiciels, mais ils sont coûteux, prennent du temps et n'éliminent pas toujours les fautes dans les systèmes complexes. Pour faire face ces défis, l'apprentissage automatique offre des solutions prometteuses pour automatiser certaines tâches du génie logiciel. Cependant, il est souvent difficile de savoir quelles approches d'automatisation conviennent le mieux à une tâche donnée. De plus, les approches actuelles sont confrontées à des défis comme la gestion de données non stationnaire ou la complexité implicite des tâches, ce qui souligne la nécessité d'études pratiques pour mieux cerner les limites de ces approches et améliorer leurs utilisations.

Dans cette thèse, nous avons mené plusieurs études empiriques et développés un outil pour explorer des tâches clés du génie logiciel et évaluer l'efficacité des approches d'automatisation actuelles. Ces travaux s'appuient sur trois constats majeurs : (1) Les performances des approches d'automatisation actuelles varient considérablement d'une tâche à l'autre, ce qui souligne l'importance d'orienter les professionnels du génie logiciel vers les approches les plus adaptées; (2) les solutions dites « généralistes », pré-entraînées sur des données variées, s'adaptent plus rapidement à de nouvelles situations et réduisent les efforts d'entraînement, ce qui les rend particulièrement utiles pour les tâches complexes ou dynamiques; et finalement (3) les techniques classiques d'automatisation peinent à s'adapter aux évolutions constantes des données, mais des approches basées sur l'apprentissage continu pourraient apporter des améliorations notables.

Nous débutons cette thèse par une analyse des différences de performances entre diverses techniques d'automatisation, appliquées à des tâches de test comme les tests de jeux vidéos et de priorisation des cas de tests. Cette analyse, basée sur des comparaisons statistiques, a conduit à trois recommandations clés pour guider les professionnels du génie logiciel dans le choix des outils. Ensuite, pour répondre au problème de temps d'entraînement longs et au manque de flexibilité des approches traditionnelles, nous avons évalué l'utilisation d'agents généralistes dans des tâches de génie logiciel. En entraînant ces agents à l'aide de petites quantités de données, nous avons démontré leur efficacité sur des tâches variées comme la localisation des fautes, la planification, ou encore lors des tests logiciels. Ces résultats nous ont permis de formuler cinq recommandations pratiques pour exploiter au mieux ces agents dans divers contextes. Enfin, nous nous sommes concentrés sur des tâches où les données évoluent constamment, comme la localisation des fautes, ce qui pose des défis particuliers pour les outils d'automatisation classiques. Nous avons développé un outil basé sur l'apprentissage continu, spécialement conçu pour s'adapter à ces évolutions. Nous avons montré empiriquement que cette approche peut améliorer la précision de localisation des fautes de manière significative, jusqu'à 87 % dans certains cas, tout en offrant une flexibilité d'utilisation selon la granularité des données.

Cette thèse constitue une avancée importante pour guider l'utilisation des approches d'automatisation dans le domaine du génie logiciel. Nous espérons que ces résultats et recommandations permettront aux professionnels du génie logiciel d'améliorer la qualité et l'efficacité des logiciels tout en exploitant pleinement les avantages des nouvelles technologies.

ABSTRACT

Software engineering focuses on developing structured and cost-efficient methodologies to ensure the delivery of high-quality, reliable software, through design, development, quality assurance, and maintenance. Selecting the right techniques is crucial, as inappropriate choices can undermine productivity and software quality. Manual peer reviews and testing improve software quality but are time-intensive and costly, often leading to undetected bugs in complex systems. To address these challenges, researchers can leverage Machine Learning techniques, such as Deep Reinforcement Learning, to automate Software Engineering tasks. Yet it is often unclear which automation techniques are suitable for the task at hand. Additionally, current automation techniques often struggle with task-specific challenges, such as non-stationary behavior and inherent complexity of tasks, highlighting the need for empirical studies to better understand the limitations of such automating techniques and optimize their applicability.

In this thesis, we conduct several empirical studies and propose a framework that tackles widely used Software Engineering tasks and current Deep Reinforcement Learning techniques, to offer valuable recommendations and best practices for applying Deep Reinforcement Learning techniques to Software Engineering tasks. To do so, we built from the following observations: (1) Deep Reinforcement Learning techniques show significant performance differences within the same Software Engineering task, emphasizing the need for guidelines to assist practitioners in selecting the most appropriate technique for specific tasks; (2) unlike traditional Deep Reinforcement Learning techniques, pre-trained generalist agents adapt faster to new tasks and reduce training costs, making them more suitable for dynamic and resource-intensive Software Engineering tasks; and (3) Deep Reinforcement Learning techniques struggle with concept drift, but Continual Learning could improve adaptability and effectiveness when applied to dynamic software artifacts.

In this thesis, we start with an empirical study to investigate the performance differences of Deep Reinforcement Learning techniques applied to software testing tasks (i.e., playtesting in games and the test case prioritization tasks). The study consists of a statistical comparison of seven state-of-the-art Deep Reinforcement Learning techniques based on both task-related and Deep Reinforcement Learning-related metrics. It leads to three general recommendations for Software Engineering practitioners looking to adopt the most suitable Deep Reinforcement Learning techniques for the Software Engineering task at hand. Then, to address the long training time and lack of adaptability of current Deep Reinforcement Learning techniques, we conduct a study that leverages Deep Reinforcement Learning generalist agents on Software Engineering tasks. Deep Reinforcement Learning generalist agents differ from traditional Deep Reinforcement Learning agents in that they are pre-trained on large task-agnostic data. We show the adaptability and efficiency of generalist agents by fine-tuning them with up to 2% training data of three different Software Engineering tasks: software testing, task scheduling and bug localization. On the top of obtained results, we provide five recommendations for Software Engineering practitioners looking to leverage Deep Reinforcement Learning generalist agents for resource-intensive Software Engineering tasks. Finally, we notice non-stationary behavior in some Software Engineering tasks (e.g., the bug localization and test case prioritization tasks), making it challenging for current Deep Reinforcement Learning techniques, even the pre-trained ones, to adapt to the drift associated with data. Therefore, we propose a Continual Learning framework tailored to the bug localization task, as this task exhibits non-stationary behavior due to the evolving nature of software projects where software bugs reside. We empirically show that our Continual Learning framework localize software bugs up to 87% more accurately than some other state-of-the-art Machine Learning techniques in non-stationary software projects. Moreover, the proposed Continual Learning setup can be used at different data granularity, enabling Software Engineering practitioners to effectively apply it to fine-grained software artifacts.

This thesis is an important first step towards the elaboration of guidelines on leveraging Deep Reinforcement learning for Software Engineering tasks. We believe that our results and recommendations will help Software Engineering practitioners improve the quality of Deep Reinforcement Learning-based software activities.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ACRONYMS

AI	Artificial Intelligence
APFD	Average Percentage of Faults Detected
AST	Abstract Syntax Tree
BERT	Bidirectional Encoder Representations from Transformers
CL	Continual Learning
CLEAR	Continual learning with experience and replay
CLES	Common Language Effect Size
CNN	Convolutional Neural Network
DAG	Directed-Acyclic Graph
DDPG	Deep Deterministic Policy Gradient
DDQN	Double Deep Q-Network
DNNs	Deep Neural Networks
DQN	Deep Q network
DRL	Deep Reinforcement Learning
DT	Decision Transformer
EWC	Elastic weight consolidation
EMOO	Evolutionary Multi-Objective Optimization
GNN	Graph Neural Network
GPT	Generative Pre-trained Transformer
IMPALA	Importance Weighted Actor-Learner Architectures
IVFPQ	InVerted File with Product Quantization
JSSP	Job Shop Scheduling Problem
LEXA	Latent Explorer Achiever
LLM	Large Language Model
LSTM	Long Short-Term Memory
MAP	Mean Average Precision
MAENT	Max-entropy sequence modelling
MDP	Markov Decision Process
MGDT	Multigame Decision Transformer Architecture
ML	Machine Learning
MSE	Mean Squared Error
NLP	Natural Language Processing
NRPA	Normalized Ranking Performance Average

PDR	Priority Dispatching Rule
PPO	Proximal Policy Optimization
ReLU	Rectified Linear Unit
RL	Reinforcement Learning
SAC	Soft Actor-Critic
SE	Software Engineering
SL	Supervised Learning
SARSA	State–Action–Reward–State–Action
TRPO	Trust-Region Policy Optimization
VIF	Variance Inflation Factor
VSM	Vector Space Model

LIST OF APPENDICES

CHAPTER 1 INTRODUCTION

1.1 Research Context

Software is deeply integrated into almost every aspect of modern life and is used globally in various sectors. By the end of 2025, the software market is projected to reach 823.92 billion USD in revenue[1]. However, a study by the Consortium for Information and Software Quality[2] reported that poor software quality cost the US 2.41 trillion USD in 2022.

The Software Engineering (SE) community initially emerged to address the growing complexity of programming involved in the development of a software product [9]. As computing expanded beyond specialized laboratory use in the late 1950s and became integral to various sectors, programming evolved into a professional field [9]. With the development of versatile programming languages in the 1960s, existing methods encountered new challenges, often being underdeveloped and poorly understood. As a result, many projects were announced but could not be completed on time, pushing some major corporations to the brink of collapse. The 1968 NATO conference introduced the terms *software engineering* and *software crisis*: software engineering is considered a solution to the software crisis. Specifically, software engineering discusses systematic and cost-effective activities to manage the growing complexities of the development of a software product and ensure high-quality, reliable software. The IEEE Standard (IEEE Std 729-1983) defines software quality as *the totality of features and characteristics of a software product that affect its ability to satisfy given needs*. How do we ensure these features and characteristics continually satisfy the given needs? Nowadays, programming is just part of the development of a high-quality software product, which involves the following activities: requirements engineering, software design, software development, software quality assurance, software maintenance and software management [10]. Throughout these activities, the techniques employed aim to simplify the development process, increase developer productivity, and enhance the quality of the final software product. However, choosing one technique over another is not trivial; employing inappropriate techniques can lead to severe consequences.

Traditionally, manual techniques such as code reviews, pair programming, and manual testing have been widely used across software development activities [11–15]. Although these approaches allow developers to apply expertise and intuition, they are often time-consuming, costly, and prone to issues such as undetected bugs in complex systems [16]. Automat-

[1]`https://www.precedenceresearch.com/software-market`
[2]`https://www.it-cisq.org/press-releases/12-06-22/`

ing the development of a software product has garnered the attention of the SE community [17–19], intending to reduce manual effort, development costs, and ultimately the time it takes to deploy new features to production. Machine Learning (ML) can be leveraged to address SE tasks through automation, notably with Deep Reinforcement Learning (DRL) techniques [1, 16, 20–23]. Employing these techniques can help alleviate the burden on developers, enhance efficiency, increase trust in the testing process, and reduce development errors.
However, current automation techniques are not always well-suited to the specific tasks they are intended to address [24]. This mismatch can be attributed to the limitations of the techniques themselves [16], the inherent complexity of the SE tasks at hand, which may be difficult to solve [25]. For instance, different DRL techniques have been employed to automate the regression testing process [1, 22]. Bagherzadeh et al. [1] leveraged state-of-the-art DRL techniques to find the optimal prioritized test cases. Shi et al. [22] employed DRL techniques to optimize test case sorting, highlighting that the execution history of the last four cycles significantly influences current cycle sorting. While DRL techniques have shown good results, it remains unclear to the SE community what factors should motivate the choice of one technique over another. Additionally, these DRL techniques often need long training time due to the large amount of training data [1,6]. Since the automation of SE tasks should reduce development costs, we believe that SE practitioners could benefit from techniques that require less amount of training data while remaining efficient at solving the task at hand. Finally, the nature of certain SE tasks (e.g., the bug localization [23] and test case prioritization [1] tasks) can exhibit non-stationary behavior [26], which must be considered when leveraging DRL techniques. This contrasts with the stationary behavior of data (i.e., stationary data) where underlying patterns and relationships in the data do not evolve significantly over time. The variability in the data should influence the selection of appropriate DRL techniques, yet current techniques are not always well-suited to handle such dynamic conditions. Empirical studies are essential for SE practitioners to evaluate the effectiveness and applicability of current DRL techniques in addressing SE tasks [1, 27, 28]. These studies should account for the inherent complexity of the tasks, the associated development costs, and the limitations of both the techniques and the tasks they aim to solve.

1.2 Research Statement

Through our investigation of current DRL techniques for software engineering tasks, we made the following observations:

- The performance of DRL techniques in software engineering tasks varies significantly, making it challenging to determine which technique is most effective for a given task. It can be beneficial for SE practitioners to have guidelines to help them choose one DRL technique over another.

- Traditional DRL techniques perform well on specific SE tasks but take a long time to train because they need a lot of task-specific data. They also struggle to adapt to new or changing tasks, making them less useful in real-world settings. In contrast, generalist DRL agents pre-trained on large task-agnostic datasets, could reduce training costs on task-specific data, while being able to adapt to such data.

- Current DRL techniques, even the ones leveraging pre-trained models, for bug localization failed to address the concept drift associated with software artifacts, leading them to underperform when software artifacts evolve. Continual Learning (CL) techniques can be utilized at different data granularities to adapt to dynamic changes in software artifacts.

1.3 Thesis Overview

DRL-based techniques for SE tasks (1) vary in performance across different DRL techniques; (2) face challenges with long training times and poor adaptation to evolving SE tasks; and (3) can be enhanced by DRL generalist agents and CL to reduce training costs and adapt to concept drifts.

1. To effectively choose a DRL technique (i.e., DRL framework/algorithm) in the context of software engineering tasks, we propose (in Chapter 4) a comprehensive comparison of different DRL frameworks applied to two software testing techniques: the playtesting in games and the test case prioritization techniques. We aim to investigate which DRL frameworks may be more suitable, thus providing meaningful insights that the SE community can use.

2. To effectively leverage generalist agents in the context of SE tasks, we empirically investigate (in Chapter 5) generalist agents, a different category of DRL algorithms, that train DRL agents on large task-agnostic datasets. This investigation aims to harness the prior knowledge and skills of DRL generalist agents applied to a range of SE tasks, to enhance their performance while reducing the cost of employing them.

3. To effectively leverage CL techniques for the bug localization task, we propose a CL framework (in Chapter 6) specifically designed for bug localization at different data

granularities (changeset-files or hunks) of software artifacts. The goal is to address the non-stationary nature of this task, which traditional DRL techniques may not fully account for, and utilize CL agents to reduce debugging time and improve software quality.

1.4 Contributions

This thesis contributes to the SE community by offering actionable recommendations and best practices for applying DRL techniques to SE tasks. In particular, the Research Questions (RQs) addressed in this thesis tackle widely used SE tasks and current DRL techniques employed, with an emphasis on the unique nature of these tasks and the limitations of the DRL techniques. The key contributions include:

Contribution 1: An empirical evaluation of different DRL frameworks on software testing tasks. Specifically we:

- Evaluate the usefulness of DRL on playtesting in games, by utilizing three state-of-the-art DRL frameworks: Stable-baselines, Keras-rl, and Tensorforce. We applied them to the BlockMaze game for bug detection and collected the number of bugs, the state coverage, the code coverage, the cumulative reward, the average training and prediction times. We have compared a total of seven DRL configurations and some of them outperform the existing work from the literature.

- Apply state-of-the-art DRL frameworks on two ranking models and collect results to evaluate their usefulness in prioritizing test cases. As metrics of comparison, we consider the Normalized Rank Percentile Average (NRPA), the Average Percentage of Faults Detected (APFD), and the average training and prediction times for each DRL configuration. The results collected are compared with the baselines and we derive conclusions regarding the most accurate DRL frameworks for test case prioritization. Our results show that in most datasets, the Stable-baselines framework performs better than Tensorforce and Keras-rl.

- We formulate three recommendations for SE practitioners looking to select a DRL framework; based on our observation of performance variations across frameworks implementing the same algorithm.

This contribution stems from our investigation of the following three RQs:

RQa$_1$: How does the choice of DRL framework affect the performance of the software testing tasks? In this question, we collect performance metrics from Stable-baselines, Keras-rl, and Tensorforce DRL frameworks and conduct statistical analysis to compare their performance against existing approaches, assessing the impact of each framework.

RQa$_2$: Which combinations of DRL frameworks-algorithms perform better (i.e., get trained accurately and solve the problem effectively)? In this question, we explore several combinations of DRL frameworks-algorithms applied to playtesting in games and the test case prioritization tasks. We conduct statistical analysis to identify which combination performs best among existing approaches and study DRL frameworks-algorithms.

RQa$_3$: How stable are the results obtained from the DRL frameworks over multiple runs? Finally in this RQ, we investigate whether or not the results we have from the same DRL algorithms in different DRL frameworks are similar in terms of NRPA, APFD, testing, and prediction times when running the trained agent for testing multiple times.

Contribution 2: An empirical evaluation of DRL generalist agents on SE tasks. Specifically we:

- Evaluate the usefulness of generalist agents on SE tasks using two pre-trained generalist agents: MGDT and IMPALA. We carefully fine-tuned these agents (on zero-shot, 1%, and 2% data budgets) on the Blockmaze and MsPacman games for playtesting and collected the time to find bugs, the cumulative reward, and the average training and testing times using model-free DRL algorithms. Similarly, we fine-tuned them on a scheduling-based task and collected the makespan, the cumulative reward, and the average training and testing times. Finally, we fine-tuned the pre-trained generalist agents to localize buggy files across six large-scale, open-source software projects. We collected metrics such as Mean Reciprocal Rank (MRR), Mean Average Precision (MAP), top@1, top@5, top@10, cumulative reward, and average training and testing times. We have evaluated a total of 59 configurations and some of them perform close to or better than the baseline specialist agents.

- Provide five recommendations for researchers looking to leverage generalist agents on SE tasks, as we found that generalist agents offer good transferability performance at low fine-tuning costs.

The goal of this contribution is to leverage generalist DRL agents on SE tasks by carefully fine-tuning their pre-trained models for playtesting in games, locating buggy source code files in software projects, and solving JSSP on two scheduling instances. To achieve this goal, we answered the following three RQs:

RQb$_1$: How do DRL generalist agents perform on software engineering tasks? In this question, we fine-tune generalist agents using different data budgets and evaluate their performance across various tasks. For video game testing, we measure the time taken to detect bugs. For task scheduling, we collect the generalist agents' makespan performance. In the bug localization task, we assessed performance using key ranking metrics, including the mean reciprocal rank, and the mean average precision. In addition to the task-related metrics, we collect the cumulative reward and the training and testing times for each SE task to gain a more complete understanding of the generalist agents' performance.

RQb$_2$: How do different DRL generalist agents perform on SE tasks compared to DRL specialist agents? In this question, based on the results of **RQb$_1$**, we compare the DRL generalist agents that performed best against existing approaches - that have more specialized techniques - based on different fine-tuning data budgets. We conduct statistical analysis between the best DRL generalist agents and existing approaches, assessing the impact of the DRL generalist agents.

RQb$_3$: How do different model-free DRL algorithms affect the performance of generalist agents on SE tasks? Finally in this question, we compare the model-free DRL algorithms used to fine-tune the DRL generalist agents against each other. We conduct statistical analysis to indicate the best model-free DRL algorithm.

Contribution 3: A CL-based framework for the bug localization task. Specifically we:

- Develop DRL-based CL agents for bug localization.
- Adopt both regularization and rehearsal approaches for the CL agents to locate buggy changeset-files and hunks in non-stationary settings.
- Enhance the performance of CL agents by incorporating prior knowledge of bug-inducing factors through a logistic regression model integrated into the agents' reward function.
- Evaluate the CL agents of the CL framework against three state-of-the art studies on non-stationary data collected from seven Apache software projects.
- We make our data and models publicly available in our replication package [29].

To evaluate CL agents leveraged for bug localization, we answer the following three RQs:

RQc$_1$: How do Deep Learning techniques for bug localization perform in non-stationary settings? The purpose of this question is to empirically evaluate the performance of current studies on non-stationary settings at changeset-files level. Between the reporting and fixing of a bug, the affected changeset-files in a software project can undergo various modifications while still containing faults. Current approaches are trained and evaluated on stationary data of these changeset-files. However, an effective bug localization technique should maintain its performance across different versions (e.g., stationary or non-stationary) of the buggy changeset-files.

RQc$_2$: Can CL techniques improve bug localization? In **RQc$_1$**, the results show that current techniques have difficulties adapting to the non-stationary nature of data across software projects. To address this problem, we employ CL agents that tackle the non-stationary nature of data by being sequentially trained on stationary and non-stationary data at different granularities across seven datasets. To mitigate the risk of catastrophic forgetting, we implement strategies that adjust to distribution shifts in the data being learned. We calculate evaluation metrics for each dataset and compare the performance of the CL agents against current techniques and each other. Additionally, we assess the performance difference of each CL agent when applied to stationary and non-stationary data of software projects.

RQc$_3$: Can prior knowledge about bug-inducing factors improve the performance of CL techniques? The purpose of this question is to evaluate the performance of the CL agents on both stationary and non-stationary data (i.e., changeset-files and hunks) by integrating prior knowledge of bug-inducing factors into their training phase. We calculate our evaluation metrics for each dataset and compare the performance of the CL agents against the baselines, both before and after including the bug-inducing factors. Additionally, for each enhanced CL agent, we calculate its performance difference when applied to stationary and non-stationary data of software projects.

The research work accomplished in this thesis led to the publication/submission of the following research papers:

- **Nouwou Mindom, P. S.**, Nikanjam, A., Khomh, F., 2023. A comparison of reinforcement learning frameworks for software testing tasks. Empirical Software Engineering, 28(5), 111.
- **Nouwou Mindom, P. S. N.**, Nikanjam, A., Khomh, F., 2024. Harnessing Pre-trained Generalist Agents for Software Engineering Tasks. Empirical Software Engineering, 30, 39.
- **Nouwou Mindom, P. S.**, Da Silva, Léuson, Nikanjam, A. and Khomh, F., 2024. Continuously Learning Bug Locations, Submitted to ACM Transactions on Software Engineering and Methodology (TOSEM).

The following papers were authored during my PhD. While they explore reinforcement learning agents, they do not align with the specific focus and narrative of the thesis and have therefore been excluded:

- **Nouwou Mindom, P. S.**, N., Nikanjam, A., Khomh, F., Mullins, J., 2021. On assessing the safety of reinforcement learning algorithms using formal methods. In 2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS) (pp. 260-269). IEEE.
- Tambon, F., Laberge, G., An, L., Nikanjam, A., **Nouwou Mindom, P. S.**, Pequignot, Y., ... Laviolette, F., 2022. How to certify machine learning based safety-critical systems? A systematic literature review. Automated Software Engineering, 29(2), 38.

1.5 Organization of the Thesis

This thesis is organized into seven chapters. **Chapter 2** introduces key concepts relevant to the topics addressed in this thesis, including Deep Neural Networks (DNNs) and various paradigms of ML, such as DRL and SL. It also covers how DNNs are trained and provides examples of DRL algorithms. Additionally, it describes the SE tasks utilized in this thesis and explains the statistical tests employed in the analysis. **Chapter 3** provides a comprehensive review of related studies and research work in the field. **Chapter 4** presents the first contribution, an empirical study on the application of state-of-art DRL algorithms from DRL frameworks on software testing tasks. Additionally, it provides recommendations for SE practitioners looking to apply them. **Chapter 5** proposes the second contribution, an empirical study on the applicability of DRL generalist agents to three SE tasks. Additionally, it provides recommendations to help SE practitioners make an informed decision when leveraging generalist agents to develop SE tasks. **Chapter 6** proposes the third contribution, a CL framework for the bug localization task. It empirically compares the effectiveness of CL agents from the framework against current techniques on stationary and non-stationary data. Additionally, it incorporates prior knowledge about bug-inducing factors to further enhance the CL agents' performance. Finally, **Chapter 7** concludes this thesis by summarizing the contributions and highlighting the limitations of the proposed contributions. It also provides possible future works to better equip SE practitioners looking to use DRL techniques when developing SE tasks.

CHAPTER 2 BACKGROUND

2.1 Chapter overview

In this chapter, we present the necessary background materials related to Deep Neural Networks (DNNs), machine learning paradigms (Supervised Learning, Unsupervised Learning, and Deep Reinforcement Learning (DRL)), and continual learning in the context of DRL and DRL frameworks. We describe the SE tasks employed in this thesis such as the testing of video games, the test case prioritization task, task scheduling, and the bug localization task. We also present and explain the statistical tests used in this thesis. This chapter is aimed at readers who are unfamiliar with these concepts.

2.2 Deep Neural Networks

With rising computational power, DNNs have become increasingly popular for capturing complex, high-dimensional data more efficiently than traditional Machine Learning (ML) models [2]. Their expressiveness makes them especially suitable for modeling complex data patterns in real-world applications. In the following, we describe the main DNNs architectures used in this thesis.

2.2.1 Feedforward Deep Neural Networks

Feedforward DNNs are composed of functions of the form $y = f[x, \phi]$, also called layers, with parameters ϕ that map multivariate inputs x to multivariate output y. Feedforward DNNs learn from available data D to approximate a function y, which should represent the underlying phenomenon of D [30]. We now consider an example of DNNs with two layers, depicted in Figure 2.1. The first layer is defined by:

$$\begin{aligned} h_1 &= a\left(\theta_{10} + \theta_{11} x\right) \\ h_2 &= a\left(\theta_{20} + \theta_{21} x\right) \\ h_3 &= a\left(\theta_{30} + \theta_{31} x\right) \end{aligned} \tag{2.1}$$

The second layer by:

$$\begin{aligned} h'_1 &= a\left(\psi_{10} + \psi_{11} h_1 + \psi_{12} h_2 + \psi_{13} h_3\right) \\ h'_2 &= a\left(\psi_{20} + \psi_{21} h_1 + \psi_{22} h_2 + \psi_{23} h_3\right) \\ h'_3 &= a\left(\psi_{30} + \psi_{31} h_1 + \psi_{32} h_2 + \psi_{33} h_3\right) \end{aligned} \tag{2.2}$$

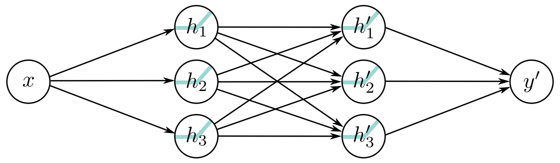


Figure 2.1 Example of Neural Network [2].

and the output by:

$$y' = \varphi'_0 + \varphi'_1 h'_1 + \varphi'_2 h'_2 + \varphi'_3 h'_3 \tag{2.3}$$

For each layer, $h_1, h_2, h_3, h'_1, h'_2$ and h'_3 are called hidden units, similar to y' are linear combinations of weights $(\theta_{11}, ..., \theta_{31})$ and $(\psi_{10}, ..., \psi_{33})$ and inputs, with eventually constant terms named bias $(\theta_{10}, \theta_{20}, \theta_{30}, \psi_{10}, \psi_{20}, \psi_{30})$. These DNNs are called "feedforward" as the information flows from the input layer to the output layer in one direction. Hence y' is obtained by linear combinations with inputs of the network's last layer. Each function is passed through an activation function $a[\cdot]$. Linear functions are not sufficient, as DNNs would be unable to solve non-linear problems such as the XOR problem. Activation functions enable DNNs to handle non-linearity in the data D. Several activation functions $a[\cdot]$ can be used, the most common ones are the Rectified Linear Unit (ReLU) [31], which is defined as

$$g(z) = \max\{0, z\}$$

or the Softmax/Sigmoid functions, which are defined for each element z_k of $\mathbf{z}$ as

$$a(z_k) = \frac{e^{z_k}}{\sum_p e^{z_p}} \quad \text{and} \quad a(z_k) = \frac{1}{1 + e^{-z_k}}.$$

2.2.2 Convolutional Deep Neural Networks

Convolutional layers are an improved version of DNNs, essential for handling 2D image data due to three key properties of images [2]. First, images are high-dimensional (e.g., 224x224 RGB, totalling 150,528 input dimensions), making DNNs impractical due to excessive memory and computation requirements. Second, neighbouring pixels in an image are statistically related, a feature DNNs cannot support. Third, images are resilient to small shifts; an object remains recognizable even if slightly moved. DNNs must re-learn features at each position, while convolutional layers use shared parameters to process local regions consistently across the image. Convolutional DNNs are characterized by the use of convolution operations in convolutional layers. In a 2D convolution, a hidden layer transforms an input matrix x into an output matrix h, where each output element $h_{i,j}$ is a weighted sum of nearby input elements. This transformation is achieved using a convolution kernel (or filter) ω, which applies the same set of weights across the input matrix x to extract spatial patterns. For example a 3×3 kernel $\Omega \in \mathbb{R}^{3\times3}$ applied to a 2D input comprising elements x_{ij} computes a single layer of hidden units h_{ij} as:

$$h_{ij} = a\left[\beta + \sum_{m=1}^{3}\sum_{n=1}^{3}\omega_{mn} \cdot x_{i+m-2,j+n-2}\right] \tag{2.4}$$

where ω_{mn} are the entries of the convolutional kernel. Convolution DNNs also make use of pooling layers, padding and stride operations. Pooling layers summarize the local output using statistical measures. One of the most commonly used operations is max pooling, which returns the highest value within a specified rectangular neighborhood [32]. Padding is employed to manage edge cases where the kernel extends beyond the input boundaries. For instance, with zero padding, additional values are added to the input boundaries, and it is assumed that these values are zero outside the valid range of the input data [2]. Stride refers to the number of positions by which the convolutional kernel is shifted when evaluating the output [2]. A stride of one means the kernel moves one position at a time, evaluating the output at every possible position

2.2.3 Transformer Architectures

Transformers are designed to process sequential data such as text. They address the challenges posed by large input sizes, as text can be represented as extensive vectors that make traditional DNNs impractical for long passages. Furthermore, compared to Recurrent DNNs or Long-Short Term Memory DNNs (also designed for sequential data), they excel at understanding context and resolving ambiguity in language, allowing words to establish meaningful connections with one another, regardless of their distance in the text [2]. Specifically, transformers use dot-product self-attention. A self-attention block $\mathrm{sa}[\cdot]$ considers N inputs $x_1, \ldots, x_N$, each of dimension $D \times 1$, and returns N outputs, which are also of size $D \times 1$. For instance in natural language processing (NLP), each input represents a word or fragment. First, a set of values is computed for each input :

$$v_m = \beta_v + \Omega_v x_m \tag{2.5}$$

where $\beta_v \in \mathbb{R}^{D\times 1}$ and $\Omega_v \in \mathbb{R}^{D\times D}$ represent biases and weights, respectively. Then the n-th output $\mathrm{sa}_n[x_1, \ldots, x_N]$ is a weighted sum of all the values $v_1, \ldots, v_N$:

$$\mathrm{sa}_n[x_1, \ldots, x_N] = \sum_{m=1}^{N} a[x_m, x_n]v_m \tag{2.6}$$

where the weight $a[x_m, x_n]$ is the attention that the n-th output pays to input x_m. The N weights $a[\cdot, x_n]$ are non-negative and sum to one. Figure 2.2 depicts the architecture of a transformer. Transformers are built with special layers called attention layers which consist of a multi-head self-attention unit followed by feedforward DNNs(see Figure 2.2). In addition, it is typical to add a LayerNorm operation after both the self-attention and feedforward DNNs. A typical task pipeline using transformer layers is as follows: First input text is split into words or fragments called tokens. Then each of these tokens is mapped to a learned embedding layer. The embeddings are passed through a series of transformer layers, collectively known as a transformer model. The overall architecture consists of blocks of encoders (on the left) and/or decoders (on the right). The encoder (e.g., BERT) transforms embeddings into encodings that support a variety of tasks while the decoder (e.g., GPT3) predicts the next token based on those encodings.

2.3 Machine Leaning Paradigms

Having described the different architectures of DNNs used in this thesis, we present the ML paradigms we will use.

2.3.1 Supervised Learning

In supervised learning, we consider an input x (features describing an object, an image, text, etc) and an output y which can be either discrete (for instance labels such as “buggy” or “not buggy”) or continuous (for instance the estimated value of a car). We aim to find a model that takes as input x and best approximates a prediction y. During the learning process, we assume that some input x and the corresponding output y are known in advance, and we want the model to correctly map the input to the output. An example of such a paradigm could be a feedforward DNN trained to classify buggy source code files based on known labels “buggy”, and “not buggy” also known as logistic regression. In that case, the output of the DNN becomes a score for each label representing the “likelihood” that the source code file belongs to this label according to the DNN. The Softmax activation can map any vector to a vector for which the dimensions are in [0, 1].

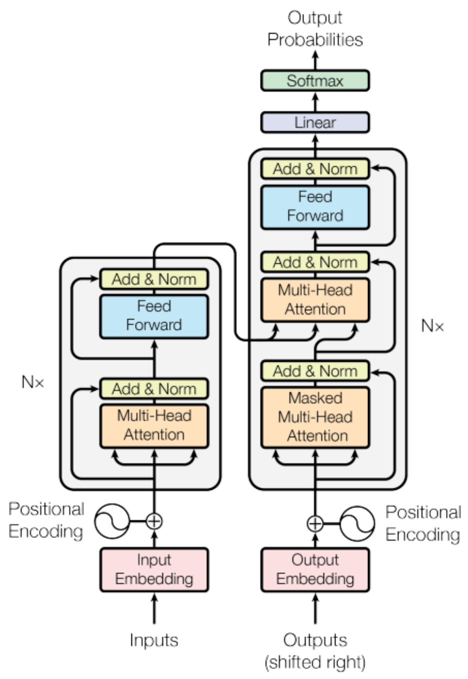


Figure 2.2 Transformer general architecture [3].

2.3.2 Unsupervised Learning

Unsupervised learning involves finding a mapping between data examples x and a set of unseen latent variables z. Unlike supervised learning, the outputs y are unknown. The latent variables capture the underlying structure of the dataset and are typically of lower dimensionality than the original data. For example, the algorithm k-means maps the data x to a cluster assignment $z \in 1, 2, ..., K$.

2.3.3 Deep Reinforcement Learning

A DRL agent interacts with the environment that can be modelled as a Markov decision process $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \gamma)$ with the following components:

State of the environment: A state $s \in \mathcal{S} = \mathbb{R}^n$ represents the agent perception of the environment.

Action: Based on the observation (i.e., state of the environment), the agent chooses among available actions in $\mathcal{A}$.

State transition distribution: $\mathcal{P} = \mathcal{P}(s_{t+1}, r_t | s_t, a_t) \;\; a_t \in \mathcal{A}$, defines the probability of the agent to move to the next state s_{t+1}, performing action a_t receives r_t as reward given that it is in state s_t . The goal of the agent is to maximize the expected rewards discounted by

γ. To make the decision to move to a state given its observation, the DRL agent follows a policy $\pi : \mathcal{S} \rightarrow \mathcal{A}$ which is a mapping from $\mathcal{S}$ to $\mathcal{A}$.

Episode: An episode is a sequence of states of the environment, actions performed by an agent and rewards (an incentive mechanism that tells the agent about the effectiveness of the action) which ends when the agent has reached a terminal state or has reached a maximum number of steps.

Policy. Given an agent, a policy π is defined as a function $\pi : S \rightarrow A$ mapping each state $s \in S$ to an action $a \in A$. The policy indicates the agent's decision in each state of the underlined task. It can be a strategy from a human expert or learned from experiences accordingly.

DRL algorithms can be classified based on the following properties similar to the work by Bagherzadeh et al. [1] :

Model-based and Model-free DRL. In model-based DRL, the agent knows the environment. It knows in advance the reaction of the environment to possible actions and the potential rewards it will get from taking each action. During training, the agent learns the optimal behavior by taking actions and observing the outcomes which include the next state and the immediate reward. On the contrary, in model-free DRL, the agent has to learn the dynamics of the environment by interacting with it. From the interaction with the environment, the agent learns an optimal policy for selecting an action. In this work, we are only interested in model-free DRL algorithms as some of the test case features (execution time) are unknown beforehand as well as the location of faults in a game.

Value-based, policy-based, and actor-critic learning. At every state, value-based methods estimate the Q-value and select the action with the best Q-value. A Q-value shows how good an action might be given a state. Regarding policy-based methods, an initial policy is parameterized, and then during training, the parameters are updated using gradient-based or gradient-free optimization techniques. Regarding actor-critic methods, the agent learns simultaneously from value-based and policy-based techniques. The policy function (actor) selects the action and the value function (critic) estimates the Q-values based on the action selected by the actor.

Action and observation space. The action space indicates the possible moves of the agent inside the environment. The observation space indicates what the agent can know about the environment. The action and observation space can be discrete or continuous. Specifically, the observation space can be a real number or high dimensional. While a discrete action space means that the agent chooses its action among distinct values, a continuous action space implies that the agent chooses actions among real value vectors. Not all DRL algorithms support discrete and continuous configurations for both the action and observation space, which limits the choice of algorithms to implement.

On-policy vs Off-policy. On-policy methods will collect data that is used to evaluate and improve a target policy and take action. On the contrary, Off-policy methods will evaluate and improve a target policy that is different from the policy used to generate the data. Off-policy learners generally use a replay buffer to update the policy.

DRL methods use Deep Neural Networks (DNNs) to approximate the value function, or the model (state transition function and reward function) and tend to be a more manageable solution space in large complex environments.

Decision transformers. A Decision Transformer (DT) is an architecture that tackles DRL as a sequential modeling problem [33]. It leverages transformer [3] architecture to predict future actions. Specifically, a trajectory is represented as a sequence of states, actions and return-to-go:

$$T = g_1, s_1, a_1 ... g_{|T|}, s_{|T|}, a_{|T|}$$

A DT learns a deterministic policy:

$$\pi(a_t \mid s_{-K,t}, g_{-K,t})$$

where K is the context length of the transformer, and $s_{-K,t}$ denotes the sequence of the last K states, which is the case for $g_{-K,t}$ as well. DT policy is parameterized through a Generative Pre-trained Transformer (GPT) [34] architecture to predict the next action sequence.

2.3.4 Training of Deep Neural Networks

When training DNN models, we seek the parameters ϕ that produce the best possible mapping from input x to output y for the task at hand. This is known as learning the network's parameters through training. To achieve this, we define a loss function $\mathcal{L}[\phi]$ that returns a single value quantifying the mismatch in the mapping. For example, the cross-entropy loss is generally used in supervised learning to measure the distance between the score of different labels and the ground truth label of y. It is defined as $-\sum ylog(\hat{y})$ for a given input x, the ground truth y and the prediction $\hat{y}$ using the approximate model. The Mean Squared Error (MSE) in supervised learning measures the average of the squares of the errors, representing the difference between the actual and predicted values. It is calculated using the formula $\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$, where n represents the number of inputs, y_i is the ground truth value of the

ith input, and $\hat{y}_i$ is the predicted value of the ith input. The objective of the optimization algorithm is to find parameters $\hat{\phi}$ that minimize the loss:

$$\hat{\phi} = \arg\min_{\phi} \left[L[\phi] \right]. \tag{2.7}$$

A common optimization algorithm is the gradient descent with initial parameters $\phi = [\phi_0, \phi_1, \ldots, \phi_N]^T$ and iterates through two steps. The computation of the derivatives of the loss with respect to each parameter:

$$\frac{\partial L}{\partial \phi} = \begin{bmatrix} \frac{\partial L}{\partial \phi_0} \\ \frac{\partial L}{\partial \phi_1} \\ \vdots \\ \frac{\partial L}{\partial \phi_N} \end{bmatrix}. \tag{2.8}$$

Then the update of the parameters according to the rule:

$$\phi \leftarrow \phi - \alpha \cdot \frac{\partial L}{\partial \phi}. \tag{2.9}$$

where the positive scalar α determines the magnitude of the update step and is called the learning rate.
A good value for α is typically determined through hyperparameter tuning. It is a process that consists of training your model sequentially with different sets of hyperparameter values to find the value that maximizes or minimizes a target variable.

2.3.5 Example of DRL algorithms

In the following, we give a high-level description of the DRL algorithms used in this thesis

Deep Q network (DQN) is an off-policy algorithm that uses a DNN to approximate the different state-action pair values (Q-values) for each possible action at a state (value-function estimation). DQN uses the Epsilon Greedy Strategy to choose actions. The Epsilon Greedy Strategy balances exploration and exploitation by sometimes choosing actions randomly or use the DQN model to choose actions. The DQN training algorithm consists of performing actions and storing the observed experience tuples in a replay memory. Then select a small batch of tuples randomly and learn from this batch using a gradient descent update step. The loss function compares the Q-value prediction and the Q-target as follows:

$$\mathcal{L}_{DQN}[\phi] = \Big(r[s_t, a_t] + \gamma \cdot \max_a \left[Q[s_{t+1}, a, \phi] \right] - Q[s_t, a_t, \phi] \Big) \tag{2.10}$$

Proximal Policy Optimization (PPO). PPO is an on-policy algorithm designed to improve training stability by restricting the extent of policy updates at each training step. To achieve this, PPO measures the difference between the current and previous policies using a ratio of their probabilities. This ratio is then clipped within a range $[1-\epsilon, 1+\epsilon]$, which limits the incentive for the new policy to deviate excessively from the previous one, thereby maintaining a "proximal" update and enhancing stability. PPO can be trained in actor-critic style where the actor takes the state value as input and outputs the respective action for the given state and the critic takes the state value as input and outputs the respective Q-value. The PPO loss function $\mathcal{L}_{PPO}$ is defined as:

$$\mathcal{L}_{PPO}(\theta) = \mathbb{E}_t\left[\min\left(r_t(\theta)\hat{A}_t,\ \text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}_t\right)\right] + c_1\mathbb{E}_t\left[(V_\theta(s_t) - V_t^{\text{target}})^2\right] - c_2 H(\pi_\theta(s_t)) \tag{2.11}$$

where:

- $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$: the probability ratio between the current policy π_θ and the previous policy $\pi_{\theta_{\text{old}}}$.
- $\hat{A}_t$: the advantage estimate at step t, which measures how much better an action a_t performed at a specific state compared to the average action.
- ϵ: the clipping parameter which restricts $r_t(\theta)$ within the range $[1-\epsilon, 1+\epsilon]$.
- $\min\left(r_t(\theta)\hat{A}_t,\ \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\right)$ ensures that updates to the policy are constrained to prevent excessive divergence from the old policy.
- $V_\theta(s_t)$: the value function predicted by the critic model for state s_t.
- V_t^{target}: the target value, often computed as the actual sum of rewards obtained in a fixed window of steps.
- c_1: a coefficient for balancing the impact of the value function loss term.
- $H(\pi_\theta(s_t))$: the entropy of the policy at state s_t, which encourages exploration by adding a penalty for low-entropy policies.
- c_2: a coefficient for the entropy bonus term.

The PPO training consists of first collecting a small batch of data. Then $\mathcal{L}_{PPO}$ is optimized for a fixed number of updates after each iteration of PPO.

Deep Deterministic Policy Gradient (DDPG). DDPG is an off-policy algorithm that can only be used for environments with continuous action spaces. It is similar to DQN but for continuous action spaces. DDPG has two learning components. The Q-learning and the policy learning components. The Q-learning component uses batches from the replay buffer, similar to DQN to minimize the mean square error loss as follows:

$$L(\phi, \mathcal{D}) = \underset{(s,a,r,s',d)\sim\mathcal{D}}{\mathrm{E}} \left[\Big(Q_\phi(s,a) - \big(r + \gamma(1-d) Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s')) \big) \Big)^2 \right], \tag{2.12}$$

where $\mu_{\theta_{\text{targ}}}$ is the target policy network. The Policy learning component aims to learn a deterministic policy $\mu_\theta(s)$ that provides the action maximizing $Q_\phi(s,a)$. Given that the action space is continuous and assuming the Q-function is differentiable with respect to the action, gradient ascent (with respect to the policy parameters) can be used to solve:

$$\max_\theta \underset{s\sim\mathcal{D}}{\mathrm{E}} \left[Q_\phi(s, \mu_\theta(s)) \right]. \tag{2.13}$$

where, the parameters of the Q-function are treated as constants.

Advantage Actor Critic (A2C). A2C is an on-policy and actor-critic algorithm. At each training step t, we obtain the current state s_t from the environment and pass it through both the actor and critic networks. The actor network takes s_t as input and outputs an action a_t, while the critic network takes s_t as input and output the expected $Q(S_t, A_t)$. The critic network also allows to estimate the advantage, so we can use the action-value in the policy loss. The loss function for the actor network is defined as follows:

$$\mathcal{L}_{\text{actor}}(\theta) = \mathbb{E}_t \left[\log \pi_\theta(a_t|s_t) \cdot \hat{A}_t \right] \tag{2.14}$$

where $\pi_\theta(a_t|s_t)$ is the probability of selecting action a_t in the state s_t according to the policy π_θ parameterized by θ, $\hat{A}_t = R_t - V_{\theta_v}(s_t)$ is the advantage function, $R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots$ is the cumulative return and $V_{\theta_v}(s_t)$ is the critic network's estimate of the value function at s_t. The loss function for the critic network is defined as follows:

$$\mathcal{L}_{\text{value}}(\theta_v) = \mathbb{E}_t \left[(R_t - V_{\theta_v}(s_t))^2 \right] \tag{2.15}$$

where $V_{\theta_v}(s_t)$ is the value estimate for state s_t by the critic network parameterized by θ_v.

V-trace is an off-policy actor-critic algorithm used to pre-train IMPALA, with the ability to correct the policy lag between actors and learners. As previously stated off-policy algorithms use the behavior policy μ to generate trajectories in the form of

$$(x_t, a_t, r_t)_{t=s}^{t=s+n}$$

then, use those trajectories to learn the value function V^π of another policy called target policy. V-trace target policy is formulated as:

$$v_s \overset{\text{def}}{=} V(x_s) + \sum_{t=s}^{s+n-1} \gamma^{t-s} (\prod_{i=s}^{t-1} c_i) \delta_t V \tag{2.16}$$

where a temporal difference for V is defined as

$$\delta_t V \overset{\text{def}}{=} \rho_t (r_t + \gamma V(x_{t+1}) - V(x_t))$$

where ρ_t and c_i are truncated Importance Sampling (IS) weights defined as:

$$\rho_t \overset{\text{def}}{=} min(\bar{\rho}, \frac{\pi(a_t \mid x_t)}{\mu(a_t \mid x_t)}),$$

$$c_i \overset{\text{def}}{=} min(\bar{c}, \frac{\pi(a_i \mid x_i)}{\mu(a_i \mid x_i)})$$

At training time s, the value parameters θ are updated by gradient descent on the mean squared error to the target function v_s in the direction of $(v_s - V_\theta(x_s))\Delta_\theta V_\theta(x_s)$, and the policy parameters ω are updated in the direction of the policy gradient:

$$\rho_s \Delta_\omega log \pi_\omega(a_s \mid x_s)(r_s + \gamma v_{s+1} - V_\theta(x_s))$$

Max-entropy sequence modelling (MAENT) was introduced by Zheng et al. [35] to fine-tune a pre-trained decision transformer via online interactions. The goal of the MAENT is to learn a stochastic policy that maximizes the likelihood of the dataset with its ability to balance the tradeoffs between exploration and exploitation. During fine-tuning, the agent has access via a replay buffer to a data distribution $\mathcal{T}$. Let τ denotes a trajectory with the length of $\mid \tau \mid$, where $a = (a_1, ..., a_{|\tau|})$, $x = (x_1, ..., x_{|\tau|})$, and $R = (R_1, ..., R_{|\tau|})$ denote the sequence of actions, states and returns of τ respectively. The constrained function to solve is formulated as follows:

$$\min_\theta \ J(\theta) \ subject \ to \ H_\theta^{\mathcal{T}}[a \mid x, R] \geq \beta \tag{2.17}$$

where β is a predefined hyperparameter, $J(\theta)$ denotes the objective function used to minimize the negative log-likelihood loss of the trajectories of the dataset:

$$J(\theta) = \frac{1}{K}\mathbb{E}_{(a,x,R)\sim\mathcal{T}}[-log\pi_\theta(a \mid x, R)]$$

and $H_\theta^{\mathcal{T}}[a \mid x, R]$ is the entropy of policy used to quantify exploration and is formulated as follows:

$$H_\theta^{\mathcal{T}}[a \mid x, R] = \frac{1}{K}\mathbb{E}_{(a,R)\sim\mathcal{T}}[H[\pi_\theta(a \mid x, R)]] \tag{2.18}$$

where $H[\pi_\theta(a_k)]$ denotes the Shannon entropy to calculate the entropy of the distribution $\pi_\theta(a_k)$.

2.4 State-of-the-art DRL Frameworks

In recent years, Lillicrap et al. [36], Mnih et al. [37] introduced multiple model-free DRL algorithms; advancing the research around DRL. Different DRL frameworks such as Stable-baselines [38, 39] and Tensorforce by Schaarschmidt et al. [40] have also been introduced to ease the implementation of DRL-based applications. These frameworks usually contain implementations of different DRL algorithms. While the developers may implement their own algorithm, in this work, we focus on comparing the implemented algorithms of existing DRL frameworks on software testing tasks. Following is a list of popular DRL frameworks:

- OpenAI baselines [41] is the most popular DRL framework given its high GitHub star rating. It provides many state-of-the-art DRL algorithms. After installing the package, training a model only requires specifying the name of the algorithm as a parameter.

- Stable-baselines [38, 39] is an improved version of OpenAI baselines with a more comprehensive documentation. In this thesis, we used version 3 of this framework, which is reported to be more reliable because of its Pytorch [42] backend that captures DNN policies. To train an agent, Stable-baselines has built-in functions that create a model depending on the DRL algorithm chosen.

- Keras-rl [43] provides the dueling extension of the DQN algorithm and SARSA algorithm that is not offered by Stable-baselines version 3. However, Keras-rl offers less algorithms than the previous frameworks.

  The training of an agent requires a few steps: the definition of the DNN that will be used for the training, the instantiation of the agent, its compilation, and finally the call of the training function.

- Tensorforce [40] provides the same algorithms as the Stable-baselines framework with some additions: Trust-Region Policy Optimization (TRPO), Dueling DQN, Reinforce, and Tensorforce Agent (TA). Tensorforce offers built-in functions to create and train an agent. Also, it offers the flexibility to train the agent without using the built-in functions, which allows it to capture the performance metrics of the agent, such as the reward. Finally, the training starts in a loop function depending on the number of episodes. Tensorforce relies on TensorFlow [44] as the backend.

- Dopamine [45] is a more recent framework that proposes an improved variant of the Deep Q-Networks (DQN) and the Soft Actor-Critic (SAC) algorithm. In addition to a TensorFlow backend for creating DNNs, Dopamine is configured using the gin [46] framework, to specify and configure hyperparameters. The training of an agent requires instantiating the model and then starting the training with built-in functions.

Based on their popularity and ease of implementation, we choose to rely on Stable-baselines, Tensorforce, and Keras-rl frameworks. Table 2.1 summarizes the implemented DRL algorithms available in these frameworks.

2.5 Playtesting in games

In the following, we present the necessary concepts to understand the process of testing video games. The process of finding bugs in a game [55, 56], is an essential activity before its release. Given the complexity of this activity, researchers have investigated various ways to automate it [16, 57]. In the following, we introduce some concepts that are important for understanding the process of testing video games through playtesting.

Definition 2.5.1. : **Game.** A game G is defined as a function $G : A^n \rightarrow (S \times R)^n$, where A is the set of actions that can be performed in the game by the agent playing it, S is the set of states of the game, R represents the set of rewards that come from the game, and n is the number of steps in the game. An agent undertakes a sequence of actions (n actions) based on the observations he has received up to the end of the game. If we view the game as an environment with which the agent interacts, each state refers to the observations of the environment the agent perceives at each time step. An action is a decision that is taken by the agent and that can be positively or negatively rewarded by the environment.

Figure 2.3 depicts an overview of the interaction between an agent and a game. Given a state s_t at time step t the agent takes an action a_t to interact with the game environment

Table 2.1 Comparison between DRL frameworks

RL Frameworks	**Algorithms**	**Learn.**	**On/Off**	**Act.**
Stable-baselines	DQN [47]	Value	Off-policy	Dis
	DDPG [36]	Policy	Off-policy	Cont
	A2C [37]	Actor-Critic	On-policy	Both
	TD3 [48]	Policy	Off-policy	Cont
	SAC [49]	Actor-Critic	Off-policy	Cont
	PPO [50]	Actor-Critic	On-policy	Both
Keras-rl	DQN [47]	Value	Off-policy	Dis
	Dueling DQN [51]	Value	Off-policy	Dis
	Double DQN [47]	Value	Off-policy	Dis
	SARSA [52]	Value	On-policy	Dis
	CDQN [53]	Value	On-policy	Cont
	DDPG [36]	Policy	Off-policy	Cont
Tensorforce	DQN [47]	Value	Off-policy	Dis
	Double DQN [47]	Value	Off-policy	Dis
	CDQN [53]	Value	On-policy	Cont
	PPO [50]	Actor-Critic	On-policy	Both
	DDPG [36]	Policy	Off-policy	Cont
	A2C [37]	Actor-Critic	On-policy	Both
	A3C [37]	Actor-Critic	On-policy	Both
	TRPO [54]	Actor-Critic	On-policy	Both
	TA [54]	Actor-Critic	On-policy	Both
	Reinforce [54]	Policy	On-policy	Both

Cont:continuous, Dis:discrete, Both: continuous and discrete

and receives a reward s_t from the game environment. The environment moves into a new state s_{t+1}, affecting the selection of the next action.

Definition 2.5.2. : **Game state.** A state in a game refers to its current status and can be represented as a fixed-length vector $(v_0, v_1, ..., v_n)$ [4]. Each element v_i of the vector represents an aspect of the state of the game such as the agent's position, and the location of the gold trophy in case of a Blockmaze game.

Definition 2.5.3. : **Game tester.** Given a game G, a set of policies Π to play G, a set of states S of G, and a set of bugs B on G, a game tester T can be defined as a function $T_G : \Pi \rightarrow S \times B$ [4].

A test case for a game is a sequence of actions. As a game might be non-deterministic (i.e., G is a stochastic function), each test case may result in various distinct states. A game tester [4] implements different strategies to explore the different states of the game to find bugs. The game tester, in this thesis, acts as an oracle indicating the existence of a bug in an output state. A game tester, therefore, generates a series of valid actions, as a test case, that leads to a state in which there might be a bug.

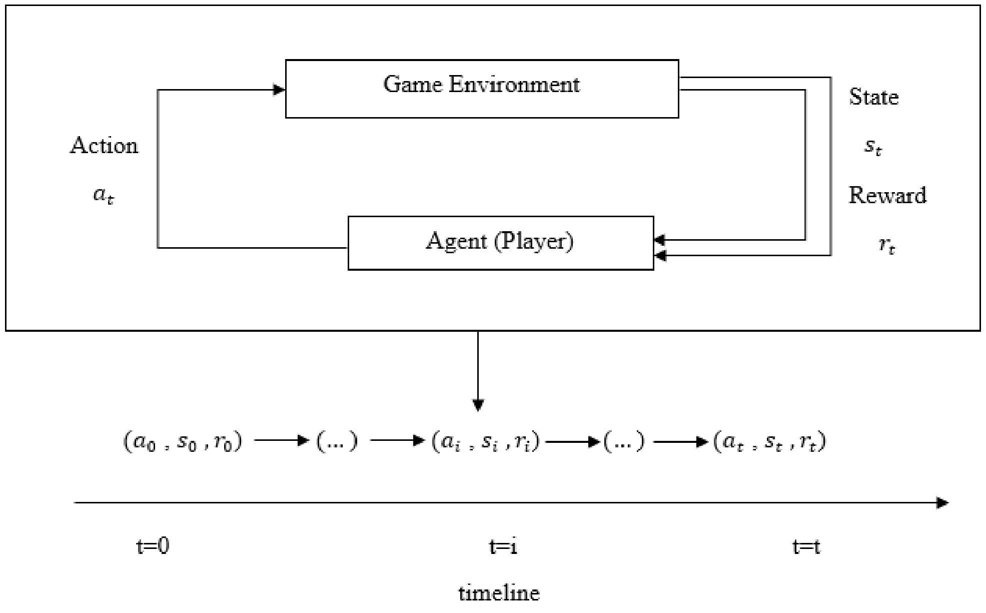

Game Environment
Action
a_t
State
s_t
Reward
r_t
Agent (Player)
$(a_0, s_0, r_0) \rightarrow (\ldots) \rightarrow (a_i, s_i, r_i) \rightarrow (\ldots) \rightarrow (a_t, s_t, r_t)$
t=0
t=i
t=t
timeline

Figure 2.3 The interaction between an agent and a game environment [4].

2.6 The test case prioritization task

Test Case Prioritization is the process of prioritizing test cases in a test suite. It allows to execute highly significant test cases first according to some measures, in order to detect faults as early as possible. In this thesis, we study test case prioritization in the context of Continuous Integration (CI).

Definition 2.6.1. : **CI Cycles.** A CI cycle is composed of a logical value and a set of test cases. The logical value indicates whether or not the cycle has failed. Failed cycles due to a test case failure are considered in this work, and we select a test case with at least one failed cycle.

Definition 2.6.2. : **Test case feature.** Each test case has an execution history and code-based features. The execution history shows a record of executions of test cases over the cycles. The execution history includes the execution verdict of a test case, the execution time, a sequence of verdicts from prior cycles, and the test age capturing the time the test case was introduced for the first time. The execution verdict indicates if the test case has failed or not. The execution time of a test case can be computed by averaging its previous execution times. The code-based features for a test case can indicate the changes that have been made, the impacted files with the number of lines of code that are relevant to predict the execution time , and can be leveraged to prioritize test cases.

Definition 2.6.3. : **Optimal ranking (Test Case prioritization)**. The test case prioritization process in this work is a ranking function that produces an ordered sequence based on the optimal ranking of test cases. The goal of prioritization is to get as close to this order as possible. The optimal ranking of a set of test cases is an order in which all test cases that fail are executed before test cases that pass. Furthermore, in this optimal ranking, test cases with a smaller time of execution should be executed sooner.

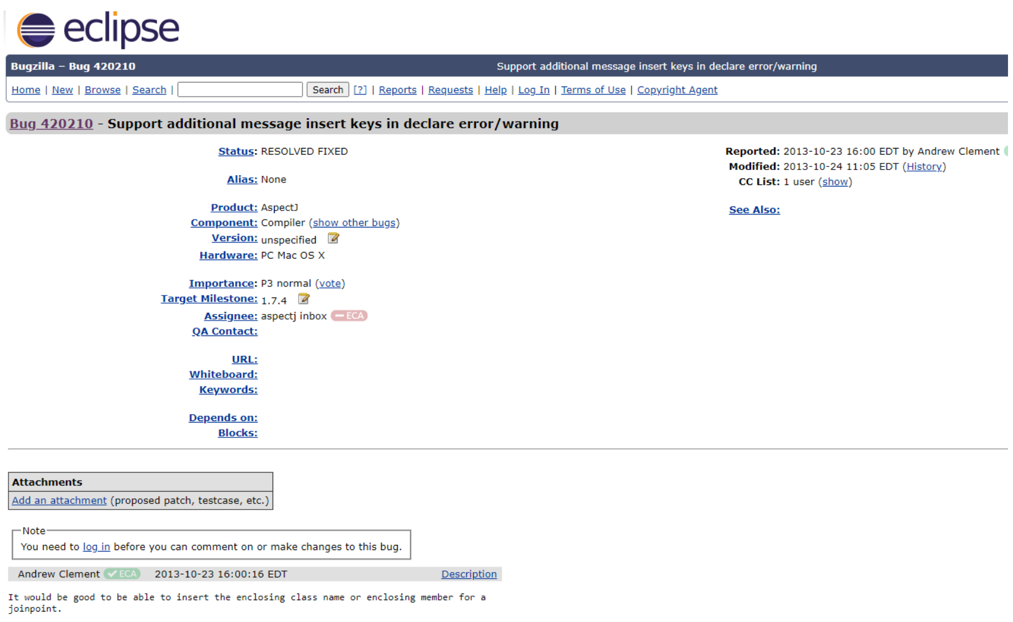
2.7 The bug localization task

The bug localization task is an essential activity in SE [58]. It involves identifying source code files that contain bugs based on provided bug reports. Figure 2.4 represents a sample bug report from the Eclipse Apache project [59]. A bug report typically contains a bug ID, title, textual description, and the version of the codebase where the bug exists.

A bug localization system retrieves all source code files from a specified version of a software project related to a bug report. For instance, if there are k source code files in the project at a specific version, the system will rank these k source code files based on their relevance to the related bug report. In this thesis, the ground truth is available, to evaluate the validity of the ranking of source code files. Therefore, the bug localization system is evaluated by comparing its output list of ranked source code files with the list of files that are effectively related to a bug report. Developers can compute metrics such as MRR to evaluate the bug localization system.

2.7.1 Bug Localization and Non-stationary Data

Bug localization systems are responsible for ranking software artifacts based on their probability of being associated with a given bug [60]. Initially, most applications rely on testing information, exploring the status of tests (passing or failing), and the related files continuously involved in the status of tests [61]. Over time, ML techniques have been adopted, improving these systems by incorporating information from several sources, including bug reports, logs, and the changeset-files itself [62]. However, even after filtering the target changeset-files, locating code snippets associated with the bug is a challenging task, leading researchers to explore more fine-grained information for bug localization [63]. Recent studies have explored the potential of elements from different sources, like hunks, from version control systems [64–66]. Changesets (i.e., changeset-files and hunks) represent a powerful source with promising findings, as they provide information from the source code level (lines added, removed, surrounding context) and metadata, like the author responsible for the applied


eclipse

Bugzilla – Bug 420210 Support additional message insert keys in declare error/warning

Home | New | Browse | Search | Search | [?] | Reports | Requests | Help | Log In | Terms of Use | Copyright Agent

Bug 420210 - Support additional message insert keys in declare error/warning

Status: RESOLVED FIXED

Alias: None

Product: AspectJ
Component: Compiler (show other bugs)
Version: unspecified
Hardware: PC Mac OS X

Importance: P3 normal (vote)
Target Milestone: 1.7.4
Assignee: aspectj inbox ECA
QA Contact:

URL:
Whiteboard:
Keywords:

Depends on:
Blocks:

Reported: 2013-10-23 16:00 EDT by Andrew Clement
Modified: 2013-10-24 11:05 EDT (History)
CC List: 1 user (show)

See Also:

Attachments
Add an attachment (proposed patch, testcase, etc.)

Note
You need to log in before you can comment on or make changes to this bug.

Andrew Clement ECA 2013-10-23 16:00:16 EDT Description

```
It would be good to be able to insert the enclosing class name or enclosing member for a
joinpoint.
```

Figure 2.4 Sample bug report [5].

changes, date, commit message, etc.

However, ML techniques face some challenges regarding the nature of information, as different terms are used for bug reports and changesets [64]. Compared to traditional ML, DL techniques have achieved better results by incorporating word embeddings, pre-trained models, and data augmentation, covering tasks related to classification, recommendation, and regression [67–70]. However, the evolution of changesets related to bug reports is not properly treated by DL models [71], knowing that such models are trained on stationary data. So, in bug localization, DL may fail to effectively handle concept drift [72] as they are trained on static data related to changesets and bug reports. A possible solution to the evolutionary nature of changesets is CL techniques that can adapt to these changes.

2.8 Job-Shop Scheduling Problem

In SE, the Job-Shop Scheduling Problem (JSSP) appears in several contexts, such as task scheduling in operating systems [73] (e.g., managing tasks or threads across processors or cores), build systems for software compilation [74] (e.g., optimizing build times by scheduling compilation tasks), resource allocation in cloud computing [75](e.g., efficiently distributing resources to reduce costs and maximize utilization), and data processing pipelines [76] (e.g., managing resources while handling data dependencies). It also affects project management [77, 78] (e.g., planning and executing tasks), network traffic scheduling [79], query scheduling in databases [80], and test case scheduling in automated testing [81] (e.g., optimizing execution times while covering all test scenarios). Each of these contexts involves optimizing resource use and managing dependencies to ensure efficient and timely task completion. In our thesis, we consider JSSP applied in the context of task scheduling in cloud computing systems to efficiently distribute cloud resources(e.g., networks, platforms, software, and applications) and optimize task completion time [73, 75, 82, 83]. In the rest of this section, we adopt the Priority Dispatching Rule (PDR) as a heuristic rule to solve the JSSP for task scheduling in cloud computing, where it is challenging to efficiently allocate cloud resources while reducing costs. In the following, we define some concepts that are important for understanding JSSP and PDR.

Definition 2.8.1. : **Job-Shop Scheduling Problem.** A JSSP instance consists of a set of jobs J and a set of machines M. A JSSP instance is denoted as $\mid J \mid \times \mid M \mid$, where each job $J_i \in J$ goes through $m_i \in M$ machines in a constraint order as follows:

$$O_{i1} \rightarrow O_{i2} \rightarrow ... \rightarrow O_{ij} \rightarrow ... \rightarrow O_{im_i}$$

where $1 \leq j \leq m_i$, and O_{ij} is an operation of $J_i \in J$ with a processing time of t_{ij}. Solving a JSSP instance means finding a schedule that starts at a startime ST_{ij} for operation O_{ij}, such that the makespan:

$$C_{max} = max_{i,j}(ST_{ij} + t_{ij})$$

is minimized.

Definition 2.8.2. : **Disjunctive graph.** A JSSP instance can be represented as a disjunctive graph [84] $G = (\mathcal{O}, \mathcal{C}, \mathcal{D})$ with

$$\mathcal{O} = \{O_{ij} \mid 1 \leq j \leq m_i\} \cup \{start, end\}$$

$\mathcal{O}$ is the set of all operations and i is the number of jobs; *start* and *end* are dummy operations with no processing time representing the start and the end of the schedule. $\mathcal{C}$ is a set of directed arcs between operations of the same job and $\mathcal{D}$ is a set of undirected arcs between operations requiring the same machine for processing. Consequently, finding a solution to a JSSP instance is equivalent to finding a direction for each undirected arc.

Definition 2.8.3. : **Priority Dispatching Rule for JSSP.** A PDR is a heuristic technique to solve the JSSP and is widely used in scheduling systems [6]. Given a set of operations, a PDR-based method can solve a JSSP instance by computing a priority index for each operation, then the operation with the highest priority is selected for dispatching. In this work, DRL is used to automatically generate PDRs.

2.9 Continual Learning in the context of DRL

In ML, CL represents a paradigm that explores learning based on the model's ability to acquire, update, accumulate, and exploit knowledge continually. Different from conventional ML models, CL is mainly known for learning from dynamic data, focusing on learning without forgetting previous knowledge (catastrophic forgetting) [85] and transferring knowledge from previous tasks for new ones [86]. Continual Learning for DRL extends CL by incorporating dynamic environments in the decision-making process through an agent that directly interacts with these dynamic environments [87]. Continual Learning for DRL faces the same challenges as CL, highlighting the need for the agent to constantly adapt itself to changes in the environment and ensure that it can still use new strategies, even based on known successful tasks.

2.10 Creation of DRL tasks

In this section, we mapped the studied SE tasks as a DRL process: the testing of two games, Blockmaze [16] and MsPacman [88], the bug localization task, the test case prioritization and a PDR-based task scheduling. We consider these tasks because previous studies have leveraged them [1, 6, 16, 23, 57], which we consider as baseline studies in this thesis.

1) The Blockmaze game can be mapped into a DRL process by defining the states (i.e., observations), actions, reward function, end of an episode, and the information related to the bug. The goal is to navigate the game until reaching the goal position.

Observation space: As mentioned in Definition 2.5.2, an observation is a set of features describing the state of the game. In our case, the observation space of the Blockmaze game has the size of a 20×20 matrix.

Action space: The action space describes the available moves that can be made in the Blockmaze by the agent. We consider the Blockmaze game with four discrete actions: north, south, east, and west.

Reward function: The reward function is the feedback from the environment regarding the agent's actions. It should be designed so that the agent can accomplish its mission. The agent is rewarded negatively when it reaches a non-valid position in the game or any other position that is not the goal position of the game. In all other cases, it receives a positive reward.

2) The MsPacman game is based on the classic Pac-man game in which the goal is to eat all the dots without touching the ghosts [89]. The environment of the MsPacman game is integrated into the gym Python toolkit [90].

Observation space: The observation space of the MsPacman game has the size of an 84×84 matrix.

Action space: The action space describes the available moves that can be made in the MsPacman game by the agent. It is a set of 5 discrete actions.

Reward function: The reward function provides +1 when the agent eats one dot and 0 otherwise [90].

3) The bug localization process can be considered as a DRL problem where the agent picks buggy files during an episode and ranks them based on their relevance to a bug report [23]. We employ the same environmental characteristics previous study by Chakraborty et al. [23]. The agent environment's state describes the bug reports and source code files associated with a particular commit SHA, its action corresponds to picking a file from the list of source code files into a ranked list. Due to the nature of DRL, passing a variable number of source code files to the generalist agent model is not feasible. Consequently, we filtered the source code files. We used ElasticSearch (ES), an open-source search and analytics engine that analyzes and indexes data with the BM25 algorithm [91]. We created an ES index from the source code files and queried it with the bug report to retrieve the top k files most textually similar to the bug report. ES ranks files based on their textual similarity to a query. In this thesis, the proposed DRL technique learns from feedback and aims to rerank the output from ES to get high-ranking metrics. Below is the definition of the states (i.e., observations), actions, reward function, and end of an episode of the bug localization task process

Observation space: An observation represents a bug report along with the top k relevant source files retrieved from ElasticSearch (ES). We employed CodeBERT [92], a transformer-based BERT model, to transform the source code and bug report texts into embeddings. Once the embeddings for the source code files $(F_1, F_2, \ldots, F_k)$ and the bug report (R) are obtained, we concatenate them to form part of the observation:

$$(E_1, E_2, \ldots, E_k) = (F_1 \| R, F_2 \| R, \ldots, F_k \| R).$$

The final observation consists of the list of embeddings $(E_1, E_2, \ldots, E_k)$ and a ranked list of source code files according to their relevance to the bug report (R). Initially, the ranked list is empty. During each transition in an episode, the agent moves one embedding from the list of embeddings $(E_1, E_2, \ldots, E_k)$ into the ranked list based on its relevance to the bug report (R).

Action space: An action corresponds to picking a file from the list of embeddings $(E_1, E_2, .., E_k)$

and moving it to the ranked list. The action space corresponds to a set of $k = 31$ files. Chakraborty et al. [23] argue that any values of k can be chosen depending on the capacity of the hardware used for training. To avoid out-of-memory errors during training the authors set k to 31.

Reward function: The agent is rewarded based on the rank of the relevant files and the distance between them in the ranked list [23]. The reward function is formulated as below:

$$\mathcal{R}(o, a) = \frac{M \times file\ relevance}{\log_2(t+1) \times distance(o)} \tag{2.19}$$

if a is an action that has not been selected before, otherwise:

$$\mathcal{R}(o, a) = -\log_2(t+1) \tag{2.20}$$

t is a step during an episode, o is an observation and a is an action. M is a value that scales the reward given to the generalist agent when it picks a relevant file. Through experimentation [23], different values$(1, 3, 6,$ and $9)$ were tested to see how they impacted the reward and, consequently, the performance of the model. The value 3 resulted in the highest reward for the DRL model. The distance distance(o) is calculated as the average of $\delta(i)$, where $\delta(i)$ represents the distance between consecutive relevant files of observation o. Given the positions of relevant files $\{f_1, f_2, \ldots, f_n\}$, $\delta(i)$ is defined as

$$\delta(i) = f_{i+1} - f_i, \quad \text{for } i = 1, 2, \ldots, n-1.$$

Finally, distance(o) is calculated as

$$\text{distance}(o) = \frac{1}{n-1} \sum_{i=1}^{n-1} \delta(i).$$

4)In this thesis, we consider a prioritization approach that consists of continuously interacting with the CI environment while improving the ranking strategy. In the CI environment, a DRL agent is used to automatically and continuously learn a ranking strategy as closely as possible to the optimal one. Specifically, the agent is trained on the CI environment by replaying the execution logs of available test cases from previous cycles in order to rank test cases in subsequent cycles. The main idea is to formulate the sequential interactions between CI and test case prioritization algorithm as a DRL problem. This way, state-of-the-art DRL techniques learn a strategy for test case prioritization, as close as possible to the optimal one, if we consider a predetermined optimal ranking as the ground truth. Using a CI environment simulator, the DRL agent is trained on the history of test execution and code-based features from previous cycles to prioritize test cases in the next cycles. We can benefit from an adaptive training process with DRL, meaning that the agent receives feedback (i.e., reward) at the end of each cycle (or when the prediction accuracy is below a particular level). To adapt the learned policy, the execution logs of test cases can be replayed several times to ensure an efficient and continuous adaptation to changes in the system and regression test suite. We leverage two ranking functions (i.e., pointwise and pairwise) that we mapped to a DRL process as follows:

Pointwise ranking function: Bagherzadeh et al. [1] designed the pointwise ranking model as a class on which the observation space, action space, and reward function are defined. This class consists of determining scores for each test case and then storing them in a temporary vector. At the end of the learning process, the test cases are sorted according to their scores stored in the temporary vector.

Observation space: The agent's observation is a record of the characteristics of a single test case with 4 numerical values.

Action space: The action describes a score associated with each test case. The agent uses this score to order the test cases. Each action is a real number between 0 and 1.

Reward function: The reward function is computed here based on the normalized distance between the assigned ranking and the optimal ranking. The values range between 0 and 1.

Pairwise ranking function: Bagherzadeh et al. [1] designed the pairwise ranking model as a class on which the observation space, action space, and reward function are defined. This class ranks the test cases using the selection sorting algorithm [93]. All test cases are divided into a pair: the sorted part on the left and the unsorted part on the right. At each time step, if a high-priority test case is found, its position is changed in the sorted part. The process continues until all test cases are sorted.

Observation space: An agent observation is a pair of test case records.

Action space: The action space values are 0 or 1. The first value indicates that the first test case in the observation has a higher priority.

Reward function: The reward function takes into account whether or not the test case with a higher priority fails. If this is the case, the agent receives the maximum reward of 1 otherwise it receives 0. In case the test cases in the pair have the same verdicts, the agent receives 0.5 as a reward when higher priority is given to the test case with less execution time otherwise it receives 0.

5) The last task we consider is a PDR task to solve JSSP. As mentioned in Definition 2.8.3, the goal of the PDR task is to dispatch all operations by minimizing the makespan. Similarly to the previous tasks, a PDR task can be mapped into a DRL process by defining the observations, actions, reward function, and end of an episode.

Observation space: The observation at each step t of the agent in the environment is a disjunctive graph:

$$G(t) = (O, C \cup D_u(t), D(t))$$

that represents the current status of solution for the task instance. $D_u(t) \subseteq D$ contains disjunctive arcs that have been assigned a direction at step t while $D(t) \subseteq D$ has the remaining arcs. At the beginning of an episode, the observation is the original JSSP instance, while at the end, it is the complete solution where all disjunctive arcs have been assigned a direction. An example of a 3×3 scheduling instance observation is depicted in Figure 2.5.

Action space: The action space describes operations that can be performed at each step t. Each job has a set of operations (as defined in Section 2.8), therefore the size of action space is $\mid J \mid$. As an episode evolves, the action space becomes smaller as more jobs are completed. An episode ends when all jobs have been completed.

Reward function: The reward function is the quality difference between the solution of each two consecutive states:

$$R = H(s_t) - H(s_{t+1})$$

where

$$H(s_t) = max_{i,j}\{C_{LB}(O_{i,j}, s_t)\}$$

is the lower bound of the makespan C_{max}. The goal is to dispatch operations step by step so that the makespan is minimized. Therefore, the cumulative reward obtained by the agent at each episode coincides with the minimum makespan [6].

2.11 Statistical Techniques for Analysis

The analysis of the results of our experiments in this thesis involves an analysis of variance (ANOVA test), Tukey's post-hoc test, Welch's ANOVA and Games-Howell post-hoc tests. In the following, we discuss these statistical techniques.

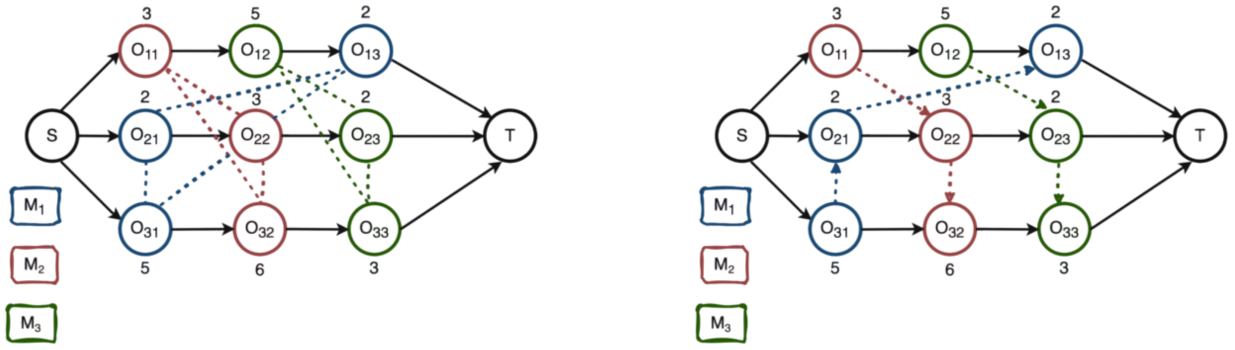


Figure 2.5 Example of a PDR task observation. The left figure represents a 3 × 3 JSSP instance. The black arrows represent conjunctive arcs and the dotted lines are disjunctive arcs grouped into machines of different colors. The right figure is a complete solution, where all disjunctive arcs are assigned with directions [6].

2.11.1 The ANOVA test

Statistical significance [94] helps determine whether observed differences in group means are due to inherent variability or experimental intervention. Two hypotheses are considered: the null hypothesis (differences are due to variability) and the alternate hypothesis (differences are due to experimental intervention). Type 1 error occurs when the null hypothesis is falsely rejected, and Type 2 error occurs when a true difference is not detected. A commonly accepted significance level for Type 1 error is 0.05 (5%). When comparing more than two groups of observations (k is the number of groups, n the number of observations in each group and N is the total number of observations), performing t-tests increases the Type 1 error rate, so ANOVA is used to maintain the error rate at an acceptable level(i.e., 5%). ANOVA tests whether group means are equal (null hypothesis) or if at least one mean differs (alternate hypothesis). It compares overall variation between and within groups using:

- The arithmetic mean of each group ($\bar{X_1}, \bar{X_2}, ..., \bar{X_k}$)
- The arithmetic mean of all observations or global mean ($\bar{X}_{GM}$)
- The sum of the squares between the group means and the global mean $SSB = n_1(\bar{X_1} - \bar{X}_{GM}) + n_2(\bar{X_2} - \bar{X}_{GM}) + ... + n_k(\bar{X_k} - \bar{X}_{GM})$
- The sum of the squares within each group $SSW = [\sum X_1^2 + \sum X_2^2 + ... + \sum X_k^2] - [\frac{\sum X_1^2}{n_1} + \frac{\sum X_2^2}{n_2} + ... + \frac{\sum X_k^2}{n_k}]$

- The degree of freedom between $dfb = k - 1$
- The degree of freedom within $dfw = N - k$
- The F ratio.
$$F = \frac{MSB}{MSW}$$
where MSB is the mean square between (calculated as $\frac{SSB}{dfb}$) and MSW is the mean squares within (calculated as $\frac{SSW}{dfw}$).

When the F ratio in an ANOVA is statistically significant, it indicates that at least one group's mean is different from the others, but it doesn't specify which one(s). To identify the specific group differences, additional calculations are needed through post hoc tests. These tests compare group means using a test statistic that is evaluated against a critical value based on significance level, degrees of freedom, and the number of groups. Post hoc tests are conducted only after a significant F ratio has been found. There is no universal post hoc test, and different tests have subtle differences.

2.11.2 Welch's ANOVA test

Welch's ANOVA [95] is a variation of the ANOVA that is used when the assumption of equal variances and sample size across groups is violated. Unlike regular ANOVA, Welch ANOVA adjusts the degrees of freedom to account for unequal variances and sample size across groups. The Welch-adjusted degrees of freedom is as follows:

$$\nu = \frac{\left(\frac{s_i^2}{n_i} + \frac{s_j^2}{n_j}\right)^2}{\frac{\left(\frac{s_i^2}{n_i}\right)^2}{n_i - 1} + \frac{\left(\frac{s_j^2}{n_j}\right)^2}{n_j - 1}}$$

s_i^2 and s_j^2 are the sample variances for groups i and j.

2.11.3 Tukey's post-hoc test

The Tukey [94] post hoc test is a statistical test that consists of doing pairwise comparisons of means while maintaining the type 1 error at the significance level of 5%. The Tukey post hoc test is performed after the ANOVA test to determine where the differences between groups lie and assume equal variances across groups. When conducting the Tukey test, q values are computed, which determine whether the difference between the largest and smallest means in a set of pairwise comparisons is statistically significant. The formula used to compute the statistic q is as follows:

$$q = \frac{(\bar{X_1} - \bar{X_2})}{\sqrt{[\frac{MSW}{2}(\frac{1}{n_1} + \frac{1}{n_2})]}}$$

2.11.4 Games-Howell post hoc test

Games-Howell test [96] is a post hoc test used after performing the Welch's ANOVA when the assumption of equal variances and equal sample size across groups is violated. It performs pairwise comparisons of means between groups while maintaining the type 1 error at the significance level of 5%. When conducting the Games-Howell test, the q statistic is computed similarly to the Tukey test and then the Welch-adjusted degrees of freedom is used to assess the significance of statistic q.

CHAPTER 3 LITERATURE REVIEW

3.1 Chapter Overview

First, we introduce various DRL algorithms, with a particular focus on the online DRL algorithms applied in this thesis. Next, we review related work on the use of DRL algorithms in software engineering tasks, including the application of generalist agents and CL techniques, with an emphasis on their relevance to the bug localization task.

3.2 Deep Reinforcement Learning algorithms

DRL is a computational approach that extends RL by integrating DNN presented in Chapter 2. In RL, an agent learns from its environment by taking actions, observing resulting states, and receiving rewards, which may be positive or negative [2]. To solve an RL problem, it is essential to find an optimal policy by using policy-based or value-based methods presented in Chapter 2. RL can be online [97] or offline [98]. In online RL the agent interacts directly with the environment to take actions while in offline RL take actions that maximize rewards by observing past sequences without ever interacting with the environment. Offline reinforcement learning is used when we cannot interact with the environment but must learn from past sequences. In this thesis, we focus on online RL. Key foundational algorithms in RL include Q-Learning [99], a value-based approach that uses the Bellman equation [100] to train its action-value function. In Q-learning, Q-values are encoded in a Q-table. However, this becomes challenging if the states and actions spaces are not small enough to be represented efficiently by tables. Algorithms such as DQN [101] presented in Chapter 2 introduce the use of DNNs to approximate Q-values, addressing the scalability issue of tabular approaches.

DQN algorithm faced some limitations especially when dealing with continuous action space. DQN assign a score for each possible action given by the state. In the case of continuous actions taking the max action of a continuous output is an optimization problem itself. Policy-based methods can be employed to address these as they output a probability over actions. DDPG [36] and REINFORCE [102] are examples of policy-based methods that directly learn to approximate a policy without having to learn a value function. Policy-based techniques like REINFORCE have high variance due to the stochastic nature of both the environment and the policy, leading to varying returns from the same starting state. To reduce this variance, REINFORCE requires many trajectories, which significantly reduces sample efficiency.

Actor-Critic methods improve on this by introducing a critic to estimate a stable value function as a baseline. This baseline reduces the variance in policy updates, allowing the actor to learn more efficiently and with fewer samples than REINFORCE alone. Actor-critic architectures include the A2C [37] and PPO [50], which controls policy updates for stability. This led to more advanced models capable of handling tasks like continuous control in environments such as the MuJoCo simulator [103].

In addition to classical DRL tasks, DRL is applied to multi-agent environments. Independent Q-Learning models provide a baseline, while newer models like Multi-Agent Deep Deterministic Policy Gradient (MADDPG) [104] handle cooperation and competition. The introduction of self-play [105] techniques where agents use former copies of its policy as an adversary allowed DRL agents to excel in games such as Go [106].

The recent rise of transformer architectures has further influenced DRL, allowing for more sophisticated sequential decision-making tasks. Models such as decision transformers presented in Chapter 2 explore how transformer-based architectures can extend DRL's potential in both temporal and spatial tasks.

3.3 Deep Reinforcement Learning for Software Engineering tasks

Software development has evolved from relying on the experience of software developers to data-driven approaches to address the increasing complexity and demands of modern software engineering (SE) tasks [107]. The shift to data-driven development is crucial, as SE data—comprising code bases, test cases, execution logs, historical changes, and bug reports—contains essential insights into software quality, development processes, and system evolution [107]. Solving SE tasks such as game testing, task scheduling, bug localization, and test case prioritization is critical for enhancing software reliability, efficiency, and maintainability. In the case of software testing task, bugs can lead to a poor gaming experience and result in financial losses for both players and game companies [16]. Many game companies depend on manual testing, which is costly and ineffective for identifying bugs in large-scale games [16]. Consequently, there is a need for automated testing techniques to enhance quality assurance in the game industry [16, 17, 19, 55].

Automated testing approaches, particularly those using DRL algorithms, have gained importance in achieving effective exploration of complex SE tasks [1, 16, 108] thanks to the availability of multiple DRL frameworks and DRL algorithms, e.g., A2C, DQN, PPO. DRL harnesses the power of DNNs to adapt and learn optimal testing strategies autonomously, which is particularly valuable in scenarios with large state spaces [16] and dynamic interactions [109]. This approach can lead to higher coverage, more efficient fault detection, and better prioritization of test cases compared to traditional methods [1]. In this thesis, we employ DQN, PPO and A2C algorithms from DRL frameworks to solve software testing tasks in Chapter 4. Further, we employ DQN and PPO algorithms in Chapter 5 to fine-tune DRL generalist agents on SE tasks. In the following, we discuss related work on using DRL algorithms for the SE tasks covered in this thesis (i.e., testing video games, task scheduling, and test case prioritization).

3.3.1 Deep Reinforcement Learning for Playtesting in Video games

As video games grow in popularity and complexity, game testing has become a crucial challenge and an essential factor for ensuring quality. Researchers studied testing video games by leveraging DRL algorithms in order to find bugs and improve the users' experience while playing.

Romdhana et al. [110] used the Stable-baselines framework for black box testing of android applications. Specifically, they developed a DRL-based tool for black-box testing of Android apps, was developed. Experiments demonstrate that ARES achieves better coverage and fault detection compared to baseline tools like TimeMachine and Q-Testing. A qualitative analysis revealed that Deep RL excels in apps with chained and blocking activities. Additionally, FATE was created to fine-tune Deep RL hyperparameters on simulated apps, reducing the computational burden of tuning on real apps.

Zheng et al. [16], developed a framework that applies evolutionary multi-objective optimization (EMOO), and DRL to facilitate automatic game testing. EMOO are designed to explore states and DRL ensures the completion of the mission of the game. Further, the authors use the Block Maze game presented in Section 2.10 and two commercial online games to evaluate wuji. This work is used as a baseline in Chapter 4. We compare the DRL part of wuji to state-of-the-art DRL algorithms from DRL frameworks. Specifically, we implement DRL algorithms from DRL frameworks to detect bugs in the Blockmaze game and assess their performance against the DRL part of wuji.

Koroglu et al. [111] proposed QBE, a Q-learning framework to automatically test mobile apps. QBE generates behavior models and uses them to train the transition prioritization matrix with two optimization goals: activity coverage and the number of crashes. Its goal is to improve the code coverage and the number of detected crashes for Android apps.

Pfau et al. [19] developed ICARUS a framework designed for autonomous video game playing, testing, and bug reporting. It details the design principles, implementation, and application in industry projects. ICARUS uses a dual discrete reinforcement learning mechanism incorporating both short-term and long-term memory across game iterations.

Bottinger et al. [112] introduced a program fuzzer that uses DRL to learn reward seed mutations for testing software. This technique obtains new inputs that can drive a program execution towards a predefined goal, e.g., maximizing code coverage.

Kim et al. [108], leveraged DRL to automatically generate test data from structural coverage. Particularly, a Double DQN agent is trained in a search-based software testing (SBST) environment to find a qualifying solution based on feedback from the fitness function.

Chen et al. [113] proposed RecBi, the first compiler bug approach via structural mutation that uses DRL. RecBi uses the A2C algorithm to mutate a given failing test program. Then, it uses that failed test program to identify compilers' bugs.

3.3.2 Deep Reinforcement Learning for Task Scheduling

In cloud systems, task scheduling aims to optimize resource utilization and achieve high quality of service (QoS) for both customers (e.g., software engineers) and providers [18, 114]. Cloud development is emerging and one main challenge that it faces is task scheduling [114].

Zhang et al. [6], another baseline study in Chapter 5, trained a specialist agent to solve the JSSP on scheduling instances. These studies are similar to our study in Chapter 5 as we examine specialist agents' performance on SE tasks. However, we compare specialist agents' performance against pre-trained generalist agents' performance. Following are some other related works leveraging DRL for task scheduling similar to our study in Chapter 5.

Mao et al. [115] explored systems capable of learning resource management strategies directly from experience. The studied approach, DeepRM frames the task of packing jobs with varying resource demands as a learning problem. Initial results indicate that DeepRM achieves comparably with state-of-the-art approaches, demonstrating quick convergence and adaptability to dynamic conditions while learning intuitive strategies for efficient resource management.

Zheng et al. [116] address the problem of efficient dispatching through DRL. They propose a novel 2-D state representation that incorporates job slack time and new reward functions for lateness and tardiness. To mitigate the cost of maintaining separate DRL models for each production line, a policy transfer approach is introduced, enhancing generalization and reducing training time. Experiments show that the approach improves performance and efficiency in dispatching tasks.

Mao et al. [117] present Decima, which uses DRL to create efficient, workload-specific scheduling policies with only a high-level objective, like minimizing job completion time. Standard DRL methods were insufficient, so new job representation techniques, scalable DRL models, and training methods for handling continuous job arrivals were developed. Decima demonstrated a 21% or greater improvement in average job completion time compared to hand-tuned heuristics, with up to a 2x boost under high load conditions.

Sun et al. [118] introduces DeepWeave, a DRL-based framework that uses Graph Neural Networks (GNNs) [119] to handle Directed-Acyclic Graph (DAG) job structures and improve scheduling. DeepWeave learns from historical workload data, enabling it to make scheduling decisions without relying on expert knowledge or predefined models. Simulations using real-life traces show that DeepWeave achieves job completion times faster than current state-of-the-art solutions.

Chraibi et al. [18] presents DMOTS-DRL, a new scheduling technique using multi-objective optimization, Dueling DQN, and dynamically prioritized experience replay to minimize makespan and power consumption. DMOTS-DRL shows superior performance compared to current techniques, achieving significant improvements in makespan and power consumption, along with enhancements in energy usage, balance, resource use, and waiting time. This demonstrates its effectiveness and reliability for cloud computing services.

Mangalampalli et al. [120] propose a DQN-based model that dynamically selects optimal resources for tasks. Inefficient scheduling can lead to increased energy consumption, SLA violations, and longer makespan. Evaluations using both random and real-world workloads demonstrate that the proposed approach outperforms baseline algorithms effectively reducing makespan, SLA violations, and energy consumption.

Lin et al. [121] propose a smart manufacturing framework that incorporates edge computing and applies it to the JSSP. The study adapts the DQN technique to an edge computing framework, extending it to handle decisions across multiple edge devices to solve the JSSP. Evaluations demonstrate that this multi-device DQN approach outperforms traditional methods relying on a single dispatching rule.

3.3.3 Deep Reinforcement Learning for Test Case Prioritization

Kim et al. [108] leveraged the Keras-rl framework to apply DRL to test data generation. This work explores using DRL in Search-Based Software Testing (SBST) as an alternative to traditional metaheuristic methods. It introduces GunPowder, a new SBST framework that reformulates the software under test as a DRL environment. The authors train a Double Deep Q-Network (DDQN) agent and conduct an empirical study to evaluate its effectiveness.

Similarly, Drozd et al. [122] used the Tensorforce framework to apply DRL to Fuzzing testing. The study harnesses both DRL advancements and fuzzing techniques, enabling deeper coverage across diverse benchmarks. The approach consists of learning mutation strategies and features a flexible, asynchronous architecture for applying various RL algorithms to fuzzing tasks with minimal performance impact.

Adamo et al. [123], build a DQN-based testing tool that generates test cases for Android applications. The tool is guided by the code coverage to generate suitable test suites. Reichstaller et al. [124], proposed a framework to test a Self-Adaptive System (SAS) where the tester is modeled as a Markov Decision Process (MDP). The MDP is then solved by using both model-free and model-based DRL algorithms to generate test cases that will adapt to SAS as they have the ability to make decisions at runtime.

Soualhia et al. [109] leveraged DRL algorithms to propose a dynamic and failure-aware framework that adjusts Hadoop's scheduling decisions based on events occurring in a cloud environment.

Spieker et al. [125] propose Retecs, a method that automatically learns to select and prioritize test cases in CI to minimize the time between code commits and developer feedback on failures. Retecs employs DRL to evaluate test cases based on their duration, execution history, and failure records. In dynamic environments, the method adapts by prioritizing error-prone test cases using a reward function informed by observations from previous CI cycles.

Bertolino et al. [126] explores the use of ML techniques for regression testing, highlighting two strategies: learning-to-rank and ranking-to-learn, the latter of which employs DRL. They introduce ten ML algorithms for CI practices and conduct a comparative analysis using data from the Apache Commons project. The results provide insights into how different features of code and testing processes influence the effectiveness of these algorithms. The study emphasizes the advantages of using RL for test prioritization, particularly its adaptability to changes in test suites and CI processes, allowing it to learn from its environment and improve prioritization dynamically.

Shi et al. [22] propose the use of DRL algorithms to optimize test case sorting through a reward mechanism, highlighting that the execution history of the last four cycles significantly influences current cycle sorting. A new reward function is introduced, leveraging weighted information from historical execution results to enhance system security. Evaluations with three industrial testing studies demonstrate that this approach improves fault detection rates, reduces execution time, and maximizes the number of fault-discovering test cases.

Each of these previous approaches from the literature either implements a DRL algorithm from scratch or uses an implemented one from a DRL framework. None of them has evaluated the performance of DRL frameworks for test case prioritization. Our work investigates various state-of-the-art DRL algorithms from popular DRL frameworks to assess DRL configurations on game and regression testing environments.

Bagherzadeh et al. [1] leveraged state-of-the-art DRL algorithms from the Stable-baselines framework in CI regression testing. They investigate pointwise, pairwise, and listwise ranking models as DRL problems to find the optimal prioritized test cases. The authors also conducted experiments on eight datasets and compared their solutions against a small subset of non-standard DRL implementations. We use this work as a baseline in Chapter 4 and implement DRL algorithms from DRL frameworks to rank test cases in a CI environment. As the authors implement the Stable-baselines framework, we leverage 2 other DRL frameworks (Tensorforce and Keras-rl) and compare them to Stable-baselines.

3.4 Generalist agents for Software Engineering tasks

Recent studies on solving SE tasks through DRL train an agent from scratch achieving a long training time due to a large amount of training data and developers spending a lot more time to solve them [6,16,23,57]. Moreover, researchers have observed that specialized DRL agents trained to excel at specific tasks often struggle to generalize to never-before-seen tasks (e.g., the testing of the same video game on which the agent was trained with added features).

Recently, researchers have started using pre-trained DL models in SE for code-related tasks. Wang et al. [127] exploited natural language pre-trained models in code-related tasks. Specifically, the natural language models are first augmented with semantic-preserving transformation sequences for pre-training purposes, then fine-tuned on two downstream tasks: code clone detection and code search tasks.

Kanade et al. [128] proposed cuBERT a contextual embedding of source code derived from BERT [129] model. CuBERT is pre-trained on a massive corpus of Python codes and then fine-tuned on downstream code-related tasks. The results show that by pre-training with cuBERT, the fine-tuned models outperformed the baseline model trained from scratch.

Similarly, Feng et al. [92] proposed codeBERT, a BERT transformer-based model pre-trained on code from github and fine-tuned for code search and documentation generation tasks.

In Chapter 5, we adopt BERT described in Section 2.2.3 as the model's architecture of the MGDT agent. Moreover, we pre-train our generalist agents to leverage prior skills during fine-tuning.

Mendonca et al. [130] proposed Latent Explorer Achiever (LEXA), a DRL-based agent capable of learning general skills in an environment where some states are estimated based on the uncertainty of the agent policy. LEXA is evaluated on robotic locomotion and manipulation tasks and outperforms prior studies.

Kalashnikov et al. [131] proposed a multi-robot learning system where multiple robots can learn a variety of tasks. The authors show that their proposed approach enables the robots to generalize to never-before-seen tasks and achieve high performance.

Other prior works include Lee et al. [8], Espeholt et al. [7], where authors developed generalist agents trained on the Atari suite environments and capable of achieving performance close to or better than the ones of specialist agents. In Chapter 5, we leverage generalist agents on SE tasks, which none of these studies had explored.

3.5 Continual Learning for bug localization

In this section, first, we discuss different techniques for bug localization that have been applied over time. Second, we discuss the challenges of dealing with concept drift in DL models.

3.5.1 Bug Localization Techniques

Previous studies have investigated different approaches to deal with bug localization. Unlike our approach in this thesis, some previous studies investigate the effort spent by developers to manually reproduce bugs, covering aspects related to bug monitoring, replaying, and reporting. For example, Moran et al. [132] propose a tool to augment crash detection reports exploring input generation for Android apps. Yu et al. [133] present a tool for supporting the reproduction of bugs based on log messages collected from system executions. Roehm et al. [134] explores the interaction between users and their applications, focusing on predicting failures based on a taxonomy of previously monitored user interactions. Neto et al. [135] improves the SZZ algorithm, which identifies bug-introducing changes, by addressing its tendency to misclassify refactoring changes as defects. Analyzing 10 Apache projects with 31,518 issues and 20,298 bug-introducing changes, the authors find that 6.5% of flagged changes and 19.9% of bug-fix modifications are refactorings. The approach enhances bug localization by reducing noise and false positives, leading to more precise defect identification.

Over time, many studies have investigated different information retrieval techniques. As we presented in Section 2.7, fine-grained information at different levels could still be associated with a bug [136–138]. Ranking changeset-files associated with a bug was one of the primary techniques used for bug localization. Chakraborty et al. [23] and Liang et al. [58], our baseline studies Chapter 6, are DL-based techniques that rank a set of changeset-files based on their relevance to a bug report. Ye et al. [139] evaluate their approach on the same software projects we consider for our evaluation in Chapter 6. Specifically, they leverage project-specific characteristics such as functional decomposition, API descriptions, bug-fixing history, code change history, and file dependency graphs to rank changeset-files relevant to a bug report using a learning-to-rank technique trained on previously resolved bug reports.

Wang et al. [140] present a CNN model based on their initial exploration of features from different dimensions between a bug report and a changeset-file.

Huo et al. [68] take a different route by proposing TRANP-CNN, a DL approach for extracting semantic features from source code projects and using them for cross-project bug localization.

Similarly, Miryeganeh et al. [141] present Globug, a framework that aims to improve the accuracy of previously pre-trained models for fault localization. The authors claim to leverage external knowledge to a given project, considering such an initial training dataset may not be rich enough (historical bug reports). This way, they advocate exploring external data (other projects) to calculate better the textual similarities of a new bug report to other historical bug reports or source code elements. These external data contain several open-source projects, later used to evaluate the potential of adopting a word embedding technique based on Doc2Vec.

Further exploring fine-grained information from source code, previous studies have investigated ranking methods associated with bugs rather than changeset-files, as recommending methods can significantly reduce the developer's inspection effort [142]. Almhana et al. [142] propose to rank the methods associated with a bug based on (i) the history of changes and bug-fixing, and (ii) the lexical similarity between the bug report description and the API documentation.

Previous studies have also investigated bug localization techniques that leverage both source code changes and associated metadata. For instance, changesets and hunks, extracted from version control systems, offer valuable metadata about ongoing modifications.

Ciborowska and Damevski [66] investigate how to leverage BERT to match the semantics reported in the bug reports with the inducing changesets (e.g., changeset-files and hunks).

Similarly, Wu et al. [65] propose ChangeLocator, a model for locating crash-inducing changes (changesets) based on crash reports.

Du and Yu [143] propose a Semantic Flow Graph, responsible for capturing the source code semantics. The authors address limitations in the representation learning of bug reports and changesets in BERT-based bug localization techniques. Specifically, they enhance the learning process by introducing a memory bank to store rich changesets from different batches, allowing the inclusion of large-scale negative samples.

Wen et al. [63] investigate how hunks, which are fine-grained segments of information consisting of changed and context lines within changeset files, could be used in a Vector Space Model (VSM) to retrieve bug-related information.

Although these previous studies present interesting and promising results, they do not cover the problem of concept drift, as we explore in Chapter 6.

3.5.2 Addressing Concept Drift

Despite promising results, some of bug localization techniques are trained on stationary data (e.g., changeset-files from a specific version of the software projects) [23,58]. Because software projects evolve over time, multiple versions of changeset files or hunks can be observed between the time a bug is reported and fixed. As a result, the data associated with bug reports and fixes is not static, which is referred to as non-stationary data distribution [144]. When ML-based techniques are evaluated on such non-stationary data, their performance may decrease because of concept drift [145]. Current ML techniques for bug localization may fail to tackle such drift in data by only considering one version of changeset-file or hunk [23, 58], which could make them less effective at ranking relevant buggy changeset-files across different versions of a software program. To adapt to changes in data distribution, models can be retrained from scratch on new changeset files (or hunks) or incrementally updated using all available changeset files (or hunks) up to the bug localization time [146–148].

Liu et al. [149] discuss the limitations of the current ML paradigms which operate on static datasets without retaining past knowledge. Resulting in a lack of memory, requiring extensive training examples and manual labelling. As the environment and tasks continuously change, the need for constant relabeling presents additional challenges. In contrast, humans learn cumulatively, integrating past experiences to enhance future learning. The authors advocate for lifelong ML as a more effective approach that mimics human learning processes, allowing for continuous knowledge accumulation and improved learning efficiency.

Mcintosh et al. [150] highlight the concept of change-level defect prediction, also known as Just-In-Time (JIT) defect prediction, which operates at a finer granularity than module-level prediction, allowing for more precise inspections and easier triaging due to the individual changes. However, JIT models assume that the properties of past fix-inducing changes are consistent with future changes, an assumption that may not hold as systems evolve. The authors investigate whether the important properties of these changes remain stable over time, revealing that JIT models lose predictive power and calibration sooner than expected. Additionally, the importance of various code change properties fluctuates, leading to potential misestimates of their future impact. To improve model performance, the authors recommend incorporating recent data into training sets and retraining JIT models regularly. They also suggest emphasizing larger data caches to better calibrate quality improvement plans.

Olewicki et al. [151] also investigate how to overcome concept drift due to the changing nature of development activities exploring lifelong learning, which is a paradigm that dynamically evolves a given model based on a stream of information [152]. For that, the authors perform case studies covering two different tools, responsible for detecting brown builds [26] and just-in-time risk prediction [153]. The authors report that the lifelong learning framework speeded up the training process, as updates used 3.3-13.7 times less data than the state-of-the-art framework.

Regarding the catastrophic forgetting problem, Kirkpatrick et al. [154] present an elastic weight consolidation algorithm, which remembers knowledge for old tasks by selectively decreasing the plasticity of weights important for the tasks under evaluation.

Specifically in the context of bug localization problems, Lee et al. [155] investigate how to assign appropriate developers to fix specific bugs, presenting the framework LBT-P. For that, the authors develop a fine-tuning method to preserve language knowledge. In Chapter 6, we aim to handle the catastrophic forgetting problem by exploring continual learning and showing relevant results compared to related work.

3.6 Chapter Summary

In this chapter, we briefly discussed the literature on DRL techniques applied to SE tasks. We also related literature about DRL generalist agents applied to SE tasks and continual learning techniques in the context of bug localization and the effort made by researchers to address concept drift. As presented in this chapter, several studies in the literature discuss DRL techniques (with specialist and generalist DRL agents) for SE tasks. However, none of them investigate their applicability across SE tasks or evaluate the implications of choosing one DRL technique over another. Therefore, there are no formal guidelines that SE practitioners could consider when leveraging DRL techniques. In the context of bug localization, none of the related studies addresses the concept drift associated with the data with continual learning techniques - which have shown to be an adequate strategy. This leaves SE practitioners with approaches that are not adequate at finding software artifacts relevant to a bug report. In this thesis, we aim to address these limitations by providing actionable guidelines for SE practitioners looking to leverage DRL techniques (with specialist and generalist DRL agents) while fully taking into account the inherent complexity (i.e., concept drift) of the task at hand.

CHAPTER 4 A COMPARISON OF REINFORCEMENT LEARNING FRAMEWORKS FOR SOFTWARE TESTING TASKS

4.1 Chapter Overview

This chapter studies various DRL algorithms across three frameworks— Stable-baselines, Keras-rl, and Tensorforce— to assess their effectiveness in software testing tasks, such as playtesting in games and test case prioritization. Developers have used DRL frameworks across various domains like software testing, but no empirical study has yet evaluated the effectiveness and performance of the implemented algorithms in these frameworks. We show that the diversity of hyperparameters available in each framework impacts their suitability for the task at hand. Moreover, we found that Tensorforce is more effective for bug detection due to its deeper exploration capabilities, while Stable-baselines excels in test case prioritization. This allows us to provide recommendations for selecting DRL frameworks, noting performance variations for the same DRL algorithm across frameworks.

4.2 Study design

In this section, we present the methodology of our study which aims to compare different implemented DRL algorithms from existing frameworks. We also introduce the two problems that we selected for this comparison.

4.2.1 Setting Objectives of the Study

Software bugs and failures have a significant financial impact on the global economy, costing trillions of dollars each year [156]. To mitigate these losses, researchers and practitioners have been working on developing efficient testing techniques to enhance the reliability of software systems before they reach the public. DRL has emerged as a promising approach in software testing, as explored by researchers like Zheng et al. [16], Bagherzadeh et al. [1], Moghadam et al. [21], and Malialis et al. [20]. DRL's growing popularity in testing can be attributed to the availability of DRL frameworks that provide pre-implemented algorithms, such as A2C, DQN, and PPO described in Chapter 2. While these frameworks offer various DRL implementations, applying them raises several questions we must tackle to apply them properly. Do the assumptions made by implemented DRL algorithms hold true for all software testing tasks, or only specific ones? How can developers and researchers determine the most suitable DRL implementation for their particular software testing task? What factors

Table 4.1 Datasets [1]

Dataset	Type	Cycles	Logs	Fail Rate(%)	Failed Cycles	Avg. Calc. Time (Avg) Enriched Features
Paint-Control	Simple	332	25,568	19.36	252	NA
IOFROL	Simple	209	32,118	28.66	203	NA
Codec	Enriched	178	2,207	0	0	1.78
Compress	Enriched	438	10,335	0.06	7	3.64
Imaging	Enriched	147	4,482	0.04	2	5.60
IO	Enriched	176	4,985	0.06	3	2.88
Lang	Enriched	301	10,884	0.01	2	5.58
Math	Enriched	57	3,822	0.01	7	9.46

influence the choice of a DRL algorithm in a DRL framework? Additionally, since DRL algorithms are often implemented differently across frameworks, can the same results be reliably achieved using different frameworks?

In this study, we perform a comprehensive comparison of different DRL algorithms implemented in three frameworks, i.e., Stable-baselines [38], Keras-rl [43], and Tensorforce [40] (see Chapter 2). We investigate which DRL algorithms/frameworks may be more suitable for detecting bugs in games and solving the test case prioritization problem by applying them to existing approaches. To do so, we analyzed how effective the studied DRL algorithms/frameworks are at testing compared to the existing approaches on different DRL environments and algorithms.

The findings of this study highlight that the diversity of hyperparameters that each framework provides impacts its suitability for each of the studied software testing tasks. We show that the Tensorforce framework tends to be more suitable for detecting bugs as it provides hyperparameters that allow a deeper exploration of the states of the environment while the Stable-baselines framework tends to be more suitable for the test case prioritization problem.

To structure the study in this chapter, we define three RQs ran through two important software testing problems:

RQa$_1$: How does the choice of DRL framework affect the performance of the software testing tasks? In this RQ, we collect performance metrics from Stable-baselines, Keras-rl, and Tensorforce DRL frameworks and conduct statistical analysis to compare their performance against existing approaches, assessing the impact of each framework.

RQa$_2$: Which combinations of DRL frameworks-algorithms perform better (i.e., get trained accurately and solve the problem effectively)? In this RQ, we explore several combinations of DRL frameworks-algorithms applied to the playtesting in games and the test case prioritization tasks. We conducted statistical analysis to identify which combination performs best

among existing approaches and studied DRL frameworks-algorithms.
RQa$_3$: How stable are the results obtained from the DRL frameworks, over multiple runs? Finally in this RQ, we investigate whether or not the results we have from the same DRL algorithms in different DRL frameworks are similar in terms of NRPA, APFD, testing and prediction times when running the trained agent for testing multiple times.

4.2.2 Problem 1: Playtesting in games using DRL

In this problem, we aim to use various DRL algorithms from different frameworks in a playtesting game environment to explore game states where bugs might hide. Our approach builds on Wuji [16], an automated game testing framework for bug detection, by analyzing the impact of different DRL algorithms on bug detection.

Datasets and Baselines

In the Blockmaze game, the player's objective is to reach the goal coin. It has 4 possible actions to choose: north, south, east, west. Every action leads to moving into a neighbor cell in the grid in the corresponding direction, except that a collision on the block (dark green) results in no movement. To evaluate the effectiveness of our DRL approaches, 25 bugs are artificially injected into the Blockmaze and randomly distributed within the environment. A bug is a position in the Blockmaze that is triggered if the robot (agent) reaches its location as in the map, as shown in Figure 5.3. A bug has no direct impact on the game but can be located in invalid locations of the game environment such as the Blockmaze obstacles or outside of the Blockmaze observation space. Invalid locations on the other hand cause the end of the game. Zheng et al. [16] studied the detection of bugs by implementing a DRL approach. The game is tested by considering the winning score. We consider their work as a baseline and compare other DRL approaches with their results.

Experimental setup

The Block Maze game has a discrete action space that limits the DRL configurations to which it can be applied. Therefore, we consider the following algorithms from Stable-baselines (its 3 versions) (SB), Keras-rl (KR) and Tensorforce (TF): DQN-SB, PPO-SB, A2C-SB, DQN-KR, DQN-TF, PPO-TF, and A2C-TF during our experiments. DRL algorithms from the studied DRL frameworks have their hyperparameter settings. We employ the same values of the optimizer (the Adam optimizer [157]), the DNN model (three fully connected linear layers with 256, 128, and 128 units as the hidden layers, connected to the output layer), the

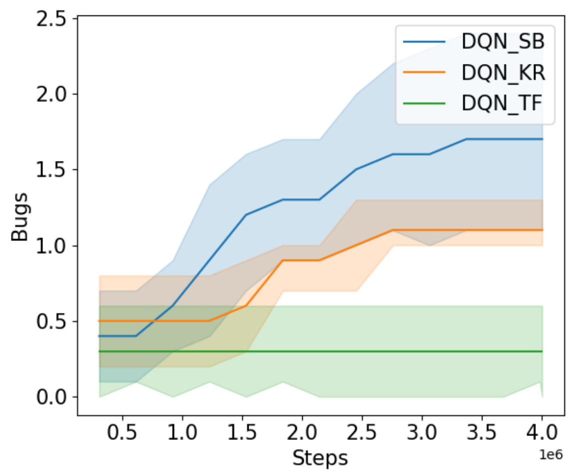


Figure 4.1 Number of bugs detected by DQN agents from different frameworks.

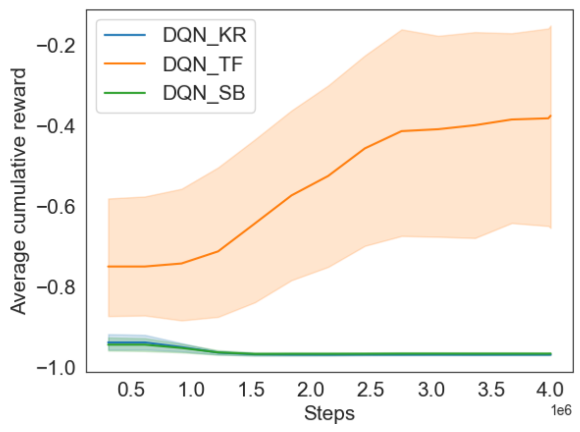


Figure 4.2 Average cumulative reward earned by
DQN agents from different frameworks.

discount factor (0.99) and the learning rate (0.25×10^3) as the baseline work [16], they are the ones we could exactly match with the different studied DRL algorithms. DQN-SB, DQN-TF, DQN-KR, PPO-TF, PPO-SB, A2C-SB, and A2C-TF have respectively 19, 21, 7, 25, 19, 18, and 23 more hyperparameters whose values are provided in our replication package [158] . We collected the results of each DRL algorithm for a total of 4 million steps and 10,000 steps. To counter the randomness effect during testing, we repeat each run 10 times to average the results. Like Zheng et al. [16], we do not train the agent while the player is playing online, instead, we simulate the game environment by implementing it using the OpenAI gym [159] interface. The DRL agents use the gym interface during training to compute the best policy to play the game. During testing, the same OpenAI gym interface as the game environment is used.

Evaluation metrics

We define the following evaluation metrics used in the Blockmaze game to evaluate the performance of the DRL configurations:

- **Number of bugs detected**: the average number of bugs detected by our DRL agents after being trained.
- **The average cumulative reward** obtained by the DRL agents after being trained.
- **The line coverage**: the lines covered by each DRL approach during testing. We use

the Python library of coverage[1] to collect line coverage. This library gives you the result per Python file. As in our replication package [158], both the gym environment and the actual game implementation are on the same file. Thus, the line coverage includes both the lines of code of the gym environment and the game implementation.

- **The state coverage**: the number of visited state during testing.
- **Training time:** We collect the time consumed by the DRL agents to train their policy, which lasts for 10,000 steps.
- **Prediction time:** We collect the time consumed by the trained DRL agents to detect bugs for 10,000 steps or until reaching the goal coin of the game environment.

4.2.3 Problem 2: Test case prioritization using DRL

In this problem, we aim to apply various DRL algorithms from different frameworks for test case prioritization in the context of CI. To do so, we build our approach on recent work using DRL for test case prioritization by Bagherzadeh et al. [1]. The authors explored various prioritization techniques that can adapt and improve continuously through interaction with the CI environment. This interaction between the CI environment and test case prioritization is framed as a DRL problem.

Datasets and Baselines

We use simple and enriched historical datasets for our experiments. Simple historical datasets represent test situations where source code is not available and contain the age, average execution time, and verdicts of test cases. Enriched historical datasets represent test situations where source code is available but due to time constraints imposed by the CI, complete coverage analysis is not possible. They are enriched with history data, execution history, and code characteristics from Apache Commons projects [126]. Table 4.1 shows the list of datasets that we employ in this study and their characteristics. The execution logs contain up to 438 CI cycles, and each CI cycle includes at least 6 test cases. Less than 6 test cases will not be relevant and can inflate the accuracy of the results [1]. The logs column indicates the number of test case execution logs which ranges from $2,207$ to $32,118$. Enriched datasets show a low rate of failed cycles and failure rate while the failure rates and number of failed cycles in simple datasets are high. The last column shows the average computation time of enriched features per cycle.

[1] `https://coverage.readthedocs.io`

We compare DRL algorithms from DRL frameworks (i.e., Keras-rl and Tensorforce) against the work by Bagherzadeh et al. [1], Spieker et al. [125] and Bertolino et al. [126]. Bagherzadeh et al. [1] applied DRL using state-of-the-art DRL algorithms from the Stable-baselines framework to solve the test case prioritization problem. Spieker et al. [125] applied the $\epsilon - greedy$ (see Chapter 2) algorithm on simple history datasets to solve the test case prioritization problem - we refer to this baseline as RL-BS1. Bertolino et al. [126] apply ensemble (i.e., Random Forest) and non-ensemble (i.e., Shallow Network, Deep Neural Network, K-Nearest Neighbor) algorithms to DRL-based and supervised learning approaches. We employ the best DRL-based and supervised learning approaches as our baselines - respectively referred to RL-BS2 and MART.

Experimental setup

We implemented our ranking models using the DRL algorithms of the selected frameworks. We used the OpenAI gym library [159] to mimic the CI environment using logs execution and relied on the implementation of the DRL algorithms provided by the Stable-baselines [160], Keras-rl [43] and Tensorforce [40] frameworks. Stable-baselines in its second version is used here as it was originally used by Bagherzadeh et al. [1]. In any case, Stable-baselines in its second version and Stable-baselines in its third version provide for their implemented DRL algorithms the same hyperparameters. Moreover, to make sure Stable-baselines in its third version meets the performance of Stable-baselines in its second version, its developers conducted experiments [161] to assess the performance of its implemented DRL algorithms and found them equivalent. So, a performance drop should not be expected by using either one of them. When applicable, we employ the default hyperparameters values of Stable-baselines for the experiments similarly to the original work [1]. Specifically, the architecture of the DNN model, the learning rate and the discount factor have the same values among all experiments. The details of all hyperparameters settings are documented in our replication package [158] . Regarding the APFD and NRPA metrics, for each dataset, we performed several experiments that correspond to the two pairwise and pointwise ranking models. It should be noted that the applicability of the DRL algorithms is restricted by the type of their action space. The pairwise ranking model involves five experiments for each data set, one for each DRL framework with DRL algorithms that can support a discrete action space (i.e., DQN-SB, DQN-KR, DQN-TF, A2C-SB, A2C-TF). Similarly, the point-ranking model involves five experiments for each dataset, one for each DRL framework with DRL algorithms that can support a continuous action space (i.e., DDPG-SB, DDPG-KR, DDPG-TF, A2C-SB, A2C-TF). The training process begins with training an agent by replaying the execution logs from the first cycle, followed by evaluating the trained agent on the second cycle. Then the logs from the second run are replayed to improve the agent, and so on.

Bagherzadeh et al. [1], trained the agent for a minimum of $200 \times n \times \log_2 n$ episodes and one million steps for training each cycle, where n refers to the number of test cases in the cycle. Training stops when the budget of steps per training instance is exhausted or when the sum of rewards in an episode cannot be improved for more than 100 consecutive episodes. After each episode, the agent is reset before the next one.

Evaluation metrics

To assess the accuracy of prioritization techniques across our DRL configurations, we use the following metrics:

Normalized Rank Percentile Average (NRPA): NRPA measures how close a predicted ranking of items is to the optimal ranking independently of the context of the problem or ranking criteria. The value can range from 0 to 1. NRPA is calculated during testing after the agent is trained. The NRPA is defined as follows: $NRPA = \frac{RPA(s_e)}{RPA(s_o)}$. In this equation s_e is the ordered sequence generated by a ranking algorithm R that takes a set of k items, and s_o is the optimal ranking of the items. RPA is defined as:

$$RPA = \frac{\sum_{m \in s} \sum_{i=idx(s,m)}^{k} |s| - idx(s_o, m) + 1}{k^2(k+1)/2} \tag{4.1}$$

where $idx(s, m)$ returns the position of m in sequence s.

Average Percentage of Faults Detected (APFD): APFD measures the weighted average of the percentage of the fault detected by the execution of test cases in a certain order. It ranges from 0 to 1. Values close to 1 imply fast fault detection. APFD is calculated during testing after the agent is trained. It is defined as follows:

$$APFD(s_e) = 1 - \frac{\sum_{t \in s_e} idx(s_e, t) * t.v}{|s_e| * m} + \frac{1}{2 * |s_e|} \tag{4.2}$$

where m is the total number of faults, t is a test case among s_e and v its execution verdict, either 0 or 1.

Training time: We collect the time consumed by the DRL agents to train their policy, which lasts for 200 episodes, for the pairwise and pointwise strategies.

Prediction time: For both pointwise and pairwise ranking models, we measured the time consumed by the DRL agents to rank a set of test cases.

4.3 Results

This section presents the experimental results obtained while answering our three RQs.

4.3.1 Playtesting in the Blockmaze game

RQa$_1$. How does selecting a DRL framework impact the performance of software testing tasks?

Figures 4.1 and 4.2 show respectively the average number of detected bugs and average cumulative reward obtained by DQN algorithms from Stable-baselines (DQN_SB), Keras-rl (DQN_KR), and Tensorforce (DQN_TF) frameworks. Indeed, in Figures 4.1 and 4.2 the x-axis represents the 4 million steps budget of training. In Figure 4.1, the y-axis is the average number of bugs detected over 10 runs of the algorithm. In Figure 4.2 the y-axis is the average cumulative reward obtained by the DRL strategy over 10 runs of the algorithm. First, compared with the baseline, which is the A2C implementation by Zheng et al. [16], the DQNs algorithm find less bugs. A2C algorithm takes advantage of all the benefits of value-based (like DQN) and policy-based DRL algorithms. Also, the A2C algorithm employs multiple workers and environments interacting concurrently whereas DQN is a single agent single environment that needs a powerful GPU to train faster. The blockmaze has a small search space and the faster convergence rate of A2C can lead to the detection of more bugs quickly. Among the DQN algorithms, the Stable-baselines's version performs better in terms of detecting bugs and Tensorforce's version performs better in terms of cumulative rewards. To explain these results, our intuition lies in the diversity of the hyperparameters provided by the DRL frameworks which affect the performance, as well as the difference between Tensorflow and Pytorch as the backend of the frameworks. Stable-baselines and Tensorforce, in their DQN implementation, have hyperparameters different from the ones used by Zheng et al. [16] which can impact the performance. For example, Stable-baselines provides a soft update coefficient to update the target network frequently and optimize the network efficiency. It can also be noted that between $[0.25E+6, 0.75E+6]$ steps DQN_KR has a faster convergence. On the official implementation of the algorithm,[2] DQN_KR is briefly trained randomly, which could explain a brief faster convergence when detecting bugs.

Figures 4.3 and 4.4 show respectively the average number of bugs and average cumulative reward obtained by the A2C algorithm from Stable-baselines (A2C_SB), Tensorforce (A2C_TF), and wuji [16] (A2C_wuji). Recall that in this chapter, we only consider the

[2]`https://github.com/Keras-RL/Keras-RL/blob/216c3145f3dc4d17877be26ca2185ce7db462bad/rl/agents/dqn.py`

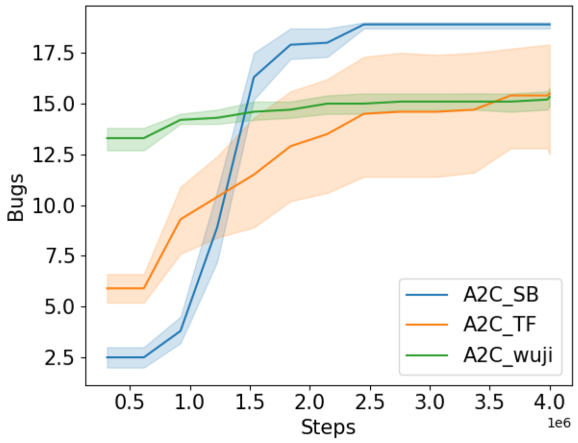


Figure 4.3 Number of bugs detected by
A2C agents from different frameworks.

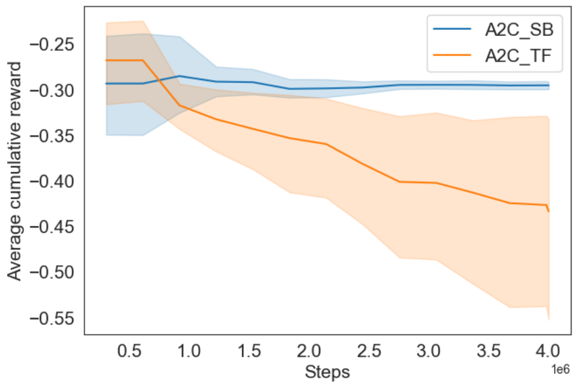


Figure 4.4 Average cumulative reward earned by
A2C agents from different frameworks.

DRL part of wuji, we then compare our results with the number of bugs detected by the wuji framework. Since the authors of wuji did not consider the average cumulative reward as a metric in the original work, we did not report it here. The reason is that we would not have any baselines to validate the results. A2C_SB performs better than A2C_wuji and A2C_TF in terms of detecting bugs. The latter slightly performs better than A2C_wuji. wuji randomly initialized DNNs policies that are used by their A2C agent to detect bugs, which explained the faster detection of bugs between $[0.25E+6, 1.25E+6]$ steps in comparison to the other A2Cs. By employing DRL algorithms from different frameworks, the data to feed the DNNs are directly collected from the environment. This difference might affect the agent's effectiveness in detecting bugs [162].

Figures 4.5 and 4.6 show respectively the average number of detected bugs and average cumulative reward obtained by the PPO algorithms from Stable-baselines (PPO_SB) and Tensorforce (PPO_TF) frameworks. The PPO algorithms detect 1 to 2 more bugs than the A2Cs. Figure 4.7 shows the statistical results of the number of bugs discovered by all the studied DRL configurations. A2C and PPO algorithms from the selected DRL frameworks have a faster convergence rate, until the number of bugs detected reaches 19. Figure 4.7 shows that their bug detection capabilities are statistically similar. The A2C implementation of wuji [16] detects 19% fewer bugs than the other A2Cs and PPOs after 4 million steps during testing. Among the studied DQN strategies, DQN_SB, DQN_KR and DQN_TF detect respectively 88%, 92%, 98% fewer bugs than the A2C implementation of wuji at the same step number.

Table 4.2 Results of Welch's ANOVA and Games-Howell post-hoc test regarding the average cumulative reward on a 10k steps budget.

A	B	mean(A)	mean(B)	pval	CLES
A2C_SB	A2C_TF	-0.40	-0.42	0.0	0.76
A2C_SB	DQN_KR	-0.40	-0.93	0.0	1.0
A2C_SB	DQN_SB	-0.40	-0.96	0.0	1.0
A2C_SB	DQN_TF	-0.40	-0.59	0.0	0.72
A2C_SB	PPO_SB	-0.40	-0.40	0.0	0.58
A2C_SB	PPO_TF	-0.40	-0.40	0.0	0.53
A2C_TF	DQN_KR	-0.42	-0.93	0.0	1.0
A2C_TF	DQN_SB	-0.42	-0.96	0.0	1.0
A2C_TF	DQN_TF	-0.42	-0.59	0.0	0.69
A2C_TF	**PPO_SB**	-0.42	-0.40	0.0	0.28
A2C_TF	**PPO_TF**	-0.42	-0.40	0.0	0.24
DQN_KR	DQN_SB	-0.93	-0.96	0.0	0.95
DQN_KR	**DQN_TF**	-0.93	-0.59	0.0	0.16
DQN_KR	**PPO_SB**	-0.93	-0.40	0.0	0.0
DQN_KR	**PPO_TF**	-0.93	-0.40	0.0	0.0
DQN_SB	**DQN_TF**	-0.96	-0.59	0.0	0.14
DQN_SB	**PPO_SB**	-0.96	-0.40	0.0	0.0
DQN_SB	**PPO_TF**	-0.96	-0.40	0.0	0.0
DQN_TF	**PPO_SB**	-0.59	-0.40	0.0	0.29
DQN_TF	**PPO_TF**	-0.59	-0.40	0.0	0.28
PPO_SB	**PPO_TF**	-0.40	-0.40	0.0	0.45

Table 4.3 State coverage of DRL algorithms on the Blockmaze game.

	A2C_SB	A2C_TF	PPO_SB	PPO_TF	DQN_SB	DQN_TF	DQN_KR
State coverage	**45.5 %**	**45.5 %**	**45.5 %**	44.4 %	9.3%	8.8%	9.9%

Furthermore, we perform Welch's ANOVA and Games-Howell post-hoc test to check for significant differences between our results. Tables 4.5 and 4.6 show respectively the results of Welch's ANOVA and Games-Howell post-hoc test analysis in terms of bugs detected and average cumulative earned by the DRL algorithms. The post hoc test shows that among the DQNs algorithms, Stable-baselines framework performs best in terms of detecting bugs, and the Tensorforce framework performs best in terms of reward earned. Among the A2Cs algorithms, Stable-baselines framework performs best in terms of detecting bugs and rewards earned. Among the PPOs algorithms, Stable-baselines performs best in terms of detecting bugs and rewards earned. Similarly, Stable-baselines has the best performance in terms of bug detection capability regardless of the DRL algorithms. Tables 4.5 and 4.6 also report CLES between the DRL configurations. CLES values show the probability that one configuration detects more bugs than another or earned more rewards.

We also analyze the coverage obtained by each DRL strategy, as well as their state coverage on the Blockmaze game. The line coverage is exactly the same for all strategies: 96%. The

Table 4.4 State coverage of DRL algorithms on the Blockmaze game on a 10K steps budget.

	PPO_SB	PPO_TF	A2C_TF	A2C_SB	DQN_SB	DQN_KR	DQN_TF
State coverage	**18.5%**	17.4%	17%	15.7%	7.4%	2.4%	1.3%

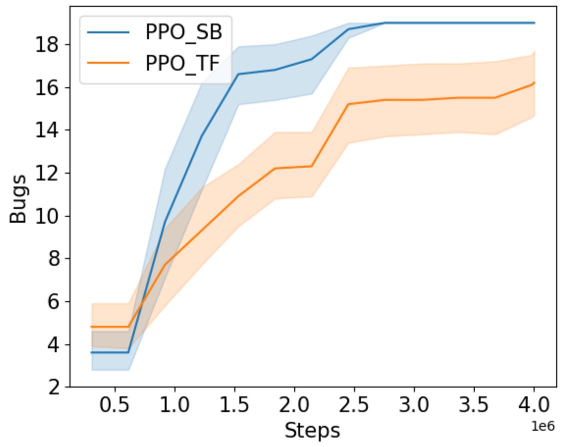


Figure 4.5 Number of bugs detected by
PPO agents from different frameworks.

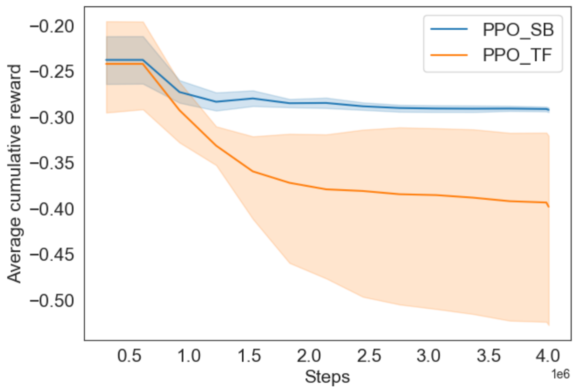


Figure 4.6 Average cumulative reward earned by
PPO agents from different frameworks.

other 4% are mostly related to the code related to the player not reaching the goal of the Blockmaze. Specifically, the lines of code on the Blockmaze gym environment where we check if the player is at the goal location. In addition, the other 4% are also related to the lines of codes instructing the termination of the game when a player reaches the goal. Finally, the line of code on the Blockmaze gym environment that converts the maze to an RGB image is not reached either, as during testing we do not require it. The Blockmaze has a total of 400 potential states to be visited by the DRL agent. Table 4.3 shows the results of the state coverage obtained by the DRL algorithms from the DRL frameworks we have evaluated. As expected, A2Cs and PPOs have the largest state coverage as they are able to detect more bugs. The state coverage obtained by the DQNs is lower as they lead to fewer bugs detected. Moreover, regarding the bugs that are not detected, this is related to the fact that the DRL configurations are not able to cover all the observation state space. The code is relatively easy to cover, as opposed to the state coverage. Thus, detecting bugs by maximizing the state coverage could lead to better performance for game testing.

In terms of winning the game (i.e., reaching the gold position of the Blockmaze as illustrated in Figure 5.3) none of our strategies are successful. Our results in Figure 4.8 show that the DRL agents earn negative rewards for all steps during testing.

For a richer analysis, we collected our evaluation metrics on a reduced number of steps during

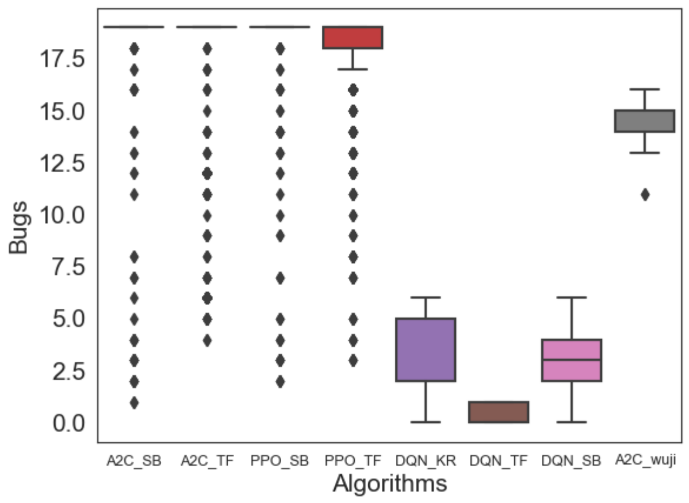


Figure 4.7 The number of bugs discovered using different strategies after 4 million steps for Blockmaze.

test time (the evaluation metrics are collected on a 10,000 steps budget instead of 4 million) in order to answer **RQa$_1$.** This analysis does not involve A2C-wuji as with this implementation the detection of bugs starts at 300000+ steps. Figures 4.9 and 4.10 show respectively the average number of detected bugs and average cumulative reward over 10 runs of the algorithm obtained by DQN algorithms from Stable-baselines (DQN-SB), Keras-rl (DQN-KR), and Tensorforce (DQN-TF) frameworks on a 10,000 budget steps. Similarly, Figures 4.11 and 4.12 show respectively the average number of detected bugs and average cumulative reward over 10 runs of the algorithm obtained by A2C algorithms from Stable-baselines (A2C-SB), and Tensorforce (A2C-TF) frameworks on a 10,000 budget steps. Finally, Figures 4.13 and 4.14 show respectively the average number of detected bugs and average cumulative reward over 10 runs of the algorithm obtained by the PPOs algorithms from Stable-baselines (PPO-SB), and Tensorforce (PPO-TF) frameworks on a 10,000 budget steps. Consistently, among the DQNs algorithms, Stable-baselines performs best in terms of detecting bugs and Tensorforce performs best in terms of rewards earned. Table 4.2 shows the results of Welch's ANOVA post hoc tests regarding the rewards earned on 10,000 steps. We report the results of Welch's ANOVA post hoc tests regarding the bugs detected by the DRL algorithms on 10,000 steps in Annex A Among A2C algorithms and w.r.t the post hoc analysis, Tensorforce performs best in terms of detecting bugs, and Stable-baselines performs best in terms of rewards earned. Among the PPOs algorithms, Stable-baselines performs best in terms of detecting bugs, and Tensorforce performs best in terms of rewards earned. Similarly as with the 4 million steps

Table 4.5 Results of Welch's ANOVA and Games-Howell post-hoc test regarding the average cumulative reward earned by DRL algorithms.

A	B	mean(A)	mean(B)	pval	CLES
A2C_SB	A2C_TF	-0.30	-0.72	0.0	0.96
A2C_SB	DQN_KR	-0.30	-0.97	0.0	1.0
A2C_SB	DQN_SB	-0.30	-0.96	0.0	1.0
A2C_SB	DQN_TF	-0.30	-0.32	0.0	0.52
A2C_SB	PPO_SB	-0.30	-0.29	0.0	0.33
A2C_SB	PPO_TF	-0.30	-0.52	0.0	0.78
A2C_TF	DQN_KR	-0.72	-0.97	0.0	0.84
A2C_TF	DQN_SB	-0.72	-0.96	0.0	0.84
A2C_TF	**DQN_TF**	-0.72	-0.32	0.0	0.22
A2C_TF	**PPO_SB**	-0.72	-0.29	0.0	0.04
A2C_TF	**PPO_TF**	-0.72	-0.52	0.0	0.31
DQN_KR	**DQN_SB**	-0.97	-0.96	0.0	0.37
DQN_KR	**DQN_TF**	-0.97	-0.32	0.0	0.07
DQN_KR	**PPO_SB**	-0.97	-0.29	0.0	0.0
DQN_KR	**PPO_TF**	-0.97	-0.52	0.0	0.07
DQN_SB	**DQN_TF**	-0.96	-0.32	0.0	0.08
DQN_SB	**PPO_SB**	-0.96	-0.29	0.0	0.0
DQN_SB	**PPO_TF**	-0.96	-0.52	0.0	0.07
DQN_TF	**PPO_SB**	-0.32	-0.29	0.0	0.48
DQN_TF	**PPO_TF**	-0.32	-0.52	0.0	0.65
PPO_SB	PPO_TF	-0.29	-0.52	0.0	0.78

budget, none the DRL configurations is able to win the game. This is shown in Table 4.2 with negative reward values.

In terms of training and prediction time, Tables 4.7 and 4.8 show the results of Welch's ANOVA test and the CLES values for each of the DRL algorithms. Keras-rl framework in its DQN implementation has the best performance in terms of prediction time. Similarly, the DQN implementation from the Stable-baselines framework has the best performance in terms of training time. Tensorforce framework performs worst in its DQN implementation in terms of both training and prediction times.

In terms of state coverage, Table 4.4 shows the results of state coverage obtained by the DRL configurations on 10,000 budget steps. Similarly, as with the 4 million steps budget, PPO and A2C have the largest state coverage. Nevertheless, but expected with fewer budget steps all DRL configurations have lower state coverage.

Table 4.6 Results of Welch's ANOVA and Games-Howell post-hoc test regarding the number bugs detected by DRL algorithms.

A	B	mean(A)	mean(B)	pval	CLES
A2C_SB	A2C_TF	18.94	16.91	0.0	0.68
A2C_SB	A2C_wuji	18.94	14.66	0.0	1.0
A2C_SB	DQN_KR	18.94	3.22	0.0	1.0
A2C_SB	DQN_SB	18.94	2.96	0.0	1.0
A2C_SB	DQN_TF	18.94	0.30	0.0	1.0
A2C_SB	PPO_SB	18.94	18.95	0.98	0.50
A2C_SB	PPO_TF	18.94	18.09	0.0	0.68
A2C_TF	A2C_wuji	16.91	14.66	0.0	0.93
A2C_TF	DQN_KR	16.91	3.22	0.0	1.0
A2C_TF	DQN_SB	16.91	2.96	0.0	1.0
A2C_TF	DQN_TF	16.91	0.30	0.0	1.0
A2C_TF	**PPO_SB**	16.91	18.95	0.0	0.32
A2C_TF	**PPO_TF**	16.91	18.09	0.0	0.40
A2C_wuji	DQN_KR	14.66	3.22	0.0	1.0
A2C_wuji	DQN_SB	14.66	2.96	0.0	1.0
A2C_wuji	DQN_TF	14.66	0.30	0.0	1.0
A2C_wuji	**PPO_SB**	14.66	18.95	0.0	0.0
A2C_wuji	**PPO_TF**	14.66	18.09	0.0	0.01
DQN_KR	DQN_SB	3.22	2.96	0.0	0.55
DQN_KR	DQN_TF	3.22	0.30	0.0	0.95
DQN_KR	**PPO_SB**	3.22	18.95	0.0	0.0
DQN_KR	**PPO_TF**	3.22	18.09	0.0	0.0
DQN_SB	DQN_TF	2.96	0.30	0.0	0.96
DQN_SB	**PPO_SB**	2.96	18.95	0.0	0.0
DQN_SB	**PPO_TF**	2.96	18.09	0.0	0.0
DQN_TF	**PPO_SB**	0.30	18.95	0.0	0.0
DQN_TF	**PPO_TF**	0.30	18.09	0.0	0.0
PPO_SB	PPO_TF	18.95	18.09	0.0	0.69

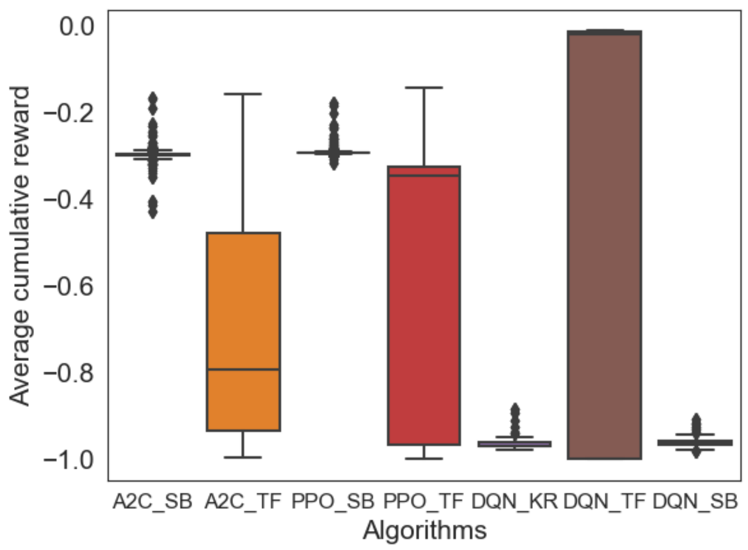


Figure 4.8 Average cumulative reward obtained by different DRL algorithms after 4 million steps for Blockmaze.

RQa$_2$. Which combinations of DRL frameworks-algorithms perform better (i.e., get trained accurately and solve the problem effectively)?

Table 4.6 shows that on average the A2Cs and PPOs algorithms perform better than the DQNs algorithms with a high bug detection number between 12.23 and 15.28. In bold are the mean of the DRL configurations that perform better with a p-value $<= 0.05$. The A2C algorithms have similar performance, same as the PPO algorithms: while PPOs detect more bugs, they do not have statistically significant results in comparison to A2Cs. The following items summarize our results per DRL algorithm where $>$ denotes greater detected bugs and CLES values are greater than 60:

A2C Algorithms:

- A2C_SB >A2C_TF > A2C_wuji

PPO Algorithms:

- PPO_SB > PPO_TF

DQN Algorithms:

- DQN_SB > DQN_TF

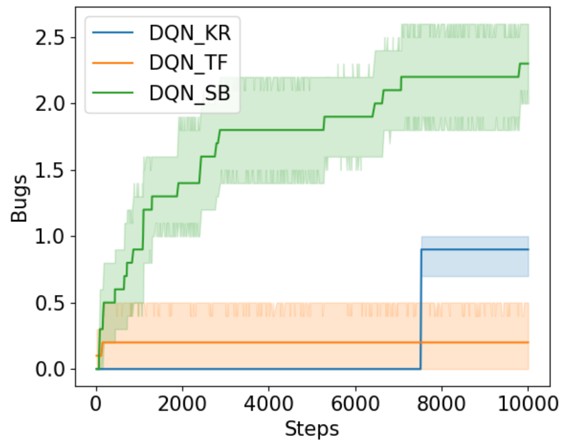


Figure 4.9 Number of bugs detected by DQN agents from different frameworks on a 10K budget steps.

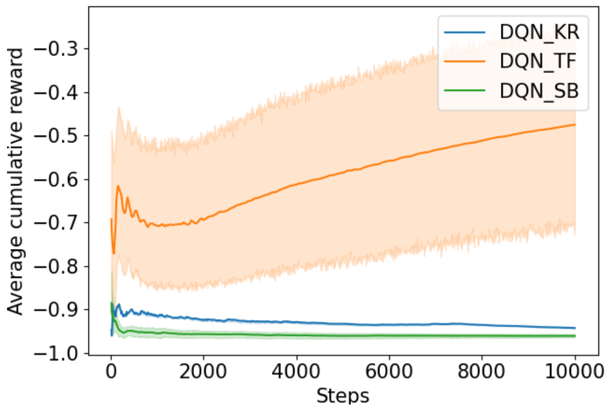


Figure 4.10 Average cumulative reward earned by DQN agents from different frameworks on a 10K budget steps.

- DQN_KR > DQN_TF

In terms of average cumulative reward, the following summarizes our results per DRL algorithm where CLES values are greater than 60.

A2C Algorithms:

- A2C_SB > A2C_TF

PPO Algorithms:

- PPO_SB > PPO_TF

Following are the results per DRL algorithm where > denotes greater detected bugs based on 10,000 steps budget and where CLES values are greater than 60.

PPO Algorithms:

- PPO_SB > PPO_TF

DQN Algorithms:

- DQN_SB > DQN_TF

In terms of average cumulative reward, the following summarizes our results per DRL algorithm on the basis of a 10,000 steps budget where CLES values are greater than 60.

A2C Algorithms:

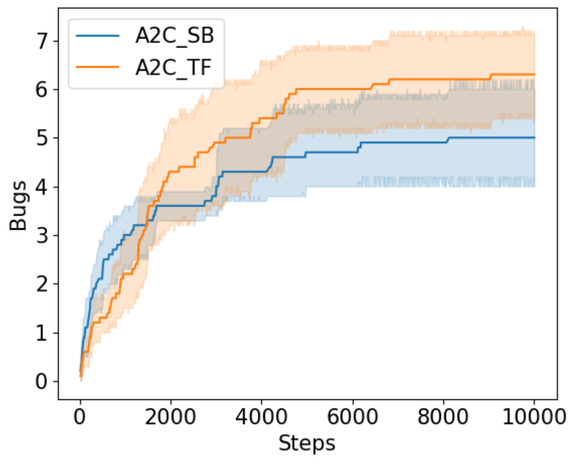


Figure 4.11 Number of bugs detected by A2C agents from different frameworks on a 10k budget.

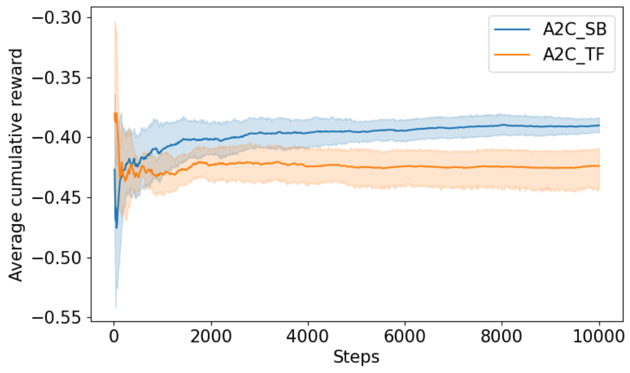


Figure 4.12 Average cumulative reward earned by A2C agents from different frameworks on a 10k budget.

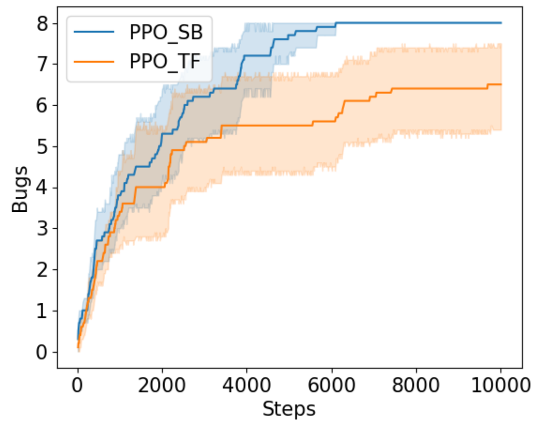


Figure 4.13 Number of bugs detected by PPO agents from different frameworks on a 10k budget.

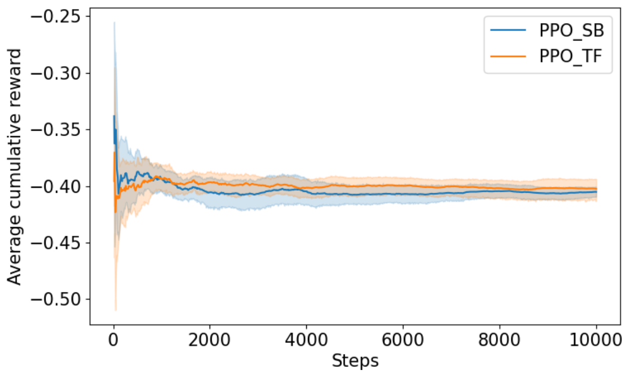


Figure 4.14 Average cumulative reward earned by PPO agents from different frameworks on a 10k budget.

Table 4.7 Results of Welch's ANOVA test of prediction time (in milliseconds) of DRL configurations on a 10k steps budget.

A	B	mean(A)	mean(B)	pval	CLES
A2C_SB	A2C_TF	13089.34	180830.26	0.0	0.0
A2C_SB	**DQN_KR**	13089.34	10759.60	0.0	1.0
A2C_SB	**DQN_SB**	13089.34	11745.30	0.0	1.0
A2C_SB	DQN_TF	13089.34	185817.34	0.0	0.0
A2C_SB	PPO_SB	13089.34	13179.68	0.97	0.39
A2C_SB	PPO_TF	13089.34	182671.67	0.0	0.0
A2C_TF	**DQN_KR**	180830.26	10759.60	0.0	1.0
A2C_TF	**DQN_SB**	180830.26	11745.30	0.0	1.0
A2C_TF	DQN_TF	180830.26	185817.34	0.01	0.07
A2C_TF	**PPO_SB**	180830.26	13179.68	0.0	1.0
A2C_TF	PPO_TF	180830.26	182671.67	0.93	0.37
DQN_KR	DQN_SB	10759.60	11745.30	0.0	0.0
DQN_KR	DQN_TF	10759.60	185817.34	0.0	0.0
DQN_KR	PPO_SB	10759.60	13179.68	0.0	0.0
DQN_KR	PPO_TF	10759.60	182671.67	0.0	0.0
DQN_SB	DQN_TF	11745.30	185817.34	0.0	0.0
DQN_SB	PPO_SB	11745.30	13179.68	0.0	0.0
DQN_SB	PPO_TF	11745.30	182671.67	0.0	0.0
DQN_TF	**PPO_SB**	185817.34	13179.68	0.0	1.0
DQN_TF	**PPO_TF**	185817.34	182671.67	0.68	0.69
PPO_SB	PPO_TF	13179.68	182671.67	0.0	0.0

- A2C_SB > A2C_TF

DQN Algorithms:

- DQN_KR > DQN_SB

Moreover, on the basis of a 4 million steps budget, we observe with CLES values equal to 1 that A2Cs and PPOs algorithms detect more bugs than DQN algorithms. Similarly, on the basis of a 10,000 steps budget, we observe with CLES values greater than 90 that A2Cs and PPOs algorithms detect more bugs than DQN algorithms. Practically it means for at least 90% of episodes, PPOs algorithms detect more bugs.

Finding 1: A2Cs and PPOs show statistically significant performance compared to DQN algorithms in finding bugs in the examined game.

Table 4.8 Results of Welch's ANOVA test of training time (in milliseconds) of DRL configurations on a 10k steps budget.

A	B	mean(A)	mean(B)	pval	CLES
A2C_SB	**A2C_TF**	16318.70	14621.82	0.0	1.0
A2C_SB	DQN_KR	16318.70	63699.69	NA	0.0
A2C_SB	DQN_SB	16318.70	1140.33	NA	1.0
A2C_SB	DQN_TF	16318.70	92807.45	0.0	0.0
A2C_SB	PPO_SB	16318.70	12168.06	NA	1.0
A2C_SB	**PPO_TF**	16318.70	14345.02	0.0	1.0
A2C_TF	DQN_KR	14621.82	63699.69	0.0	0.0
A2C_TF	**DQN_SB**	14621.82	1140.33	0.0	1.0
A2C_TF	DQN_TF	14621.82	92807.45	0.0	0.0
A2C_TF	**PPO_SB**	14621.82	12168.06	0.0	1.0
A2C_TF	**PPO_TF**	14621.82	14345.02	0.77	0.68
DQN_KR	DQN_SB	63699.69	1140.33	NA	1.0
DQN_KR	DQN_TF	63699.69	92807.45	0.0	0.0
DQN_KR	PPO_SB	63699.69	12168.06	NA	1.0
DQN_KR	**PPO_TF**	63699.69	14345.02	0.0	1.0
DQN_SB	DQN_TF	1140.33	92807.45	0.0	0.0
DQN_SB	PPO_SB	1140.33	12168.06	NA	0.0
DQN_SB	PPO_TF	1140.33	14345.02	0.0	0.0
DQN_TF	**PPO_SB**	92807.45	12168.06	0.0	1.0
DQN_TF	**PPO_TF**	92807.45	14345.02	0.0	1.0
PPO_SB	PPO_TF	12168.06	14345.02	0.0	0.0

RQa$_3$. How stable are the results obtained from the DRL frameworks, over multiple runs?

Our findings show that we do not get similar results from the same DRL algorithm over the DRL frameworks. We explain this by the fact that each DRL framework that we used in this study does not provide the same hyperparameters regarding the DRL algorithm. Some of the hyperparameters are similar but not all of them. For example, the DQN algorithm from Stable-baselines has an additional hyperparameter called "gradient_steps" to perform the gradient process as there are steps done during a rollout, instead of doing it after a complete rollout is done. These other hyperparameters, even with default values, can slightly improve efficiency as we observe in our results.

4.3.2 Test case prioritization

Tables 4.9 and 4.10 show the averages and standard deviations of APFD and NRPA for the eight datasets, using different configurations (i.e., combinations of a ranking model, DRL framework, and algorithm). The first column reports different DRL algorithms, and the second column reports the ranking models followed by four datasets per table (a total of eight datasets). Each dataset column is subdivided into the DRL frameworks. In the rest of this section, we use *[ranking model]-[RL algorithm]-[RL framework]* to refer to DRL configurations. For example, *Pairwise-DQN-KR* corresponds to a configuration of the pairwise ranking model and the DQN algorithm from the Keras-rl framework. For each dataset (column), the relative performance rank of configurations in terms of APFD or NRPA are expressed with $\textcircled{n}$, where a lower rank indicates better performance. Again, we analyze the differences in the results by using Welch's ANOVA and the Games-Howell post-hoc test. Table 4.11 and 4.12 show the overall training times for the first 10 cycles across datasets. Similarly, Tables 4.13 and 4.14 show the averages and standard deviations of prediction time (ranking) for the first 10 cycles across datasets. Each cell value represents a configuration as mentioned before. For each dataset, the relative performance ranks of configurations in terms of training/prediction time are expressed with $\textcircled{n}$, where a lower rank $\textcircled{n}$ indicates better performance.

RQa$_1$. How does selecting a DRL framework impact the performance of software testing tasks?

As shown in Table 4.9, pairwise configurations perform best across Stable-baselines's algorithms. Pairwise-A2C-SB yields the best averages. Based on the post-hoc test, Pairwise-A2C-SB performs best across all datasets. Similarly, the Stable-baselines framework performs best regarding the pointwise ranking model. While Pairwise-A2C-SB has the best performance overall, Tensorforce has good performance on IOFROL dataset when implementing Pairwise-A2C configuration. IOFROL is a simple dataset with a high number of execution logs. When using this dataset, the training time of the DRL agent is long, which might explain why Tensorforce configurations perform well. Specifically, despite the high number of execution logs of the IOFROL dataset, Tensorforce still has good performance. To show the importance of selecting the best DRL configuration, we measured the effect size of the differences between pairs of configurations based on CLES. As shown in Table 4.15, the CLES values among one of the worst and best cases for the six enriched datasets are over 80%, whereas they are 66% and 71% for the simple Paint-Control and IOFROL datasets, respectively. These results show that, for each dataset, we have, with high probability, a DRL configuration that has adequately learned a ranking strategy.

In terms of training time, as shown in Tables 4.11 and 4.12, both pairwise and pointwise configurations perform well for some datasets/frameworks. Figures 4.16 and 4.17 show the statistical analysis of the training time involving Pairwise-DQN and Pointwise-DDPG con-

Table 4.9 The average performance of different configurations in terms of APFD and NRPA, along with the results of the three baselines (RL-BS1, RL-BS2, and MART) for PAINT, IOFROL, CODEC, and IMAG datasets. The index in each cell shows the position of a configuration (row) with respect to others for each dataset (column) in terms of NRPA or APFD, based on statistical testing.

		PAINT (APFD)			IOFROL (APFD)			CODEC (NRPA)			IMAG (NRPA)		
	RM	SB	KR	TF	SB	KR	TF	SB	KR	TF	SB	KR	TF
DQN	PA	0.66±.2 ②	0.49±.1 ⑦	0.56±.2 ⑤	0.53±.1 ③	0.49±.09 ⑦	0.50±.1 ⑤	0.94±.06 ②	0.79±.07 ⑦	0.86±.07 ⑤	0.95±.06 ①	0.77±.06 ⑥	0.87±.06 ④
DDPG	PO	0.62±.2 ③	0.57±.2 ④	0.52±.2 ⑥	0.53±.1 ③	0.52±.1 ④	0.49±.1 ⑥	0.89±.07 ③	0.76±.08 ⑨	0.78±.07 ⑧	0.85±.07 ⑤	0.75±.07 ⑨	0.77±.05 ⑦
A2C	PA	0.70±.2 ①	NA	0.57±.2 ④	0.54±.1 ②	NA	0.67±.2 ①	0.96 ± .04 ①	NA	0.78±.07 ⑧	0.95±.05 ②	NA	0.76±.10 ⑧
	PO	0.57±.2 ④	NA	0.44±.1 ⑧	0.52±.1 ④	NA	0.49±.1 ⑥	0.89 ± .06 ④	NA	0.83±.08 ⑥	0.92 ± .05 ③	NA	0.85±.07 ⑤
Optimal	NA	0.79±.14			0.89±.14			NA			NA		
RL-BS1	PO	0.63±.16			0.74±.24			NA			NA		
RL-BS2	PO	NA			NA			0.90±.05			0.89±.09		
MART	PA	NA			NA			0.96±.03			0.90±.05		

Table 4.10 The average performance of different configurations in terms of APFD and NRPA, along with the results of the three baselines (RL-BS1, RL-BS2, and MART) for IO, COMP, LANG, and MATH datasets. The index in each cell shows the position of a configuration (row) with respect to others for each dataset (column) in terms of NRPA or APFD, based on statistical testing.

		IO (NRPA)			COMP (NRPA)			LANG (NRPA)			MATH (NRPA)		
	RM	SB	KR	TF	SB	KR	TF	SB	KR	TF	SB	KR	TF
DQN	PA	0.97±.02 ②	0.76±.05 ⑦	0.85±.06 ⑥	0.97±.03 ①	0.77±.05 ⑦	0.85±.06 ⑥	0.94±.04 ②	0.78±.05 ⑦	0.86±.06 ④	0.93±.05 ②	0.77±.05 ⑦	0.88±.07 ③
DDPG	PO	0.90±.07 ④	0.72±.07 ⑩	0.75±.05 ⑨	0.86±.07 ④	0.74±.06 ⑩	0.76±.06 ⑨	0.84±.08 ⑤	0.76±.05 ⑨	0.76±.06 ⑧	0.87±.06 ⑤	0.74±.06 ⑨	0.76±.05 ⑧
A2C	PA	0.98±.02 ①	NA	0.75±.09 ⑧	0.97±.02 ②	NA	0.76 ± .08 ⑧	0.95±.03 ①	NA	0.73±.1 ⑩	0.95±.04 ①	NA	0.57±.3 ⑩
	PO	0.92±.04 ③	NA	0.87±.06 ⑤	0.89±.05 ③	NA	0.85±.07 ⑤	0.89±.05 ③	NA	0.83±.07 ⑥	0.88±.05 ④	NA	0.82±.06 ⑥
Optimal	NA	NA			NA			NA			NA		
RL-BS1	PO	NA			NA			NA			NA		
RL-BS2	PO	0.84±.13			0.90±.05			0.89±.07			0.95±.02		
MART	PA	0.93±.02			0.96±.02			0.94±.02			0.95±.02		

Table 4.11 Average training time (in minutes) of DRL configurations for the first 10 cycles across PAINT, IOFROL, CODEC, and IMAG datasets.

	RM	PAINT			IOFROL			CODEC			IMAG		
		SB	KR	TF	SB	KR	TF	SB	KR	TF	SB	KR	TF
DQN	PA	6.0±3.9 ④	9.2±5.8 ⑥	44.3±27.5 ⑨	45.6±52.4 ④	68.3±76.8 ⑤	315.1±356.8 ⑨	2.1±1.4 ④	3.4±2.4 ⑤	16.6±11.1 ⑨	0.9±1.6 ③	2.8±2.9 ⑤	23.7±20.4 ⑩
DDPG	PO	2.2±1.4 ③	10.5±6.4 ⑦	8.1±.3 ⑤	19.4±22.3 ③	76.6±86.4 ⑥	347.9±398.4 ⑩	0.8±.6 ②	3.7±2.3 ⑥	16.8±10.8 ⑩	0.8±.7 ②	3.3±3.2 ⑥	14.1±14.0 ⑧
A2C	PA	1.6±.9 ②	NA	49.1±22.7 ⑩	11.1±12.1 ②	NA	221.3±299.2 ⑦	0.6 ± .3 ①	NA	8.5 ± .8 ⑦	0.6±.4 ①	NA	14.7 ± 14.8 ⑨
	PO	1.5±.3 ①	NA	36.0±21.8 ⑧	1.3±.3 ①	NA	282.4±332.1 ⑧	1.7±.2 ③	NA	14.8±9.9 ⑧	1.7 ± .5 ④	NA	14.0±13.5 ⑦

Table 4.12 Average training time (in minutes) of DRL configurations for the first 10 cycles across IO, COMP, LANG, and MATH datasets.

	RM	IO			COMP			LANG			MATH		
		SB	KR	TF	SB	KR	TF	SB	KR	TF	SB	KR	TF
DQN	PA	2.8 ± 1.4 ④	4.2 ± 2.1 ⑤	7.8±.7 ⑦	2.8 ± 1.4 ④	4.2 ± 2.2 ⑤	8.3±.9 ⑦	4.0 ± 4.7 ④	17.3 ± 27.7 ⑦	8.5±1.6 ⑥	15.7 ± 17.7 ⑤	23.0 ± 26.1 ⑥	11.7±5.9 ④
DDPG	PO	1.1±.5 ③	4.9±2.4 ⑥	20.7±10.4 ⑨	1.1±.5 ③	4.9±2.4 ⑥	21.9±11.2 ⑨	1.5±1.8 ③	6.7±8.1 ⑤	31.4±37.5 ⑨	6.4±7.4 ③	26.0±29.3 ⑦	131.9±166.0 ⑩
A2C	PA	0.8±.3 ②	NA	22.4 ± 11.7 ⑩	0.8± .3 ②	NA	22.1 ± 11.4 ⑩	1.1±1.2 ②	NA	43.4 ± 41.2 ⑩	4.0±.4.4 ②	NA	116.2 ± 130.6 ⑨
	PO	0.7±.3 ①	NA	18.5 ± 9.7 ⑧	0.7±.3 ①	NA	18.3±9.5 ⑧	1.0±1.1 ①	NA	21.1±32.5 ⑧	3.6±3.9 ①	NA	114.5 ± 131.2 ⑧

Table 4.13 The average of prediction (ranking) time (in seconds) of DRL configurations for the first 10 cycles across PAINT, IOFROL, CODEC, and IMAG datasets.

	RM	PAINT			IOFROL			CODEC			IMAG		
		SB	KR	TF	SB	KR	TF	SB	KR	TF	SB	KR	TF
DQN	PA	1.8±.2 ⑤	0.1±.07 ②	8.1±.9 ⑩	4.5±3.2 ⑥	1.1±1.1 ③	16.6±11.5 ⑩	1.5±.09 ④	0.06±.03 ②	7.4±.8 ⑦	1.5±.1 ④	0.05±.03 ②	7.7±.9 ⑧
DDPG	PO	0.8±.3 ③	0.07±.3 ①	8.1±.3 ⑨	0.9±.1 ②	0.2±.2 ①	10.3±2.0 ⑨	0.7±.03 ③	0.05±.03 ①	9.4±.7 ⑩	0.7±.1 ③	0.05±.02 ①	8.7±.8 ⑩
A2C	PA	1.8±.5 ⑥	NA	7.9±1.1 ⑧	3.0±2.0 ⑤	NA	8.1±1.7 ⑧	1.5 ± .3 ⑤	NA	8.4 ± .7 ⑧	1.6±.4 ⑤	NA	8.3 ± .8 ⑨
	PO	1.5±.3 ④	NA	7.4±.6 ⑦	1.3±.3 ④	NA	7.8±1.3 ⑦	1.7±.2 ⑥	NA	8.4±.9 ⑨	1.7 ± .5 ⑥	NA	7.7±.8 ⑦

Table 4.14 The average of prediction (ranking) time (in seconds) of DRL configurations for the first 10 cycles across IO, COMP, LANG, and MATH datasets.

	RM	IO			COMP			LANG			MATH		
		SB	KR	TF	SB	KR	TF	SB	KR	TF	SB	KR	TF
DQN	PA	1.6 ± .1 ⑤	0.07 ± .03 ②	7.8±.7 ⑧	1.6 ± .1 ⑥	0.07 ± .03 ②	8.3±.9 ⑧	1.7 ± .2 ⑥	1.4 ± 2.3 ③	8.5±1.6 ⑨	2.2 ± .9 ⑥	0.3 ± .3 ②	11.7±5.9 ⑩
DDPG	PO	0.7±.1 ③	0.05±.02 ①	8.9±.7 ⑩	0.7±.1 ③	0.05±.03 ①	9.8±.9 ⑩	0.7±.1 ②	0.06±.03 ①	9.6±.8 ⑩	0.8±.3 ③	0.1±.07 ①	9.6±1.1 ⑨
A2C	PA	1.4±.3 ④	NA	8.8 ± .5 ⑨	1.5± .3 ⑤	NA	8.4 ± .4 ⑨	1.6±.4 ⑤	NA	8.4 ± 1.2 ⑧	2.0±.9 ⑤	NA	9.0 ± 1.4 ⑧
	PO	1.6±.3 ⑥	NA	7.6 ± .8 ⑦	1.4±.3 ④	NA	7.3±.7 ⑦	1.5±.3 ④	NA	7.7±.6 ⑦	1.6±.5 ④	NA	8.3 ± .9 ⑦

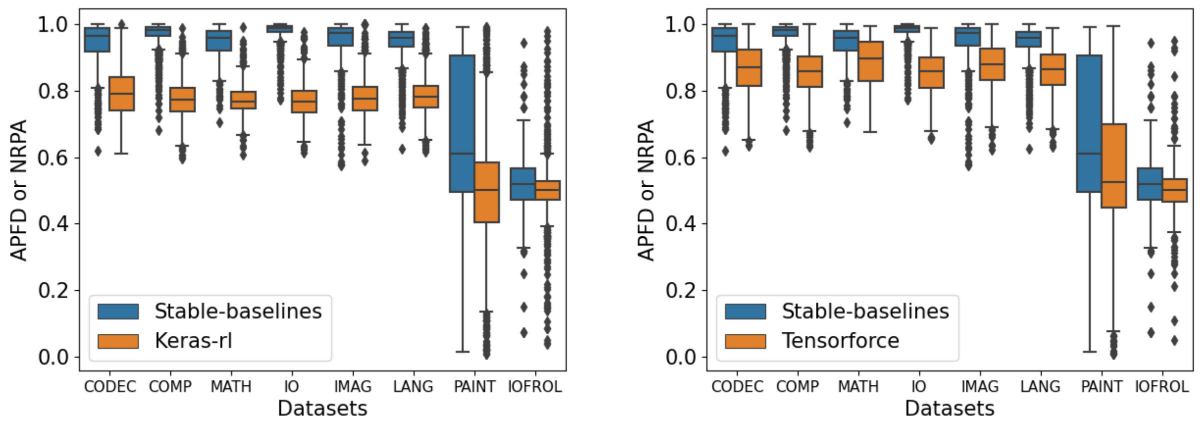


Figure 4.15 APFD (simple datasets) or NRPA (enriched datasets) of DQN-PAIRWISE configuration across DRL frameworks for all datasets: Stable-baselines vs. Keras-rl (left) and Stable-baselines vs. Tensorforce (right).

Table 4.15 Common Language Effect Size between one of the worst and best configurations for each dataset based on accuracy.

Dataset	Best Conf.	Worst Cons.	CLES
Paint-Control	PAIRWISE-A2C-SB	POINTWISE-A2C-TF	.66
IOFROL	PAIRWISE-A2C-TF	PAIRWISE-DQN-KR	.71
Codec	PAIRWISE-A2C-SB	POINTWISE-DDPG-KR	.97
Compress	PAIRWISE-A2C-SB	POINTWISE-DDPG-KR	**.99**
Imaging	PAIRWISE-DQN-SB	POINTWISE-DDPG-KR	.98
IO	PAIRWISE-A2C-SB	POINTWISE-DDPG-KR	**.99**
Lang	PAIRWISE-A2C-SB	PAIRWISE-A2C-TF	.81
Math	PAIRWISE-A2C-SB	PAIRWISE-DQN-TF	.81

figurations, respectively. The results show that Pointwise-DDPG-SB performs best followed by Pointwise-DDPG-KR. Regarding the DQN configurations, similarly, the Stable-baselines framework performs best. It is worth mentioning that, since DRL agents are trained offline, the training time does not add any delay to the CI build process. In terms of prediction time, as shown in Table 4.13 and 4.14, similar to the training time, both configurations (pairwise or pointwise) perform well for some of the datasets/frameworks. Based on the post-hoc test, Pairwise-DQN-SB performs best on average followed by Pairwise-DQN-KR. The prediction time among pointwise and pairwise configurations goes up to 11 seconds, notably for Pairwise-DQN-TF, which is non-negligible for CI builds.

The last three rows of Table 4.9 and 4.10 show the averages and standard deviations of baselines configurations in terms of NRPA and APFD values collected from [1], for the datasets on which they were originally experimented. In Annex A we report the results

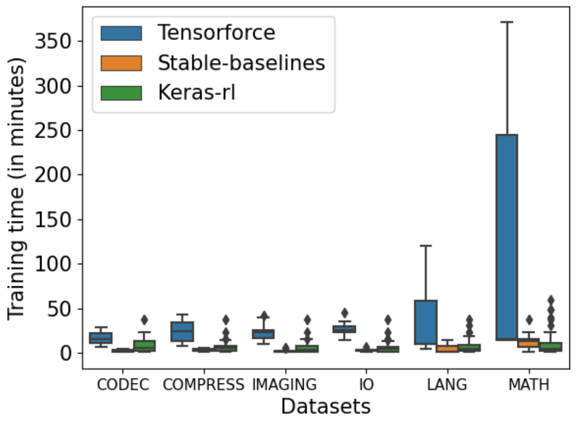


Figure 4.16 Training time of Pairwise-DQN configuration accross DRL frameworks for enriched datasets.

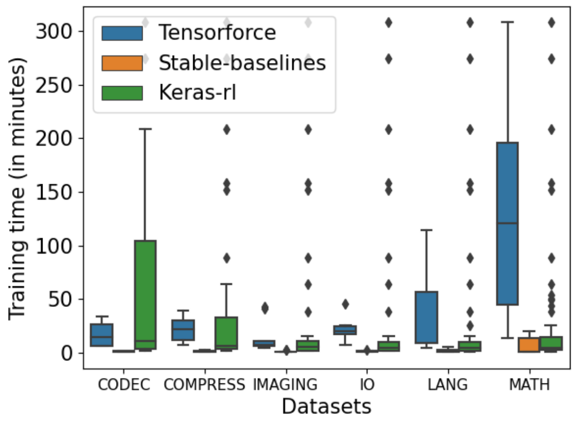


Figure 4.17 Training time Pointwise-DDPG configuration accross DRL frameworks for enriched datasets.

of CLES between the best configuration of each framework and selected baselines for all datasets, to assess the effect size of differences.

The row RL-BS1 in Table 4.9 and 4.10 shows the results of an RL-based solution reported by Bagherzadeh et al. [1]. For the Paint-Control dataset, Pairwise-A2C-SB fares slightly better than RL-BS1 with a CLES of 60.2. Also, both solutions (RL-BS1, Pairwise-A2C-SB) are close to the optimal ranking (the row labelled as "Optimal" in Table 4.9 and 4.10). For dataset IOFROL, RL-BS1 performs better than Pairwise-A2C-SB: however, both solutions do not perform well as their values are lower than the optimal ranking. RL-BS1 performs better than Pairwise-DQN-KR for both simple datasets. Moreover, RL-BS1 and Pairwise-A2C-TF perform equivalently on the IOFROL dataset. These results are anyway lower than the optimal ranking. As pointed out by Bagherzadeh et al. [1], the test execution history provided by simple datasets is not sufficient enough to learn an accurate test prioritization policy. The row of RL-BS2 in Tables 4.9 and 4.10 shows the results of an RL-based solution reported by Bagherzadeh et al. [1]. For all datasets, Pairwise-A2C-SB fares significantly better than RL-BS2 with CLES values between 71.7 and 91.0. In contrast, RL-BS2 performs better than Pairwise-A2C-TF and Pairwise-DQN-KR for all datasets: CLES values between Pairwise-A2C-TF and RL-BS2 range between 16.3 and 33.5, and between 23.8 and 34.8 for Pairwise-DQN-KR and RL-BS2. Thus, according to these results, Pairwise-A2C-SB improves the baselines in the use of DRL for test case prioritization. The row labelled by MART (MART ranking model) in Table 4.9 and 4.10 provides the results of the best ML-based solution reported by Bagherzadeh et al. [1]. For the MATH dataset, Pairwise-A2C-SB performs equivalently as MART. We observe 58.8 as the CLES value for the MATH dataset. For other datasets, Pairwise-A2C-SB fares better than MART. The CLES of Pairwise-A2C-SB vs. MART ranges between 58.8 to 85.7 with an average of 0.711, i.e., in 71.1% of the cycles, Pairwise-A2C-SB fares better than MART. Then, we can conclude that Pairwise-A2C-SB advances state-of-the-art compared to the best ML-based ranking technique (MART). Pairwise-A2C-TF and Pairwise-DQN-KR solutions perform similarly to MART with 0.549 and 0.584 CLES averages respectively.

Finding 2: The performance of DQN algorithms is close to the A2C algorithms when applying to the Pairwise ranking model.

RQa$_2$. Which combinations of DRL frameworks-algorithms perform better (i.e., get trained accurately and solve the problem effectively)?

Figure 4.15 shows the statistical results of APFD and NRPA metrics for the Pairwise-DQN configuration. The results show that the Stable-baselines framework performs better for all enriched datasets. Similarly, Figure 4.19 shows the statistical results of APFD and NRPA metrics regarding the DDPG-Pointwise configuration. According to the reported results, Stable-baselines perform best. Moreover, to analyze the accuracy of DRL algorithms w.r.t the relative performance, we performed two sets of Welch's ANOVA and Games-Howell post-hoc tests corresponding to the pairwise and pointwise ranking models, based on the result of all algorithms across datasets. Tables 4.16 and 4.17 show for each configuration the calculated mean, p-value and CLES. The results show that for the enriched datasets A2C-SB has better performance on both the pairwise and pointwise ranking models. Regarding the simple datasets, none of the DRL configurations has learned an adequate ranking strategy, as the highest CLES value is 0.63. This is explained by the fact that it cannot always be possible to learn a proper policy from simple data.

To compare the DRL configurations based on their training time, we perform a similar analysis as discussed above for the 10 first cycles. We summarize the results as follows:

- Pairwise and simple datasets:
  - DQN-SB > DQN-KR > DQN-TF
  - A2C-SB > A2C-TF
- Pairwise and enriched datasets:
  - DQN-SB > DQN-KR > DQN-TF

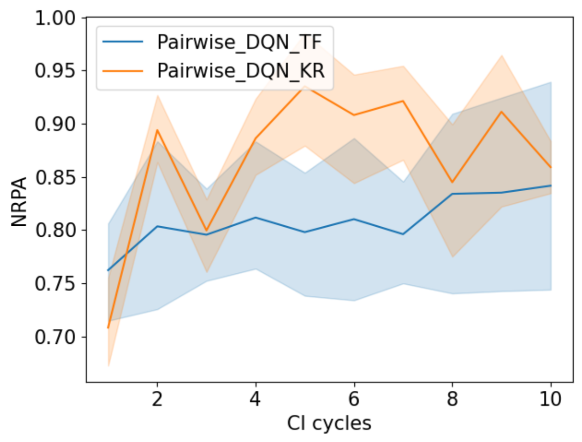


Figure 4.18 NRPA of the Pairwise-DQN configurations for the first 10 CI cycles on the CODEC dataset.

  - A2C-SB > A2C-TF

- Pointwise and simple datasets:
  - DDPG-SB > DDPG-KR > DDPG-TF
  - A2C-SB > A2C-TF
- Pointwise and enriched datasets:
  - DDPG-SB > DDPG-KR > DDPG-TF
  - A2C-SB > A2C-TF

To compare the DRL configurations based on their prediction time, here is the result for the 10 first cycles:

- Pairwise and simple datasets:
  - DQN-KR > DQN-SB > DQN-TF
  - A2C-SB > A2C-TF
- Pairwise and enriched datasets:
  - DQN-KR > DQN-SB > DQN-TF
  - A2C-SB > A2C-TF
- Pointwise and simple datasets:

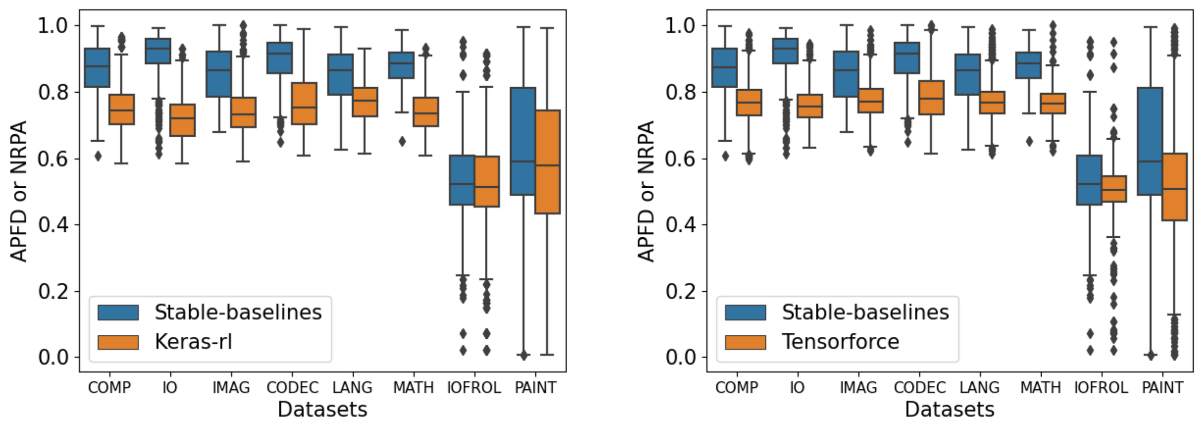


Figure 4.19 APFD (simple datasets) or NRPA (enriched datasets) of DDPG-POINTWISE configuration across DRL frameworks for all datasets: Stable-baselines vs. Keras-rl (left) and Stable-baselines vs. Tensorforce (right).

  - DDPG-KR > DDPG-SB > DDPG-TF
  - A2C-SB > A2C-TF

- Pointwise and enriched datasets:
  - DDPG-KR > DDPG-SB > DDPG-TF
  - A2C-SB > A2C-TF

Based on the results presented, we can conclude that both pairwise and pointwise configurations perform well with Stable-baselines and Keras-rl frameworks in terms of prediction times. Nevertheless, Tensorforce configurations need more time for training and prediction times.

Finding 3: Overall, the Stable-baselines framework has better performance than Keras-rl and Tensorforce framework, therefore, it can be recommended when using DRL for test case prioritization.

RQa$_3$. How stable are the results obtained from the DRL frameworks, over multiple runs?

Figure 4.20, 4.21, 4.18 show the results of the Pairwise-DQN configurations from Tensorforce and Keras-rl frameworks in terms of NRPA, accumulated reward obtained by agents during

Table 4.16 Results of Welch ANOVA and Games-Howell post-hoc tests on pairwise and pointwise ranking models for enriched datasets.

	A	B	mean(A)	mean(B)	pval	CLES
Pairwise and enriched datasets	**A2C-SB**	A2C-TF	0.96	0.77	0.0	1.0
	A2C-SB	DQN-KR	0.96	0.77	0.0	1.0
	A2C-SB	DQN-SB	0.96	0.95	0.0	0.56
	A2C-SB	DQN-TF	0.96	0.86	0.0	0.92
	A2C-TF	**DQN-KR**	0.77	0.77	0.0	0.47
	A2C-TF	**DQN-SB**	0.77	0.95	0.0	0.01
	A2C-TF	**DQN-TF**	0.77	0.86	0.0	0.17
	DQN-KR	**DQN-SB**	0.77	0.95	0.0	0.01
	DQN-KR	**DQN-TF**	0.77	0.86	0.0	0.18
	DQN-SB	DQN-TF	0.95	0.86	0.0	0.88
Pointwise and enriched datasets	DDPG-KR	**DDPG-SB**	0.75	0.87	0.0	0.13
	DDPG-KR	**DDPG-TF**	0.75	0.77	0.0	0.42
	DDPG-SB	DDPG-TF	0.87	0.77	0.0	0.83
	A2C-SB	A2C-TF	0.90	0.84	0.0	0.73
	A2C-SB	DDPG-KR	0.90	0.75	0.0	0.95
	A2C-SB	DDPG-SB	0.90	0.87	0.0	0.62
	A2C-SB	DDPG-TF	0.90	0.77	0.0	0.94
	A2C-TF	DDPG-KR	0.84	0.75	0.0	0.82
	A2C-TF	DDPG-SB	0.84	0.87	0.0	0.41
	A2C-TF	DDPG-TF	0.84	0.77	0.0	0.78

training and accumulated reward obtained by agents during testing on CODEC dataset. The results are collected for the first 10 CI cycles over 5 different runs. Regarding the DQN algorithm, Keras-rl and Tensorforce have the same performance in terms of reward but perform differently in terms of NRPA. Similarly, as with the other DRL algorithms, we do not observe stable results across the DRL frameworks.

4.4 Recommendations about frameworks/algorithms selection

We now provide recommendations regarding the selection of DRL frameworks/algorithms for researchers and practitioners. The results of our analysis indicate that there are some differences in using the same algorithm from different DRL frameworks. This is due to the diversity of hyperparameters that are offered by different DRL frameworks. Among the studied DRL frameworks, the DQN algorithm from Keras-rl has the least number of hyperparameters (13 in total), leading to less flexibility in improving the agent's training process thus explaining its poor performance. Moreover, Table 4.18 shows the results of the tuning of some hyperparameters provided by DQN-KR. In bold are the values of the hyperparameters we initially used for our experiments. Then every time we vary each of them individually (see Column "Values" for their values) and collect the average number of bugs and state coverage. The results show that fine-tuning the hyperparameters do not make DQN-KR significantly more performant. DQN-SB still has better performance. A DRL

Table 4.17 Results of Welch ANOVA and Games-Howell post-hoc tests on pairwise and pointwise ranking models for simple datasets.

	A	B	mean(A)	mean(B)	pval	CLES
Pairwise and simple datasets	**A2C-TF**	A2C-SB	0.60	0.54	0.0	0.57
	A2C-TF	DQN-KR	0.60	0.50	0.0	0.63
	A2C-TF	DQN-SB	0.60	0.61	1.0	0.49
	A2C-TF	DQN-TF	0.60	0.55	0.0	0.56
	A2C-SB	DQN-KR	0.54	0.50	0.0	0.59
	A2C-SB	**DQN-SB**	0.54	0.61	0.0	0.42
	A2C-SB	DQN-TF	0.54	0.55	1.0	0.49
	DQN-KR	**DQN-SB**	0.50	0.61	0.0	0.36
	DQN-KR	**DQN-TF**	0.50	0.55	0.0	0.43
	DQN-SB	DQN-TF	0.61	0.55	0.0	0.57
Pointwise and simple datasets	A2C-TF	**DDPG-TF**	0.46	0.51	0.0	0.42
	A2C-TF	**A2C-SB**	0.46	0.55	0.0	0.37
	A2C-TF	**DDPG-KR**	0.46	0.55	0.0	0.37
	A2C-TF	**DDPG-SB**	0.46	0.59	0.0	0.32
	DDPG-TF	**A2C-SB**	0.51	0.55	0.0	0.44
	DDPG-TF	**DDPG-KR**	0.51	0.55	0.0	0.44
	DDPG-TF	**DDPG-SB**	0.51	0.59	0.0	0.39
	A2C-SB	DDPG-KR	0.55	0.55	1.0	0.50
	A2C-SB	**DDPG-SB**	0.55	0.59	0.0	0.44
	DDPG-KR	**DDPG-SB**	0.55	0.59	0.00	0.44

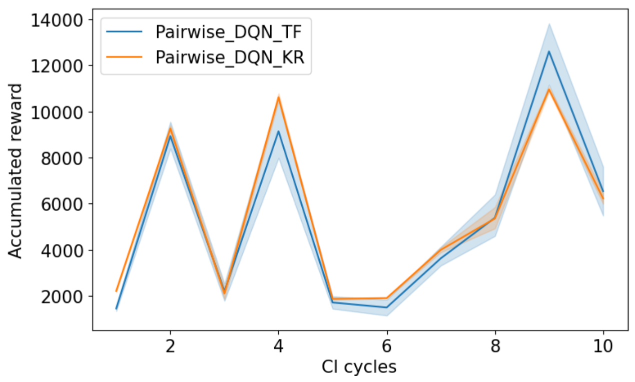


Figure 4.20 Accumulated reward during training of the Pairwise-DQN configurations
for the first 10 CI cycles on CODEC dataset.

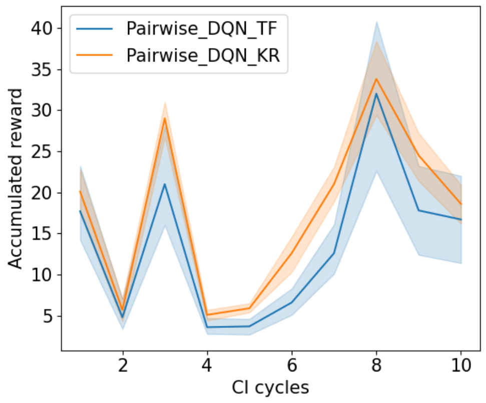


Figure 4.21 Accumulated reward during testing of the Pairwise-DQN configurations for the first 10 CI cycles on the CODEC dataset.

Table 4.18 Results of state coverage and the number of bugs detected, performed by DQN-KR configuration for the game testing problem on a 10k steps budget over 5 runs.

Hyperparameters	Value	Average number of bugs	Average state coverage
Learning rate	**0.00025**	0.22	10.2
	0.01	1	14.4
	0.001	0.6	6
Batch size	**128**	0.22	10.2
	64	0.4	12
	32	0	6.4
Policy	**Epsilon greedy**	0.22	10.2
	Boltzman	0.4	4.4
Enable dueling dqn	**False**	0.22	10.2
	True, dueling type=avg	0	6.8
	True, dueling type=max	1	9.6
	True, dueling type=naive	0.8	11.4
Test policy	**GreedyQPolicy()**	0.22	10.2
	Epsilon greedy	0.2	9.4
	Boltzman	0.4	8.8
Gamma	**0.99**	0.22	10.2
	0.25	0	8
	0.025	0.6	6.4

framework should offer a large number of hyperparameters to provide flexibility for tuning DRL agents and improve their efficiency.

Recommendation 1: When applying a DRL algorithm from a DRL framework to an SE problem, we recommend choosing the DRL framework that offers the largest number of hyperparameters to have the flexibility to improve the agent efficiency.

In this chapter, we studied two problems whose characteristics can be found in Annex A. Regardless of the studied frameworks, PPO's and A2C's algorithms have shown good performance when applied to the game testing problem. The PPO has shown slightly better performance as it has detected 1 to 2 more bugs than the A2C when used to detect bugs in the Block Maze game. Regarding the studied test case prioritization problem, Pairwise-A2C-SB yields the best performance. The implementations of PPO and A2C algorithms show good performance on discrete action space. The studied problems have been implemented using both kinds of reward distribution. In the game testing problem, the agent is positively rewarded only when it reaches the goal otherwise it is rewarded with small negative values. The results have shown that this kind of reward does not incentivize the agent to reach the goal regardless of the DRL implementation applied. In the test case prioritization problem, the agent is positively rewarded with small values even when it fails to rank test cases. Some DRL configurations have performed very well in terms of APFD or NRPA, close to the optimal value.

Table 4.19 DQN algorithm hyperparameters: differences between Stable-baselines and Tensorforce.

Frameworks	Hyperparameters	Definitions
Stable-baselines	Soft Update Coefficient	The target network of the DQN algorithm is updated frequently by a little amount.
	Gradient Step	Number of gradient steps to do after each rollout.
	Exploration Fraction	Fraction of the training period over which the exploration rate is reduced
	Gradient Clipping	The maximum value for the gradient clipping
	Tensorboard Log	The log location for tensorboard
	Evaluate Environment	Whether to create a second environment that will be used for evaluating the agent periodically.
	Seed	Seed for the pseudo random generators
	Device	Device on which the code should be run
	Model Setup	Whether or not to build the network at the creation of the instance
Tensorforce	Variable Noise	Alternative exploration mechanism by adding Gaussian noise to all trainable variables
	State Preprocessing	State preprocessing as layer or list of layers
	Reward Preprocessing	Reward preprocessing as layer or list of layers
	Return of processing	Return processing as layer or list of layers,
	L2 Regularization	L2 regularization loss weight
	Entropy Regularization	To discourage the policy distribution from being too certain
	Huber Loss	Threshold of the Huber loss function

Recommendation 2: When designing a DRL problem, we recommend keeping the cumulative reward of the agent positive for better performance.

To showcase the difference between employing a simple DRL algorithm from the two frameworks, we performed some additional analysis of the hyperparameters offered by Stable-baselines and Tensorforce regarding the DQN algorithm and conducted some experiments. Here are our findings:

- Stable-baselines provides a total of **25 hyperparameters** while Tensorforce provides **22 hyperparameters**.
- Table 4.19 describes for Stable-baselines and Tensorforce, the hyperparameters that differ from each other. An interesting hyperparameter is the *variable noise* from Tensorforce, which adds Gaussian noise [163] to all trainable variables as an exploration strategy. Adding noise to DRL agents during training has been shown to improve their exploration of the environment and their gains of reward throughout training [164].
- We consider the *variable noise=0.5* as an additional hyperparameter for the DQN algorithm from Tensorforce. Therefore, we collected the number of detected bugs and average reward of the DQN agent from Tensorforce for $50,000$ steps of training on the

Table 4.20 Results of state coverage and the number of bugs detected, performed by DQN-TF configuration for the game testing problem on a 10k steps budget over 5 runs.

Hyperparameters	Value	Average number of bugs	Average state coverage
Learning rate	**0.00025**	0.2	4.4
	0.01	1.6	19.6
	0.001	0.4	12.6
Batch size	**128**	0.2	4.4
	64	0.4	7.4
	32	0	3.4
Variable noise	**0**	0.2	4.4
	0.5	8	74
	1	7.8	73.8
Gamma	**0.99**	0.2	4.4
	0.5	0	3.8
	0.1	0	2.8

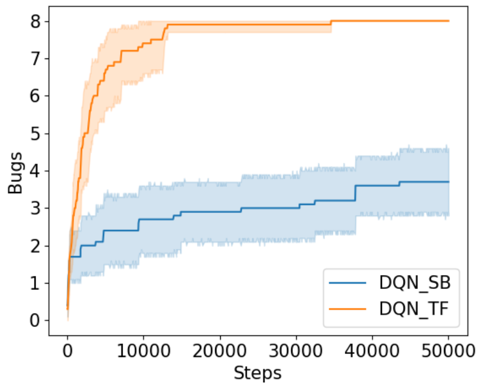


Figure 4.22 Number of bugs detected by DQN Stable-baselines and DQN (with Gaussian noise) Tensorforce.

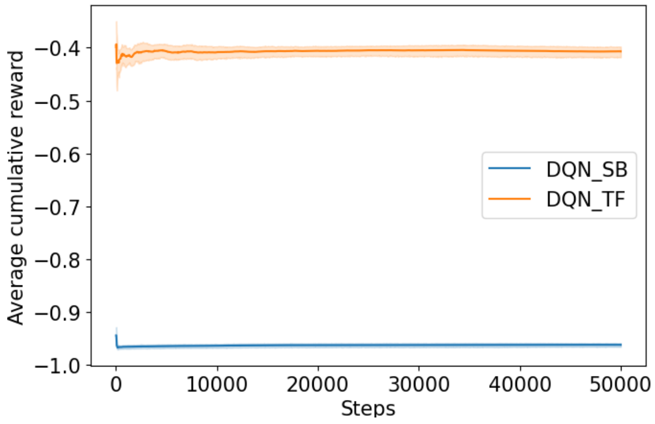


Figure 4.23 Average cumulative reward earned by DQN Stable-baselines and DQN (with Gaussian noise) Tensorforce.

Block Maze game. Figures 4.22 and 4.23 show that the DQN agent from Tensorforce is able to detect more bugs than initially (see Figure 4.1 and 4.2), with more gained reward.

- Furthermore we conducted more experiments to assess the effects of hyperparameters tuning on DQN-TF implementation regarding the game testing problem. Table 4.20 shows the number of bugs and state coverage resulting from the hyperparameters tuning. As other results, in bold are the values of the hyperparameters we initially used for our experiments. Then every time we vary each of them individually (see Column "Value" for their values). As shown on Table 4.20, the only parameter that stands out is the variable noise which boosts DQN-TF performance. Such results indicate

Table 4.21 Results of state coverage and the number of bugs detected, performed by A2C-TF configuration for the game testing problem on a 10k steps budget over 5 runs.

Hyperparameters	Value	Average number of bugs	Average state coverage
Learning rate	**0.00025**	6.6	57.4
	0.01	1.8	26.8
	0.001	1.6	18
Batch size	**128**	6.6	57.4
	64	5	55.4
	32	3.2	44.4
Variable noise	**0**	6.6	57.4
	0.5	8	74
	1	8	74
Gamma	**0.99**	6.6	57.4
	0.5	8	74
	0	8	74
Entropy regularization	**0**	6.6	57.4
	0.5	8	74
	1	8	74
L2 regularization	**0**	6.6	57.4
	0.5	7.8	73.5
	1	7.2	72
Exploration	**0**	6.6	57.4
	0.5	7	72
	1	8	74

that a DRL framework with effective exploration strategies could improve the agent performance.

- Similarly, Tables 4.21 and 4.22 show the results of hyperparameters tuning regarding PPO-TF and A2C-TF implementations. The results on these tables show up to 2 more bugs detected when applying different values of the hyperparameters that the Tensorforce framework offers (i.e., variable noise, discount factor, entropy/l2 regularization, and exploration).

Recommendation 3: In the context of testing through exploration, we recommend a DRL framework that offers effective exploration strategies such as Tensorforce.

Table 4.22 Results of state coverage and number of bugs detected, performed by PPO-TF configuration for the game testing problem on a 10k steps budget over 5 runs.

Hyperparameters	Value	Average number of bugs	Average state coverage
Learning rate	**0.00025**	6.2	68.4
	0.01	4	52
	0.001	5.8	63.4
Batch size	**128**	6.2	68.4
	64	5.6	64.8
	32	8	71.4
Variable noise	**0**	6.2	68.4
	0.5	8	74
	1	8	74
Gamma	**0.99**	6.2	68.4
	0.5	7.2	71.4
	0.1	7	70.6
Entropy regularization	**0**	6.2	68.4
	0.5	8	74
	1	8	74
L2 regularization	**0**	6.2	68.4
	0.5	8	74
	1	8	71.4
Exploration	**0**	6.2	68.4
	0.5	6.8	68.8
	1	8	74
Likelihood ratio clipping	**0.25**	6.2	68.4
	0.5	7.8	73.8
	1	7.2	69.8

4.5 Threats to validity

Conclusion validity. Conclusion limitations concern the degree to which the statistical conclusions about which algorithms/frameworks perform best are accurate. We use Welch's ANOVA and Games-Howell's post-hoc test as statistical tests. The significance level is set to 0.05 which is standard across the literature as shown by Welch et al. [95], Games et al. [96]. The non-deterministic nature of DRL algorithms can threaten the conclusions made in this work. We address this by collecting results from 10 independent runs in the case of the game testing problem. Regarding the test case prioritization problem, the results are collected from 5 independent runs and on multiple cycles (the MATH dataset has 55 cycles which is the least number of cycles among all datasets).

Internal validity. Regarding the game testing problem, the fact that we only consider the DRL part of the wuji framework for comparison with the DRL strategies we studied, might threaten the validity of this work. Although we only compare the algorithms on sub-optimal solutions, it is necessary to make a fair comparison among the DRL algorithms. A potential limitation is the number of frameworks used and the algorithms chosen among these frameworks. We have chosen to evaluate some of the available frameworks and have not evaluated all the algorithms they offer. However, the frameworks used are among the most popular on GitHub, as well as the algorithms (see Section 2.3.3). This ensures good coverage in terms of the usage of DRL in SE. In the future, we plan to expand our study to cover more algorithms.

Construct validity. A potential threat to validity is related to our evaluation metrics, which are standard across the literature. We use these metrics to make a fair comparison amongst frameworks/algorithms under identical circumstances. We discussed some of their limitations and how they can be interpreted in Sections 4.2.2 and 4.2.3.

External validity. Since our goal is to compare DRL frameworks and their implemented algorithms for SE testing tasks, a potential limitation is the choice of the testing tasks for comparing frameworks. We address this threat by choosing game testing and test case prioritization problems that are totally different in SE testing to achieve enough diversity. While test case prioritization focuses on optimizing the order of test cases, one needs to find bugs in the game testing as early as possible. The results we found on the two studied problems might mitigate this threat as we consistently found some algorithms performing similarly among the frameworks. For example, the A2C algorithm had good performance whether applied to the game testing problem or test case prioritization problem.

Reliability validity. To allow other researchers to replicate or build on our research, we provide a detailed replication package [158] including the code and obtained results.

4.6 Chapter Summary

In this chapter, we study the application of state-of-the-art implemented DRL algorithms from well-known frameworks on two important software testing tasks: test case prioritization and game testing. We rely on two baseline studies to apply and evaluate the performance of DRL algorithms from several frameworks (i) in terms of detecting bugs in a game, and (ii) in the context of a CI environment to rank test cases. Our results show that the same algorithm from different DRL frameworks can have different performances. Each framework provides hyperparameters unique to its implementation, therefore depending on the underlying SE tasks, the framework that has the most suitable hyperparameters will lead to better performance. We formulate recommendations to help SE practitioners make informed decisions when leveraging DRL frameworks for the development of SE tasks.

CHAPTER 5 HARNESSING PRE-TRAINED GENERALIST AGENTS FOR SOFTWARE ENGINEERING TASKS

5.1 Chapter Overview

In the previous Chapter, we trained DRL agents (also known as specialist agents) with traditional DRL algorithms (i.e., A2C, PPO, DQN). Despite the good results obtained that led to meaningful recommendations for SE practitioners, the specialist agents require long training time due to a large amount of training data, and they can struggle to generalize to new tasks [6, 16, 23, 57]. SE practitioners could also benefit from techniques (1) that do not require long training time and (2) that can easily generalize to a new SE task.

In this Chapter, we explore another category of DRL agents called DRL generalist agents and empirically compare their performance against the performance of specialist agents [6, 16, 23, 57] on three SE tasks: Playtesting in games, bug localization across software projects and the minimization of makespan of scheduling operations. Generalist DRL agents, trained on task-agnostic datasets, can adapt to unseen tasks and maintain high performance with few fine-tuning [7, 8] steps. The fine-tuning process involved some of the DRL algorithms employed in Chapter 4 without training an agent from scratch. We show that prior knowledge, fine-tuning budget, and scalable architecture of generalist agents enhance their performance on SE tasks, enabling them to perform better or close to the performance of specialist agents with few fine-tuning steps.

5.2 Study design

In this section, we present the methodology of our study which aims to investigate which generalist agents (or which of its different configurations) with regards to the fine-tuning budget allowed, perform better than or are close to the specialist agents' performance.

5.2.1 Setting Objectives of the Study

Researchers and practitioners have developed various techniques to enhance the reliability, effectiveness, and quality of software systems. Approaches such as coverage-based testing [165] and search-based testing [166] aim to ensure that software products behave as expected. Studies by Munro et al. [167] and Singh et al. [168] explored techniques for detecting code smells, while Gao et al. [169] and Kaur et al. [170] proposed task scheduling techniques for optimizing cloud computing performance. More recently, DRL has gained the attention of SE researchers, with algorithms like PPO [50], A2C [37], and DQN [47] being applied to train agents to solve SE tasks. While specialist DRL agents have shown promising results in SE tasks, their training often requires significant time due to the large amount of data needed [1, 6]. Moreover, researchers have observed that specialist DRL agents trained to excel at specific tasks often struggle to generalize to never-before-seen tasks [171, 172], which can leave them unable to solve real-world tasks, as these tasks are likely to evolve constantly because of changing environments [173].

A generalist agent trained on multiple tasks that can solve never-before-seen tasks is a goal for the AI community [174]. In other domains, like NLP [129] and Computer Vision [175], researchers have built generalist models that can adapt to never-before-seen tasks and achieve high performance, by training them on large task-agnostic datasets with a light fine-tuning step. Unlike DL-based generalist models [174, 175], DRL-based generalist agents are autonomous agents that learn optimal strategies in an environment (e.g., optimally rank test cases in a continuous integration environment [1]) and can adapt to never-before-seen environments without extensive domain-specific fine-tuning. In this study, To evaluate the usefulness of generalist agents on SE tasks, we utilized two pre-trained generalist agents: MGDT and IMPALA. We carefully fine-tuned them (on zero-shot, 1%, and 2% data budgets) on the Blockmaze and MsPacman games for playtesting in games and collected the time to find bugs, the cumulative reward, and the average training and testing times using model-free DRL algorithms. Similarly, we fine-tuned them on a scheduling-based task and collected the makespan, the cumulative reward, and the average training and testing times. Finally, we fine-tuned the pre-trained generalist agents to localize buggy files across six large-scale, open-source software projects (Birt, Eclipse, SWT, AspectJ, JDT, and Tomcat). We collected metrics such as Mean Reciprocal Rank (MRR), Mean Average Precision (MAP), top@1, top@5, top@10, cumulative reward, and average training and testing times.

The findings of this study show that generalist agents are effective for playtesting in games by efficiently exploring games and quickly identifying bugs. Additionally, on the task scheduling process, generalist agents achieved the most significant reductions in makespan across all fine-tuning data budgets in the studied scheduling instances. Finally, in the bug localization task, the generalist agents show comparable or greater performance compared to the baseline specialists.

To structure the study in this chapter, we define the following RQs:

RQb$_1$: How do generalist agents perform on software engineering tasks? In this RQ, we fine-tune the DRL generalist agents on different data budgets and collect the time to find bugs in the Blockmaze and MsPacman games. We also collect the average makespan obtained by the fine-tuned generalist agents on the PDR-based scheduling tasks. Regarding the bug localization task, we collect the MAP, MRR, top@1, top@5 and top@10 metrics. Finally, we collect the average cumulative reward and the training and testing times of the DRL generalist agents for all the studied tasks

RQb$_2$: How do different DRL generalist agents perform on SE tasks compared to DRL specialist agents? In this RQ, based on the results of **RQb$_1$**, we compare the pre-trained generalist agent that performed best against the baselines, based on the fine-tuning data budgets w.r.t to the evaluation metrics for each task. We calculate Common Language Effect Size (CLES) [176], [177], between the best configuration and the baseline to assess the effect size of differences.

RQb$_3$: How do different model-free DRL algorithms affect the performance of DRL generalist agents on SE tasks? Finally in this RQ, we compare the model-free DRL algorithms used to fine-tune the DRL generalist agents against each other. We use Tukey's post-hoc test [94] to indicate the best online method. A difference with p-value $<= 0.05$ is considered significant in our assessments. When the variances in our results are not equal, we employ the Welch's ANOVA and Games-Howell post-hoc test [95], [96] instead of Tukey's test, as Welch's ANOVA do not assume equal variance (see Chapter 2).

5.2.2 Methodology

We employ DRL generalist agents in several SE tasks. More specifically, the DRL generalist agents are fine-tuned on selected SE tasks in order to accomplish the task at hand. In this chapter, we leverage two generalist agents IMPALA [7] and MGDT [8].

Overview of IMPALA

IMPALA (Figure 5.1) [7] is based on an actor-critic architecture with a set of actors and a set of learners. The actors generate trajectories of experiences and the learners use the experiences to learn an off-policy π. At the beginning of the training, each actor updates its own policy μ to the latest learner policy π and runs it for n steps in its environment. At the end of n steps, each actor sends the collected observations, actions, rewards, as well as its policy distribution μ and an initial Long Short-Term Memory (LSTM) state to the learners via a queue. The learners update their own policy π based on experiences collected from the actors. An IMPALA model architecture consists of multiple convolutional networks for feature extraction, followed by an LSTM [178] and a fully connected output layer. This architecture of IMPALA allows the learners to be accelerated on GPUs and the actors distributed across multiple machines. Given that the pre-trained version of IMPALA is not available, we pre-trained it on 57 games of the Atari learning environment [179] using V-trace [7], an off-policy actor-critic algorithm (see Chapter 2). We pre-trained the generalist agent, IMPALA, using the 57 games of the Atari suite environments [179] for 50M steps. Please note that the MsPacman game is excluded from the pre-training since it is among our evaluation tasks.

Overview of the Multi-Game Decision Transformer (MGDT)

MGDT, as illustrated in Figure 5.2, is a decision-making agent similar to a DT that receives at time t an observation o_t, chooses an action a_t, and receives a reward r_t. The MGDT is an offline pre-trained DRL agent with the goal of learning a single optimal policy distribution $P^*_\theta(a_t \mid o_{\leq t}, a_{<t}, r_{<t})$ with parameters θ that maximizes the agent's total future return $R_t = \sum_{k>t} r^k$. Further, during training, the trajectories are modeled in the form of a sequence of tokens:

$$s = \langle ..., o_t^1, ..., x_t^M, \hat{R}_t, a_t, r_t, ... \rangle$$

where t represents a time-step, M is the number of image patches per observation and $\hat{R}_t$ is the MGDT agent target return for the rest of the sequence. The returns, actions and rewards are converted to discrete tokens after being generated via multinomial distributions. The scalar rewards are converted to ternary quantities $\{-1, 0, +1\}$, and the returns are uniformly quantized into a discrete range shared by the environment considered by Lee et al. [8]. We did not pre-train the MGDT agent as the checkpoint of the pre-training done by Lee et al. [8] is available online.[1] For pre-training, Lee et al. [8] used data from the Atari trajectories of 41 environments introduced by Agarwal et al. [180] on which a DQN agent was trained [181]. The data used for pre-training are from two training runs, each with roll-outs from 50 policy checkpoints for a total of 4.1 billion steps. Similar to the case of IMPALA, the MsPacman game is excluded from the pre-training since it is among the evaluation tasks [8].

Online fine-tuning on SE tasks

In our study, we fine-tune DRL generalist agents via online interactions (i.e., with model-free DRL algorithms) to perform SE tasks. Regarding the MGDT agent, we employ MAENT [35], PPO [50] and DQN [47] as the learning policy of the DRL agent. Regarding the IMPALA agent, in addition to V-trace, we leverage PPO as the agent's learning policy. We did not

[1]https://github.com/google-research/google-research/tree/master/multi_game_dt

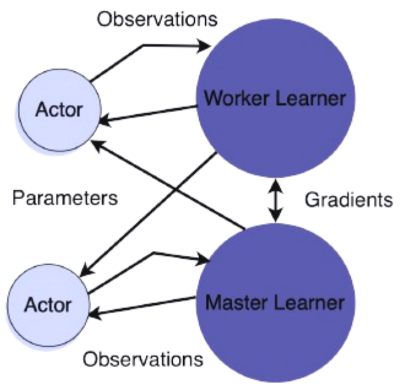


Figure 5.1 IMPALA architecture [7].

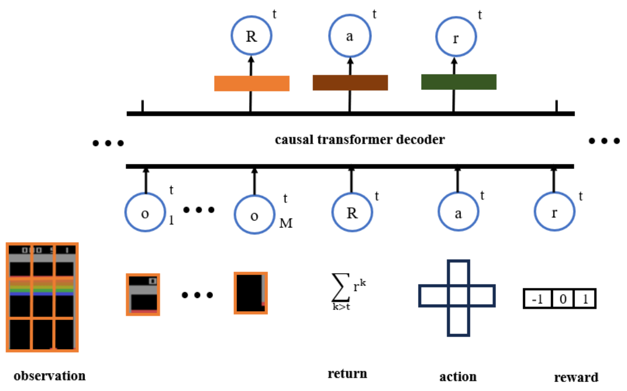


Figure 5.2 Multigame Decision Transformer Architecture (MGDT) [8].

use DQN with IMPALA because IMPALA has an actor-critic architecture incompatible with DQN's value-based learning policy.

Algorithms 1 and 2 summarize the overall fine-tuning procedure we implement to leverage the DRL generalist agents on our SE tasks. With Algorithm 1, at each iteration we generate

Algorithm 1: Online fine-tuning of the MGDT agent

```
Input : model parameters $\theta$, replay buffer $\mathcal{R}$, number of iterations $\mathcal{I}$, context length
        $\mathcal{K}$, batch size $\mathcal{B}$
1 for t=1,..., $\mathcal{I}$ do
2     Generate trajectories and fill $\mathcal{R}$
3     Sample $\mathcal{B}$ trajectories out of $\mathcal{R}$
4     for each sampled trajectory $\tau$ do
5         $(a, x, R)$ ← a length $\mathcal{K}$ sub-trajectory sampled uniformly from $\tau$
6     $\theta$ ← one gradient update using the sampled sub-trajectory $\{(a, x, R)\}s$.
```

trajectories to fill up the replay buffer. Then, we sample $\mathcal{K}$ trajectories to perform one gradient update of the policy of either one of the online training methods we used (i.e., MAENT, PPO, and DQN). With Algorithm 2, each actor generates trajectories that are

Algorithm 2: Online fine-tuning of the IMPALA agent

```
Input : model parameters $\theta$, replay buffer $\mathcal{R}$, number of iterations $\mathcal{I}$,
        $\{Actor_1, ..., Actor_N\}$, Learner $\mathcal{L}$
1 for t=1,..., $\mathcal{I}$ do
2     In parallel $Actor_i$ generates trajectories $\{(x_1, a_1, r_1), ..., (x_n, a_n, r_n)\}$
3     $\mathcal{R} \leftarrow \{(x_1, a_1, r_1), ..., (x_n, a_n, r_n)\}$
4     $Actor_i$ retrieves policy parameters from $\mathcal{L}$
5     $\theta$ ← one gradient update of $\mathcal{L}$ using trajectories in $\mathcal{R}$
```

sent to the learner (PPO or V-trace) to update its policy gradients.

Baselines studies

In order to assess the performance of our fine-tuned generalist agents on SE tasks, we consider three baseline studies that we replicate, collect the performance of the specialist DRL agents being used and compare them against the pre-trained generalist. We picked these baseline studies because they target challenging problems and the source code of their proposed approaches is available.

1) Zheng et al. [16] proposed Wuji, an automated game testing framework that combines Evolutionary Multi-Objective Optimization (EMOO) and DRL to detect bugs in a game.
Approach: Wuji randomly initializes a population of policies (represented by DNNs), adopts EMOO to diversify the state's exploration, and then uses DRL to improve the capability of the agent to accomplish its mission which is to play the game.
Zheng et al. [16] evaluated their approach on three games namely a Blockmaze game, L10 a Chinese ghost story[2] and NSH a Treacherous Water Online.[3] Given that the two latest are commercial games, with closed source, we do not consider them in our study. We replicated Wuji and collected the time to find bugs, and its training and testing times performance on the Blockmaze game.

2) Tufano et al. [57] proposed RELINE, an approach that leverages DRL algorithms to detect performance bugs. Specifically, the authors injected artificial performance bugs in two games, Cartpole [182], and MsPacman [88], and investigated whether or not the DRL agents can detect the bugs. Further, the authors studied a 3D kart racing game, Supertuxkart [183] and investigated whether or not their approach can find parts of the game resulting in a drop in the number of frames per second.
Approach: RELINE leverages DQN [47] to train a DRL agent. Specifically, DQN adopts a Convolutional Neural Network (CNN) architecture that takes as input screenshots of the game and returns one of the possible actions of the game. Moreover, to incentivize the agent to look for bugs, an additional +50 is added to the reward earned by the agent every time it finds a bug during an episode.
In our study, we consider MsPacman as one of our evaluation studies. We did not consider Cartpole because our generalist agents are only compatible with 2D observation space. We replicated RELINE and collected the time to find bugs, and its training and testing times performance on the MsPacman game.

3) Zhang et al. [6] leveraged DRL to automatically learn a PDR-based scheduling task for solving JSSP. They tested their approach on scheduling instances of various sizes $(6\times6), (10\times 10), (15\times15), (20\times20), (30\times20), (50\times20), (100\times20)$ as well as on JSSP benchmarks and collected the makespan metric.
Approach: Zhang et al. [6] used PPO algorithm to train a DRL agent that solves the JSSP. They proposed a Graph Neural Network (GNN) based architecture to encode the nodes of the disjunctive graphs of each JSSP instance. Particularly, the GNN captures the feature embedding of each node in a non-linear fashion.
In our study, we consider two instances $((6\times6), (30\times20))$ to compare the performance of our

[2]https://xqn.163.com/
[3]http://n.163.com/

fine-tuned generalist agents against a PPO-based specialist agent utilized by Zhang et al [6]. We picked two scheduling instances each among the small and medium sizes. We replicated Zhang et al.'s [6] approach and collected the makespan metric, the cumulative reward earned, as well as the training and testing times performance. Since the authors of Wuji [16] and RELINE [57] did not consider the cumulative reward as a metric in the original study, we did not report it here. The reason is that we would not have any baselines to compare our results.

4) Chakraborty et al. [23] propose RLOCATOR, an approach that leveraged DRL to locate buggy source code files across six software projects (Birt, Eclipse, SWT, AspectJ, JDT, and Tomcat)
Approach: Chakraborty et al. [23] used the A2C algorithm (with and without entropy) to train a DRL agent to rank source code files based on their relevance to a bug report. A2C with entropy incorporates the entropy of the probability of the possible action to the loss of the actor model. They adopt a CNN architecture followed by a Long Short-Term Memory (LSTM). The CNN architecture captures the relation between source code embedding, bug report embedding, and across different concatenated embeddings. The LSTM is used to make the DRL model aware of previous actions.
In our study, we consider the six software projects (Birt, Eclipse, SWT, AspectJ, JDT, and Tomcat) to compare the performance of our fine-tuned generalist agents against the two configurations of specialist agents (RLOCATOR with and without entropy) proposed by Chakraborty et al. [23]. We replicated Chakraborty et al.'s [23] approach and collected the MRR, MAP, top@1, top@5, top@10, the cumulative reward earned, as well as the training and testing times performance. Since the authors of RLOCATOR did not consider the cumulative reward as a metric in the original study, we did not report it here.

Datasets

In this section, we describe the training dataset used to fine-tune the selected DRL generalist agents for each task. We fine-tuned the pre-trained generalist agents on trajectories collected from the Blockmaze game, MsPacman game, six open-source Java repositories and a PDR-based scheduling environment.

1) A Blockmaze game, Figure 5.3, from Zheng et al. [16], is selected for the evaluation of the generalist's agents. In the Blockmaze game, the agent's objective is to reach the goal coin, and it has 4 possible actions to choose from: north, south, east, and west. Every action leads to moving into a neighbor cell in the grid in the corresponding direction, except that a collision on a block (shown by dark green in Figure 5.3) results in no movement. To evaluate

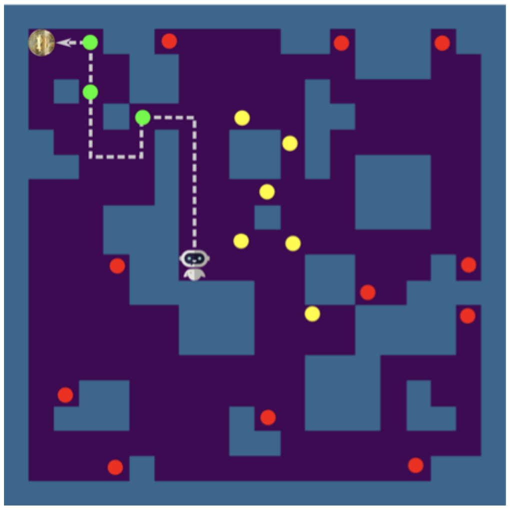


Figure 5.3 Blockmaze with bugs (red, green and yellow dots).

the effectiveness of the fine-tuned generalist agents, 25 bugs are artificially injected into the Blockmaze, and randomly distributed within the task environment. A bug is a position in the Blockmaze that is triggered if the robot (agent) reaches its location in the map, as shown in Figure 5.3 with dots with colours green, red and yellow. A bug has no direct impact on the game but can be located in invalid locations of the game environment such as the Blockmaze obstacles or outside of the Blockmaze observation space. Invalid locations, on the other hand, cause the end of the game. Therefore, in this study, we consider two types of bugs: Type 1 that refers to exploratory bugs that measure the exploration capabilities of the agent, and Type 2 that refers to bugs at invalid locations of the Blockmaze.

2) The objective of the MsPacman [57] game is to eat all the dots without touching the ghosts. The possible actions to choose from are: north, south, east, west, and none. Four performance bugs are artificially injected into the MsPacman game, at four gates within the task environment (see white arrows in Figure 5.4). These kinds of bugs (i.e., performance bugs on the MsPacman game) cause the number of observation frames per second to drop when playing the game. Specifically, a bug is a designated area within the environment, and whenever this area is reached, the agent will be rewarded with additional +50 points. For the MsPacman game, we consider 4 types of bugs representing the four gates (Figure 5.4) where bugs can be spotted.

3) We consider a well-established benchmark for bug localization to evaluate our approach [64]. Previous and related studies also evaluate their approaches on this benchmark, considering its quality and the diversity of bug scenarios [23]. The benchmark covers six open-source Java repositories from Apache: Birt, Eclipse, SWT, AspectJ, JDT, and Tomcat. We use

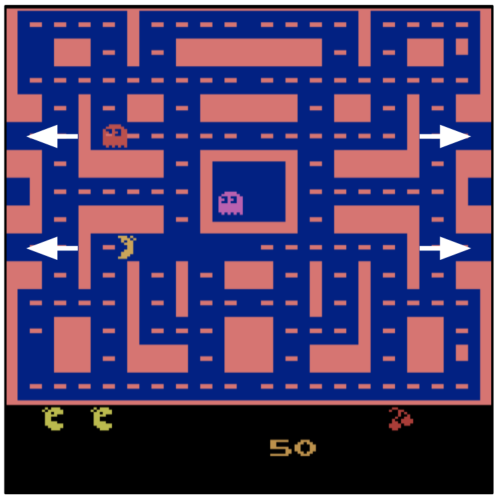


Figure 5.4 MsPacman game with bugs (white arrows).

a dataset division of a 60:40 percent split for training and testing similar to the baseline specialist agent [23]. From each commit associated with a bug report, we checkout on the commits and access the full version of the buggy source code. We report the number of bug reports on each repository in our replication package [184]

4) Finally, we consider a PDR-based scheduling [6] technique to solve the JSSP. The objective of the task is to dispatch a set of operations of jobs while minimizing the makespan. The instances we considered ((6×6), (30×20)) are generated following the Taillard's method [185]. We used the same generated instances that have been used in the baseline study [6] for training and testing.

Experimental setup

We trained the baseline specialist agent on the Blockmaze game until there was no performance improvement after 7 consecutive training rounds. This process took 12 hours in terms of training time metric (see Section 5) on our hardware setup. We fine-tuned the generalist agents on the Blockmaze game zero-shot, 1% and 2% fine-tuning data budgets of the baseline [16], corresponding to 432 seconds and 864 seconds of training time. We evaluated the baseline specialist and the generalist agents on the Blockmaze game for 300,000 steps.

We trained the baseline specialist agent [57] on the MsPacman game for 1,000 episodes (i.e., an episode lasts for a maximum of 27,000 steps). Based on the baseline budget training, we fine-tuned (i.e., zero-shot, 1% and 2% fine-tuning data budgets) the DRL generalist agents on the MsPacman game for 0, 10 and 20 episodes. We evaluated the baseline specialist and the fine-tuned generalist agents on the MsPacman game for 1000 episodes.

We trained the baseline specialist agent [23] on the bug localization task for 7500 episodes (i.e., an episode lasts for a maximum of k=31 steps). We fine-tuned (i.e., zero-shot, 1% and 2% fine-tuning data budgets) the DRL generalist agents on the bug localization task for 0, 75 and 150 episodes. We evaluated the baseline specialist and the fine-tuned generalist agents on the bug localization task for all unique values in each test dataset that have exactly k=31 corresponding source code files.

Finally, we trained the baseline specialist agent [6] on the task scheduling for 10,000 iterations (i.e., corresponding to 36,000 steps on the (6×6) and 6M steps on the (30×20)). We fine-tuned (i.e., zero-shot, 1% and 2% fine-tuning data budgets) the DRL generalist agents on the task scheduling for 0, 360, and 720 steps for the (6×6) instance and 0, 60,000, and 120,000 steps for the (30×20) instance. We evaluated the baseline specialist and the fine-tuned generalist agents on 100 generated instances.

Across all studied tasks, we trained the baseline specialist agents similar to the baseline authors [6, 16, 23, 57]. Given the difference in training budgets of the baseline specialist agents and, hence of the generalist agents, we report in Table 5.2 the amount of training time the baseline specialist and generalist agents achieved. To counter the randomness effect during testing, we repeat each run 5 times to average the results.

As for the hyperparameters, we did not perform hyperparameters tuning, as in this chapter we only study whether or not DRL generalist agents can be leveraged on the task at hand with little effort for fine-tuning. In our replication package [184], we summarize the hyperparameters we used on each generalist agent, as well as the ones we used for each online training algorithm considered in this chapter (DQN, PPO, MAENT, and V_TRACE).

Evaluation metrics

Depending on the task at hand, we use the following metrics to evaluate the performance of the fine-tuned generalist agents against the baselines specialist agents:

- **Time to find a bug:** We collect the time (in seconds) consumed by the fine-tuned generalist agents to find the first occurrence of each type of bug, averaged over five different runs. When playtesting in games, the objective of generalist agents is to achieve the shortest time to find the first occurrence of each type of bug. Specifically when playing games each generalist agent should be as fast as possible at finding the first occurrence of each type of bug. We collect this metric for the Blockmaze and MsPacman games.

- **The average cumulative reward:** We report the reward obtained by the fine-tuned generalist agents as well as the specialist agents. We collect this metric for all our tasks.
- **Makespan:** We report the makespan obtained by the fine-tuned generalist agents as well as the specialist agents. We collect this metric for only the PDR-based scheduling.
- **Mean Reciprocal Rank (MRR):** is a metric used to evaluate the effectiveness of information retrieval based on the files' ranking reported by a model within its list of results. For example, in our case, we aim to evaluate how frequently the buggy files are ranked in the first positions of a report. An MRR close to 1 means the model usually ranks the right files in the first positions, while a value close to 0 indicates poor performance. We collect this metric for only the bug localization task.
- **Mean Average Precision (MAP):** measures the overall mean performance of a model regarding a set of bug reports, unlike MRR, which considers the best rank of relevant files. Knowing that multiple buggy files can be associated with a bug, ideally, the model should rank all of them in its report. This way, MAP represents a more descriptive and unbiased metric than MRR. For a given bug, we compute the average value based on the individual precision values of each buggy file. We collect this metric for only the bug localization task.
- **Top K:** computes the overall ranking performance of a model based on whether a buggy file is properly ranked in the top N results. In our study, considering multiple files might be buggy, we check whether at least one is reported in the top N results. Following previous studies, we consider three values of N: 1, 5, and 10 [23]. We collect this metric for only the bug localization task.
- **Training time (in seconds):** We collect the time consumed by the DRL generalist agents to fine-tune their policies on the task at hand.
- **Testing time (in seconds):** During the testing, we assess the performance of fine-tuned generalist agents on the task at hand. We collect the testing time for 300,000 steps for the Blockmaze game and 100 instances for the PDR-based scheduling task. Regarding the MsPacman game, once the generalist agents have been fine-tuned, we collected the time consumed by each of them to play the game for an additional 1000 episodes, similarly to the baseline [57]. Regarding the bug localization task, we collect the time taken by our fine-tuned generalist agents to rank the $k = 31$ source code files for each unique value in the test datasets.

5.3 Results

This section presents the experimental results obtained while answering our three RQs.

5.3.1 RQb$_1$: How do generalist agents perform on software engineering tasks?

We leverage generalist DRL agents for SE tasks by fine-tuning them on playtesting specific games (i.e., the Blockmaze and MsPacman games), for solving the PDR scheduling task (on their (6×6) and (30×20) instances) and to localize files involved in bug reports. During this process, we compute task-related and non task-related metrics. Across all tasks, we collected the general metrics, namely the training and testing times as well as the cumulative reward earned to evaluate the effectiveness and efficiency of the generalist agents.

Specifically on playtesting in games, we collect the time taken to find each type of bug. Figures 5.5 and 5.6 illustrate the time taken by fine-tuned generalist agents to find bugs (i.e., averaged over all the times taken to find the first occurrence of each type of bug) across different data budgets in the Blockmaze and MsPacman games, respectively. In Figure 5.5, the time to find the first occurrence of bugs is log scale for better visualization. Overall, in both games MGDT and IMPALA agents are faster at finding bugs at 2% fine-tuning data budgets (see Figure 5.5 and 5.6). Table 5.1 reports the statistical results of the performance of the generalist agents in terms of time to find bugs for the Blockmaze and MsPacman games at different fine-tuning data budgets. At a 2% fine-tuning budget, MGDT and IMPALA agents achieved their best performance at finding bugs (i.e., lowest time to find bugs), showing a slight improvement (finding bugs faster) of at least 6% (see Table 5.1) compared to zero-shot and 1% fine-tuning data budget for both games. Specifically on the bug localization task, we collect the MRR. MAP, top@1, top@5, and top@10 metrics. With our fine-tuning setup, some generalist agents perform up to 76% better than the specialist agents in terms of MRR. Figure 5.7 showcases the performance of the baseline and generalist agents in terms of MRR on the Tomcat project. For instance, on the Tomcat project, as shown in Figure 5.7, the MGDT agents show greater performance with the MRR metric, suggesting that MGDT agents are effective in prioritizing the most relevant bugs - enabling developers to resolve bugs more efficiently. Specifically on the (6×6) and (30×20) instances of the PDR task, we collect the makespan performed during task scheduling. For instance, the generalist agent configurations have similar average makespan performance across all three fine-tuning data budgets as shown in Figures 5.8 and 5.9 - which showcase the performance of the baseline and generalist agents in terms of makespan. The consistent makespan performance across varying data budgets implies that the generalist agents maintain reliability, which can be useful in

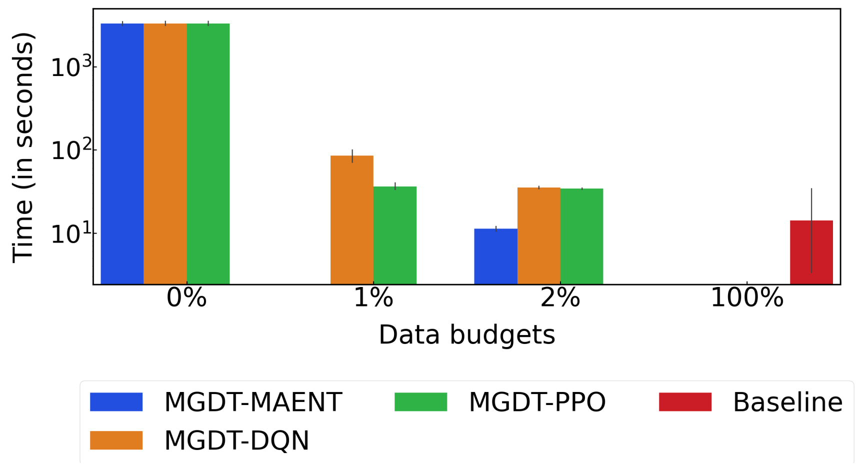


Figure 5.5 Time (log scale in seconds) to find bugs by the MGDT agent configurations and the baseline on the Blockmaze game.

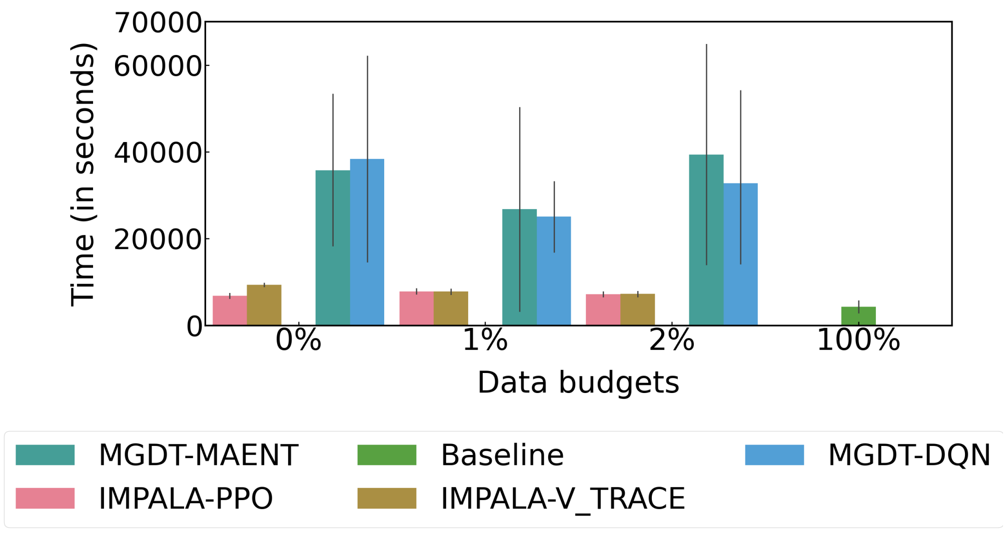


Figure 5.6 Time (in seconds) to find bugs by the MGDT and IMPALA agents configurations and the baseline on the MsPacman game.

Table 5.1 Results of post-hoc test analysis of the time taken to find bugs by the generalist agents on the Blockmaze and MsPacman games for different fine-tuning data budgets (in bold are DRL configurations where the p-value is < 0.05 and have greater performance w.r.t the effect size).

Games	Generalist agents	Data budgets		mean(A)	mean(B)	pval	CLES
		A	B				
MsPacman game	IMPALA	zero-shot	1%	8,095.84	**7,801.87**	0.55	0.53
		zero-shot	2%	8,095.84	**7,227.89**	0.01	0.58
		1%	2%	7,801.87	**7,227.89**	0.14	0.54
	MGDT	zero-shot	1%	37,081.75	**26,199.27**	0.65	0.63
		zero-shot	2%	37,081.75	**36,087.10**	1.0	0.46
		1%	2%	**26,199.27**	36,087.10	0.73	0.42
Blockmaze game	MGDT	zero-shot	1%	3,303.0	**60.90**	0.0	1.0
		zero-shot	2%	3,303.0	**26.88**	0.0	1.0
		1%	2%	60.90	**26.88**	0.01	0.83

mean(A) and mean(B) refer to the time to find bugs at different fine-tuning data budgets.

real-world applications where environmental settings can vary, but consistent and efficient task scheduling is still required. As mentioned in Section 5, there are differences in training budgets for the generalist agents due to the difference in training budgets of the baseline specialist agents. Hence, we provide, in Table 5.2 the average training time of the generalist agent configurations on the studied SE tasks at 1% and 2% fine-tuning data budgets. The (6 $\times$ 6) instance of the PDR scheduling task (i.e., across 6 out of 10 configurations), followed by the Blockmaze game (i.e., across 2 out of 10 configurations), required less time to be fine-tuned by the generalist agents compared to other tasks. This outcome is expected because of the small size of the environment of these tasks - there is less complexity for the generalist agents to learn and adapt to during fine-tuning. The details of the fine-tuning approaches we used are presented in Section 1 and Section 2 for each pre-trained generalist agent. In the next subsections, we discuss in detail the performance of the fine-tuned generalist agents in terms of the metrics mentioned in Section 5.

Finding 1: To leverage generalist agents for the studied SE tasks, we fine-tune them on zero-shot, 1%, and 2% data budgets of specialist agents. At a 2% fine-tuning data budget, the generalist agents demonstrate comparable or superior performance compared to their performance at lesser fine-tuning data budgets.

Table 5.2 Average training time performance of the generalist on the studied SE tasks. In bold are the values of the generalist agent configurations with greater performance per fine-tuning data budgets.

Data budgets	Generalist agents	Playtesting games task		PDR-based scheduling task		Bug localization task					
		Blockmaze game	MsPacman game	(6 x 6) instance	(30 x 20) instance	JDT	AspectJ	SWT	Tomcat	Eclipse	Birt
1%	IMPALA-PPO	432.0	10,139.16	**112.0**	1,368.64	13,093.92	11,238.56	17,717.64	15,419.46	15,738.24	16,522.93
	IMPALA-V_TRACE	432.0	10,244.02	**111.0**	1,311.37	4,110.44	6,284.15	9,256.44	6,834.29	8,435.69	7,811.66
	MGDT-DQN	432.0	4,455.75	**2,283.77**	67,745.03	5,847.19	5,975.08	6,278.84	5,888.29	5,792.02	6,028.91
	MGDT-MAENT	432.0	4,673.86	642.91	17,974.24	638.89	**612.09**	816.53	671.42	676.25	656.78
	MGDT-PPO	**432.0**	5,456.84	7,739.87	158,210.39	3,596.03	3,700.46	3,808.74	3,612.16	3,699.78	3,723.90
2%	IMPALA-PPO	864.0	10,029.11	**109.61**	2,874.49	26,533.92	21,615.80	37,773.88	27,584.77	35,048.25	32,439.0
	IMPALA-V_TRACE	864.0	9,917.51	**120.26**	2,600.39	10,149.58	9,768.44	22,316.18	12,496.27	19,369.63	16,615.61
	MGDT-DQN	864.0	10,489.13	**4,573.57**	143,357.15	11,540.92	11,719.93	12,300.44	11,591.67	11,366.72	11,912.61
	MGDT-MAENT	864.0	8,139.46	1,302.92	41,965.29	1,141.05	**1,086.13**	1,392.42	1,156.17	1,203.99	1,167.77
	MGDT-PPO	**864.0**	12,705.73	15,626.85	277,500.96	7,005.81	7,249.32	7,389.04	7,050.27	7,261.99	7,309.24

Table 5.3 Results of post-hoc tests analysis of the time taken to find bugs by the baseline and the generalist agents on MsPacman game (in bold are DRL configurations where the p-value is < 0.05 and have superior performance w.r.t the effect size).

Data budgets	A	B	mean(A)	mean(B)	pval	CLES
zero-shot	Baseline	IMPALA-PPO	4275.81	6846.81	0.11	0.16
	Baseline	IMPALA-V_TRACE	**4275.81**	9344.86	**0.01**	0.02
	Baseline	MGDT-DQN	4275.81	38364.88	0.26	0.13
	Baseline	MGDT-MAENT	4275.81	35798.62	0.15	0.09
	IMPALA-PPO	IMPALA-V_TRACE	**6846.81**	9344.86	**0.0**	0.32
	IMPALA-PPO	MGDT-DQN	6846.81	38364.88	0.30	0.23
	IMPALA-PPO	MGDT-MAENT	6846.81	35798.62	0.19	0.18
	IMPALA-V_TRACE	MGDT-DQN	9344.86	38364.88	0.36	0.25
	IMPALA-V_TRACE	MGDT-MAENT	9344.86	35798.62	0.24	0.21
	MGDT-DQN	MGDT-MAENT	38364.88	35798.62	1.0	0.53
1%	**Baseline**	IMPALA-PPO	**4275.81**	7807.09	**0.03**	0.08
	Baseline	IMPALA-V_TRACE	**4275.81**	7796.65	**0.03**	0.08
	Baseline	MGDT-DQN	4275.81	25057.12	0.46	0.06
	Baseline	MGDT-MAENT	4275.81	26770.35	0.58	0.22
	IMPALA-PPO	IMPALA-V_TRACE	7807.09	7796.65	1.0	0.50
	IMPALA-PPO	MGDT-DQN	7807.09	25057.12	0.54	0.14
	IMPALA-PPO	MGDT-MAENT	7807.09	26770.35	0.69	0.31
	IMPALA-V_TRACE	MGDT-DQN	7796.65	25057.12	0.54	0.14
	IMPALA-V_TRACE	MGDT-MAENT	7796.65	26770.35	0.69	0.31
	MGDT-DQN	MGDT-MAENT	25057.12	26770.35	1.0	0.47
2%	Baseline	IMPALA-PPO	4275.81	7193.88	0.07	0.13
	Baseline	IMPALA-V_TRACE	4275.81	7262.05	0.06	0.12
	Baseline	MGDT-DQN	4275.81	32777.13	0.26	0.13
	Baseline	MGDT-MAENT	4275.81	39397.07	0.30	0.15
	IMPALA-PPO	IMPALA-V_TRACE	7193.88	7262.05	1.0	0.50
	IMPALA-PPO	MGDT-DQN	7193.88	32777.13	0.33	0.24
	IMPALA-PPO	MGDT-MAENT	7193.88	39397.07	0.36	0.25
	IMPALA-V_TRACE	MGDT-DQN	7262.05	32777.13	0.33	0.24
	IMPALA-V_TRACE	MGDT-MAENT	7262.05	39397.07	0.36	0.25
	MGDT-DQN	MGDT-MAENT	32777.13	39397.07	1.0	0.44

mean(A) and mean(B) refer to the time to find bugs.

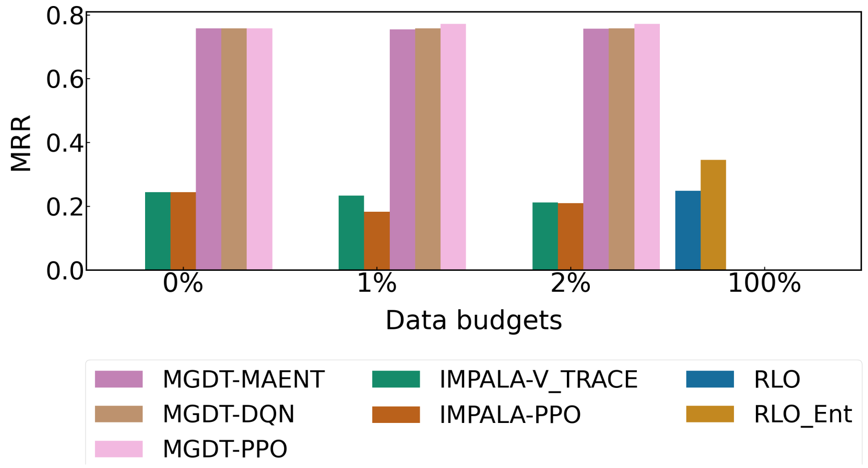


Figure 5.7 MRR performance of the generalist and specialist agents on Tomcat project. RLO and RLO_Ent refer respectively to RLOCATOR without and with entropy.

5.3.2 RQb$_2$: How do different DRL generalist agents perform on SE tasks compared to DRL specialist agents?

In both the Blockmaze and MsPacman games, specialist agents tend to perform better in terms of time to find bugs at lower fine-tuning data budgets, likely due to their optimized, task-specific architectures. Table 5.3 shows the performance of the generalist agents and the baseline specialist on the MsPacman game in terms of time to find bugs. The baseline specialist finds bugs at least 47% faster than the generalist agents. However, at a 2% fine-tuning budget, MGDT agents find bugs 3-8% faster than specialist agents in the Blockmaze game, suggesting that generalist agents can become more effective as the amount of fine-tuning data budgets increases. Across both games, MGDT and IMPALA agents get fine-tuned at least 40% faster than specialist agents at zero-shot and 1% fine-tuning data budgets. With rapid fine-tuning, generalist agents can be leveraged by game developers to test games that require constant and small updates to ensure that the game environment is up to date. On the other hand, specialist agents achieve on average shorter testing times when finding the first occurrences of all types of bugs (i.e., type 1 and 2 for the blockmaze game - type 1, 2, 3 and 4 for the MsPacman game), primarily due to their smaller model sizes (i.e., 68K parameters) versus the larger model size of generalist agents (i.e., 197M parameters), which require more computational resources and result in longer testing times.

Statistical analysis: Table 5.4 reports the statistical results of the performance of the generalist agents and the baseline specialist in terms of time to find bugs for the Blockmaze game. At the 2% fine-tuning data budget, MGDT generalist agents show a slight but not statistically significant improvement over the baseline specialist in terms of time to find bugs

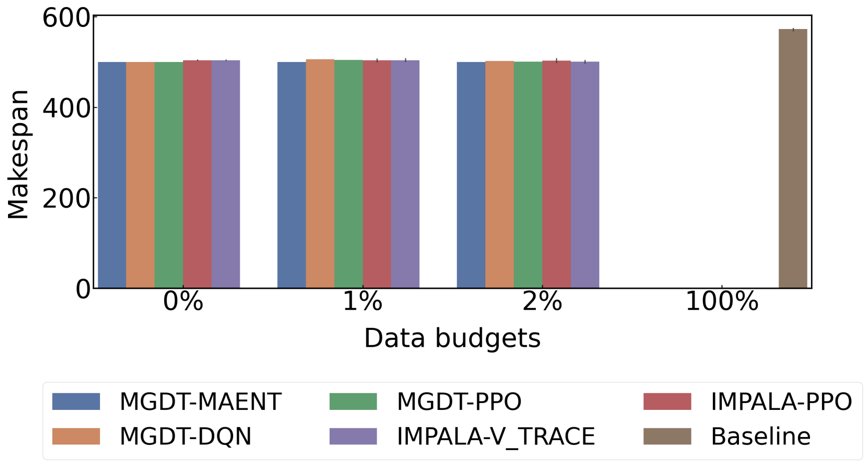


Figure 5.8 Makespan performance by the baseline [6], the MGDT and IMPALA agents configurations on the 6 × 6 instance.

in the Blockmaze game. For training time, MGDT agents significantly outperform the baseline specialist in the MsPacman game at the 1% fine-tuning data budget with a CLES value close to 1 making them suitable for rapid model deployment.

Summary of generalist vs. specialist agents' performance in playtesting games: Different pre-trained DRL generalist and specialist agents show varied performance in playtesting video games. In Blockmaze, MGDT agents show a slight advantage, finding bugs 3-8% faster at 2% fine-tuning data budgets compared to the baseline specialist. MGDT and IMPALA agents achieve significantly lower training times in the Blockmaze game, and MGDT agents' performance in the MsPacman game shows an improvement at 1% fine-tuning data budgets.

In our analysis of the bug localization task, the specialist agents perform better than the generalist agents on average over five runs in terms of top@1, top@10 and MRR by at least 2% on the AspectJ (with the top@10 metric), SWT (with the top@1 metric), JDT (with the MRR metric) and Tomcat (with the top@10 metric) projects. Meanwhile, with all the other metrics (MAP, top@5, training and testing time) at least one generalist agent achieves on average comparable or greater performance by at least 4% improvement over the performance of the specialist agents. For instance, at a 2% fine-tuning data budget, IMPALA takes 83% less time to complete its evaluation on the AspectJ project, showing a better adaptation to the task with more fine-tuning.

Statistical analysis: The performance difference between the generalist (i.e., V_TRACE and MAENT configurations) and specialist agents is significant across software projects and fine-tuning budgets with a large effect size close to 100 percent. We provide the results of the

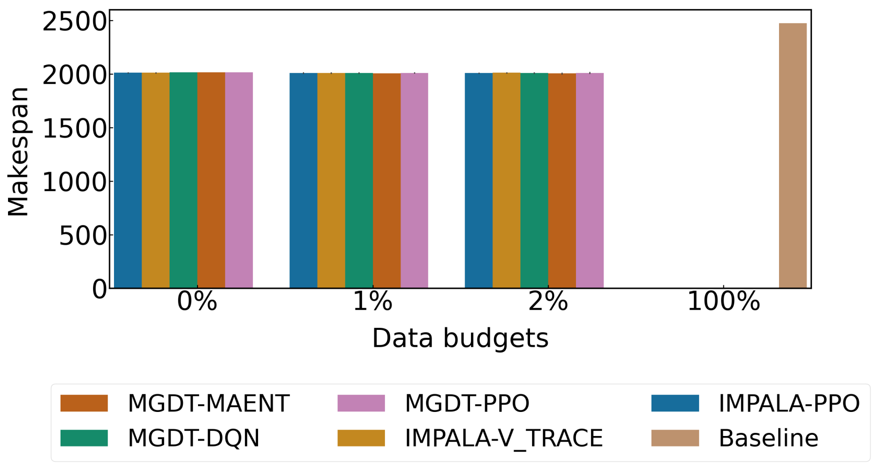


Figure 5.9 Makespan performance by the baseline [6], the MGDT and IMPALA agents configurations on the 30 × 20 instance.

post hoc test analysis across all software projects, w.r.t our evaluation metrics (see Section 5) in our replication package [184]. Therefore, fine-tuning the generalist agents with source code and bug reports dataset makes it suitable for bug localization.

Summary of generalist vs. specialist agents' performance in bug localization task: Some generalist agents show significantly greater performance than the specialist agents across all software projects with 5 out of 8 metrics.

In our analysis of the task-based scheduling across the 6 × 6 and 30 × 20 instances, we observed performance differences between generalist agents and the baseline specialist [6] across various metrics. In terms of makespan performance, both IMPALA and MGDT generalist agents outperformed the baseline on average. For the 6 × 6 instance, IMPALA agents reduced makespan by approximately $12.7 - 13.3\%$, and MGDT agents achieved reductions of about $12.3 - 13.5\%$ across all fine-tuning data budgets. Similarly, in the 30 × 20 instance, IMPALA agents reduce the makespan by approximately $20.6 - 20.8\%$, with MGDT agents achieving reductions of about $20.3 - 20.8\%$ over the baseline specialist. Generalist agents are designed to adapt to a wide range of tasks, making them highly scalable [7,8]. In the larger 30 × 20 instance of the PDR task, this adaptability becomes even more important. The generalist agents' ability to leverage their pre-trained knowledge and adapt to the specific requirements of the scheduling task allows them to achieve greater reductions in makespan in the 30 × 20 instance compared to the 6 × 6 instance of the PDR task. In terms of training time, IMPALA agents take on average at least 10% less time compared to the baseline specialist in both instances across all fine-tuning data budgets. Moreover, IMPALA and MGDT agents earned on average 37% more rewards compared to the baseline specialist in

Table 5.4 Results of post-hoc test analysis of the time taken to find bugs by the baseline specialist and MGDT generalist agents on the Blockmaze game for different fine-tuning data budgets (in bold are DRL configurations where the p-value is < 0.05 and have greater performance w.r.t the effect size).

Data budgets	A	B	mean(A)	mean(B)	pval	CLES
zero-shot	**Baseline**	MGDT-DQN	**14.21**	3303.0	**0.0**	0
	Baseline	MGDT-MAENT	**14.21**	3303.0	**0.0**	0.0
	Baseline	MGDT-PPO	**14.21**	3303.0	**0.0**	0.0
	MGDT-DQN	MGDT-MAENT	3303.0	3303.0	1.0	0.50
	MGDT-DQN	MGDT-PPO	3303.0	3303.0	1.0	0.50
	MGDT-MAENT	MGDT-PPO	3303.0	3303.0	1.0	0.50
1%	**Baseline**	MGDT-DQN	**14.21**	85.48	**0.0**	0.01
	Baseline	MGDT-PPO	14.21	36.32	0.18	0.17
	MGDT-DQN	**MGDT-PPO**	85.48	36.32	**0.01**	0.99
2%	Baseline	MGDT-DQN	14.21	35.19	0.29	0.18
	Baseline	MGDT-MAENT	14.21	11.32	0.99	0.55
	Baseline	MGDT-PPO	14.21	34.12	0.33	0.19
	MGDT-DQN	**MGDT-MAENT**	35.19	**11.32**	0.0	1.0
	MGDT-DQN	MGDT-PPO	35.19	34.12	0.59	0.72
	MGDT-MAENT	MGDT-PPO	11.32	34.12	0.0	0.0

mean(A) and mean(B) refer to the time to find bugs.

both instances. However, both IMPALA and MGDT agents exhibited longer testing times compared to the baseline specialist across all fine-tuning data budgets due to the size of their models' parameters.

Statistical analysis: Tables 5.5, 5.6, and 5.7 show the results of the post-hoc test analysis for the generalist agents and the baseline specialist on the 6 $\times$ 6 instance of the PDR task in terms of the cumulative reward earned, training time and makespan. Regarding the 30 $\times$ 20 instance, since we observe similar statistical results as the 6 $\times$ 6 instance, we do not report its results (post-hoc test analysis is available in our replication package [184]). In terms of makespan and cumulative reward performance, both generalist agent configurations significantly outperformed the baseline across all fine-tuning data budgets, with CLES values close to 0 in both instances. The CLES values indicate that with a 0% chance, the baseline will have a higher makespan and lower cumulative reward earned than the generalist agents.

Summary of generalist vs. specialist agents' performance on the PDR task: Different DRL generalist agents show notable advantages over the baseline specialist in the PDR task across studied instances. In both the 6 $\times$ 6 and 30 $\times$ 20 instances, IMPALA and MGDT generalist agents consistently reduce makespan by approximately $12.3 - 13.5\%$ and $20.3 - 20.8\%$, respectively, compared to the baseline specialist across all fine-tuning data

Table 5.5 Results of post-hoc tests analysis of makespan performance by the baseline and generalist agents on the 6×6 instance (in bold are DRL configurations where p-value is $<$ 0.05 and have greater performance w.r.t the effect size).

Data budgets	A	B	mean(A)	mean(B)	pval	CLES
zero-shot	Baseline	**IMPALA-PPO**	572.26	**504.0**	**0.0**	**1.0**
	Baseline	**IMPALA-V_TRACE**	572.26	**504.0**	**0.0**	**1.0**
	Baseline	**MGDT-DQN**	572.26	**499.82**	**0.0**	**1.0**
	Baseline	**MGDT-MAENT**	572.26	**499.82**	**0.0**	**1.0**
	Baseline	**MGDT-PPO**	572.26	**499.82**	**0.0**	**1.0**
	IMPALA-PPO	IMPALA-V_TRACE	504.18	504.18	1.0	0.50
	IMPALA-PPO	**MGDT-DQN**	504.18	**499.82**	**0.0**	**1.0**
	IMPALA-PPO	**MGDT-MAENT**	504.18	**499.82**	**0.0**	**1.0**
	IMPALA-PPO	**MGDT-PPO**	504.18	**499.82**	**0.0**	**1.0**
	IMPALA-V_TRACE	**MGDT-DQN**	504.18	**499.82**	**0.0**	**1.0**
	IMPALA-V_TRACE	**MGDT-MAENT**	504.18	**499.82**	**0.0**	**1.0**
	IMPALA-V_TRACE	**MGDT-PPO**	504.18	**499.82**	**0.0**	**1.0**
	MGDT-DQN	MGDT-MAENT	499.82	499.82	1.0	0.50
	MGDT-DQN	MGDT-PPO	499.82	499.82	1.0	0.50
	MGDT-MAENT	MGDT-PPO	499.82	499.82	1.0	0.50
1%	Baseline	**IMPALA-PPO**	572.26	**503.92**	**0.0**	**1.0**
	Baseline	**IMPALA-V_TRACE**	572.26	**504.68**	**0.0**	**1.0**
	Baseline	**MGDT-DQN**	572.26	**505.99**	**0.0**	**1.0**
	Baseline	**MGDT-MAENT**	572.26	**500.11**	**0.0**	**1.0**
	Baseline	**MGDT-PPO**	572.26	**504.74**	**0.0**	**1.0**
	IMPALA-PPO	IMPALA-V_TRACE	503.92	504.68	1.0	0.48
	IMPALA-PPO	MGDT-DQN	503.92	505.99	0.81	0.20
	IMPALA-PPO	MGDT-MAENT	503.92	500.11	0.24	1.0
	IMPALA-PPO	MGDT-PPO	503.92	504.74	1.0	0.40
	IMPALA-V_TRACE	MGDT-DQN	504.68	505.99	0.90	0.20
	IMPALA-V_TRACE	MGDT-MAENT	504.68	500.11	0.16	0.80
	IMPALA-V_TRACE	MGDT-PPO	504.68	504.74	1.0	0.40
	MGDT-DQN	**MGDT-MAENT**	505.99	**500.11**	**0.02**	**1.0**
	MGDT-DQN	MGDT-PPO	505.99	504.74	0.97	1.0
	MGDT-MAENT	MGDT-PPO	500.11	504.74	0.10	0.0
2%	Baseline	**IMPALA-PPO**	572.26	**503.06**	**0.0**	**1.0**
	Baseline	**IMPALA-V_TRACE**	572.26	**501.19**	**0.0**	**1.0**
	Baseline	**MGDT-DQN**	572.26	**502.14**	**0.0**	**1.0**
	Baseline	**MGDT-MAENT**	572.26	**500.11**	**0.0**	**1.0**
	Baseline	**MGDT-PPO**	572.26	**500.69**	**0.0**	**1.0**
	IMPALA-PPO	IMPALA-V_TRACE	503.06	501.19	0.94	0.56
	IMPALA-PPO	MGDT-DQN	503.06	502.14	1.0	0.40
	IMPALA-PPO	MGDT-MAENT	503.06	500.11	0.70	0.60
	IMPALA-PPO	MGDT-PPO	503.06	500.69	0.85	0.60
	IMPALA-V_TRACE	MGDT-DQN	501.19	502.14	1.0	0.60
	IMPALA-V_TRACE	MGDT-MAENT	501.19	500.11	0.99	0.60
	IMPALA-V_TRACE	MGDT-PPO	501.19	500.69	1.0	0.60
	MGDT-DQN	MGDT-MAENT	502.14	500.11	0.91	1.0
	MGDT-DQN	MGDT-PPO	502.14	500.69	0.98	1.0
	MGDT-MAENT	MGDT-PPO	499.82	500.69	1.0	0.0

mean(A) and mean(B) refer to makespan values.

Table 5.6 Results of post-hoc tests analysis of training time performance by the baseline and generalist agents on PDR task on the 6×6 instance(in bold are DRL configurations where p-value is < 0.05 and have greater performance w.r.t the effect size).

Data budgets	A	B	mean(A)	mean(B)	pval	CLES
1%	Baseline	**IMPALA-PPO**	3962.78	**112.0**	**0.03**	**1.0**
	Baseline	**IMPALA-V_TRACE**	3962.78	**110.17**	**0.03**	**1.0**
	Baseline	MGDT-DQN	3962.78	2283.77	0.31	0.80
	Baseline	**MGDT-MAENT**	3962.78	**642.91**	**0.05**	**1.0**
	Baseline	MGDT-PPO	**3962.78**	7739.87	**0.03**	**0.0**
	IMPALA-PPO	IMPALA-V_TRACE	112.0	110.17	0.97	0.56
	IMPALA-PPO	MGDT-DQN	**112.0**	2283.77	**0.0**	**0.0**
	IMPALA-PPO	MGDT-MAENT	**112.0**	642.91	**0.0**	**0.0**
	IMPALA-PPO	MGDT-PPO	**112.0**	7739.87	**0.0**	**0.0**
	IMPALA-V_TRACE	MGDT-DQN	**110.17**	2283.77	**0.0**	**0.0**
	IMPALA-V_TRACE	MGDT-MAENT	**110.17**	642.91	**0.0**	**0.0**
	IMPALA-V_TRACE	MGDT-PPO	**110.17**	7739.87	**0.0**	**0.0**
	MGDT-DQN	**MGDT-MAENT**	2283.77	**642.91**	**0.0**	**1.0**
	MGDT-DQN	MGDT-PPO	**2283.77**	7739.87	**0.0**	**0.0**
	MGDT-MAENT	MGDT-PPO	**642.91**	7739.87	**0.0**	**0.0**
2%	Baseline	**IMPALA-PPO**	3962.78	**109.61**	**0.03**	**1.0**
	Baseline	**IMPALA-V_TRACE**	3962.78	**120.26**	**0.03**	**1.0**
	Baseline	MGDT-DQN	3962.78	4573.57	0.93	0.52
	Baseline	MGDT-MAENT	3962.78	1302.92	0.09	0.80
	Baseline	MGDT-PPO	**3962.78**	15626.85	**0.0**	**0.0**
	IMPALA-PPO	IMPALA-V_TRACE	109.61	120.26	0.21	0.08
	IMPALA-PPO	MGDT-DQN	**109.61**	4573.57	**0.0**	**0.0**
	IMPALA-PPO	MGDT-MAENT	**109.61**	1302.92	**0.0**	**0.0**
	IMPALA-PPO	MGDT-PPO	**109.61**	15626.85	**0.0**	**0.0**
	IMPALA-V_TRACE	MGDT-DQN	**120.26**	4573.57	**0.0**	**0.0**
	IMPALA-V_TRACE	MGDT-MAENT	**120.26**	1302.92	**0.0**	**0.0**
	IMPALA-V_TRACE	MGDT-PPO	**120.26**	15626.85	**0.0**	**0.0**
	MGDT-DQN	**MGDT-MAENT**	4573.57	**1302.92**	**0.0**	**1.0**
	MGDT-DQN	MGDT-PPO	**4573.57**	15626.85	**0.0**	**0.0**
	MGDT-MAENT	MGDT-PPO	**1302.92**	15626.85	**0.0**	**0.0**

mean(A) and mean(B) refer to training time values.

budgets. They also achieve at least 10% faster training times and earn 37% more rewards on average. However, generalist agents exhibit longer testing times compared to the baseline specialist. Statistical analysis confirms significant performance improvements of generalist agents over the baseline specialist in terms of makespan and cumulative rewards across all fine-tuning data budgets for both instances.

Table 5.7 Results of post-hoc tests analysis of cumulative reward performance by the baseline and the fine-tuned generalist agents on the PDR task on the 6×6 instance(in bold are DRL configurations where p-value is < 0.05 and have greater performance w.r.t the effect size).

Data budgets	A	B	mean(A)	mean(B)	pval	CLES
zero-shot	Baseline	**IMPALA-PPO**	-572.66	**-391.52**	**0.0**	**0.0**
	Baseline	**IMPALA-V_TRACE**	-572.67	**-391.52**	**0.0**	**0.0**
	Baseline	**MGDT-DQN**	-572.67	**-391.57**	**0.0**	**0.0**
	Baseline	**MGDT-MAENT**	-572.67	**-391.57**	**0.0**	**0.0**
	Baseline	**MGDT-PPO**	-572.67	**-391.57**	**0.0**	**0.0**
	IMPALA-PPO	IMPALA-V_TRACE	-391.52	-391.52	1.0	0.50
	IMPALA-PPO	MGDT-DQN	-391.52	-391.47	1.0	0.60
	IMPALA-PPO	MGDT-MAENT	-391.52	-391.47	1.0	0.60
	IMPALA-PPO	MGDT-PPO	-391.52	-391.47	1.0	0.60
	IMPALA-V_TRACE	MGDT-DQN	-391.52	-391.47	1.0	0.60
	IMPALA-V_TRACE	MGDT-MAENT	-391.52	-391.47	1.0	0.60
	IMPALA-V_TRACE	MGDT-PPO	-391.52	-391.47	1.0	0.60
	MGDT-DQN	MGDT-MAENT	-391.47	-391.47	1.0	0.50
	MGDT-DQN	MGDT-PPO	-391.47	-391.47	1.0	0.50
	MGDT-MAENT	MGDT-PPO	-391.47	-391.47	1.0	0.50
1%	Baseline	**IMPALA-PPO**	-572.67	**-391.80**	**0.0**	**0.0**
	Baseline	**IMPALA-V_TRACE**	-572.67	**-391.32**	**0.0**	**0.0**
	Baseline	**MGDT-DQN**	-572.67	**-390.68**	**0.0**	**0.0**
	Baseline	**MGDT-MAENT**	-572.67	**-391.47**	**0.0**	**0.0**
	Baseline	**MGDT-PPO**	-572.67	**-393.07**	**0.0**	**0.0**
	IMPALA-PPO	IMPALA-V_TRACE	-391.80	-391.32	1.0	0.20
	IMPALA-PPO	MGDT-DQN	-391.80	-390.68	0.85	0.0
	IMPALA-PPO	MGDT-MAENT	-391.80	-391.47	1.0	0.0
	IMPALA-PPO	MGDT-PPO	-391.80	-393.07	0.77	1.0
	IMPALA-V_TRACE	MGDT-DQN	-391.32	-390.68	0.98	0.0
	IMPALA-V_TRACE	MGDT-MAENT	-391.32	-391.47	1.0	0.80
	IMPALA-V_TRACE	MGDT-PPO	-391.32	-393.07	0.47	1.0
	MGDT-DQN	MGDT-MAENT	-390.68	-391.47	0.96	1.0
	MGDT-DQN	MGDT-PPO	-390.68	-393.07	0.17	1.0
	MGDT-MAENT	MGDT-PPO	-391.47	-393.07	0.56	1.0
2%	Baseline	**IMPALA-PPO**	-572.67	**-391.51**	**0.0**	**0.0**
	Baseline	**IMPALA-V_TRACE**	-572.67	**-391.83**	**0.0**	**0.0**
	Baseline	**MGDT-DQN**	-572.67	**-391.72**	**0.0**	**0.0**
	Baseline	**MGDT-MAENT**	-572.67	**-391.47**	**0.0**	**0.0**
	Baseline	**MGDT-PPO**	-572.67	**-393.16**	**0.0**	**0.0**
	IMPALA-PPO	IMPALA-V_TRACE	-391.51	-391.83	1.0	0.60
	IMPALA-PPO	MGDT-DQN	-391.51	-391.72	1.0	0.70
	IMPALA-PPO	MGDT-MAENT	-391.51	-391.47	1.0	0.40
	IMPALA-PPO	MGDT-PPO	-391.51	-393.16	0.54	1.0
	IMPALA-V_TRACE	MGDT-DQN	-391.83	-391.72	1.0	0.60
	IMPALA-V_TRACE	MGDT-MAENT	-391.83	-391.47	1.0	0.40
	IMPALA-V_TRACE	MGDT-PPO	-391.83	-393.16	0.74	1.0
	MGDT-DQN	MGDT-MAENT	-391.72	-391.47	1.0	0.0
	MGDT-DQN	MGDT-PPO	-391.72	-393.16	0.68	1.0
	MGDT-MAENT	MGDT-PPO	-391.47	-393.16	0.52	1.0

mean(A) and mean(B) refer to cumulative reward values.

Finding 2: Compared to baseline specialists, the MGDT generalist agent finds bugs 3-8% faster at 2% fine-tuning data budgets in the Blockmaze game, while also achieving the most significant reductions in makespan across all fine-tuning data budgets in the studied scheduling instances. Additionally, in the bug localization task, the generalist agents show comparable or greater performance compared to the baseline specialists with 5 out of 8 metrics. However, in the MsPacman game, the baseline specialist finds bugs at least 47% faster than the generalist agents.

5.3.3 RQb$_3$: How do different model-free DRL algorithms affect the performance of pre-trained generalist agents on SE tasks?

In the Blockmaze game, MGDT-DQN took similar or less time on average to find the first occurrence of type 1 bugs compared to MGDT-MAENT and MGDT-PPO across all fine-tuning data budgets. However, MGDT-MAENT performed better than the other MGDT configurations at the 2% fine-tuning data budget by taking at least three times less to find the first occurrence of type 2 bugs. DQN algorithm creates a mapping between states and actions, optimizing the actions that lead to the highest cumulative rewards. This helps the MGDT-DQN configuration to identify states associated with type 1 bugs, which are easier to find and relate to exploration capabilities. MGDT-MAENT's ability to balance between exploration and exploitation enables it to identify complex bugs at invalid locations (type 2). IMPALA agents did not find any bugs after fine-tuning in the Blockmaze game but achieved the best performance in the MsPacman game across all fine-tuning data budgets, continuously learning from new experiences during evaluation and adapting their behavior to improve performance [8]. This highlights the advantage of using IMPALA agents for games with mechanics that evolve over time.

Statistical analysis: Tables 5.8 and 5.9 report the results of post-hoc tests analysis of cumulative reward performance by each model-free algorithm on the Blockmaze and MsPacman games respectively. In terms of cumulative reward earned, MGDT-DQN and MGDT-MAENT significantly performed better than IMPALA agents in the 1% and 2% fine-tuning data budgets, as reported in Tables 5.8 and 5.9 across both games. Table 5.10 reports the results of post-hoc tests analysis of testing time performance by each model-free algorithm on the MsPacman game. IMPALA agents demonstrated greater testing time performance across all fine-tuning data budgets, with large effect sizes in the MsPacman game, as detailed in Table 5.10. MGDT agents' pre-training configurations and balanced exploration-exploitation policies lead to better rewards earned on both games, while the scalable architecture of IM-

Table 5.8 Results of post-hoc tests analysis of cumulative reward performance by each model-free algorithm on the Blockmaze game (in bold are DRL configurations where p-value is < 0.05 and have greater performance w.r.t the effect size).

Data budgets	A	B	mean(A)	mean(B)	pval	CLES
zero-shot	IMPALA-PPO	**MGDT-DQN**	-400.0	**-149.43**	**0.0**	**0.0**
	IMPALA-PPO	**MGDT-MAENT**	-400.0	**-149.43**	**0.0**	**0.0**
	IMPALA-PPO	**MGDT-PPO**	-400.0	**-149.43**	**0.0**	**0.0**
	IMPALA-V_TRACE	**MGDT-DQN**	-400.0	**-149.43**	**0.0**	**0.0**
	IMPALA-V_TRACE	**MGDT-MAENT**	-400.0	**-149.43**	**0.0**	**0.0**
	IMPALA-V_TRACE	**MGDT-PPO**	-400.0	**-149.43**	**0.0**	**0.0**
1%	IMPALA-PPO	**IMPALA-V_TRACE**	-400.0	**-396.04**	**1.0**	**0.20**
	IMPALA-PPO	**MGDT-DQN**	-400.0	**-148.84**	**0.0**	**0.0**
	IMPALA-PPO	**MGDT-MAENT**	-400.0	**-141.53**	**0.0**	**0.0**
	IMPALA-PPO	**MGDT-PPO**	-400.0	**-122.44**	**0.0**	**0.0**
	IMPALA-V_TRACE	**MGDT-DQN**	-396.04	**-148.84**	**0.0**	**0.0**
	IMPALA-V_TRACE	**MGDT-MAENT**	-396.04	**-141.53**	**0.0**	**0.0**
	IMPALA-V_TRACE	**MGDT-PPO**	-396.04	**-122.44**	**0.0**	**0.0**
	MGDT-DQN	MGDT-MAENT	-148.84	-141.53	1.0	0.20
	MGDT-DQN	MGDT-PPO	-148.84	-122.44	0.82	0.0
	MGDT-MAENT	MGDT-PPO	-141.53	-122.44	0.94	0.80
2%	IMPALA-PPO	**IMPALA-V_TRACE**	-400.0	**-395.54**	**0.03**	**0.13**
	IMPALA-PPO	**MGDT-DQN**	-400.0	**-148.72**	**0.0**	**0.0**
	IMPALA-PPO	**MGDT-MAENT**	-400.0	**-297.01**	**0.0**	**0.0**
	IMPALA-PPO	**MGDT-PPO**	-400.0	**-122.39**	**0.0**	**0.0**
	IMPALA-V_TRACE	**MGDT-DQN**	-395.54	**-148.72**	**0.0**	**0.0**
	IMPALA-V_TRACE	**MGDT-MAENT**	-395.54	**-297.01**	**0.0**	**0.0**
	IMPALA-V_TRACE	**MGDT-PPO**	-395.54	**-122.39**	**0.0**	**0.0**
	MGDT-DQN	MGDT-MAENT	**-148.72**	-297.01	**0.0**	**1.0**
	MGDT-DQN	**MGDT-PPO**	-148.72	**-122.39**	**0.0**	**0.0**
	MGDT-MAENT	**MGDT-PPO**	-297.01	**-122.39**	**0.0**	**0.0**

mean(A) and mean(B) refer to the cumulative reward values.

PALA agents provides an advantage in testing time performance on the MsPacman game across all fine-tuning data budgets.

Summary of generalist agents' performance in Playtesting in games: MGDT agents achieve the lowest training times in MsPacman game and find bugs the fastest in Blockmaze, notably MGDT-MAENT on a 2% fine-tuning budget. IMPALA agents achieve the best testing times in the MsPacman game.

Regarding the bug localization task, IMPALA agent configurations take at least 57% less time on average to complete their evaluation compared to MGDT agent configurations in terms of testing time on the 1% and 2% fine-tuning data budgets across all software projects. This is because IMPALA's architecture is optimized for rapid adaptation and efficient processing, making it more effective at utilizing fine-tuning data to reduce testing time significantly.

Table 5.9 Results of post-hoc tests analysis of cumulative reward performance by each model-free algorithm on the MsPacman game (in bold are DRL configurations where p-value is $<$ 0.05 and have greater performance w.r.t the effect size).

Data budgets	A	B	mean(A)	mean(B)	pval	CLES
zero-shot	IMPALA-PPO	**IMPALA-V_TRACE**	88.43	**196.03**	**0.02**	**0.0**
	IMPALA-PPO	**MGDT-DQN**	88.43	**151.56**	**0.0**	**0.0**
	IMPALA-PPO	**MGDT-MAENT**	88.43	**192.81**	**0.03**	**0.0**
	IMPALA-PPO	MGDT-PPO	**88.43**	45.0	**0.0**	**1.0**
	IMPALA-V_TRACE	MGDT-DQN	196.03	151.56	0.26	1.0
	IMPALA-V_TRACE	MGDT-MAENT	196.03	192.81	1.0	0.52
	IMPALA-V_TRACE	MGDT-PPO	**196.03**	45.0	**0.01**	**1.0**
	MGDT-DQN	MGDT-MAENT	151.56	192.81	0.42	0.32
	MGDT-DQN	MGDT-PPO	**151.56**	45.0	**0.0**	**1.0**
	MGDT-MAENT	MGDT-PPO	**192.81**	45.0	**0.01**	**1.0**
1%	IMPALA-PPO	IMPALA-V_TRACE	88.38	87.85	0.93	0.68
	IMPALA-PPO	**MGDT-DQN**	88.38	**150.93**	**0.0**	**0.0**
	IMPALA-PPO	**MGDT-MAENT**	88.38	**169.09**	**0.02**	**0.0**
	IMPALA-PPO	MGDT-PPO	**88.38**	45.0	**0.0**	**1.0**
	IMPALA-V_TRACE	**MGDT-DQN**	87.85	**150.93**	**0.0**	**0.0**
	IMPALA-V_TRACE	**MGDT-MAENT**	87.85	**169.09**	**0.02**	**0.0**
	IMPALA-V_TRACE	MGDT-PPO	**87.85**	45.0	**0.0**	**1.0**
	MGDT-DQN	MGDT-MAENT	150.93	169.09	0.74	0.44
	MGDT-DQN	MGDT-PPO	**150.93**	45.0	**0.0**	**1.0**
	MGDT-MAENT	MGDT-PPO	**169.09**	45.0	**0.01**	**1.0**
2%	**IMPALA-PPO**	IMPALA-V_TRACE	**181.82**	89.02	**0.0**	**1.0**
	IMPALA-PPO	MGDT-DQN	181.82	150.07	0.09	1.0
	IMPALA-PPO	MGDT-MAENT	181.82	199.65	0.93	0.44
	IMPALA-PPO	MGDT-PPO	**181.82**	45.0	**0.0**	**1.0**
	IMPALA-V_TRACE	**MGDT-DQN**	89.02	**150.07**	**0.0**	**0.0**
	IMPALA-V_TRACE	**MGDT-MAENT**	89.02	**199.65**	**0.03**	**0.0**
	IMPALA-V_TRACE	MGDT-PPO	**89.02**	45.0	**0.0**	**1.0**
	MGDT-DQN	MGDT-MAENT	150.07	199.65	0.31	0.20
	MGDT-DQN	MGDT-PPO	**150.07**	45.0	**0.0**	**1.0**
	MGDT-MAENT	MGDT-PPO	**199.65**	45.0	**0.01**	**1.0**

mean(A) and mean(B) refer to cumulative reward values.

Table 5.10 Results of post-hoc tests analysis of testing time performance of the baseline and each model-free algorithm on MsPacman game (in bold are DRL configurations where the p-value is < 0.05 and have superior performance w.r.t the effect size).

Data budgets	A	B	mean(A)	mean(B)	pval	CLES
zero-shot	Baseline	IMPALA-PPO	25,735.54	29,623.51	0.43	0.18
	Baseline	IMPALA-V_TRACE	25,735.54	38,893.19	0.07	0.03
	Baseline	MGDT-DQN	**25,735.54**	389,826.64	**0.0**	**0.0**
	Baseline	MGDT-MAENT	**25,735.54**	350,826.03	**0.0**	**0.0**
	Baseline	MGDT-PPO	**25,735.54**	677,430.43	**0.0**	**0.0**
	IMPALA-PPO	IMPALA-V_TRACE	29,623.51	38,893.19	0.22	0.12
	IMPALA-PPO	MGDT-DQN	**29,623.51**	389,826.64	**0.0**	**0.0**
	IMPALA-PPO	MGDT-MAENT	**29,623.51**	350,826.03	**0.0**	**0.0**
	IMPALA-PPO	MGDT-PPO	**29,623.51**	677,430.43	**0.0**	**0.0**
	IMPALA-V_TRACE	MGDT-DQN	**38,893.19**	389,826.64	**0.0**	**0.0**
	IMPALA-V_TRACE	MGDT-MAENT	**38,893.19**	350,826.03	**0.0**	**0.0**
	IMPALA-V_TRACE	MGDT-PPO	**38,893.19**	677,430.43	**0.0**	**0.0**
	MGDT-DQN	MGDT-MAENT	389,826.64	350,826.03	0.07	0.94
	MGDT-DQN	MGDT-PPO	**389,826.64**	677,430.43	**0.0**	**0.0**
	MGDT-MAENT	MGDT-PPO	**350,826.03**	677,430.43	**0.0**	**0.0**
1%	**Baseline**	IMPALA-PPO	**25,735.54**	30,141.35	**0.03**	**0.02**
	Baseline	IMPALA-V_TRACE	**25,735.54**	30,089.51	**0.02**	**0.02**
	Baseline	MGDT-DQN	**25,735.54**	350,428.81	**0.0**	**0.0**
	Baseline	MGDT-MAENT	**25,735.54**	423,350.88	**0.0**	**0.0**
	Baseline	MGDT-PPO	**25,735.54**	573,670.40	**0.0**	**0.0**
	IMPALA-PPO	IMPALA-V_TRACE	30,141.35	30,089.51	1.0	0.51
	IMPALA-PPO	MGDT-DQN	**30,141.35**	350,428.81	**0.0**	**0.0**
	IMPALA-PPO	MGDT-MAENT	**30,141.35**	423,350.88	**0.0**	**0.0**
	IMPALA-PPO	MGDT-PPO	**30,141.35**	573,670.40	**0.0**	**0.0**
	IMPALA-V_TRACE	MGDT-DQN	**30,089.51**	350,428.81	**0.0**	**0.0**
	IMPALA-V_TRACE	MGDT-MAENT	**30,089.51**	423,350.88	**0.0**	**0.0**
	IMPALA-V_TRACE	MGDT-PPO	**30,089.51**	573,670.40	**0.0**	**0.0**
	MGDT-DQN	MGDT-MAENT	350,428.81	423,350.88	0.44	0.19
	MGDT-DQN	MGDT-PPO	**350,428.81**	573,670.40	**0.0**	**0.0**
	MGDT-MAENT	MGDT-PPO	**423,350.88**	573,670.40	**0.05**	**0.02**
2%	**Baseline**	IMPALA-PPO	**25,735.54**	32,387.04	**0.02**	**0.01**
	Baseline	IMPALA-V_TRACE	25,735.54	32,143.17	0.15	0.08
	Baseline	MGDT-DQN	**25,735.54**	365,766.97	**0.0**	**0.0**
	Baseline	MGDT-MAENT	**25,735.54**	391,992.99	**0.0**	**0.0**
	Baseline	MGDT-PPO	**25,735.54**	598,646.42	**0.0**	**0.0**
	IMPALA-PPO	IMPALA-V_TRACE	32,387.04	32,143.17	1.0	0.52
	IMPALA-PPO	MGDT-DQN	**32,387.04**	365,766.97	**0.0**	**0.0**
	IMPALA-PPO	MGDT-MAENT	**32,387.04**	391,992.99	**0.0**	**0.0**
	IMPALA-PPO	MGDT-PPO	**32,387.04**	598,646.42	**0.0**	**0.0**
	IMPALA-V_TRACE	MGDT-DQN	**32,143.17**	365,766.97	**0.0**	**0.0**
	IMPALA-V_TRACE	MGDT-MAENT	**32,143.17**	391,992.99	**0.0**	**0.0**
	IMPALA-V_TRACE	MGDT-PPO	**32,143.17**	598,646.42	**0.0**	**0.0**
	MGDT-DQN	MGDT-MAENT	365,766.97	391,992.99	0.93	0.34
	MGDT-DQN	MGDT-PPO	**365,766.97**	598,646.42	**0.0**	**0.0**
	MGDT-MAENT	MGDT-PPO	**391,992.99**	598,646.42	**0.01**	**0.0**

mean(A) and mean(B) refer to testing time values.

Nevertheless, on all the other metrics, MGDT agents achieve comparable or better performance due to their design that balances generalization and specialization. Their ability to achieve higher top@1 scores can help developers save more effort in analyzing buggy source files.

Statistical analysis: Among the generalist agents (i.e., between MGDT agents and between IMPALA agents), the performance difference between the generalist agents' configurations is not significant except with the training time metric where MGDT-MAENT and IMPALA-V_TRACE achieve significantly greater performance. This suggests that the MAENT and V_TRACE algorithms optimize their policies more effectively within the MGDT and IMPALA agents, respectively. This optimization likely leads to more efficient training processes, allowing these configurations to achieve faster training times compared to other generalist agent configurations.

Summary of generalist agents' performance in the bug localization task: IMPALA agent configurations take at least 57% less time on average for testing compared to MGDT agents. However, MGDT agents achieve comparable or better performance on other metrics, such as the top@1 metric, by balancing generalization and specialization.

Regarding the PDR task, both IMPALA and MGDT agent configurations achieve similar performance on average in terms of makespan and cumulative reward earned across all fine-tuning data budgets for both the 6×6 and 30×20 instances. For the 6×6 instance, IMPALA and MGDT agents achieve similar performance in terms of testing time. However, for the 30×20 instance, IMPALA-V_TRACE and IMPALA-PPO achieve the longest testing times suggesting that their architecture requires more processing effort to manage larger state and action spaces. MGDT-PPO achieves the longest training times across all fine-tuning data budgets for both instances because the PPO algorithm involves multiple policy updates during training.

Statistical analysis: At the zero-shot fine-tuning data budget, the MGDT agents perform better than IMPALA agents with CLES values equal to 1 in terms of makespan performance in the 6×6 instance (see Table 5.5) suggesting that MGDT pre-training setup is better suited for small task scheduling activities. In terms of training time the IMPALA configurations with their scalable architecture perform significantly better than MGDT agents across all fine-tuning data budgets on both studies instances. In terms of training time, the IMPALA agents perform significantly better than MGDT agents across all fine-tuning data budgets.

Summary of generalist agents' performance in the PDR task: For the 6×6 instance, MGDT agents outperform IMPALA ones significantly in makespan performance at zero-shot (CLES = 1). In both studied instances IMPALA agents achieve significantly better training across all fine-tuning data budgets.

Finding 3: MGDT agents achieve comparable or better performance in terms of makespan compared to the IMPALA agents on both scheduling instances at 1% and 2% fine-tuning data budgets. Further, MGDT-MAENT performs best in terms of time taken to find bugs on the playtesting task at 2% fine-tuning data budget. In the bug localization task, IMPALA agents significantly reduce testing time, while MGDT agents achieve comparable or better performance with the other metrics across all fine-tuning data budgets and software projects.

5.4 Discussions

Beyond the SE tasks studied in this chapter, generalist agents can help in test case generation [186] by creating test cases to ensure software reliability. By understanding diverse scenarios during their training, generalist agents can leverage their transfer learning skills [8] to generate diverse test cases that include edge cases and potential failure points in the software with little effort during fine-tuning. Generalist agents can also help in analyzing user behaviors and interactions with software [187] by rapidly adjusting user interfaces based on behavior analysis, leveraging their rapid adaptation skills [188, 189] acquired by being trained across diverse datasets. Generalist agents [8] show good performance at recognizing complex patterns in never-before-seen tasks due to their diverse training and abundance of task-agnostic data [7, 8] in the Atari set for instance. In software defect prediction [190], this ability can allow them to distinguish between true defect-prone code changes and false positives accurately with minimal effort compared to training from scratch, reducing false positives. In the following, we discuss recommendations for researchers and practitioners looking to leverage generalist agents for SE tasks. The provided claims are contextualized to only reflect the obtained results on the studied tasks (i.e., playtesting in games, task scheduling and bug localization).

The exploration capability of the MGDT generalist agent has enabled it to find bugs faster in parts of the Blockmaze game environment where the IMPALA agent did not, across all fine-tuning data budgets.

Recommendation 1: For exploration-intensive tasks, generalist agents with sequence modeling representation of data can be considered for their potential to adapt rapidly and explore unknown areas in an environment.

MGDT and IMPALA agents have shown good performance on the studied bug localization task and PDR-based scheduling in both evaluated scheduling instances across all fine-tuning data budgets. Compared to the Blockmaze and MsPacman games, the PDR-based scheduling addresses the issue of sparse rewards by frequently rewarding the agent according to its makespan during an episode. Similarly, generalist agents are frequently rewarded based on how relevant the file picked is to the bug report.

Recommendation 2: Generalist agents might be effective for tasks where the distribution of rewards is dense.

IMPALA's scalability property allowed it to outperform MGDT in the MsPacman game. Specifically, during evaluation, IMPALA effectively leveraged its ability to scale, continuously learning from new experiences, adapting its behavior, and improving its performance to achieve the desired outcomes.

Recommendation 3: Scalable generalist agents could be beneficial in continuous learning settings for improved data efficiency.

Our results also indicate that the performance of the generalist agents varies among the model-free DRL algorithms used for fine-tuning. In the MsPacman game and the bug localization task, the V_TRACE algorithm outperforms the classic model-free algorithm PPO. Similarly, MAENT outperforms DQN and PPO. As mentioned in Section 5.3.3, MAENT and V_TRACE algorithms optimize their policies more effectively within the MGDT and IMPALA agents, respectively, as they were adapted specifically to the generalist agents' architecture.

Recommednation 4: When fine-tuning a generalist on a task, it might be advantageous to use a model-free DRL algorithm adapted to the generalist agent's architecture for sample-efficient policy optimization.

Our results indicate that the generalist agents significantly outperform the specialist ones on the studied PDR-based scheduling tasks across all fine-tuning data budgets, by achieving the lowest makespan time. Similarly, in the bug localization task, the generalist agents show comparable or greater performance compared to the baseline specialists with 5 out of 8 metrics. The generalist agents were pre-trained on the Atari suite environments which includes a wide range of games, each with its unique mechanics, objectives, and challenges. This diverse training environment helps generalist agents acquire the ability to adapt to different game scenarios which translates well to scheduling, where tasks (i.e., jobs) can vary in complexity and priority. The Atari suite exposed generalist agents to a wide variety of visual images and behavioral patterns related to game mechanics, which means generalist agents can effectively localize bugs by recognizing buggy source code patterns across software projects.

Recommendation 5: Generalist agents could be effective in solving scheduling and bug localization problems, leveraging their diverse pre-training and pattern recognition.

5.5 Threats to validity

Conclusion validity. The conclusion's limitations concern the degree of accuracy of the statistical findings on the best-performing fine-tuned general agents. We use Tukey and Welch's ANOVA post-hoc tests as statistical tests. The significance level is set to 0.05 which is standard across the literature as shown by Welch et al. [95], Games et al. [96]. The non-deterministic nature of DRL algorithms can also threaten the conclusions made in this Chapter. We address this by collecting results from 5 independent runs for all our experiments.

Internal validity. A potential limitation is the number of pre-trained generalist agents used. We have chosen to evaluate only two generalist agents because of the availability of their source code. Moreover, the goal of our study was to show that generalist agents can be leveraged in SE tasks for resource efficiency, thus using any other generalist agents does not invalidate our findings.

Construct validity. A potential threat to construct validity stems from our evaluation criteria. However, these criteria are standard in the literature. We use these measures(i.e., time to find a bug, the average cumulative reward, the makespan, MRR, MAP, top@1, top@5, top@10, and the training and testing time) to make a fair comparison between the generalist agents in identical circumstances. We discussed how they should be interpreted in Section 5

External validity. Since our goal is to investigate whether or not pre-trained generalist agents can be leveraged for SE tasks, a potential limitation is the choice of the tasks used to evaluate the pre-trained generalist agents. We address this threat by choosing playtesting in games, bug localization and scheduling tasks which are three different types of SE tasks, to achieve diversity. While in scheduling we aim to minimize the makespan, in the playtesting in games task we aim to minimize the time to find bugs in a game and in bug localization we aim to localize buggy source code files relevant to bug reports. Another potential external threat to the validity, which concerns that our recommendations may not generalize, is that we applied the generalist agents to only three increasingly used SE tasks (i.e., playtesting in games, bug localization and PDR tasks). We addressed this threat by contextualizing our recommendations to ensure that they reflect our obtained results for the examined tasks and highlight that their generalization may be limited to similar contexts.

Reliability validity. To allow other researchers to replicate or build on our research, we provide a detailed replication package [184] including the code and obtained results.

5.6 Chapter Summary

In this chapter, we investigate the applicability of DRL generalist agents to three increasingly used SE tasks: playtesting in two games, bug localization task, and minimization of makespan in two instances of task scheduling problems. We compare the efficiency of pre-trained generalist agents against that of four baseline approaches in terms of (i) the time to find bugs, (ii) the minimization of the makespan of task scheduling instances, (iii) MRR, (iv) MAP, (v) top@1, (vi) top@5, and (vii) top@10. Our results show that the fine-tuned generalist agents can perform close to or better than the specialist agents on some SE tasks. Our results also highlight the difficulties of some model-free DRL algorithms in handling the complexity of generalist agents, which suggests further investigation. We formulate recommendations, contextualized to reflect our obtained results, to help SE practitioners make an informed decision when leveraging generalist agents to develop SE tasks.

CHAPTER 6 CONTINUOUSLY LEARNING BUG LOCATIONS

6.1 Chapter Overview

In the previous chapters, we studied two types of DRL agents: (1) DRL specialist agents trained from scratch with DRL algorithms (e.g., DQN, PPO, A2C) to solve task scheduling, software testing, and bug localization tasks. Despite their good performance on the task at hand, these agents achieved long training time and struggled to generalize to unseen tasks. This led us to study another category of DRL agents. (2) DRL generalist agents are pre-trained on large and diverse datasets, during which they acquire skills to easily adapt to unseen tasks with little to no effort during fine-tuning. The fine-tuning process is conducted using DRL algorithms such as DQN, PPO, and A2C without having to train the agents from scratch - addressing the cost of training using DRL specialist agents. Depending on the task at hand, DRL generalist agents show greater performance with our evaluation metrics. Investigating DRL specialist and generalist agents led to meaningful recommendations for SE practitioners looking to leverage either one of the agents, depending on the SE task at hand. However, these prior studies did not fully account for the inherent characteristics of certain tasks that could hinder the performance of both DRL specialist and generalist agents. One critical complexity observed in SE tasks, such as task scheduling, software testing, and bug localization, is the non-stationary nature of their data. In such scenarios, concept drift—shifts in the underlying data distribution—can degrade the performance of existing DRL and machine learning techniques.

In this chapter, we propose a Continual Learning (CL) approach designed to address the challenges posed by concept drift. Our investigation focuses specifically on the bug localization task, offering a tailored solution to improve the adaptability and robustness of DRL agents in dynamic, non-stationary environments.

Automatically locating buggy changesets associated with bug reports is crucial in the software development process. Our proposed framework leverages CL agents in a multi-subtask setting for bug localization, where each subtask operates on either stationary or non-stationary data. We empirically evaluate the performance of the CL agents within this framework against several benchmark techniques: a bug localization approach using a BERT model, a DRL-based model utilizing the A2C algorithm, and a DL-based function-level interaction model for semantic bug localization.

To further enhance the performance of the CL agents, we incorporate supervised learning regression to identify and integrate the most significant bug-inducing factors. Our empirical evaluation, conducted across seven widely used software projects, demonstrates that the proposed CL agents outperform DL-based techniques across five ranking metrics in non-stationary settings.

Additionally, our results highlight the effectiveness of the studied CL techniques in localizing changesets relevant to bug reports, while mitigating catastrophic forgetting across tasks. The CL agents also offer a significant advantage in reducing computational effort during training compared to traditional approaches.

6.2 Study Design

In this section, we present the methodology of our study which aims to evaluate the CL agents of the proposed CL framework for bug localization.

6.2.1 Setting Objectives of the Study

Over the years, researchers have investigated the process of locating bugs at both the changeset-file level (i.e., collections of source code files modified in a specific commit) [23, 58, 191–193], and the code segment level [194, 195]. While these studies have reported promising results, they primarily focus on fault localization in stationary settings [23, 58]. This presents a notable limitation, as software projects are inherently non-stationary. Between the time a bug is reported and resolved, multiple versions of changeset files or hunks may exist, leading to evolving and dynamic data. Consequently, the data associated with bug reports and fixes is subject to non-stationary distribution shifts, a phenomenon increasingly recognized in the literature [144]. To adapt to changes in data distribution, ML models can be retrained from scratch on new changeset files (or hunks) or incrementally updated using all available changeset files (or hunks) up to the point of bug localization. However, these approaches are highly resource-intensive, as the rapid evolution of modern software generates expansive search spaces and repositories containing numerous commits [146–148]. Researchers such as Rolnick et al. [196] and Kirkpatrick et al. [154] have implemented DRL-based CL agents, like CLEAR [196] and EWC [154], by cyclically training them across diverse and dynamic environments with shifting data distributions. During the evaluation, these CL agents achieved performance comparable to agents trained separately in each environment, effectively handling distribution shifts while reducing training costs [154, 196, 197]. Inspired by these findings, we hypothesize that DRL-based CL agents have the potential to improve the bug localization process. However, their application in this context remains unexplored, leaving the software engineering community with little understanding of how these agents can be adapted or how effective they might be in addressing this challenge.

This Chapter proposes a CL framework that incorporates two well-established approaches to CL, tailored for DRL in multi-task, non-stationary environments: CLEAR [196] and EWC [154]. CLEAR employs an actor-critic model that combines new and replayed experiences, updating the model using the V-Trace off-policy algorithm [7], which corrects for off-policy distribution shifts via importance weights. EWC, on the other hand, is a weight consolidation technique that maintains the model's adaptability (i.e., its ability to acquire new knowledge) by restricting significant parameters to stay close to their original values. CLEAR and EWC are adapted to train CL agents cyclically on stationary and non-stationary data. We evaluate the effectiveness of the CL agents by comparing them against three baseline Deep Learning (DL) techniques. The evaluation focuses on performance and computational efficiency during training across different data granularities (i.e., changeset-files and hunks). To enhance these CL agents, by training a logistic regression model using known bug-inducing factors (e.g., source and executable lines of code, cyclomatic complexity, number of lines modified, and pre-release bugs), selecting only non-collinear, statistically significant factors. These factors are effective indicators of potential bugs in software programs [198]. The scalar output of the logistic regression model was then incorporated into the reward function of the DRL agents, boosting their effectiveness in identifying and localizing bugs.

To structure the study in this chapter, we define the following RQs:

RQc$_1$: How do DL techniques for bug localization perform in non-stationary settings? The purpose of this RQ is to empirically evaluate the performance of our baseline studies on non-stationary settings at changeset-files level. Between the time a bug is reported and when it is fixed, the affected changeset-files in a software project can undergo various changes, such as content modifications [199] or function renaming [200], yet still be associated with the bug report. RLOCATOR AND FLIM baseline studies trained and evaluated their proposed approaches on changeset-files collected when bugs are fixed only. Nonetheless, a bug localization technique should remain effective regardless of the different versions of the buggy changeset-files. We calculate the relevant metrics described in Section 6.2.3 for FLIM and RLOCATOR baseline studies.

RQc$_2$: Can CL techniques improve bug localization? In **RQc$_1$**, the results show that FLIM and RLOCATOR (i.e., our baseline studies) techniques have difficulties adapting to the concept drift associated with data. One way to solve this problem is to employ CL agents that address the concept drift associated with data by being sequentially trained on both stationary and non-stationary data at the changeset-files and hunks level. Therefore, the purpose of this RQ is to evaluate the performance of CL agents on both stationary and non-stationary data across all datasets. Moreover, since CL agents may suffer from catastrophic forgetting [85], we adopt rehearsal and regularization based techniques (CLEAR and EWC respectively) that mitigate catastrophic forgetting through policies that adjust distribution shifts in the data to be learned. We calculate our evaluation metrics for each dataset and compare the performance of the CL agents against all baselines and each other (CLEAR vs. EWC). Additionally, for each CL agent, we calculate the performance differences between stationary and non-stationary data. Our goal is to ensure that the agents effectively localize bugs in both types of environments.

RQc$_3$: Can prior knowledge about bug-inducing factors improve the performance of CL techniques? The purpose of this RQ is to evaluate the performance of the CL agents on both stationary and non-stationary data (i.e., changeset-files and hunks) by integrating prior knowledge of bug-inducing factors into their training phase. Bug-inducing factors are good indicators of whether changeset-files and hunks are likely to be buggy or not [63, 198]. Thus, incorporating them in any bug localization technique can improve its performance. We calculate our evaluation metrics for each dataset and compare the performance of the CL agents against the baselines, both before and after including the bug-inducing factors. Similarly to **RQc$_2$**, for each CL agent (with regression), we calculate the performance differences in percentage between stationary and non-stationary data.

6.2.2 Motivating Example

To motivate the problem under investigation, let's consider a scenario from the Eclipse Apache project [59]. This project, due to its complexity, requires developers to collaborate closely, each handling specific tasks. However, bugs are occasionally reported, and developers must then spend valuable time analyzing and fixing these issues. In this context, developer **A**, responsible for maintaining the project, decides to implement a bug localization technique to streamline the debugging process and improve productivity. The goal is to reduce the manual effort involved in identifying the bug's location in the code.

To achieve this, developer **A** adopts a bug localization technique, such as FLIM [58] or RLOCATOR [23], both of which rank changeset-files (stationary data) according to their relevance to a given bug report, with and without using entropy.

For instance, when bug #420210 is reported [5], developer **A** uses the previously implemented bug localization technique to assist in the bug-fixing process. The technique works by analyzing the list of changeset-files associated with the latest commit following the bug report. It then ranks these changeset-files based on their likelihood of containing the bug.

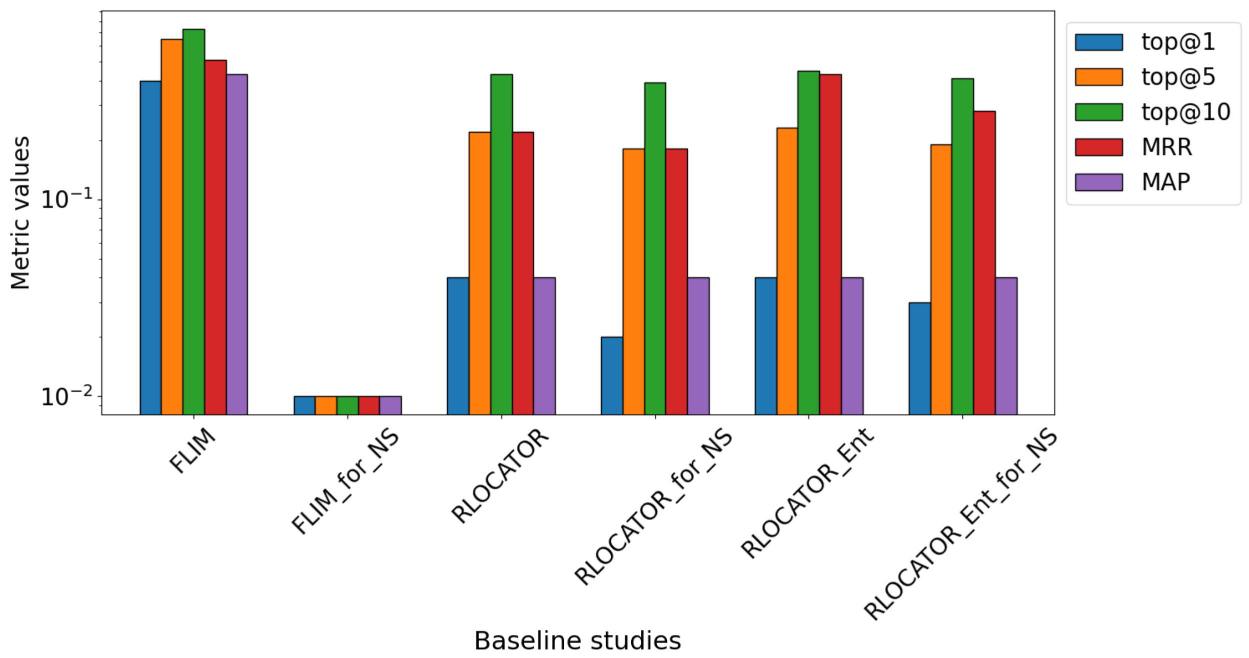


Figure 6.1 Performance of state-of-art baseline studies on Eclipse project.

Figure 6.1 illustrates the performance of FLIM, RLOCATOR, and RLOCATOR with entropy (RLOCATOR_Ent) in ranking changeset-files on the Apache Eclipse project for bug reports similar to bug #384108. Based on the ranking, developer **A** can prioritize the files most likely to be related to the bug, thereby accelerating the bug-fixing process.

However, when another bug is reported (bug #384108 [201]), the scenario becomes more complex. Developer **B**, who is working on a different task, modifies certain changeset-files that developer **A** had previously identified as important for bug #384108. These modifications (e.g., code updates, code additions, or removals) introduce concept drift into the data used to rank bug-prone files. As a result, the bug localization technique's performance may degrade.

Due to the concept drift in the data, the performance of DL-based models used by developer **A** tends to decrease. Figure 6.1 shows a noticeable decline in the performance of FLIM, RLOCATOR (with and without entropy), on non-stationary data (i.e., FLIM_for_NS, RLOCATOR_for_NS, and RLOCATOR_Ent_for_NS). Specifically, there is a drop of at least 9% in top@1, top@5, and mean reciprocal rank (MRR) metrics.

This highlights the challenge of maintaining the effectiveness of bug localization techniques in the face of dynamic code changes and concept drift, underscoring the need for methods that can adapt to such changes over time.

To address or minimize the concept drift associated with the data, we propose a solution involving CL agents capable of adapting to both stationary and non-stationary data. On non-stationary data, the performance of the CL agents decreased by 4 - 62%. They also outperformed FLIM and RLOCATOR by 14-167% in terms of top@1, top@5, and MRR metrics. Our CL agents can adapt more effectively to non-stationary data by reducing catastrophic forgetting while demanding less computational resources.

6.2.3 Methodology

To leverage CL for bug localization we powered our proposed technique with two CL agents to rank changeset-files and hunks related to bug reports. We were inspired by state-of-the-art CL techniques: CL with experience and replay [196] and elastic weight consolidation [154] that we used to train DRL agents cyclically in sequence on stationary and non-stationary data. Figure 6.2 illustrates our proposed CL framework. In the rest of this section, we detail the CL framework and used components.

Continual learning with experience and replay (CLEAR)

CLEAR is a CL technique to mitigate catastrophic forgetting by training an actor-critic deep neural network on a mixture of new and replayed experiences. Inspired by IMPALA [7] (Importance Weighted Actor-Learner Architecture) presented in Chapter 5, CLEAR uses the actor-network to generate trajectories of the environments (e.g ranking tasks) which are sent to the critic-network to learn and off-policy algorithm V_TRACE. V_TRACE mitigates catastrophic forgetting by correcting the distribution shift of the experiences generated by the actor. As stated in Section 2, off-policy algorithms like V_TRACE use the behavior policy μ to generate trajectories in the form of:

$$(x_t, a_t, r_t)_{t=s}^{t=s+n}$$

then, use those trajectories to learn the value function V^π of another policy called target policy. V_TRACE target policy is formulated as:

$$v_s \overset{\text{def}}{=} V(x_s) + \sum_{t=s}^{s+n-1} \gamma^{t-s} (\prod_{i=s}^{t-1} c_i) \delta_t V \tag{6.1}$$

where γ is the discount factor and a temporal difference for V is defined as

$$\delta_t V \overset{\text{def}}{=} \rho_t (r_t + \gamma V(x_{t+1}) - V(x_t))$$

where ρ_t and c_i are truncated Importance Sampling (IS) weights defined as:

$$\rho_t \overset{\text{def}}{=} min(\bar{\rho}, \frac{\pi(a_t \mid x_t)}{\mu(a_t \mid x_t)}),$$

$$c_i \overset{\text{def}}{=} min(\bar{c}, \frac{\pi(a_i \mid x_i)}{\mu(a_i \mid x_i)})$$

At training time s, the value parameters θ are updated by the gradient descent on $l2$ loss to the target function v_s in the direction of

$$(v_s - V_\theta(x_s))\Delta_\theta V_\theta(x_s),$$

and the policy parameters ω are updated in the direction of the policy gradient:

$$\rho_s \Delta_\omega log\pi_\omega(a_s \mid x_s)(r_s + \gamma v_{s+1} - V_\theta(x_s))$$

In bug localization setting, the actor generates trajectories that are a mixture of stationary and non-stationary data. With two tasks to be learned (i.e., ranking source code files from stationary and non-stationary data), we leverage CLEAR to train a CL agent that learns cyclically in sequence the optimal ranking strategy on both tasks. In this chapter, we adopt the CLEAR implementation with the default parameters.

Elastic weight consolidation (EWC)

EWC is a CL algorithm that enables plasticity on deep neural networks by constraining important parameters to stay close to their old values. More specifically, EWC works by over-parameterizing the neural networks such that there is a solution to a task B that is close to the solution of a previous task A. When learning task B, the performance of task A is protected by constraining the parameters of the neural network to stay in a region of low error for task A, i.e., centered around the parameters obtained after learning task A. To define which parameters are more important for a task B given the learned parameters θ^*_A of a task B, the EWC loss function is defined as follows:

$$\mathcal{L}(\theta) = \mathcal{L}_B(\theta) + \sum_i \frac{\lambda}{2} F_i \left(\theta_i - \theta^*_{A,i}\right)^2 \tag{6.2}$$

where $\mathcal{L}_B(\theta)$ is the loss function for task B, F_i is the Fisher information matrix, λ sets how important the old task is compared to the new one, and i labels each parameter.

In the context of DRL, the DQN algorithm is augmented with EWC to achieve CL across the Atari 2600 task set [179].

To the best of our knowledge, EWC implementation is not publicly available, therefore we utilized it in this chapter from a popular implementation made by Powers et al. [197]. In their implementation, they use the IMPALA architecture to achieve comparable results. In this chapter, we adopt the EWC implementation with the default parameters. Similar to CLEAR, the CL agent is exposed to experiences from stationary and non-stationary data. To apply the EWC algorithm, the Fisher information matrix is computed before switching to each bug localization task.

Bug-inducing factors for bug localization

Bug-inducing factors associated with changesets have shown to be predictors of bugs across software projects [63, 198, 202]. Taba et al. [198] used antipattern information as metrics to improve traditional bug prediction models at the changeset-files level. Table 6.1 presents the bug-inducing factors metrics used by Taba et al. [198]. They combine metrics representing bug-inducing factors to construct a logistic regression model for predicting the presence of bugs. In this model, the independent variables correspond to the bug-inducing factor metrics, while the dependent variable is a binary variable indicating whether a file contains one or more bugs. Following a similar approach, we collected these metrics at both the changeset-file and hunk levels. A detailed description of the bug-inducing factors is provided below:

- **Lines of Code (LOC):** For a given buggy version, we do a checkout at the buggy commit and compute the associated lines of code.
- **Executable LOC (MLOC):** This metric is generated based on the previous one (LOC), but this time considering only executable lines, excluding lines with comments and blank ones.
- **Cyclomatic Complexity (VG):** This metric computes the number of linearly independent paths of the given source code [203]. For computing this and the previous metrics, we consider the CCC tool [204].
- **Pre-released Bugs (PRE):** Based on the set of files updated when fixing a given bug, we compute the number of times these files were changed in other bug fixes. For that, we get the changed files in each commit associated with a bug fix; then, we search for occurrences considering the files associated with the bug report under analysis.

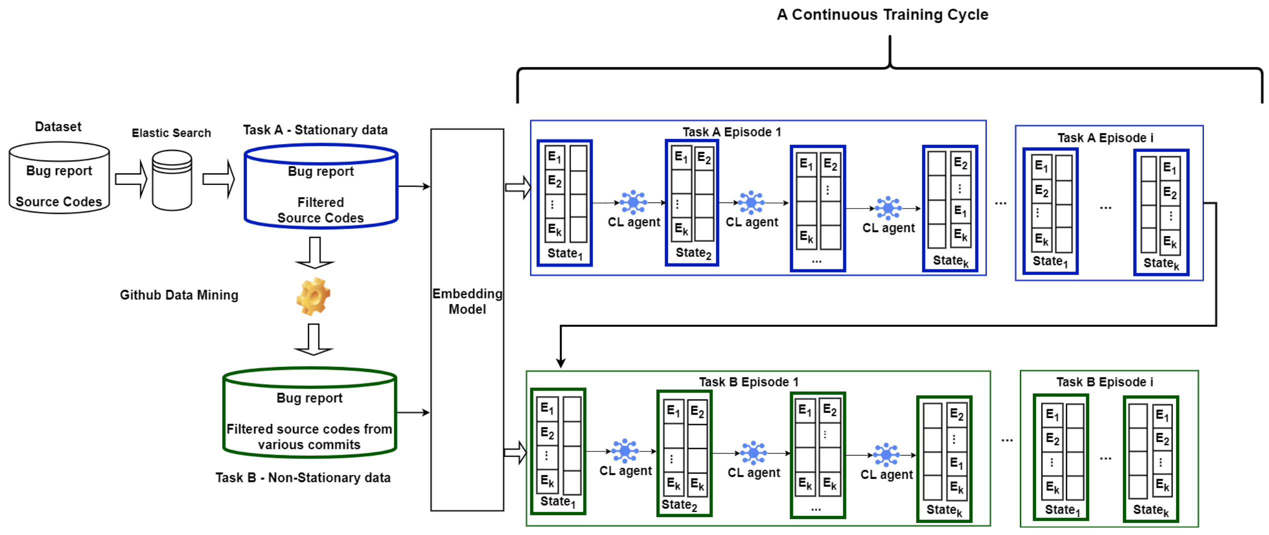


Figure 6.2 Illustration of the CL Framework.

Table 6.1 Measured Source Code and Bug Report Metrics

	Metrics	Description
Source Code	LOC	Source Lines of Code
	MLOC	Executable Lines of Code
	VG	Cyclomatic complexity
Bug Report	PRE	Number of pre-released bugs
	Churn	Number of lines of code modified (added, removed)

- **Code Churn (Churn):** We compute the number of lines of code added, modified, or deleted. We get the diff associated with the bug commit under analysis and get the required information.

Since the metrics collected can be dependent and highly collinear, we follow the iterative process described below to retain only important and non-collinear metrics. The following elaborates more on the process.

1. Removing statistically insignificant variables: In this step, we first build the logistic regression model with all metrics mentioned above as variables of the model; then, in an iterative process, we remove the statistically insignificant metrics. We use the threshold p-value < 0.05 to determine whether a variable is statistically significant or not.

2. Collinearity analysis: In a regression model, two or more variables are multicollinear when they are highly correlated. We removed the highly collinear variables and only

kept independent variables that produce an effect on the dependent variable [198]. We used the Variance Inflation Factor (VIF) to measure the level of multicollinearity of the regression model. We set the maximum VIF value to be 2.5, as suggested in [198]. Independent variables with VIF values that exceed 2.5 are considered highly collinear and, hence, are removed.

After removing statistically insignificant and highly collinear variables, we narrow down our list of independent variables to the Churn and the PRE metrics. In this study, we integrated the output of the logistic regression model as an additional component of the reward function of the CL agents. To validate this approach, we conduct ablation studies in Section 6.3.4. These studies compare the retrieval performance of the CL agents under three conditions: (1) when each of the bug-inducing factors is individually added to the reward function of the CL agents, (2) when the output of a logistic regression model incorporating all bug-inducing factors as independent variables is added to the reward function of the CL agents and (3) when the output of a logistic regression model, using only statistically significant and non-collinear bug-inducing factors, is added to the reward function of the CL agents. When considering bug-inducing factors as prior knowledge for the CL agents, we use a logistic model to learn the probability that a changeset is buggy. The reward function for this case is as follows:

$$\mathcal{R}(o,a) = \mathcal{R}(o,a) + \textit{bug probability indicator} \tag{6.3}$$

where *bug probability indicator* is the output of the logistic regression model. The bug probability indicator is a scalar. At each step during an episode, we add the bug probability indicator to the reward earned by the CL agent.

Baselines studies

As baselines for our study, we employ three methods detailed below, each representing state-of-the-art approaches utilizing DL techniques for bug localization.

- Rlocator [23]: a DRL model based on an MDP to directly optimize the ranking of considered metrics. For evaluation, we consider the model provided with and without Entropy for A2C. A2C with entropy incorporates the entropy of the probability of the possible action to the loss of the actor model. Moreover, we evaluated the CL agents against RLOCATOR on AspectJ, Birt, Eclipse, SWT, and Tomcat projects.

- FLIM [58]: a framework that extracts semantic features from code at the function level and calculates the relevance between natural and programming language. Then, it uses a DL model to fuse the function-level semantic features with IR features to calculate the final relevance. The DL model employed by FLIM uses 19 independent variables. These variables capture the similarity between a source code file and a bug report, the API specifications of the source code file, as well as the recency and frequency of bug fixes associated with the source code files. In this work, we evaluated the CL agents against FLIM on AspectJ, Birt, Eclipse, SWT, and Tomcat projects.

- FBL-BERT [66]: A supervised learning approach that uses BERT model for bug localization at changeset-files, hunks, and commits (i.e., set of hunks) levels. The study explores three strategies (i.e., QD, QARC, and QARCL) for encoding code modifications and data granularities to optimize BERT's performance: 1) QD considers a changeset as a single document, with a special token pre-appended to signal the start of the code sequence for the model 2) QARC encodes a changeset by dividing it into lines grouped by their type: added (+), removed (-), or context (empty space). Each group is prefixed with a special token, and the sequences are then concatenated to form the model's input 3) In QARCL, similar to the QARC strategy, a changeset is divided into lines, but the original line order is preserved, with special tokens pre-appended at each change in the type of modification. We compare the performance of the CL agents on the AspectJ, Zxing, PDE, SWT, and Tomcat projects against all three strategies at the changeset-files and hunks levels. We do not evaluate the CL agents against FBL-BERT at the commits level as the CL agent action consists of picking one changeset-file or hunk at a time during an episode, as presented in Chapter 2.

Datasets

We evaluate our approach using bugs reported in widely used software projects, leveraging their associated changeset-files and hunks [63,64]. These software projects are commonly used in previous and related studies, valued for their quality and the diversity of bug scenarios they present [23, 58, 63, 66]. From the eight open-source Java repositories available from Apache (AspectJ, Birt, JDT, PDE, Eclipse, Zxing, Tomcat, and Birt), we selected seven, excluding JDT due to the absence of non-stationary data. Specifically, when analyzing changes between bug report dates and commit fix dates in the JDT project, we found no modifications in the files associated with the reported bug fixes, resulting in no intersection with the ground truth. Among the different information provided for a given bug, we highlight the bug commit and its associated changeset-files — key elements needed for computing our metrics on stationary data (see Section 6.2.3). From the commits associated with bugs, we checkout on the commits and access the full version of the buggy code.

Table 6.2 Benchmark statistics

Project	#Bugs reported	#Changeset-files and hunks
AspectJ	593	27,127
Birt	4,178	1,939
Eclipse	6,495	6,645
SWT	4,151	598
Tomcat	1,056	41,271
PDE	60	123,053
Zxing	20	19,084

For the non-stationary data, we analyze all commits that modify the changeset-files associated with a bug report. The objective is to identify all changeset-files potentially impacted by a bug. To achieve this, we collect commits made between the dates of the bug report and the corresponding bug-fixing commit. Once we have the list of commits, we extract the different versions of the target changeset-files and use them as non-stationary data for our model. To collect the hunks associated with each changeset-files we collect the output of the git diff command in which added lines of code are annotated with +, and removed lines with -. Table 6.2 shows for each project (i.e., AspectJ, Birt, PDE, Eclipse, Zxing, Birt, and Tomcat), the total number of changed buggy changeset-files and hunks relevant to the bug reports. To create our training and testing datasets, we sort the bug reports based on their report date, following an ascending order. Then, we divide the dataset into a 60:40 percent split (training and test) similar to our baselines [23].

Evaluation metrics

To evaluate our proposed model and compare it with the baseline studies, we rely on the ground truth provided by the benchmark considered for this study (see Section 6.2.3). For each bug reported, the dataset provides the changed files to fix the bug. This way, we can evaluate the performance of our model based on five criteria, which are widely recognized and used by related studies when conducting bug localization studies [23, 58, 139, 197, 205].

- **Mean Reciprocal Rank (MRR)** is a metric used to evaluate the effectiveness of information retrieval based on the files' ranking reported by a model within its list of results. For example, in our case, we aim to evaluate how frequently the buggy files are ranked in the first positions of a report. An MRR close to 1 means the model usually ranks the right files in the first positions, while a value close to 0 indicates poor performance.

Table 6.3 Tomcat

		top@1		top@5		top@10		MRR		MAP	
		ST	NS	ST	NS	ST	NS	ST	NS	ST	NS
FLIM	-files	0.47	0.00	0.70	0.00	0.77	0.00	0.57	0.01	0.51	0.01
RLO.	-files	0.04	0.04	0.24	0.21	0.48	0.42	0.24	0.18	0.04	0.04
RLO. + Reg.	-files	0.04	0.04	0.23	0.22	0.46	0.43	0.25	0.19	0.04	0.04
RLO. + Ent.	-files	0.04	0.04	0.21	0.22	0.45	0.45	0.34	0.28	0.04	0.04
RLO. + Ent. + Reg.	-files	0.04	0.04	0.21	0.21	0.41	0.44	0.27	0.32	0.04	0.04
CLEAR	-files	0.05	0.05	0.24	0.22	0.44	0.48	0.45	0.25	0.05	0.05
	hunks	0.03	0.03	0.21	0.20	0.42	0.40	0.39	0.29	0.04	0.04
CLEAR + Reg.	-files	0.04	0.06	0.21	0.29	0.32	0.43	0.31	0.27	0.04	0.06
	hunks	0.05	0.07	0.22	0.28	0.35	0.42	0.26	0.17	0.04	0.06
EWC	-files	0.03	0.03	0.21	0.20	0.43	0.42	0.30	0.34	0.04	0.04
	hunks	0.06	0.04	0.23	0.22	0.46	0.44	0.28	0.22	0.05	0.04
EWC+Reg.	-files	0.04	0.04	0.20	0.24	0.42	0.42	0.18	0.26	0.04	0.05
	hunks	0.05	0.05	0.23	0.24	0.48	0.48	0.20	0.32	0.05	0.05

Table 6.4 Birt

		top@1		top@5		top@10		MRR		MAP	
		ST	NS	ST	NS	ST	NS	ST	NS	ST	NS
FLIM	-files	0.13	0.14	0.30	0.14	0.38	0.57	0.21	0.21	0.16	0.24
RLO.	-files	0.05	0.02	0.24	0.16	0.51	0.33	0.27	0.16	0.05	0.03
RLO. + Reg.	-files	0.05	0.03	0.25	0.19	0.51	0.33	0.44	0.16	0.05	0.03
RLO. + Ent.	-files	0.05	0.02	0.25	0.14	0.52	0.28	0.34	0.30	0.05	0.03
RLO. + Ent. + Reg.	-files	0.05	0.03	0.25	0.18	0.52	0.37	0.38	0.42	0.05	0.03
CLEAR	-files	0.05	0.05	0.26	0.25	0.52	0.49	0.31	0.46	0.05	0.05
	hunks	0.05	0.07	0.23	0.25	0.50	0.47	0.29	0.24	0.05	0.05
CLEAR + Reg.	-files	0.05	0.06	0.25	0.25	0.49	0.46	0.36	0.33	0.05	0.05
	hunks	0.05	0.02	0.28	0.23	0.54	0.42	0.44	0.39	0.05	0.04
EWC	-files	0.05	0.03	0.27	0.17	0.53	0.39	0.45	0.33	0.05	0.04
	hunks	0.05	0.02	0.25	0.19	0.51	0.41	0.36	0.17	0.05	0.04
EWC+Reg.	-files	0.06	0.06	0.27	0.20	0.52	0.41	0.45	0.26	0.05	0.04
	hunks	0.05	0.03	0.28	0.17	0.51	0.40	0.50	0.17	0.05	0.04

Algorithms evaluation across software projects. "RLO." refers to the baseline RLOCATOR. "+ Reg" indicates the inclusion of logistic regression, while "+ Ent" denotes the addition of entropy. "-files"=changeset-files. The best and second best average performances are highlighted in dark grey and light grey respectively. For each metric, the ST column contains the performance of algorithms on stationary data, and the NS column contains the performance on non-stationary data.

Table 6.5 Eclipse

		top@1		top@5		top@10		MRR		MAP	
		ST	NS	ST	NS	ST	NS	ST	NS	ST	NS
FLIM	-files	0.40	0.00	0.65	0.00	0.73	0.00	0.51	0.00	0.43	0.00
RLO.	-files	0.04	0.02	0.22	0.18	0.43	0.39	0.22	0.18	0.04	0.04
RLO. + Reg.	-files	0.04	0.05	0.22	0.19	0.43	0.36	0.28	0.48	0.04	0.04
RLO. + Ent.	-files	0.04	0.03	0.23	0.19	0.45	0.41	0.43	0.28	0.04	0.04
RLO. + Ent. + Reg.	-files	0.04	0.04	0.22	0.19	0.43	0.38	0.56	0.47	0.04	0.04
CLEAR	-files	0.05	0.03	0.22	0.19	0.44	0.38	0.34	0.27	0.04	0.04
	hunks	0.05	0.03	0.22	0.16	0.44	0.37	0.35	0.34	0.04	0.04
CLEAR + Reg.	-files	0.04	0.04	0.21	0.20	0.35	0.31	0.34	0.24	0.04	0.04
	hunks	0.04	0.04	0.20	0.21	0.34	0.36	0.31	0.23	0.04	0.04
EWC	-files	0.04	0.05	0.21	0.19	0.45	0.39	0.48	0.25	0.04	0.04
	hunks	0.04	0.05	0.21	0.20	0.45	0.40	0.26	0.38	0.04	0.04
EWC+Reg.	-files	0.04	0.04	0.21	0.19	0.44	0.38	0.26	0.28	0.04	0.04
	hunks	0.05	0.05	0.23	0.22	0.44	0.42	0.31	0.30	0.05	0.04

Table 6.6 AspectJ

		top@1		top@5		top@10		MRR		MAP	
		ST	NS	ST	NS	ST	NS	ST	NS	ST	NS
FLIM	-files	0.14	0.00	0.44	0.37	0.63	1.00	0.29	0.17	0.24	0.05
RLO.	-files	0.04	0.03	0.20	0.17	0.33	0.33	0.36	0.30	0.04	0.03
RLO. + Reg.	-files	0.04	0.03	0.21	0.17	0.45	0.30	0.32	0.30	0.04	0.04
RLO. + Ent.	-files	0.04	0.03	0.22	0.15	0.47	0.31	0.36	0.24	0.04	0.03
RLO. + Ent. + Reg.	-files	0.04	0.02	0.23	0.14	0.46	0.29	0.32	0.12	0.04	0.03
CLEAR	-files	0.04	0.03	0.22	0.22	0.45	0.40	0.37	0.14	0.04	0.04
	hunks	0.04	0.06	0.25	0.21	0.48	0.40	0.32	0.29	0.05	0.04
CLEAR + Reg.	-files	0.04	0.04	0.24	0.17	0.48	0.37	0.39	0.26	0.05	0.04
	hunks	0.06	0.05	0.23	0.24	0.44	0.46	0.37	0.30	0.05	0.05
EWC	-files	0.04	0.04	0.25	0.20	0.48	0.41	0.34	0.37	0.05	0.04
	hunks	0.05	0.03	0.23	0.14	0.46	0.33	0.41	0.30	0.05	0.03
EWC+Reg.	-files	0.04	0.05	0.26	0.23	0.49	0.44	0.43	0.42	0.05	0.04
	hunks	0.05	0.04	0.22	0.22	0.43	0.45	0.33	0.45	0.04	0.04

Algorithms evaluation across software projects. "RLO." refers to the baseline RLOCATOR. "+ Reg" indicates the inclusion of logistic regression, while "+ Ent" denotes the addition of entropy. "-files"=changeset-files. The best and second best average performances are highlighted in dark grey and light grey respectively. For each metric, the ST column contains the performance of algorithms on stationary data, and the NS column contains the performance on non-stationary data.

- **Mean Average Precision (MAP)** measures the overall mean performance of a model regarding a set of bug reports, unlike MRR, which considers the best rank of relevant files. Knowing that multiple buggy files can be associated with a bug, ideally, the model should rank all of them in its report. This way, MAP represents a more descriptive and unbiased metric than MRR. For a given bug, we compute the average value based on the individual precision values of each buggy file.

- **Top K** computes the overall ranking performance of a model based on whether a buggy file is properly ranked in the top N results. In our study, considering multiple files might be buggy, we check whether at least one is reported in the top N results. Following previous studies, we consider three values of N: 1, 5, and 10 [23].

- **Training time** measures the time consumed by the CL agents to get trained. We collected the training time for 7500 episodes per task, same as the baseline agent [23].

- **The forgetting metric.** We evaluate the forgetting of the CL agents as part of our evaluation metrics. Specifically, at the training stage, we measure how much each CL agent forgets previously learned bug localization skills on non-stationary data while learning bug localization on stationary data. The **forgetting metric** [197] $\mathcal{F}$ is computed as follows:
$$\mathcal{F} = \frac{r_{i,j-1,end} - r_{i,j,end}}{\mid r_{i,all,max} \mid} \tag{6.4}$$
where $r_{i,j,end}$ is expected return achieved on task i after training on task j and $r_{i,all,max}$ is the maximum expected return achieved on task i. $r_{i,all,max}$ ensures the normalization of the rewards across tasks for better comparison among them.

To counter the randomness effect when collecting our evaluation metrics, we repeat each run 5 times to average the results. We employ Welch's ANOVA and Games-Howell post-hoc test [95], [96] to indicate the best bug localization technique. While Welch's ANOVA test checks for significant differences between the CL agents across the datasets, the Games-Howell post-hoc test compares each pair of configurations. A difference with p-value ≤ 0.05 is considered significant in our assessments.

6.3 Results

This section presents the experimental results obtained while answering our three RQs.

Table 6.7 Algorithms evaluation on SWT project. "RLO." refers to the baseline RLOCATOR. "+ Reg" indicates the inclusion of logistic regression, while "+ Ent" denotes the addition of entropy. "-files"=changeset-files. The best and second best average performances are highlighted in dark grey and light grey respectively. For each metric, the ST column contains the performance of algorithms on stationary data, and the NS column contains the performance on non-stationary data.

		top@1		top@5		top@10		MRR		MAP	
		ST	NS	ST	NS	ST	NS	ST	NS	ST	NS
FLIM	-files	0.32	0.00	0.62	0.00	0.73	0.50	0.46	0.05	0.39	0.10
RLO.	-files	0.04	0.07	0.22	0.37	0.44	0.88	0.26	0.32	0.04	0.08
RLO. + Reg.	-files	0.04	0.10	0.22	0.34	0.44	0.70	0.26	0.37	0.05	0.07
RLO. + Ent.	-files	0.04	0.10	0.23	0.35	0.43	0.83	0.19	0.58	0.04	0.10
RLO. + Ent. + Reg.	-files	0.04	0.10	0.22	0.41	0.44	0.80	0.64	0.68	0.04	0.10
CLEAR	-files	0.05	0.17	0.22	0.51	0.45	0.86	0.46	0.59	0.04	0.10
	hunks	0.03	0.08	0.21	0.45	0.44	0.91	0.25	0.54	0.04	0.09
CLEAR + Reg.	-files	0.05	0.13	0.24	0.41	0.43	0.91	0.24	0.41	0.05	0.09
	hunks	0.04	0.09	0.22	0.46	0.45	0.87	0.23	0.40	0.04	0.10
EWC	-files	0.04	0.15	0.21	0.35	0.44	0.85	0.33	0.40	0.04	0.08
	hunks	0.04	0.13	0.24	0.56	0.45	1.0	0.31	0.56	0.05	0.11
EWC+Reg.	-files	0.05	0.12	0.25	0.47	0.49	0.94	0.36	0.48	0.05	0.10
	hunks	0.04	0.12	0.22	0.44	0.43	0.91	0.19	0.51	0.04	0.09

6.3.1 RQc$_1$: How do DL techniques for bug localization perform in non-stationary settings?

Table 6.3, 6.4, 6.5, 6.6, and 6.7 show the performance of the baselines studies (i.e., FLIM [23] and RLOCATOR [58]) on stationary and non-stationary data at the changeset-files level across Tomcat, Birt, Eclipse, AspectJ and SWT projects respectively. FLIM's performance significantly decreases (by 17 - 194%) on non-stationary data compared to stationary data, as evaluated across the Tomcat (see Table 6.3), Eclipse (see Table 6.5), AspectJ (see Table 6.6) and SWT (see Table 6.7) projects using the top@1, top@5, MRR, and MAP metrics. RLOCATOR (with and without entropy) demonstrates a similar trend on AspectJ, Birt, and Eclipse projects with a decrease in performance by 9 - 85% on non-stationary data compared to stationary data using the top@1, top@5, top@10, and MRR metrics.

Overall, both studied baseline techniques struggle to adapt to non-stationary data, except for RLOCATOR (with and without entropy) on SWT, and FLIM on Birt and AspectJ (top@10 metric). As detailed in Section 5, FLIM utilizes 19 features within its DL framework. Among these features is the Pre-released Bugs metric (refer to Section 6.2.3), which is scaled by the maximum number of Pre-released Bugs of the source code files. For the Birt and AspectJ projects, over 80% of the source code files have Pre-released Bugs greater than 0. This likely accounts for FLIM's accurate ranking of buggy changeset-files in non-stationary setting for these datasets. RLOCATOR (with and without entropy) can adapt to non-

Table 6.8 Algorithms evaluation on PDE and Zxing project. -files=changeset-files. The best and second best average performances are highlighted in dark grey and light grey respectively per project. For each metric, the ST column contains the performance of algorithms on stationary data, and the NS column contains the performance on non-stationary data.

			top@1		top@5		top@10		MRR		MAP	
			ST	NS	ST	NS	ST	NS	ST	NS	ST	NS
PDE	CLEAR	-files	0.08	0.08	0.40	0.41	0.87	0.87	0.51	0.28	0.08	0.09
		hunks	0.07	0.08	0.42	0.44	0.81	0.83	0.38	0.45	0.08	0.09
	CLEAR + Reg.	-files	0.08	0.08	0.42	0.39	0.77	0.77	0.33	0.41	0.08	0.08
		hunks	0.09	0.09	0.39	0.41	0.69	0.73	0.36	0.33	0.08	0.09
	EWC	-files	0.09	0.09	0.44	0.43	0.89	0.88	0.45	0.33	0.09	0.09
		hunks	0.07	0.09	0.42	0.45	0.85	0.87	0.47	0.29	0.08	0.09
	EWC + Reg.	-files	0.10	0.09	0.43	0.44	0.85	0.86	0.32	0.39	0.09	0.09
		hunks	0.09	0.10	0.44	0.45	0.85	0.88	0.34	0.33	0.09	0.09
Zxing	CLEAR	-files	0.12	0.11	0.65	0.56	1.0	1.0	0.40	0.60	0.12	0.11
		hunks	0.11	0.11	0.58	0.58	1.0	1.0	0.54	0.55	0.12	0.11
	CLEAR + Reg.	-files	0.12	0.12	0.63	0.61	1.0	1.0	0.42	0.49	0.12	0.12
		hunks	0.14	0.12	0.65	0.59	1.0	1.0	0.51	0.51	0.13	0.12
	EWC	-files	0.12	0.14	0.62	0.56	1.0	1.0	0.58	0.47	0.12	0.12
		hunks	0.12	0.12	0.62	0.57	1.0	1.0	0.55	0.54	0.12	0.12
	EWC + Reg.	-files	0.13	0.12	0.63	0.59	1.0	1.0	0.58	0.64	0.13	0.12
		hunks	0.10	0.12	0.61	0.56	1.0	1.0	0.55	0.57	0.12	0.11

stationary data on SWT, possibly because SWT project has less amount of changeset-files (see Table 6.2), allowing the A2C algorithm employed to stabilize more easily despite the non-stationary nature of the data. Chen et al. [206] observed similar performances, where DRL models performed relatively well in non-stationary setting due to the limited data available for training in each episode.

Finding 1. At the changeset-files level, FLIM and RLOCATOR (with and without entropy) struggle to adapt to non-stationary data in most projects (three out of five) and metrics (four out of five).

6.3.2 RQc$_2$: Can CL techniques improve bug localization?

Table 6.3, 6.4, 6.5, 6.6, and 6.7 show the performance of the CL agents (i.e., CLEAR and EWC) on stationary and non-stationary data (i.e., changeset-files and hunks) across the studied datasets against FLIM and RLOCATOR. CLEAR and EWC show up to a 35% performance improvement over RLOCATOR (with and without entropy) on the stationary data of the AspectJ project, measured by top@5, top@10, MRR, and MAP metrics. In contrast, for other projects (i.e., Tomcat, Birt, SWT, and Eclipse), CLEAR and EWC perform similarly to RLOCATOR (with and without entropy) in top@1, top@5, and MAP metrics. The CL agents show better or competitive performance compared to RLOCATOR (with and with-

Table 6.9 Algorithms evaluation across software projects with different data granularities. In bold are the best average performances per algorithm per metric for each project and data granularity. "-files"=changeset-files. "P@"=top@.

	-files	SWT	AspectJ	Tomcat	**hunks**	SWT	AspectJ	Tomcat
MRR		0.53	**0.26**	0.35		0.40	**0.30**	0.34
MAP		0.07	0.04	0.05		0.06	0.04	0.04
P@1	CLEAR	0.08	0.04	0.05	CLEAR	0.04	0.04	0.03
P@5		**0.36**	**0.22**	**0.23**		0.33	**0.23**	0.21
P@10		**0.65**	0.42	**0.46**		0.68	**0.44**	0.41
MRR		0.36	0.35	0.32		0.43	0.35	0.25
MAP		0.06	0.04	0.04		0.08	0.04	0.05
P@1	EWC	0.05	0.04	0.03	EWC	0.07	0.04	0.05
P@5		0.28	**0.22**	0.21		**0.40**	0.19	**0.22**
P@10		**0.65**	**0.44**	0.42		**0.75**	0.40	**0.45**
MRR		**0.55**	0.17	**0.44**		**0.51**	0.17	0.38
MAP		**0.13**	**0.08**	**0.11**		**0.13**	0.08	**0.13**
P@1	QARC	**0.53**	0.15	0.36	QARC	**0.47**	**0.15**	0.29
P@5		0.17	0.09	0.18		0.18	0.08	0.18
P@10		0.15	0.10	0.18		0.16	0.11	0.20
MRR		0.48	0.17	0.44		**0.51**	0.17	**0.48**
MAP		0.12	**0.08**	**0.11**		**0.13**	0.08	**0.13**
P@1	QARCL	0.44	0.15	**0.37**	QARCL	**0.47**	**0.15**	**0.37**
P@5		0.15	0.10	0.15		0.18	0.08	0.20
P@10		0.14	0.11	0.15		0.16	0.11	0.20
MRR		0.54	0.18	0.40		0.49	0.17	0.40
MAP		**0.13**	**0.08**	0.09		**0.13**	0.08	0.12
P@1	QD	**0.53**	**0.16**	0.29	QD	0.42	**0.15**	0.28
P@5		0.16	0.10	0.15		0.19	0.10	0.18
P@10		0.14	0.11	0.17		0.17	0.10	0.20

out entropy), indicating their capability to retain their effectiveness on stationary data even when the software project undergoes significant changes (non-stationary data) and concept drift. This is due to the CL agents' capability of protecting knowledge from previous cycles during training. For non-stationary data in the Eclipse (top@1, top@5, top@10, and MRR metrics) and Tomcat (top@1, top@5, top@10, and MRR metrics), as well as the Birt (top@5, MRR metrics), SWT (top@1, top@5, top@10, and MRR metrics) and AspectJ (top@1, MRR metrics), CLEAR and EWC outperform FLIM by at least 14%. Similarly, CLEAR and EWC significantly outperform RLOCATOR (with and without entropy) on non-stationary data in the AspectJ project using top@5, top@10, and MRR metrics. For the Birt project, CLEAR outperforms RLOCATOR with entropy by 14% (top@5, top@10, and MRR metrics), while EWC outperforms RLOCATOR without entropy by 87% (MRR metric). In the Eclipse project, CLEAR outperforms RLOCATOR (with and without entropy) by 14% (top@5 and MRR metrics). Furthermore, EWC significantly outperforms RLOCATOR without entropy

Table 6.10 Algorithms evaluation across PDE and Zxing projects with different data granularities. In bold are the best average performances per algorithm per metric for each project and data granularity. "-files"=changeset-files. "P@"=top@.

	-files	PDE	Zxing	**hunks**	PDE	Zxing
MRR	CLEAR	**0.39**	0.50	CLEAR	**0.42**	**0.55**
MAP		0.08	0.12		0.08	0.11
P@1		**0.08**	0.11		0.08	0.11
P@5		0.41	0.61		**0.43**	0.58
P@10		0.87	**1.00**		0.82	**1.00**
MRR	EWC	**0.39**	**0.53**	EWC	0.38	**0.55**
MAP		**0.09**	0.12		0.09	0.12
P@1		0.09	0.13		0.08	0.12
P@5		**0.44**	0.59		**0.43**	**0.59**
P@10		**0.88**	**1.00**		**0.86**	**1.00**
MRR	QARC	0.17	0.24	QARC	0.21	0.44
MAP		0.05	0.18		0.08	**0.29**
P@1		0.07	0.20		0.10	**0.40**
P@5		0.09	0.14		0.17	0.31
P@10		0.12	0.34		0.19	0.32
MRR	QARCL	0.26	0.35	QARCL	0.26	0.47
MAP		0.08	**0.21**		0.08	**0.29**
P@1		0.17	0.20		0.17	**0.40**
P@5		0.11	**0.39**		0.11	0.39
P@10		0.18	0.38		0.18	0.42
MRR	QD	0.21	0.29	QD	0.26	0.32
MAP		0.07	0.15		**0.10**	0.14
P@1		0.13	0.20		**0.20**	0.20
P@5		0.15	0.27		0.18	0.29
P@10		0.15	0.38		0.17	0.30

by 14% (top@10 and MRR metrics) in the Tomcat project. Based on the obtained results, CL agents show that they can adapt to software project changes that lead to non-stationary data distributions, ensuring that the bug localization process remains effective over time.

Table 6.9 and 6.10 report the performance of the CL agents and FBL-BERT for QARC, QARCL and QD at changeset-files and hunks level across SWT, AspectJ, Tomcat, PDE and Zxing software projects - the performance on stationary and non-stationary data are averaged for comparison with FBL-BERT. At the changeset-files level, the CL agents significantly outperform FBL-BERT for QARC, QARCL, and QD on AspectJ (by at least 80% in top@5 and top@10), on Zxing (by at least 89% in top@10) and on SWT (by at least 80% in top@10). On Tomcat, the CL agents significantly outperform QARCL and QD by at least 22% in both top@5 and top@10. On PDE, the CL agents significantly outperform QARCL and QD by at least 13% in top@1, top@5 and top@10. At the hunk level across the SWT, Tomcat, PDE,

Table 6.11 Algorithms evaluation across software projects with different data granularity when incorporating bug-inducing factors on CL agents. In bold are the best average performances per algorithm per metric for each project and data granularity. "-files"=changeset-files. "P@"=top@.

	-files	SWT	AspectJ	Tomcat	**hunks**	SWT	AspectJ	Tomcat
MRR	CLEAR+Reg.	0.33	0.32	0.29	CLEAR+Reg.	0.32	0.33	0.22
MAP		0.07	0.04	0.05		0.07	0.05	0.05
P@1		0.07	0.04	0.05		0.07	0.05	0.06
P@5		0.33	0.21	**0.25**		**0.34**	**0.23**	**0.25**
P@10		**0.67**	0.43	0.38		0.66	**0.45**	0.39
MRR	EWC+Reg.	0.42	**0.42**	0.22	EWC+Reg.	0.35	**0.39**	0.26
MAP		0.07	0.05	0.04		0.06	0.04	0.05
P@1		0.08	0.05	0.04		0.08	0.04	0.05
P@5		**0.36**	**0.24**	0.22		0.33	0.22	0.24
P@10		**0.71**	**0.47**	**0.42**		**0.67**	0.44	**0.48**
MRR	QARC	**0.55**	0.17	**0.44**	QARC	**0.51**	0.17	0.38
MAP		**0.13**	**0.08**	**0.11**		**0.13**	0.08	**0.13**
P@1		**0.53**	0.15	0.36		**0.47**	**0.15**	0.29
P@5		0.17	0.09	0.18		0.18	0.08	0.18
P@10		0.15	0.10	0.18		0.16	0.11	0.20
MRR	QARCL	0.48	0.17	0.44	QARCL	**0.51**	0.17	**0.48**
MAP		0.12	**0.08**	**0.11**		**0.13**	0.08	**0.13**
P@1		0.44	0.15	**0.37**		**0.47**	**0.15**	**0.37**
P@5		0.15	0.10	0.15		0.18	0.08	0.20
P@10		0.14	0.11	0.15		0.16	0.11	0.20
MRR	QD	0.54	0.18	0.40	QD	0.49	0.17	0.40
MAP		**0.13**	**0.08**	0.09		**0.13**	0.08	0.12
P@1		**0.53**	**0.16**	0.29		0.42	**0.15**	0.28
P@5		0.16	0.10	0.15		0.19	0.10	0.18
P@10		0.14	0.11	0.17		0.17	0.10	0.20

Zxing and AspectJ projects, the CL agents significantly outperform QARC, QARCL, and QD in top@10 by at least 67%. However, FBL-BERT for QARC, QARCL, and QD shows better performance than CL agents in terms of MRR and top@1 at both the changeset-files and hunk level across all studied projects. FBL-BERT first accounts for all changeset-files and hunks (stationary and non-stationary) committed to a software project up to the present point in time then uses the InVerted File with Product Quantization (IVFPQ) [207] algorithm to refine the search space by identifying the top-N most relevant ones. This allows FBL-BERT to focus on a smaller, more relevant set of changeset-files (or hunks) for bug localization, improving precision. The refined search helps FBL-BERT to quickly identify the most relevant changeset-files (or hunks), leading to better MRR and top@1 performance compared to the CL agents. Nevertheless, the consistent performance of the CL agents, with top@5 and top@10 metrics, indicates their effectiveness in capturing a high proportion of the most relevant changesets within the top-ranked selections.

The performance difference on stationary vs. non-stationary at changeset-files level for CLEAR is up to 14% on Eclipse, 40% on the Zxing, 38% on Birt, 59% on PDE, 57% on

Table 6.12 Algorithms evaluation across PDE and Zxing projects with different data granularity when incorporating bug-inducing factors on CL agents. In bold are the best average performances per algorithm per metric for each project and data granularity. "-files"=changeset-files. "P@"=top@.

	-files	PDE	Zxing	**hunks**	PDE	Zxing
MRR	CLEAR+Reg	**0.37**	0.46	CLEAR+Reg	**0.35**	0.51
MAP		0.08	0.12		0.09	0.12
P@1		0.08	0.12		0.09	0.13
P@5		0.41	**0.62**		0.40	**0.62**
P@10		0.77	**1.00**		0.71	**1.00**
MRR	EWC+Reg	0.36	**0.61**	EWC+Reg	0.33	**0.56**
MAP		**0.09**	0.12		0.09	0.12
P@1		0.10	0.12		0.09	0.11
P@5		**0.43**	0.61		**0.44**	0.59
P@10		**0.85**	1.00		**0.87**	**1.00**
MRR	QARC	0.17	0.24	QARC	0.21	0.44
MAP		0.05	0.18		0.08	**0.29**
P@1		0.07	**0.20**		0.10	**0.40**
P@5		0.09	0.14		0.17	0.31
P@10		0.12	0.34		0.19	0.32
MRR	QARCL	0.26	0.35	QARCL	0.26	0.47
MAP		0.08	**0.21**		0.08	**0.29**
P@1		**0.17**	**0.20**		0.17	**0.40**
P@5		0.11	0.39		0.11	0.39
P@10		0.18	0.38		0.18	0.42
MRR	QD	0.21	0.29	QD	0.26	0.32
MAP		0.07	0.15		**0.10**	0.14
P@1		0.13	**0.20**		**0.20**	0.20
P@5		0.15	0.27		0.18	0.29
P@10		0.15	0.38		0.17	0.30

Tomcat, 90% AspectJ, and 109% on SWT projects with all ranking metrics. As for EWC, it is up to 12% on Tomcat, 15% on AspectJ, 19% on Zxing, 33% on PDE, 45% on Birt, 115% on SWT, and 63% on the Eclipse projects at changeset-files level. At the hunks level, the performance difference on stationary vs. non-stationary for CLEAR is up to 15% on Birt, 19% on PDE, 9% on Zxing, 17% on Eclipse, 30% on Tomcat, 22% on AspectJ, and 72% on SWT projects. As for EWC, the performance difference is up to 28% on Tomcat, 45% on PDE, 10% on Zxing, 44% on AspectJ, 71% on Birt, 100% on SWT, and 41% on the Eclipse projects at the hunks level. CLEAR seems to keep old and new policies close together better than EWC for all projects and all data granularities except Tomcat (at the changeset-files and hunks levels), AspectJ (at the changeset-files level), PDE (at the changeset-files level) and Zxing (at the changeset-files level). A possible explanation lies in the complexity of the projects. For complex projects such as Tomcat (given its large number of changed files as reported in Table 6.2), it can become challenging for the behavioral cloning used by CLEAR to keep track of all the new and old experiences while preventing the DNNs from drifting.

Table 6.13 Ablation study 1: different bug report metrics added to CLEAR. In bold are the best average performances per algorithm per metric for each project.

	Eclipse		SWT		Birt		AspectJ		Tomcat	
	top@1									
	ST	NS	ST	NS	ST	NS	ST	NS	ST	NS
CLEAR+Churn	0.04	0.05	0.05	0.14	**0.07**	0.06	0.06	0.03	0.04	0.04
CLEAR+PRE	**0.05**	**0.05**	0.04	0.11	0.06	0.06	**0.06**	**0.06**	0.03	0.04
CLEAR+Reg.	0.04	0.04	**0.05**	**0.13**	0.05	**0.06**	0.04	0.04	**0.04**	**0.06**
	top@5									
	ST	NS	ST	NS	ST	NS	ST	NS	ST	NS
CLEAR+Churn	**0.24**	0.20	0.23	**0.49**	0.24	0.23	0.23	0.20	0.23	0.22
CLEAR+PRE	0.19	0.18	0.23	0.47	0.24	**0.29**	0.23	**0.25**	0.21	0.27
CLEAR+Reg.	0.21	**0.20**	**0.24**	0.41	**0.25**	0.25	**0.24**	0.17	**0.21**	**0.29**
	top@10									
	ST	NS	ST	NS	ST	NS	ST	NS	ST	NS
CLEAR+Churn	**0.44**	**0.42**	**0.44**	0.82	**0.54**	0.46	**0.48**	0.40	**0.44**	0.41
CLEAR+PRE	0.35	0.32	0.43	0.83	0.47	**0.47**	0.45	**0.49**	0.39	**0.46**
CLEAR+Reg.	0.35	0.31	0.43	**0.91**	0.49	**0.47**	**0.48**	0.37	0.32	0.43
	MAP									
	ST	NS	ST	NS	ST	NS	ST	NS	ST	NS
CLEAR+Churn	0.04	0.04	0.05	0.09	0.05	0.04	0.05	0.04	0.04	0.04
CLEAR+PRE	0.04	0.04	0.04	0.08	0.05	**0.06**	0.05	**0.05**	0.04	0.05
CLEAR+Reg.	**0.04**	**0.04**	**0.05**	**0.09**	**0.05**	0.05	**0.05**	0.04	**0.04**	**0.06**
	MRR									
	ST	NS	ST	NS	ST	NS	ST	NS	ST	NS
CLEAR+Churn	0.26	0.24	0.23	0.40	**0.44**	**0.42**	0.33	0.18	**0.42**	0.23
CLEAR+PRE	0.26	0.22	**0.39**	0.30	0.36	0.34	0.36	**0.33**	0.39	**0.28**
CLEAR+Reg.	**0.34**	**0.24**	0.24	**0.41**	0.36	0.33	**0.39**	0.26	0.31	**0.28**

Regarding the computational efforts of the CL agents, as shown in Table 6.19 and 6.20, they require up to 5x less time to get trained at different data granularities compared to the baseline studies across all software projects except the Zxing project (with the FBL-BERT baseline study). The Zxing project has the fewest number of bug reports (see Table 6.2) compared to other software projects, which explains its fastest fine-tuning time on a BERT-based approach like FBL-BERT.

Finding 2: The CL agents demonstrate substantial performance enhancements over RLOCATOR (with and without entropy), on stationary data for the AspectJ project, as measured by top@5, top@10, MRR, and MAP metrics, while achieving comparable performance on other projects. On non-stationary data, CLEAR and EWC outperform FLIM and RLOCATOR (with and without entropy) on AspectJ and SWT projects as measured by top@5, top@10, and MRR metrics. Finally, they significantly outperform FBL-BERT with the top@5 and top@10 metrics across AspectJ, SWT, PDE, Zxing, and Tomcat at the hunks level, while FBL-BERT shows better performance as measured by MRR and top@1 metrics.

Table 6.14 Ablation study 2: different bug report metrics added to EWC. In bold are the best average performances per algorithm per metric for each project.

	Eclipse		SWT		Birt		AspectJ		Tomcat	
	top@1									
	ST	NS	ST	NS	ST	NS	ST	NS	ST	NS
EWC+Churn	0.03	0.02	0.04	**0.13**	0.05	0.05	**0.04**	0.04	0.04	0.04
EWC+PRE	**0.04**	**0.04**	**0.05**	0.06	0.05	0.03	**0.04**	0.04	**0.05**	**0.05**
EWC+Reg.	**0.04**	**0.04**	**0.05**	**0.13**	**0.06**	**0.06**	**0.04**	**0.05**	0.04	0.04
	top@5									
	ST	NS	ST	NS	ST	NS	ST	NS	ST	NS
EWC+Churn	**0.22**	**0.19**	0.23	0.41	0.26	0.19	0.21	0.20	0.22	0.25
EWC+PRE	0.21	**0.19**	0.22	0.39	**0.27**	0.19	0.24	0.20	**0.25**	**0.26**
EWC+Reg.	**0.22**	**0.19**	**0.25**	**0.47**	**0.27**	**0.20**	**0.26**	**0.23**	0.20	0.24
	top@10									
	ST	NS	ST	NS	ST	NS	ST	NS	ST	NS
EWC+Churn	**0.44**	**0.40**	0.45	0.78	0.52	0.40	0.43	0.41	0.44	0.47
EWC+PRE	0.43	0.39	0.44	0.81	**0.55**	0.40	0.48	0.42	**0.50**	**0.51**
EWC+Reg.	**0.44**	0.38	**0.49**	**0.94**	0.52	**0.41**	**0.49**	**0.44**	0.42	0.42
	MAP									
	ST	NS	ST	NS	ST	NS	ST	NS	ST	NS
EWC+Churn	**0.04**	**0.04**	0.04	0.08	**0.05**	**0.04**	0.04	**0.04**	0.04	**0.05**
EWC+PRE	**0.04**	**0.04**	0.04	0.08	**0.05**	**0.04**	**0.05**	**0.04**	**0.05**	**0.05**
EWC+Reg.	**0.04**	**0.04**	**0.05**	**0.10**	**0.05**	**0.04**	**0.05**	**0.04**	0.04	**0.05**
	MRR									
	ST	NS	ST	NS	ST	NS	ST	NS	ST	NS
EWC+Churn	**0.31**	0.23	0.33	0.44	0.23	**0.38**	0.36	0.17	0.25	**0.32**
EWC+PRE	**0.31**	**0.28**	0.34	0.44	0.38	0.32	0.30	0.17	**0.29**	0.21
EWC+Reg.	0.26	**0.28**	**0.36**	**0.48**	**0.45**	0.26	**0.43**	**0.42**	0.18	0.26

6.3.3 RQc$_3$: Can prior knowledge about bug-inducing factors improve the performance of CL techniques?

Table 6.3, 6.4, 6.5, 6.6, 6.7, and 6.8 show the performance of the CL agents with regression across the studied software projects at the changeset-files and hunks levels. The performance difference on stationary vs. non-stationary data at the changeset-files level for CLEAR with regression is up to 18% on the Birt project (see Table 6.4), 34% on the Eclipse project (see Table 6.5), 29% on the Tomcat project (see Table 6.3), 40% AspectJ project (see Table 6.6), 15% on the Zxing project (see Table 6.8), 21% PDE project (see Table 6.8), and 85% on SWT project (see Table 6.7) as measured by all ranking metrics. As for EWC with regression, the performance difference is up to 36% on the Tomcat project, 10% on the AspectJ project, 53% on the Birt project, 62% on SWT, 6% on the Zxing project (see Table 6.8), 19% PDE project (see Table 6.8) and 14% on the Eclipse project as measured by all metrics at the changeset-files level. The performance difference on stationary vs. non-stationary at the hunks level for CLEAR with regression is up to 27% on the Birt project, 30% on the Eclipse project, 21% on the Tomcat project, 18% on the AspectJ project, 9% on the Zxing project

Table 6.15 Ablation study 3: different bug report metrics added to CLEAR. In bold are the best average performances per algorithm per metric for each project.

	Eclipse		SWT		Birt		AspectJ		Tomcat	
	top@1									
	ST	NS	ST	NS	ST	NS	ST	NS	ST	NS
CLEAR+LOC	**0.04**	**0.04**	0.04	0.12	0.05	0.03	**0.04**	0.02	0.05	0.04
CLEAR+MLOC	**0.04**	**0.04**	0.04	0.12	**0.07**	0.03	**0.04**	0.03	0.04	0.08
CLEAR+VG	**0.04**	**0.04**	**0.05**	0.12	0.06	0.04	**0.04**	**0.04**	0.05	**0.09**
CLEAR+ALL	**0.04**	**0.04**	**0.05**	0.10	0.05	0.04	**0.04**	0.03	**0.06**	0.05
CLEAR+Reg.	**0.04**	**0.04**	**0.05**	**0.13**	0.05	**0.06**	**0.04**	**0.04**	0.04	0.06
	top@5									
	ST	NS	ST	NS	ST	NS	ST	NS	ST	NS
CLEAR+LOC	**0.23**	0.23	0.23	0.36	**0.28**	0.18	0.21	0.14	0.22	0.21
CLEAR+MLOC	0.22	0.17	0.21	0.40	0.26	0.20	0.23	0.18	**0.24**	**0.30**
CLEAR+VG	0.20	0.19	0.23	**0.52**	0.27	**0.29**	0.22	**0.27**	0.20	0.29
CLEAR+ALL	**0.23**	**0.20**	**0.24**	0.35	0.25	0.23	0.20	0.22	0.22	0.26
CLEAR+Reg.	0.21	**0.20**	**0.24**	0.41	0.25	0.25	**0.24**	0.17	0.21	0.29
	top@10									
	ST	NS	ST	NS	ST	NS	ST	NS	ST	NS
CLEAR+LOC	0.43	**0.44**	0.45	0.79	0.53	0.35	0.47	0.33	**0.45**	0.40
CLEAR+MLOC	**0.47**	0.39	0.43	0.75	**0.54**	0.42	0.45	0.38	0.44	**0.51**
CLEAR+VG	0.35	0.34	0.43	0.87	0.43	**0.54**	0.40	0.45	0.28	0.43
CLEAR+ALL	0.45	0.42	**0.46**	0.73	0.50	0.42	0.44	**0.48**	0.42	0.47
CLEAR+Reg.	0.35	0.31	0.43	**0.91**	0.49	0.47	**0.48**	0.37	0.32	0.43
	MAP									
	ST	NS	ST	NS	ST	NS	ST	NS	ST	NS
CLEAR+LOC	0.04	**0.05**	**0.05**	**0.09**	**0.05**	0.04	**0.05**	0.03	0.04	0.04
CLEAR+MLOC	0.04	0.04	0.04	0.07	**0.05**	0.04	**0.05**	0.04	**0.04**	0.05
CLEAR+VG	0.04	0.04	0.04	**0.09**	**0.05**	**0.05**	**0.05**	**0.10**	**0.04**	**0.08**
CLEAR+ALL	**0.04**	0.04	**0.05**	**0.09**	**0.05**	0.04	0.04	0.04	**0.04**	0.05
CLEAR+Reg.	**0.04**	**0.04**	**0.05**	**0.09**	**0.05**	**0.05**	**0.05**	0.04	**0.04**	0.06
	MRR									
	ST	NS	ST	NS	ST	NS	ST	NS	ST	NS
CLEAR+LOC	0.33	0.37	0.26	0.41	0.27	0.26	0.35	0.34	0.25	0.24
CLEAR+MLOC	0.31	**0.40**	0.20	**0.48**	**0.51**	0.22	**0.39**	0.30	0.32	0.24
CLEAR+VG	0.33	0.23	**0.42**	0.36	0.32	**0.36**	0.28	**0.39**	0.26	**0.37**
CLEAR+ALL	0.30	0.28	0.32	0.24	0.36	0.31	0.32	0.24	**0.33**	0.29
CLEAR+Reg.	**0.34**	0.24	0.24	0.41	0.36	0.33	**0.39**	0.26	0.31	**0.28**

(see Table 6.8), 5% on the PDE project (see Table 6.8) and 74% on SWT as measured by all metrics. As for EWC with regression, the performance difference is up to 43% on the Tomcat project, 33% on the AspectJ project, 98% on the Birt project, 114% on the SWT project, 8% on the Zxing project (see Table 6.8), 3% on the PDE project (see Table 6.8), and 22% on the Eclipse project as measured by all metrics at the hunks level.

Our results show an improvement in performance difference on stationary and non-stationary data for up to 38% across AspectJ, Birt, PDE, Zxing, and Tomcat projects when incorporating bug-inducing factors as prior knowledge of CLEAR at the changeset-files and hunks levels. With EWC, our results show a performance improvement of up to 42% on AspectJ, PDE, Zxing, and Eclipse projects at the changeset-files and hunks levels. Suggesting that

Table 6.16 Ablation study 4: different bug report metrics added to EWC. In bold are the best average performances per algorithm per metric for each project.

	Eclipse		SWT		Birt		AspectJ		Tomcat	
	top@1									
	ST	NS	ST	NS	ST	NS	ST	NS	ST	NS
EWC+LOC	**0.04**	**0.04**	**0.05**	**0.21**	0.04	0.05	0.05	0.04	0.04	0.04
EWC+MLOC	**0.04**	0.03	**0.05**	0.17	0.05	**0.06**	0.05	0.04	0.04	0.04
EWC+VG	**0.04**	0.03	0.04	0.17	0.05	0.05	0.04	0.04	**0.05**	**0.05**
EWC+ALL	**0.04**	0.03	**0.05**	0.13	0.04	0.03	**0.06**	0.02	**0.05**	0.03
EWC+Reg.	**0.04**	**0.04**	**0.05**	0.13	**0.06**	**0.06**	0.04	**0.05**	0.04	0.04
	top@5									
	ST	NS	ST	NS	ST	NS	ST	NS	ST	NS
EWC+LOC	**0.24**	0.16	0.23	**0.57**	**0.27**	0.18	0.25	0.21	**0.24**	0.21
EWC+MLOC	0.23	0.18	0.22	0.44	0.26	**0.20**	0.25	0.21	0.22	0.18
EWC+VG	0.19	0.18	0.22	0.50	0.25	0.19	0.23	0.21	**0.24**	0.21
EWC+ALL	0.22	**0.20**	0.21	0.52	0.23	**0.20**	0.24	0.22	0.21	0.22
EWC+Reg.	0.22	0.19	**0.25**	0.47	**0.27**	**0.20**	**0.26**	**0.23**	0.20	**0.24**
	top@10									
	ST	NS	ST	NS	ST	NS	ST	NS	ST	NS
EWC+LOC	**0.45**	0.38	0.46	0.97	0.51	0.39	0.47	0.38	**0.48**	0.44
EWC+MLOC	**0.45**	**0.39**	0.44	0.84	**0.53**	0.39	**0.50**	0.39	0.42	0.41
EWC+VG	0.43	0.36	0.42	**1.0**	0.52	0.38	0.49	0.40	0.45	**0.45**
EWC+ALL	**0.45**	**0.39**	0.43	**1.0**	0.48	**0.45**	0.470	**0.47**	0.47	**0.45**
EWC+Reg.	0.44	0.38	**0.49**	0.94	0.52	0.41	0.49	0.44	0.42	0.42
	MAP									
	ST	NS	ST	NS	ST	NS	ST	NS	ST	NS
EWC+LOC	**0.04**	0.03	0.04	**0.10**	**0.05**	**0.04**	0.04	**0.04**	**0.04**	0.04
EWC+MLOC	**0.04**	0.03	0.04	0.08	**0.05**	**0.04**	**0.05**	**0.04**	**0.04**	0.04
EWC+VG	**0.04**	0.03	0.04	0.09	**0.05**	0.03	0.04	**0.04**	**0.04**	0.04
EWC+ALL	**0.04**	0.03	0.04	**0.10**	0.04	**0.04**	0.04	**0.04**	**0.04**	0.04
EWC+Reg.	**0.04**	**0.04**	**0.05**	**0.10**	**0.05**	**0.04**	**0.05**	**0.04**	**0.04**	**0.05**
	MRR									
	ST	NS	ST	NS	ST	NS	ST	NS	ST	NS
EWC+LOC	0.26	0.24	0.40	0.48	0.27	0.20	0.28	0.19	0.27	0.28
EWC+MLOC	0.25	0.34	0.42	0.38	0.36	**0.38**	0.34	**0.42**	0.26	0.34
EWC+VG	0.25	0.30	**0.48**	**0.62**	0.42	0.24	0.32	0.18	0.21	0.40
EWC+ALL	**0.42**	**0.44**	0.34	0.59	**0.52**	0.37	0.35	0.13	**0.33**	**0.43**
EWC+Reg.	0.26	0.28	0.36	0.48	0.45	0.26	**0.43**	**0.42**	0.18	0.26

Table 6.17 Ablation study 5: different bug report metrics added to CLEAR. In bold are the best average performances per algorithm per metric for each project.

	PDE		Zxing	
	top@1			
	ST	NS	ST	NS
CLEAR+Churn	**0.08**	0.08	0.11	**0.12**
CLEAR+PRE	0.06	**0.10**	0.11	0.11
CLEAR+Reg.	**0.08**	0.08	**0.12**	**0.12**
	top@5			
	ST	NS	ST	NS
CLEAR+Churn	0.40	0.41	0.62	0.56
CLEAR+PRE	0.37	**0.47**	0.61	0.55
CLEAR+Reg.	**0.42**	0.39	**0.63**	**0.61**
	top@10			
	ST	NS	ST	NS
CLEAR+Churn	0.64	0.67	**1.00**	**1.00**
CLEAR+PRE	0.72	**0.90**	**1.00**	**1.00**
CLEAR+Reg.	**0.77**	0.77	**1.00**	**1.00**
	MAP			
	ST	NS	ST	NS
CLEAR+Churn	**0.08**	0.09	**0.12**	0.11
CLEAR+PRE	0.07	**0.10**	**0.12**	0.11
CLEAR+Reg.	**0.08**	0.08	**0.12**	**0.12**
	MRR			
	ST	NS	ST	NS
CLEAR+Churn	**0.49**	0.34	**0.62**	0.47
CLEAR+PRE	0.36	0.38	0.56	0.48
CLEAR+Reg.	0.33	**0.41**	0.42	**0.49**

prior knowledge about bug-inducing factors can provide a critical appropriate context that helps the CL agents better understand and adapt to changes in the data of software projects.

The CL agents with regression significantly outperform RLOCATOR (with and without entropy) on stationary data of Birt, and AspectJ projects at the changeset-files level, as measured by the MRR metric by up to 50%. On stationary data of the other projects (at the changeset-files level), they achieve comparable performance against RLOCATOR (with and without entropy) as measured by top@1 and top@5 metrics. Given that we adopted the same environment as RLOCATOR, we enhanced it as well with prior knowledge of bug-inducing factors. Table 6.3, 6.4, 6.6, 6.7, and 6.5 show the performance of RLOCATOR (with and without entropy) across Tomcat, Birt, AspectJ, SWT and Eclipse projects when incorporating prior knowledge of bug inducing factors through regression at the changeset-files level. Our results show that the CL agents with regression significantly outperform RLOCATOR (with and without entropy) with regression on stationary data of Birt, AspectJ projects with the MRR metric by up to 29%, suggesting CLEAR and EWC (with regression) utilize bug-inducing factors better than RLOCATOR. This is likely due to our CL mechanisms that are

Table 6.18 Ablation study 6: different bug report metrics added to EWC. In bold are the best average performances per algorithm per metric for each project.

	PDE		Zxing	
	top@1			
	ST	NS	ST	NS
EWC+Churn	**0.10**	0.08	**0.13**	0.09
EWC+PRE	**0.10**	0.07	0.12	0.11
EWC+Reg.	**0.10**	**0.09**	**0.13**	**0.12**
	top@5			
	ST	NS	ST	NS
EWC+Churn	**0.46**	0.42	0.60	0.54
EWC+PRE	0.43	0.44	**0.64**	0.55
EWC+Reg.	0.43	**0.44**	0.63	**0.59**
	top@10			
	ST	NS	ST	NS
EWC+Churn	**0.85**	0.84	**1.00**	**1.00**
EWC+PRE	0.83	**0.89**	**1.00**	**1.00**
EWC+Reg.	**0.85**	0.86	**1.00**	**1.00**
	MAP			
	ST	NS	ST	NS
EWC+Churn	**0.09**	0.08	0.12	0.11
EWC+PRE	0.08	**0.09**	**0.13**	0.11
EWC+Reg.	**0.09**	**0.09**	**0.13**	**0.12**
	MRR			
	ST	NS	ST	NS
EWC+Churn	**0.52**	0.36	**0.71**	**0.64**
EWC+PRE	0.43	0.36	0.49	0.45
EWC+Reg.	0.32	**0.39**	0.58	**0.64**

capable of retaining and applying learned knowledge.

On non-stationary data, CLEAR with regression significantly outperforms RLOCATOR (with and without regression as well as with entropy) on Tomcat (with top@1 and top@5 metrics), Birt (top@5 and MRR metrics), AspectJ (top@1 metric), and Eclipse (top@5 metric) projects at the changeset-files level. Similarly, EWC CLEAR with regression significantly outperforms RLOCATOR (with and without regression as well as with entropy) on Birt (top@1 metric), AspectJ (top@1, top@5, top@10 and MRR metrics), and Eclipse (top@1 metric) projects at the changeset-files level. As of the comparison with FLIM, both CLEAR and EWC with regression perform better on non-stationary data of all projects except AspectJ (top@5 and top@10 metrics), and Birt (top@1, top@10, and MAP metrics). Only a small subset of source code files in the Tomcat, SWT, and AspectJ projects have Pre-released Bugs greater than 0 (see Section 6.3.1). This limitation prevents FLIM from capturing sufficient information to accurately localize bugs in non-stationary settings. In contrast, our CL agents demonstrate better adaptation to new knowledge in these non-stationary settings. Similarly to their versions without regression, CLEAR and EWC require much less computational

Table 6.19 Computational effort (mean ± std) in hours of studied algorithms across software projects with changeset-files data granularity. "RLO." refers to the baseline RLOCATOR. "+ Reg" indicates the inclusion of logistic regression, while "+ Ent" denotes the addition of entropy. The best and second best average performances are highlighted in dark grey and light grey, respectively.

	AspectJ	SWT	Birt	Eclipse	Tomcat
FLIM	16.1 ± 2.4	11.2 ± 0.1	11.4 ± 0.4	12.0 ± 0.1	10.5 ± 0.2
RLO.	8.5 ± 0.07	13.6 ± 1.6	10.7 ± 0.8	12.4 ± 0.2	9.1 ± 0.04
RLO. + Reg.	4.4 ± 1.0	14.0 ± 1.5	10.1 ± 0.7	13.1 ± 1.5	9.0 ± 0.2
RLO. + Ent.	12.6 ± 1.2	16.8 ± 2.2	15.6 ± 2.3	12.3 ± 0.1	9.8 ± 0.9
RLO. + Ent. + Reg.	11.4 ± 1.2	16.7 ± 0.4	15.8 ± 0.4	10.6 ± 0.05	11.0 ± 1.5
CLEAR	0.32±0.01	0.70±0.05	0.43 ± 0.00	0.52 ± 0.00	0.71±0.05
CLEAR + Reg.	0.68 ± 0.04	1.38 ± 0.30	0.80 ± 0.02	3.15 ± 0.31	1.46 ± 0.09
EWC	0.36±0.05	0.77±0.02	0.47 ± 0.01	0.59 ± 0.02	0.79±0.02
EWC + Reg.	0.60 ± 0.09	1.89 ± 0.50	0.88 ± 0.06	2.09 ± 1.04	1.48 ± 0.06

effort (up to 5x less effort).

Table 6.11 and 6.12 report the performance of the CL agents with regression and FBL-BERT for QARC, QARCL and QD at changeset-files and hunks levels across SWT, AspectJ, Tomcat, PDE, and Zxing software projects - the performance on stationary and non-stationary data are averaged for comparison with FBL-BERT. Compared to FBL-BERT for QARC, QARCL, and QD, the CL agents with regression show similar performance as their performance without regression at both changeset-files and hunks levels. They perform significantly better than the FBL-BERT metrics across the SWT project (with top@10), Tomcat project (with top@10 metric), AspectJ project (with top@5 and top@10 metrics), PDE project (with top@5 and top@10 metrics) and Zxing project (with top@5 and top@10 metrics).

Given the diverse nature of software projects, developers must select the CL agent that best fits their project. Our results indicate significant performance differences between the CL agents, suggesting that the characteristics of the software project might be influencing this. To better assist developers in choosing the appropriate CL agent, we evaluate the overall forgetting performance of EWC and CLEAR with and without regression across the studied software projects at the changeset-files and hunks levels. Table 6.23 shows the performance of the CL agents in terms of forgetting across the software projects at the changeset-files and hunks levels. The range of forgetting values obtained when incorporating bug-inducing factors into the CL agents is similar to the results reported by Powers et al. [197]. This indicates that EWC+Reg. and CLEAR+Reg. at the changeset-files and hunks levels are more effective compared to EWC and CLEAR at mitigating forgetting while addressing the concept drift associated with data across software projects, which confirms the findings of our study. Moreover, the forgetting values of CL agents suggest that for complex projects

Table 6.20 Computational effort (mean ± std) in hours of studied algorithms across software projects with different data granularities. The best and second best average performances are highlighted in dark grey and light grey, respectively.

	Granularity	SWT	Tomcat	AspectJ	PDE	Zxing
CLEAR	hunks	0.54±0.01	0.49±0.03	0.32±0.08	0.41±0.0	0.26±0.0
	changeset-files	0.77±0.02	0.79±0.02	0.36±0.05	0.42±0.0	0.28±0.0
EWC	hunks	0.55±0.03	0.56±0.02	0.33±0.04	0.44±0.0	0.29±0.0
	changeset-files	1.25±0.0	1.68±0.1	3.18±0.07	0.46±0.0	0.31±0.0
CLEAR+Reg.	hunks	0.66±0.01	0.76±0.03	0.49±0.0	0.44±0.02	0.26±0.0
	changeset-files	1.38±0.3	1.46±0.09	0.68±0.04	0.49±0.01	0.27±0.0
EWC+Reg.	hunks	0.71±0.01	0.80±0.01	0.54±0.02	0.45±0.0	0.28±0.0
	changeset-files	1.89±0.5	1.48±0.06	0.60±0.09	0.47±0.0	0.30±0.0
QARC	hunks	2.34±0.1	1.97±0.01	3.17±0.1	1.56±0.0	0.08±0.0
	changeset-files	1.20±0.02	1.60±0.0	3.28±0.07	0.97±0.0	0.05±0.0
QARCL	hunks	2.31±0.12	1.88±0.02	3.16±0.1	1.43±0.02	0.08±0.0
	changeset-files	1.20±0.05	1.59±0.01	3.23±0.1	0.96±0.0	0.05±0.0
QD	hunks	2.13±0.0	1.82±0.03	3.29±0.1	1.41±0.0	0.08±0.0
	changeset-files	1.19±0.04	1.59±0.01	3.22±0.12	0.91±0.0	0.06±0.0

such as Tomcat — given its high number of file changes in commits between the time a bug is fixed and is reported (reported in Table 6.2), any of the CL agents with regression applied might be more suited.

Finding 3: Incorporating prior knowledge about bug-inducing factors through regression improves the performance of both CLEAR and EWC, outperforming their non-regression counterparts on four out of seven projects at the changeset-files level and three out of seven projects at the hunks level across all ranking metrics.

6.3.4 Ablation study on using bug-inducing factors

To validate our regression approach for enhancing CL agents, we analyze the impact of different bug-inducing factor configurations, as shown in Table 6.1, when added to the reward function. For each CL agent (i.e., CLEAR and EWC), we consider as an additional component of their reward functions 1) each bug-inducing factor value (i.e., LOC, MLOC, VG, PRE, Churn), 2) the sum of all bug-inducing factor values (i.e., LOC+MLOC+VG+PRE+Churn), and 3) the output of the logistic regression model with PRE and Churn as independent, statistically significant and non-collinear variables (the configuration used in this study to answer our RQs).

Table 6.13, 6.14, 6.15, 6.16, 6.17, 6.18, 6.21 and 6.22 report the performance of the CL agents, enhanced with different configurations of the bug-inducing factors, across the studied software

Table 6.21 Ablation study 7: different bug report metrics added to CLEAR. In bold are the best average performances per algorithm per metric for each project.

	PDE		Zxing	
	top@1			
	ST	NS	ST	NS
CLEAR+LOC	0.09	**0.10**	0.11	0.12
CLEAR+MLOC	0.07	**0.10**	0.08	0.11
CLEAR+VG	0.09	**0.10**	**0.12**	0.12
CLEAR+ALL	**0.10**	**0.10**	0.10	**0.13**
CLEAR+Reg.	0.08	0.08	**0.12**	0.12
	top@5			
	ST	NS	ST	NS
CLEAR+LOC	0.41	**0.48**	0.60	0.55
CLEAR+MLOC	0.41	**0.48**	0.53	0.58
CLEAR+VG	0.42	0.45	0.48	0.47
CLEAR+ALL	**0.45**	0.42	0.54	0.60
CLEAR+Reg.	0.42	0.39	**0.63**	**0.61**
	top@10			
	ST	NS	ST	NS
CLEAR+LOC	0.81	0.85	**1.00**	**1.00**
CLEAR+MLOC	0.80	**0.91**	**1.00**	**1.00**
CLEAR+VG	0.66	0.72	**1.00**	**1.00**
CLEAR+ALL	**0.86**	0.82	**1.00**	**1.00**
CLEAR+Reg.	0.77	0.77	**1.00**	**1.00**
	MAP			
	ST	NS	ST	NS
CLEAR+LOC	**0.09**	**0.10**	**0.12**	0.11
CLEAR+MLOC	0.08	0.09	0.11	0.11
CLEAR+VG	**0.09**	0.09	**0.12**	0.11
CLEAR+ALL	**0.09**	0.09	0.11	**0.12**
CLEAR+Reg.	0.08	0.08	**0.12**	**0.12**
	MRR			
	ST	NS	ST	NS
CLEAR+LOC	**0.60**	0.46	0.43	**0.58**
CLEAR+MLOC	0.37	0.30	**0.56**	0.50
CLEAR+VG	0.29	0.33	0.50	0.47
CLEAR+ALL	0.48	**0.48**	0.48	0.44
CLEAR+Reg.	0.33	0.41	0.42	0.49

projects on stationary and non-stationary changeset-files. Compared to other configurations, CLEAR with regression shows improvements of up to 26% in top@1 (across SWT, Birt, and Tomcat), 27% in top@5 (across Eclipse, SWT, Zxing and AspectJ), 21% in top@10 (across SWT, Zxing and AspectJ), 25% in MAP (across all projects), and 26% in MRR (across Eclipse and Tomcat). Similarly, compared to other configurations, EWC with regression shows improvements of up to 73% in top@1 (across SWT, and Birt and AspectJ), 18% in top@5 (across Eclipse, SWT, and AspectJ), 15% in top@10 (across SWT and Zxing), 22% in MAP (across all projects), and 105% in MRR (across AspectJ). Therefore, boosting the CL agents with bug-inducing factors through regression is suitable for bug localization.

Table 6.22 Ablation study 8: different bug report metrics added to EWC. In bold are the best average performances per algorithm per metric for each project.

	PDE		Zxing	
	top@1			
	ST	NS	ST	NS
EWC+LOC	0.08	0.08	0.13	0.11
EWC+MLOC	0.09	0.08	0.12	**0.12**
EWC+VG	0.08	**0.10**	**0.14**	0.10
EWC+ALL	0.09	0.07	0.11	0.11
EWC+Reg.	**0.10**	0.09	0.13	**0.12**
	top@5			
	ST	NS	ST	NS
EWC+LOC	**0.43**	0.42	0.60	0.57
EWC+MLOC	**0.43**	0.44	0.57	0.58
EWC+VG	0.39	**0.45**	**0.64**	**0.59**
EWC+ALL	**0.43**	0.44	0.57	**0.59**
EWC+Reg.	**0.43**	0.44	0.63	**0.59**
	top@10			
	ST	NS	ST	NS
EWC+LOC	0.81	0.82	**1.00**	**1.00**
EWC+MLOC	0.87	0.88	**1.00**	**1.00**
EWC+VG	0.79	**0.91**	**1.00**	**1.00**
EWC+ALL	**0.86**	0.88	**1.00**	**1.00**
EWC+Reg.	0.85	0.86	**1.00**	**1.00**
	MAP			
	ST	NS	ST	NS
EWC+LOC	0.08	0.08	0.12	**0.12**
EWC+MLOC	**0.09**	**0.09**	0.12	0.11
EWC+VG	0.08	**0.09**	0.12	**0.12**
EWC+ALL	**0.09**	**0.09**	0.12	**0.12**
EWC+Reg.	**0.09**	**0.09**	**0.13**	**0.12**
	MRR			
	ST	NS	ST	NS
EWC+LOC	**0.53**	**0.45**	0.57	0.36
EWC+MLOC	0.52	0.37	**0.59**	0.55
EWC+VG	0.27	0.31	0.57	0.63
EWC+ALL	0.41	0.30	0.48	0.48
EWC+Reg.	0.32	0.39	0.58	**0.64**

Table 6.23 Forgetting ($\mathcal{F}$) statistics (mean ± std) of the CL agents across datasets. The best results are highlighted in dark grey. "-files"=changeset-files.

-files	CLEAR	CLEAR + Reg.	EWC	EWC + Reg.	hunks	CLEAR	CLEAR + Reg.	EWC	EWC + Reg.
AspectJ	3.9±0.1	1.3 ± 0.3	3.6±0.1	1.1 ± 0.0	AspectJ	3.7±0.2	1.2±0.2	3.6±0.0	1.1±0.1
SWT	3.1 ± 1.4	0.9 ± 0.2	3.6 ± 0.3	1.0 ± 0.1	SWT	3.6±0.1	1.1±0.1	3.2±1.5	1.0±0.1
Birt	3.7±0.3	0.7±0.0	3.6±0.3	0.7 ± 0.0	Birt	3.9±0.5	0.7±0.0	3.7±0.2	0.8±0.4
Eclipse	3.6 ± 0.2	1.1±0.2	3.5 ± 0.2	0.5 ± 0.2	Eclipse	3.7±0.2	0.8 ± 0.4	3.6±0.2	0.2±0.1
Tomcat	3.1 ± 1.1	1.1 ± 0.1	3.7±0.4	0.7 ± 0.5	Tomcat	3.9±0.1	0.6±0.5	3.8±0.1	0.5±0.5
PDE	5.6 ± 0.9	0.7 ± 0.1	5.3 ± 2.6	0.5 ± 0.3	PDE	5.6 ± 2.1	0.8 ± 0.1	6.0 ± 1.5	0.3 ± 0.3
Zxing	4.8 ± 0.7	0.5 ± 0.2	3.5 ± 1.4	0.3 ± 0.3	Zxing	5.3 ± 1.2	0.5 ± 0.3	4.4 ± 0.4	0.1 ± 0.1

6.3.5 Discussions

DL-based bug localization techniques are becoming more prevalent to assist developers in maintaining software projects. Nevertheless, they often do not consider the dynamic nature of software projects, typically depending on stationary data. Our research shows that CL agents can perform at the same level or better than traditional DL-based techniques while significantly lowering the computational effort needed for training by at least half. Further, without additional cost to the training, we incorporate bug-inducing factors into the reward function of CL agents. Our results show an improvement in the performance of the CL agents.

Beyond the bug localization task, other software engineering tasks (e.g., software defect prediction [208], malware detection [209], and test case prioritization [1]) might be affected by concept drift associated with data that can cause DL-based techniques to drift and degrade in performance [1,151,208–210]. For example, Gangwar et al. [208] show that software defect prediction models become outdated due to the concept drift associated with software metrics data. We recommend utilizing CL agents as a tool for software defect prediction, training them on temporally diverse data from the given project to predict defects in future releases. Singh et al. [209] investigate how malware evolves over time to defeat detection, which can lead to non-stationary malware data. These non-stationary malware data can nullify the performance of existing DL-based techniques for malware detection which are trained on static malware data [209]. In this context, CL agents trained on malware data, collected over time, can learn useful malware patterns to detect future malware. Bagherzadeh et al. [1] employ DRL to learn a test case prioritization strategy. The test case prioritization approach trains the model continuously on continuous integration cycles to address the concept drift associated with the evolution of test cases. Their results show up to 177.5 hours of training for some datasets, which can be more for larger datasets. The simple architecture of the CL agents can be an option in the context of test case prioritization to reduce the computational effort for training while remaining effective at prioritizing test cases. Our proposed solution can be applied to other SE tasks, as well as our findings that CL agents can adapt to changes in data and retain appropriate context when evaluated on either stationary or non-stationary data.

6.4 Threats to validity

Our study has some limitations that we discuss in detail in this section.

Conclusion validity. In gen eral, the conclusion's limitations concern the degree of accuracy of the statistical findings on the different steps in our study. Regarding the logistic regression model, we remove the insignificant variable based on a threshold $p-$value of 0.05, as commonly used by other studies. In the same way, to handle the multicollinearity of the model's variables, we compute the VIF considering a maximum value of 2.5, as suggested in the literature. We use Welch's ANOVA post-hoc tests as statistical tests. The significance level is set to 0.05 which is standard across the literature as shown by Welch et al. [95] and Games et al. [96]. The non-deterministic nature of DL-based algorithms can also threaten the conclusions made in this work. We address this by collecting results from 5 independent runs for all our experiments and reporting the average.

Internal validity. In this study, we excluded the JDT project from our evaluation because we could not collect relevant non-stationary data for it. Specifically, when collecting non-stationary data from the JDT project (i.e., all changeset-files modified by commits made between the dates of the bug report and the corresponding bug-fixing commit), we found that none of the changeset-files matched our oracle. Here, the oracle refers to the ground truth list of changeset-files that are directly related to resolving a specific bug report.

Construct validity. Regarding evaluating our model, the measures adopted here might not reflect real-world situations. Such a threat is mitigated by considering evaluation measures commonly and largely used by related studies [23, 58, 139]. Furthermore, these measures represent the best available ones for measuring and comparing the performance of our model and related studies on information retrieval-based bug localization.

External validity. Our results are based on a dataset of bugs from open-source projects written in Java, which may limit their generalizability to other projects. To further validate our findings, additional evaluations are needed on projects written in different programming languages and developed in varied settings with diverse characteristics. However, previous studies have also conducted their investigations based on the same dataset evaluated here and the assessed methods are language-agnostic.

Reliability validity. To allow other researchers to reproduce or build on our research, we provide a replication package [29].

6.5 Chapter Summary

This chapter presents a CL framework for bug localization. We compare the effectiveness of two configurations of CL agents, with and without regression, against three baseline studies on stationary and non-stationary changesets. The CL agents employ rehearsal and regularization approaches to mitigate catastrophic forgetting while being trained cyclically in sequence on stationary and non-stationary changesets from seven software projects. Our results show that CL agents can achieve comparable or better performance than the baselines on both stationary and non-stationary data across five software projects while requiring less computational effort during training (on six out of seven projects). Additionally, incorporating bug-inducing factors as prior knowledge for the CL agents significantly improves their performance.

CHAPTER 7 CONCLUSION

The performance of DRL techniques varies significantly across software testing tasks, making it difficult for practitioners to select the most effective technique for a given task. Traditional DRL techniques require extensive task-specific data and long training times, limiting their adaptability to changing or new tasks. DRL generalist agents, pre-trained on task-agnostic datasets, offer potential benefits like reduced training costs and better adaptability. However, existing DRL techniques, including those with pre-trained models, struggle with concept drift in bug localization, leading to suboptimal performance as software artifacts evolve. Continual learning techniques could address this issue by adapting to dynamic changes at various data granularities. SE practitioners could benefit from practical guidelines for effectively leveraging DRL techniques while fully addressing the complexity of the task at hand. Therefore, our thesis was:

DRL-based techniques for SE tasks (1) vary in performance across different DRL techniques; (2) face challenges with long training times and poor adaptation to evolving SE tasks; and (3) can be enhanced by DRL generalist agents and CL to reduce training costs and adapt to concept drifts.

To prove our hypothesis, we conducted two empirical studies, each investigating the limitations of current DRL techniques applied to the complex nature of the examined SE tasks. Additionally, we proposed a CL framework, and then empirically evaluated its effectiveness in addressing concept drift in software projects while improving bug localization.

7.1 Summary of the findings

In the following, we summarize the contributions of this thesis.

7.1.1 An empirical evaluation of different DRL frameworks on software testing tasks.

This contribution was tackled in Chapter 4, where we empirically investigated the applicability of DRL frameworks to software testing tasks. The objective was to provide guidance on effectively applying DRL frameworks to software testing tasks: by evaluating whether the assumptions underlying DRL algorithms are applicable across all software testing tasks or only specific ones, identifying key factors affecting the choice of a DRL algorithm within a given framework and assessing the consistency of results across different DRL frameworks, examining whether similar outcomes can be reliably achieved despite variations in implementation.

We leveraged Stable-baselines, Keras-rl, and Tensorforce DRL frameworks and applied them to find bugs in the blockmaze game and prioritize test cases. Specifically on the Blockmaze game, we evaluated the performance of DRL algorithms (i.e., A2C, PPO and DQN) from Stable-baselines, Keras-rl, and Tensorforce by collecting metrics such as the number of bugs detected, average cumulative reward, state and line coverage, and training/testing times. To prioritize test cases with pairwise and pointwise ranking techniques, we collected the averages and standard deviations of APFD and NRPA of seven enriched and simple datasets. We conducted Welch's ANOVA with Games-Howell post-hoc test on these collected metrics to determine significant differences among DRL strategies, with a significance level set at 0.05. We also compared the consistency of results across different DRL frameworks for metrics like bug detection and testing times to identify whether the same algorithms produced similar outcomes across frameworks.

Our results show that using the same algorithm from different DRL frameworks can result in performance variations due to the diversity of hyperparameters available. Notably, the DQN algorithm from Keras-rl has only 13 hyperparameters, limiting its flexibility during training in the Blockmaze game and contributing to its weaker performance compared to DQN from the Stable-baseline3 framework. We show that the type of reward function (i.e., in testing the Blockmaze game we used negative rewards when not reaching the goal, while in prioritizing test cases we used small positive rewards even when test cases are not ranked correctly) used to design the software testing task influences the performance of the DRL agents. In the test case prioritization task, we show that the reward function influences the good performance of the DRL implementations with the NRPA and APFD metrics.

7.1.2 An empirical evaluation of DRL generalist agents on SE tasks

This contribution was tackled in Chapter 5. We empirically investigated the performance of DRL generalist agents against the performance of specialist agents on three SE tasks: Playtesting in two games, bug localization across five software projects and the minimization of makespan of scheduling operations. We employed another category of DRL agents, DRL generalist agents and extended the range of SE tasks compared to our study in 4. The objective was to address the limitations of specialist DRL agents, which, despite their effectiveness in specific SE tasks, require substantial training time and data. Therefore, we utilized DRL generalist agents capable of adapting to never-before-seen tasks with minimal fine-tuning, to provide recommendations for researchers looking to leverage them on SE tasks.

We fine-tuned DRL generalist agents (i.e., MGDT and IMPALA) on zero-shot, 1%, and 2% data budgets and evaluated their performance by measuring the time to find bugs in Blockmaze and MsPacman games, makespan in PDR-based scheduling tasks, and metrics such as MAP, MRR, top@1, top@5, and top@10 for bug localization. Additionally, training and testing times and cumulative rewards are collected for all tasks. We employed DQN, PPO, V_TRACE and MAENT algorithms to fine-tune the DRL generalist agents. Using statistical tests, we compared the DRL generalist agents' performance at the studied fine-tuning data budgets, and against DRL specialist agents.

Results indicated that the MGDT agent and its configurations (MGDT-DQN, MGDT-PPO, and MGDT-MAENT) found bugs more quickly in the Blockmaze game, while IMPALA failed to detect any, demonstrating the adaptability and exploratory capability of DRL generalist agents using sequence modeling. Additionally, DRL generalist agents significantly outperformed specialist DRL agents in the PDR-based scheduling task, achieving at least a 12.3% reduction in makespan, and in the bug localization task, showing superior performance across all software projects in 5 out of 8 metrics. These findings highlight the effectiveness of DRL generalist agents in leveraging pre-training and pattern recognition to deliver high performance with minimal fine-tuning.

7.1.3 A CL-based framework for the bug localization task

This contribution was tackled in Chapter 6. We proposed a CL framework, that consists of cyclically trained CL agents in a multiple sub-tasks setting for bug localization (each of which operates on either stationary or non-stationary data), comparing them against three current techniques. In this chapter, the objective was to reduce developer effort needed to locate buggy source code by extending the bug localization task discussed in the previous chapter, to include more fine-grained code segments such as hunks. Additionally, the study aimed to effectively address the non-stationary nature of the bug localization task where multiple versions of changeset files or hunks exist between the reporting and fixing of bugs. We aimed to show that current techniques have difficulties handling non-stationary data distribution. We utilized DRL-based CL techniques specifically designed to handle distribution shifts in data while reducing training costs and accelerating the debugging process. Finally, we aimed to further reduce debugging time by using bug-inducing factors associated with source code.

We leveraged CLEAR and EWC, two state-of-the-art DRL-based CL techniques and adapted them to locate buggy changesets-files and hunks in seven software projects in both stationary and non-stationary environments. For each CL agent of the proposed framework (i.e., involving CLEAR or EWC) and current techniques, we calculated evaluation metrics such as MRR, MAP, top@1, top@5, top@10 and the training time. For each software project, we statistically compared the performance of the CL agents against current techniques in both stationary and non-stationary environments. We also calculated performance differences achieved in both stationary and non-stationary environments. We employed logistic regression to determine the best combination of bug-inducing factors needed to enhance the performance of the CL agents.

Results showed that current techniques (i.e., FLIM and RLOCATOR) struggle to adapt to the non-stationary nature of 3 out of 5 software projects with 4 out of 5 metrics due to concept drift, highlighting the need for techniques that can handle such challenges. We showed that CL agents significantly outperformed current techniques either at changeset-files or hunks level with some of our evaluation metrics across all software projects. For instance at the hunks level, against FBL-BERT, which accounts for non-stationary data, the CL agents performed better with the top@10 metric by at least 67% across the SWT, Tomcat, PDE, Zxing, and AspectJ projects. When incorporating bug-inducing factors as prior knowledge for the CL agents, results showed an improvement of up to 20% with all evaluation metrics across at least one software project. The CL agents required 2-5x less training time compared to current techniques with six out of seven software projects, demonstrating that, with or without the integration of bug-inducing factors, these agents can effectively locate buggy source code and reduce debugging time.

7.2 Limitations

There are other challenges associated with applying DRL techniques to SE tasks that are not covered in this thesis. For instance, we did not conduct the hyperparameter tuning process, discussed in Chapter 2.3.4, in order to train the studied DRL agents with the optimal hyperparameters. Moreover, while we try to leverage DRL techniques for SE tasks by accounting for several SE tasks and DRL algorithms, we could not account for all. Additionally, while the studied DRL techniques and their limitations should still be a concern in the near future when applying them to SE tasks, new DRL techniques could be introduced with the rise of new paradigms in DRL. For instance, LLMs powered with DRL have been used to identify vulnerable codes and repair them. Investigating the applicability of DRL techniques for SE tasks is an ongoing effort. In addition to these general concerns, several other factors could influence the results obtained in our studies:

1. *Diversity of environments and datasets:* During the evaluation of the DRL agents, we used datasets and environments made available by previous studies. Although the results achieved with these environments and datasets are encouraging, more evaluations with additional models and datasets are needed to provide more generalizable recommendations to the SE community. For each SE task, the characteristics of the environment (observation, reward, action, etc.) should be expanded. The datasets used for each study in this thesis for tasks such as test case prioritization and bug localization tasks should also be expanded.

2. *DRL Paradigms generalization:* All of our studies involved online DRL algorithms. Although we conduct these studies to be as general as possible using online DRL algorithms, they should be expanded to other paradigms such as offline DRL to improve their generalizability further.

3. *Cost of the analysis:* All of the studies conducted in this thesis rely on multiple DRL algorithms, multiple SE tasks, and multiple training instances. Although training a DRL agent multiple times is essential to account for the stochastic nature of the DRL environment, this process introduces additional computational costs when solving SE tasks. In Chapter 5, we mitigate this cost by fine-tuning DRL generalist agents with up to 2% data budgets, which helps reduce the amount of data needed. However, the size of the generalist agent model can still slow down the fine-tuning process, particularly for more complex tasks. As a result, our studies face limitations when applied to SE tasks that are highly computationally expensive, where extensive DRL training and fine-tuning can become impractical.

7.3 Opportunities for Future Research

In light of this thesis, establishing clear and actionable guidelines for applying DRL techniques to SE tasks is crucial to advance the state-of-the-art in SE applications. These guidelines will empower SE practitioners to push the boundaries of innovation, reduce development costs, and enhance software quality, driving greater efficiency and impact in the field. Yet, there is still a long road ahead to ensure that DRL techniques are harnessed effectively to maximize software quality and minimize development costs. The comprehensive studies in this thesis are just the beginning, and we hope they inspire the SE community to build upon them. Drawing from our findings, we outline promising avenues for future research to advance the field.

1. *Expanding generalizability of the proposed studies:* One main avenue of improvement

is to expand the generalizability of the proposed studies. In Chapter 4, the PPO algorithm from the Stable-baselines framework has detected 1 to 2 more bugs than the PPO algorithm from Tensorforce. Nevertheless, with hyperparameter tuning, changing the variable noise parameter to the PPO algorithm from Tensorforce, we have increased its detection capability to be better than the PPO from Stable-baselines framework. Future works should investigate both implementations in more detail. Additionally, the DQN algorithm showed poor bug detection performance in the Blockmaze game but demonstrated strong ranking capability in the test case prioritization problem for certain pairwise configurations and enriched datasets. The DQN algorithm's suitability for the pairwise ranking technique may be linked to its discrete action space, which is smaller and simpler than that of the Blockmaze game environment, suggesting that the nature of the action space could influence its performance. Another avenue of research would be to explore the impact of the action space nature of software testing tasks on the performance of DRL agents. In Chapter 5, future works should consider investigating larger models for DRL generalist agents. Moreover, while the studied DRL generalist agents were applied to testing video games, bug localization and task scheduling, they can help solve other SE tasks. For instance, applying them to the test case prioritization task discussed in the previous chapter could be interesting. Another avenue could focus on extending the study to include more DRL algorithms to fine-tune the DRL generalist agents. In Chapter 6, we trained CL agents with default hyperparameters. Therefore, optimization and tuning to get the best CL agents for the bug localization task is a potential research avenue for this study. Concept drift is also prevalent in other SE tasks, hence there is a need for effective techniques that can adapt to data that evolves over time. Another research avenue for this study is to use the CL framework for other SE tasks such as the test case prioritization task discussed in Chapter 4. We evaluated CL agents at two data granularity levels: changeset-files and hunks. While the commits level (i.e., a set of hunks), which spans code changes across multiple files, has also been studied [66], we focus on selecting and ranking individual files or hunks. Future work could explore environments where agents select and rank entire commits level data. Finally, we used datasets from existing software projects made available by previous studies. We observe that the performance of CL agents varies depending on the size of available data. The availability of more diverse datasets could be useful in this research area to evaluate existing techniques. It could be also interesting to expand the proposed studies by using multi-agent DRL techniques instead of single agents, implying the modification of each SE task definition we introduced in Section 2.10.

2. *Investigate applicability of other DRL paradigms:* As highlighted in the limitations, this thesis focuses on online DRL algorithms. This can be limiting as the environment describing the SE task could not always be available for training and/or testing. Additionally, online DRL algorithms can not learn from existing data, which could accelerate the training process. Thus, future works could tackle other DRL paradigms (e.g., offline DRL) for SE tasks and analyze their applicability.

3. *Investigate the applicability of LLMs powered with DRL for SE tasks:* LLMs have significantly impacted the SE community, with much research conducted on leveraging them for SE tasks [211–215]. Specifically, LLMs are now involved at all stages of the development of a software product [10]: in requirements engineering, software design, software development, software quality assurance, software maintenance, and software management. LLMs for SE typically involve fine-tuning them on the task at hand for task understanding, content generation etc [10]. DRL techniques can be employed to fine-tune LLMs on the task at hand [10,216,217]. Islam et al. [216] developed SecRepair, a system powered by LLMs, which helps developers identify and fix code vulnerabilities, providing detailed explanations and code comments. The system leverages the PPO algorithm to fine-tune the LLM by optimizing the learning environment with a semantically aware reward function, ensuring the generated code comments are concise while preserving the original meaning. Steenhoek et al. [217] proposed an approach that leverages DRL to optimize test cases generated by LLMs. Specifically, the PPO algorithm is used to fine-tune the LLMs to generate higher-quality test cases. While DRL techniques have been used to fine-tune LLMs at different stages of the development of a software product (e.g., software maintenance, software quality assurance), their applicability and effectiveness are unknown [10]. Future works could explore the applicability of DRL techniques in the context of LLMs for SE. Additionally, it is unknown why other fine-tuning techniques such as Supervised Fine-Tuning [218] are preferred over DRL techniques [10] that can be explored as future works.

4. *Leveraging claims made at each study to further explore the applicability of DRL algorithms:* Following each study, we offer actionable guidelines and a framework for SE practitioners aiming to apply DRL techniques. Delving deeper into some of these solutions could pave the road for future research. In Chapter 4 we recommend keeping the cumulative reward positive for better performance of DRL agents. Effectively designing the reward function of a DRL agent is crucial, as it relies on it to learn the optimal policy [219]. Existing works have studied several approaches to design informative rewards for DRL agents [220–223]. For instance, Devidze et al. [223] propose to learn an

intrinsic reward function with exploration-based bonuses to enhance the DRL agent's learning and maximize its performance for extrinsic rewards. An interesting research avenue could be to explore these reward shaping techniques in the context of SE and evaluate their capabilities to enhance the performance of the DRL agent on the task at hand. In Chapter 5, we recommend using DRL generalist agents to solve scheduling and bug localization problems. The DRL generalist agents we leveraged were pre-trained using games of the Atari learning environment [179]. It could be interesting for future works to extend the pre-training datasets with samples of scheduling instances or source code and investigate whether such contextual data further improves performance. In Chapter 6 we propose a CL framework to train CL agents cyclically in sequence on two sub-tasks involving stationary and non-stationary data. Multiple versions of source code data can be observed between the time a bug is reported and fixed. We employed all source code data at two different points in time: At the time the bug reported was fixed and before it was fixed. A potential avenue for future work could involve defining additional sub-tasks by developing a strategy to segment source code data based on different versions. This would allow the investigation of the impact of version-based segmentation on identifying buggy source code.

5. *Benchmark of DRL techniques for SE:* Future benchmarking efforts should focus on systematically evaluating the generalizability and adaptability of DRL techniques in SE tasks, incorporating different DRL algorithms, hyperparameter configurations, and action space complexities - ensuring reproducibility and comparability across implementations.

REFERENCES

[1] M. Bagherzadeh, N. Kahani, and L. Briand, “Reinforcement learning for test case prioritization,” IEEE Transactions on Software Engineering, vol. 48, no. 8, pp. 2836–2856, 2021.

[2] S. J. Prince, Understanding Deep Learning. The MIT Press, 2023. [Online]. Available: http://udlbook.com

[3] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, “Attention is all you need,” Advances in neural information processing systems, vol. 30, 2017.

[4] P. S. Nouwou Mindom, A. Nikanjam, and F. Khomh, “A comparison of reinforcement learning frameworks for software testing tasks,” Empirical Software Engineering, vol. 28, no. 5, p. 111, 2023.

[5] “Link for bug - 420210,” https://bugs.eclipse.org/bugs/show_bug.cgi?id=420210, 2013.

[6] C. Zhang, W. Song, Z. Cao, J. Zhang, P. S. Tan, and X. Chi, “Learning to dispatch for job shop scheduling via deep reinforcement learning,” Advances in Neural Information Processing Systems, vol. 33, pp. 1621–1632, 2020.

[7] L. Espeholt et al., “Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures,” in International conference on machine learning. PMLR, 2018, pp. 1407–1416.

[8] K.-H. Lee et al., “Multi-game decision transformers,” Advances in Neural Information Processing Systems, vol. 35, pp. 27 921–27 936, 2022.

[9] N. Wirth, “A brief history of software engineering,” IEEE Annals of the History of Computing, vol. 30, no. 3, pp. 32–39, 2008.

[10] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, “Large language models for software engineering: A systematic literature review,” ACM Transactions on Software Engineering and Methodology, 2023.

[11] V. Garousi and M. V. Mäntylä, “A systematic literature review of literature reviews in software testing,” Information and Software Technology, vol. 80, pp. 195–216, 2016.

[12] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in 2013 35th International Conference on Software Engineering (ICSE). IEEE, 2013, pp. 712–721.

[13] N. Salleh, E. Mendes, and J. Grundy, "Empirical studies of pair programming for cs/se teaching in higher education: A systematic literature review," IEEE Transactions on Software Engineering, vol. 37, no. 4, pp. 509–525, 2010.

[14] S. Aleem, L. F. Capretz, and F. Ahmed, "Critical success factors to improve the game development process from a developer's perspective," Journal of computer science and technology, vol. 31, pp. 925–950, 2016.

[15] M. P. Robillard et al., "On-demand developer documentation," in 2017 IEEE International conference on software maintenance and evolution (ICSME). IEEE, 2017, pp. 479–483.

[16] 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.

[17] R. E. S. Santos, C. V. C. Magalhães, L. F. Capretz, J. S. Correia-Neto, F. Q. B. da Silva, and A. Saher, "Computer games are serious business and so is their quality: Particularities of software testing in game development from the perspective of practitioners," in Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ser. ESEM '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3239235.3268923

[18] A. Chraibi, S. Ben Alla, A. Touhafi, and A. Ezzati, "A novel dynamic multi-objective task scheduling optimization based on dueling dqn and per," The Journal of Supercomputing, vol. 79, no. 18, pp. 21 368–21 423, 2023.

[19] J. Pfau, J. D. Smeddinck, and R. Malaka, "Automated game testing with icarus: Intelligent completion of adventure riddles via unsupervised solving," in Extended abstracts publication of the annual symposium on computer-human interaction in play, 2017, pp. 153–164.

[20] K. Malialis, S. Devlin, and D. Kudenko, "Distributed reinforcement learning for adaptive and robust network intrusion response," Connection Science, vol. 27, no. 3, pp. 234–252, 2015.

[21] M. H. Moghadam, M. Saadatmand, M. Borg, M. Bohlin, and B. Lisper, "An autonomous performance testing framework using self-adaptive fuzzy reinforcement learning," Software quality journal, pp. 1–33, 2021.

[22] T. Shi, L. Xiao, and K. Wu, "Reinforcement learning based test case prioritization for enhancing the security of software," in 2020 IEEE 7th International Conference on Data Science and Advanced Analytics (DSAA). IEEE, 2020, pp. 663–672.

[23] P. Chakraborty, M. Alfadel, and M. Nagappan, "Rlocator: Reinforcement learning for bug localization," arXiv preprint arXiv:2305.05586, 2023.

[24] C. Watson, N. Cooper, D. N. Palacio, K. Moran, and D. Poshyvanyk, "A systematic literature review on the use of deep learning in software engineering research," ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 31, no. 2, pp. 1–58, 2022.

[25] E. H. Houssein, A. G. Gad, Y. M. Wazery, and P. N. Suganthan, "Task scheduling in cloud computing based on meta-heuristics: review, taxonomy, open challenges, and future trends," Swarm and Evolutionary Computation, vol. 62, p. 100841, 2021.

[26] D. Olewicki, M. Nayrolles, and B. Adams, "Towards language-independent brown build detection," in Proceedings of the 44th International Conference on Software Engineering, 2022, pp. 2177–2188.

[27] M. M. Morovati, F. Tambon, M. Taraghi, A. Nikanjam, and F. Khomh, "Common challenges of deep reinforcement learning applications development: an empirical study," Empirical Software Engineering, vol. 29, no. 4, p. 95, 2024.

[28] A. H. Yahmed, A. A. Abbassi, A. Nikanjam, H. Li, and F. Khomh, "Deploying deep reinforcement learning systems: a taxonomy of challenges," in 2023 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, 2023, pp. 26–38.

[29] "Replication package," https://zenodo.org/records/14271134, 2024.

[30] I. Goodfellow, Y. Bengio, and A. Courville, Deep learning. MIT press, 2016.

[31] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," in Proceedings of the fourteenth international conference on artificial intelligence and statistics. JMLR Workshop and Conference Proceedings, 2011, pp. 315–323.

[32] Zhou and Chellappa, "Computation of optical flow using a neural network," in IEEE 1988 international conference on neural networks. IEEE, 1988, pp. 71–78.

[33] L. Chen, K. Lu, A. Rajeswaran, K. Lee, A. Grover, M. Laskin, P. Abbeel, A. Srinivas, and I. Mordatch, "Decision transformer: Reinforcement learning via sequence modeling," Advances in neural information processing systems, vol. 34, pp. 15 084–15 097, 2021.

[34] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever et al., "Improving language understanding by generative pre-training," 2018.

[35] Q. Zheng, A. Zhang, and A. Grover, "Online decision transformer," in International Conference on Machine Learning. PMLR, 2022, pp. 27 042–27 059.

[36] T. Lillicrap, "Continuous control with deep reinforcement learning," arXiv preprint arXiv:1509.02971, 2015.

[37] 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.

[38] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann, "Stable-baselines3: Reliable reinforcement learning implementations," Journal of Machine Learning Research, vol. 22, no. 268, pp. 1–8, 2021. [Online]. Available: http://jmlr.org/papers/v22/20-1364.html

[39] A. Hill et al., "Stable baselines," https://github.com/hill-a/stable-baselines, 2018.

[40] M. Schaarschmidt, A. Kuhnle, B. Ellis, K. Fricke, F. Gessert, and E. Yoneki, "Lift: Reinforcement learning in computer systems by learning from demonstrations," arXiv preprint arXiv:1808.07903, 2018.

[41] P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, Y. Wu, and P. Zhokhov, "Openai baselines," https://github.com/openai/baselines, 2017.

[42] A. Paszke et al., "Pytorch: An imperative style, high-performance deep learning library," in Advances in Neural Information Processing Systems 32, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[43] M. Plappert, "keras-rl," https://github.com/keras-rl/keras-rl, 2016.

[44] M. Abadi et al., "Tensorflow, large-scale machine learning on heterogeneous systems," 11 2015.

[45] P. S. Castro, S. Moitra, C. Gelada, S. Kumar, and M. G. Bellemare, "Dopamine: A research framework for deep reinforcement learning," arXiv preprint arXiv:1812.06110, 2018.

[46] "gin-config," https://github.com/google/gin-config, 2018.

[47] 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.

[48] S. Fujimoto, H. Hoof, and D. Meger, "Addressing function approximation error in actor-critic methods," in International Conference on Machine Learning. PMLR, 2018, pp. 1587–1596.

[49] 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.

[50] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," arXiv preprint arXiv:1707.06347, 2017.

[51] Z. Wang, T. Schaul, M. Hessel, H. Hasselt, M. Lanctot, and N. Freitas, "Dueling network architectures for deep reinforcement learning," in International conference on machine learning. PMLR, 2016, pp. 1995–2003.

[52] R. S. Sutton, A. G. Barto et al., Introduction to reinforcement learning. MIT press Cambridge, 1998, vol. 135.

[53] S. Gu, T. Lillicrap, I. Sutskever, and S. Levine, "Continuous deep q-learning with model-based acceleration," in International conference on machine learning. PMLR, 2016, pp. 2829–2838.

[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] C. Politowski, Y.-G. Guéhéneuc, and F. Petrillo, "Towards automated video game testing: Still a long way to go," in Proceedings of the 6th international ICSE workshop

on games and software engineering: engineering fun, inspiration, and motivation, 2022, pp. 37–43.

[56] M. Yampolsky and W. Scacchi, "Learning game design and software engineering through a game prototyping experience: a case study," in Proceedings of the 5th International Workshop on Games and Software engineering, 2016, pp. 15–21.

[57] R. Tufano, S. Scalabrino, L. Pascarella, E. Aghajani, R. Oliveto, and G. Bavota, "Using reinforcement learning for load testing of video games," in Proceedings of the 44th International Conference on Software Engineering, 2022, pp. 2303–2314.

[58] H. Liang, D. Hang, and X. Li, "Modeling function-level interactions for file-level bug localization," Empirical Software Engineering, vol. 27, no. 7, p. 186, 2022.

[59] "Eclipse apache project," https://github.com/eclipse-platform/eclipse.platform.ui.git, 2022.

[60] D. Wang, M. Galster, and M. Morales-Trujillo, "A systematic mapping study of bug reproduction and localization," Information and Software Technology, p. 107338, 2023.

[61] W. E. Wong, V. Debroy, and B. Choi, "A family of code coverage-based heuristics for effective fault localization," Journal of Systems and Software, vol. 83, no. 2, pp. 188–208, 2010.

[62] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," IEEE Transactions on Software Engineering, vol. 42, no. 8, pp. 707–740, 2016.

[63] M. Wen, R. Wu, and S.-C. Cheung, "Locus: Locating bugs from software changes," in Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, 2016, pp. 262–273.

[64] X. Ye, R. Bunescu, and C. Liu, "Learning to rank relevant files for bug reports using domain knowledge," in Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering, 2014, pp. 689–699.

[65] R. Wu, M. Wen, S.-C. Cheung, and H. Zhang, "Changelocator: locate crash-inducing changes based on crash reports," Empirical Software Engineering, vol. 23, pp. 2866–2900, 2018.

[66] A. Ciborowska and K. Damevski, "Fast changeset-based bug localization with bert," in Proceedings of the 44th International Conference on Software Engineering, 2022, pp. 946–957.

[67] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Bug localization with combination of deep learning and information retrieval," in 2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC). IEEE, 2017, pp. 218–229.

[68] X. Huo, F. Thung, M. Li, D. Lo, and S.-T. Shi, "Deep transfer bug localization," IEEE Transactions on software engineering, vol. 47, no. 7, pp. 1368–1380, 2019.

[69] Z. Zhang, Y. Lei, X. Mao, and P. Li, "Cnn-fl: An effective approach for localizing faults using convolutional neural networks," in 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, 2019, pp. 445–455.

[70] Y. Xiao, J. Keung, K. E. Bennin, and Q. Mi, "Improving bug localization with word embedding and enhanced convolutional neural networks," Information and Software Technology, vol. 105, pp. 17–29, 2019.

[71] X.-W. Chen and X. Lin, "Big data deep learning: challenges and perspectives," IEEE access, vol. 2, pp. 514–525, 2014.

[72] A. Munappy, J. Bosch, H. H. Olsson, A. Arpteg, and B. Brinne, "Data management challenges for deep learning," in 2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA). IEEE, 2019, pp. 140–147.

[73] M. Samadi, S. Royuela, L. M. Pinho, T. Carvalho, and E. Quiñones, "Time-predictable task-to-thread mapping in multi-core processors," Journal of Systems Architecture, vol. 148, p. 103068, 2024.

[74] S. Qu, S. Zhao, B. Li, Y. He, X. Cai, L. Zhang, and Y. Wang, "Cim-mlc: A multi-level compilation stack for computing-in-memory accelerators," in Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, 2024, pp. 185–200.

[75] S. A. Murad, Z. R. M. Azmi, A. J. M. Muzahid, M. M. H. Sarker, M. S. U. Miah, M. K. B. Bhuiyan, N. Rahimi, and A. K. Bairagi, "Priority based job scheduling technique that utilizes gaps to increase the efficiency of job distribution in cloud computing," Sustainable Computing: Informatics and Systems, vol. 41, p. 100942, 2024.

[76] C.-H. Chiu, Z. Xiong, Z. Guo, T.-W. Huang, and Y. Lin, “An efficient task-parallel pipeline programming framework,” in Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region, 2024, pp. 95–106.

[77] M. G. Sánchez, E. Lalla-Ruiz, A. F. Gil, C. Castro, and S. Voß, “Resource-constrained multi-project scheduling problem: A survey,” European Journal of Operational Research, vol. 309, no. 3, pp. 958–976, 2023.

[78] L. Peters and A. M. Moreno, “Educating software engineering managers-revisited what software project managers need to know today,” in 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, vol. 2. IEEE, 2015, pp. 353–359.

[79] X. Wang, H. Yao, T. Mai, S. Guo, and Y. Liu, “Reinforcement learning-based particle swarm optimization for end-to-end traffic scheduling in tsn-5g networks,” IEEE/ACM Transactions on Networking, vol. 31, no. 6, pp. 3254–3268, 2023.

[80] B. Wagner, A. Kohn, and T. Neumann, “Self-tuning query scheduling for analytical workloads,” in Proceedings of the 2021 International Conference on Management of Data, 2021, pp. 1879–1891.

[81] O. Schwahn, N. Coppik, S. Winter, and N. Suri, “Assessing the state and improving the art of parallel testing for c,” in Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2019, pp. 123–133.

[82] G. Rjoub, J. Bentahar, O. Abdel Wahab, and A. Saleh Bataineh, “Deep and reinforcement learning for automated task scheduling in large-scale cloud computing systems,” Concurrency and Computation: Practice and Experience, vol. 33, no. 23, p. e5919, 2021.

[83] X. Liu, L. Fan, J. Xu, X. Li, L. Gong, J. Grundy, and Y. Yang, “Fogworkflowsim: An automated simulation toolkit for workflow performance evaluation in fog computing,” in 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2019, pp. 1114–1117.

[84] J. Blazewicz, E. Pesch, and M. Sterna, “The disjunctive graph machine representation of the job shop scheduling problem,” European Journal of Operational Research, vol. 127, no. 2, pp. 317–331, 2000.

[85] M. McCloskey and N. J. Cohen, “Catastrophic interference in connectionist networks: The sequential learning problem,” in Psychology of learning and motivation. Elsevier, 1989, vol. 24, pp. 109–165.

[86] D. L. Silver, Q. Yang, and L. Li, "Lifelong machine learning systems: Beyond learning algorithms," in 2013 AAAI spring symposium series, 2013.

[87] D. Abel, A. Barreto, B. Van Roy, D. Precup, H. P. van Hasselt, and S. Singh, "A definition of continual reinforcement learning," Advances in Neural Information Processing Systems, vol. 36, 2024.

[88] "Mspacman," https://www.gymlibrary.dev/environments/atari/ms_pacman/, 2018.

[89] P. Rohlfshagen, J. Liu, D. Perez-Liebana, and S. M. Lucas, "Pac-man conquers academia: Two decades of research using a classic arcade game," IEEE Transactions on Games, vol. 10, no. 3, pp. 233–256, 2017.

[90] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.

[91] C. Gormley and Z. Tong, Elasticsearch: the definitive guide: a distributed real-time search and analytics engine. " O'Reilly Media, Inc.", 2015.

[92] Z. Feng et al., "Codebert: A pre-trained model for programming and natural languages," arXiv preprint arXiv:2002.08155, 2020.

[93] D. E. Knuth, The art of computer programming. Pearson Education, 1997, vol. 3.

[94] A. M. Brown, "A new software for carrying out one-way anova post hoc tests," Computer methods and programs in biomedicine, vol. 79, no. 1, pp. 89–95, 2005.

[95] B. L. Welch, "The generalization of 'student's'problem when several different population varlances are involved," Biometrika, vol. 34, no. 1-2, pp. 28–35, 1947.

[96] P. A. Games and J. F. Howell, "Pairwise multiple comparison procedures with unequal n's and/or variances: a monte carlo study," Journal of Educational Statistics, vol. 1, no. 2, pp. 113–125, 1976.

[97] 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.

[98] S. Levine, A. Kumar, G. Tucker, and J. Fu, "Offline reinforcement learning: Tutorial, review, and perspectives on open problems," arXiv preprint arXiv:2005.01643, 2020.

[99] C. J. C. H. Watkins, "Learning from delayed rewards," 1989.

[100] B. O'Donoghue, I. Osband, R. Munos, and V. Mnih, "The uncertainty bellman equation and exploration," in International conference on machine learning, 2018, pp. 3836–3845.

[101] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in Proceedings of the AAAI conference on artificial intelligence, vol. 30, no. 1, 2016.

[102] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," Machine learning, vol. 8, pp. 229–256, 1992.

[103] E. Todorov, T. Erez, and Y. Tassa, "Mujoco: A physics engine for model-based control," in 2012 IEEE/RSJ International Conference on Intelligent Robots and Systems. IEEE, 2012, pp. 5026–5033.

[104] S. Li, Y. Wu, X. Cui, H. Dong, F. Fang, and S. Russell, "Robust multi-agent reinforcement learning via minimax deep deterministic policy gradient," in Proceedings of the AAAI conference on artificial intelligence, vol. 33, no. 01, 2019, pp. 4213–4220.

[105] D. Silver et al., "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," arXiv preprint arXiv:1712.01815, 2017.

[106] ——, "A general reinforcement learning algorithm that masters chess, shogi, and go through self-play," Science, vol. 362, no. 6419, pp. 1140–1144, 2018.

[107] S. Wang, L. Huang, A. Gao, J. Ge, T. Zhang, H. Feng, I. Satyarth, M. Li, H. Zhang, and V. Ng, "Machine/deep learning for software engineering: A systematic literature review," IEEE Transactions on Software Engineering, vol. 49, no. 3, pp. 1188–1231, 2022.

[108] J. Kim, M. Kwon, and S. Yoo, "Generating test input with deep reinforcement learning," in 2018 IEEE/ACM 11th International Workshop on Search-Based Software Testing (SBST). IEEE, 2018, pp. 51–58.

[109] M. Soualhia, F. Khomh, and S. Tahar, "A dynamic and failure-aware task scheduling framework for hadoop," IEEE Transactions on Cloud Computing, vol. 8, no. 2, pp. 553–569, 2020.

[110] A. Romdhana, A. Merlo, M. Ceccato, and P. Tonella, "Deep reinforcement learning for black-box testing of android apps," ACM Transactions on Software Engineering and Methodology, 2022.

[111] Y. Koroglu, A. Sen, O. Muslu, Y. Mete, C. Ulker, T. Tanriverdi, and Y. Donmez, "Qbe: Qlearning-based exploration of android applications," in 2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST). IEEE, 2018, pp. 105–115.

[112] K. Böttinger, P. Godefroid, and R. Singh, "Deep reinforcement fuzzing," in 2018 IEEE Security and Privacy Workshops (SPW). IEEE, 2018, pp. 116–122.

[113] J. Chen, H. Ma, and L. Zhang, "Enhanced compiler bug isolation via memoized search," in Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, 2020, pp. 78–89.

[114] D. A. Agarwal and S. Jain, "Efficient optimal algorithm of task scheduling in cloud computing environment," arXiv preprint arXiv:1404.2076, 2014.

[115] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, "Resource management with deep reinforcement learning," in Proceedings of the 15th ACM workshop on hot topics in networks, 2016, pp. 50–56.

[116] S. Zheng, C. Gupta, and S. Serita, "Manufacturing dispatching using reinforcement and transfer learning," in Machine learning and knowledge discovery in databases: European conference, ECML pKDD 2019, wüRzburg, Germany, September 16–20, 2019, proceedings, part III. Springer, 2020, pp. 655–671.

[117] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh, "Learning scheduling algorithms for data processing clusters," in Proceedings of the ACM special interest group on data communication, 2019, pp. 270–288.

[118] P. Sun, Z. Guo, J. Wang, J. Li, J. Lan, and Y. Hu, "Deepweave: Accelerating job completion time with deep reinforcement learning-based coflow scheduling," in Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence, 2021, pp. 3314–3320.

[119] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," IEEE transactions on neural networks, vol. 20, no. 1, pp. 61–80, 2008.

[120] S. Mangalampalli, G. R. Karri, M. Kumar, O. I. Khalaf, C. A. T. Romero, and G. A. Sahib, "Drlbtsa: Deep reinforcement learning based task-scheduling algorithm in cloud computing," Multimedia Tools and Applications, vol. 83, no. 3, pp. 8359–8387, 2024.

[121] C.-C. Lin, D.-J. Deng, Y.-L. Chih, and H.-T. Chiu, “Smart manufacturing scheduling with edge computing using multiclass deep q network,” IEEE Transactions on Industrial Informatics, vol. 15, no. 7, pp. 4276–4284, 2019.

[122] W. Drozd and M. D. Wagner, “Fuzzergym: A competitive framework for fuzzing and learning,” arXiv preprint arXiv:1807.07490, 2018.

[123] D. Adamo, M. K. Khan, S. Koppula, and R. Bryce, “Reinforcement learning for android gui testing,” in Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation, 2018, pp. 2–8.

[124] A. Reichstaller and A. Knapp, “Risk-based testing of self-adaptive systems using run-time predictions,” in 2018 IEEE 12th international conference on self-adaptive and self-organizing systems (SASO). IEEE, 2018, pp. 80–89.

[125] H. Spieker, A. Gotlieb, D. Marijan, and M. Mossige, “Reinforcement learning for automatic test case prioritization and selection in continuous integration,” in Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2017, pp. 12–22.

[126] A. Bertolino, A. Guerriero, B. Miranda, R. Pietrantuono, and S. Russo, “Learning-to-rank vs ranking-to-learn: strategies for regression testing in continuous integration,” in Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, 2020, pp. 1–12.

[127] D. Wang, Z. Jia, S. Li, Y. Yu, Y. Xiong, W. Dong, and X. Liao, “Bridging pre-trained models and downstream tasks for source code understanding,” in Proceedings of the 44th International Conference on Software Engineering, 2022, pp. 287–298.

[128] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi, “Learning and evaluating contextual embedding of source code,” in International conference on machine learning. PMLR, 2020, pp. 5110–5121.

[129] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “Bert: Pre-training of deep bidirectional transformers for language understanding,” arXiv preprint arXiv:1810.04805, 2018.

[130] R. Mendonca, O. Rybkin, K. Daniilidis, D. Hafner, and D. Pathak, “Discovering and achieving goals via world models,” Advances in Neural Information Processing Systems, vol. 34, pp. 24 379–24 391, 2021.

[131] D. Kalashnikov, J. Varley, Y. Chebotar, B. Swanson, R. Jonschkowski, C. Finn, S. Levine, and K. Hausman, “Mt-opt: Continuous multi-task robotic reinforcement learning at scale,” arXiv preprint arXiv:2104.08212, 2021.

[132] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, C. Vendome, and D. Poshyvanyk, “Automatically discovering, reporting and reproducing android application crashes,” in 2016 IEEE international conference on software testing, verification and validation (icst). IEEE, 2016, pp. 33–44.

[133] T. Yu, T. S. Zaman, and C. Wang, “Descry: reproducing system-level concurrency failures,” in Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, 2017, pp. 694–704.

[134] T. Roehm, N. Gurbanova, B. Bruegge, C. Joubert, and W. Maalej, “Monitoring user interactions for supporting failure reproduction,” in 2013 21st International Conference on Program Comprehension (ICPC). IEEE, 2013, pp. 73–82.

[135] E. C. Neto, D. A. Da Costa, and U. Kulesza, “The impact of refactoring changes on the szz algorithm: An empirical study,” in 2018 IEEE 25th international conference on software analysis, evolution and reengineering (SANER). IEEE, 2018, pp. 380–390.

[136] S. Wang, F. Khomh, and Y. Zou, “Improving bug localization using correlations in crash reports,” in 2013 10th Working Conference on Mining Software Repositories (MSR). IEEE, 2013, pp. 247–256.

[137] M. Rath, D. Lo, and P. Mäder, “Analyzing requirements and traceability information to improve bug localization,” in Proceedings of the 15th International Conference on Mining Software Repositories, 2018, pp. 442–453.

[138] C. Mills, J. Pantiuchina, E. Parra, G. Bavota, and S. Haiduc, “Are bug reports enough for text retrieval-based bug localization?” in 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, 2018, pp. 381–392.

[139] X. Ye, R. Bunescu, and C. Liu, “Mapping bug reports to relevant files: A ranking model, a fine-grained benchmark, and feature evaluation,” IEEE Transactions on Software Engineering, vol. 42, no. 4, pp. 379–402, 2015.

[140] B. Wang, L. Xu, M. Yan, C. Liu, and L. Liu, “Multi-dimension convolutional neural network for bug localization,” IEEE Transactions on Services Computing, vol. 15, no. 3, pp. 1649–1663, 2020.

[141] N. Miryeganeh, S. Hashtroudi, and H. Hemmati, "Globug: Using global data in fault localization," Journal of Systems and Software, vol. 177, p. 110961, 2021.

[142] R. Almhana, M. Kessentini, and W. Mkaouer, "Method-level bug localization using hybrid multi-objective search," Information and Software Technology, vol. 131, p. 106474, 2021.

[143] Y. Du and Z. Yu, "Pre-training code representation with semantic flow graph for effective bug localization," in Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2023, pp. 579–591.

[144] R. Kurle, B. Cseke, A. Klushyn, P. Van Der Smagt, and S. Günnemann, "Continual learning with bayesian neural networks for non-stationary data," in International Conference on Learning Representations, 2019.

[145] J. Lu, A. Liu, F. Dong, F. Gu, J. Gama, and G. Zhang, "Learning under concept drift: A review," IEEE transactions on knowledge and data engineering, vol. 31, no. 12, pp. 2346–2363, 2018.

[146] D. Olewicki, S. Habchi, M. Nayrolles, M. Faramarzi, S. Chandar, and B. Adams, "Towards lifelong learning for software analytics models: Empirical study on brown build and risk prediction," arXiv preprint arXiv:2305.09824, 2023.

[147] C. Rosen, B. Grawi, and E. Shihab, "Commit guru: analytics and risk prediction of software commits," in Proceedings of the 2015 10th joint meeting on foundations of software engineering, 2015, pp. 966–969.

[148] T. Savor, M. Douglas, M. Gentili, L. Williams, K. Beck, and M. Stumm, "Continuous deployment at facebook and oanda," in Proceedings of the 38th International Conference on software engineering companion, 2016, pp. 21–30.

[149] B. Liu, "Lifelong machine learning: a paradigm for continuous learning," Frontiers of Computer Science, vol. 11, no. 3, pp. 359–361, 2017.

[150] S. McIntosh and Y. Kamei, "Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction," in Proceedings of the 40th international conference on software engineering, 2018, pp. 560–560.

[151] D. Olewicki, S. Habchi, M. Nayrolles, M. Faramarzi, S. Chandar, and B. Adams, "On the costs and benefits of adopting lifelong learning for software analytics - empirical

study on brown build and risk prediction," in Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice, ser. ICSE-SEIP '24, 2024.

[152] G. I. Parisi, R. Kemker, J. L. Part, C. Kanan, and S. Wermter, "Continual lifelong learning with neural networks: A review," Neural networks, vol. 113, pp. 54–71, 2019.

[153] M. Nayrolles and A. Hamou-Lhadj, "Clever: Combining code metrics with clone detection for just-in-time fault prevention and resolution in large industrial projects," in Proceedings of the 15th international conference on mining software repositories, 2018, pp. 153–164.

[154] J. Kirkpatrick et al., "Overcoming catastrophic forgetting in neural networks," Proceedings of the national academy of sciences, vol. 114, no. 13, pp. 3521–3526, 2017.

[155] J. Lee, K. Han, and H. Yu, "A light bug triage framework for applying large pre-trained language model," in Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, 2022, pp. 1–11.

[156] "Report software failure," https://www.techrepublic.com/article/report-software-failure-caused-1-7-trillion-in-financial-losses-in-2017/, 2017.

[157] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," arXiv preprint arXiv:1412.6980, 2014.

[158] "Replication package, drl frameworks for se," https://github.com/npaulinastevia/DRL_se, 2022.

[159] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.

[160] A. Hill et al., "Stable baselines. 2018," URL: https://github.com/hill-a/stable-baselines, 2019.

[161] "Stable-baselines3 migration," https://stable-baselines3.readthedocs.io/en/master/guide/migration.html, 2021.

[162] N. M. Nguyen, "Improving model-based rl with adaptive rollout using uncertainty estimation," 2018.

[163] F. P. Such, V. Madhavan, E. Conti, J. Lehman, K. O. Stanley, and J. Clune, "Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning," arXiv preprint arXiv:1712.06567, 2017.

[164] M. Fortunato et al., “Noisy networks for exploration,” arXiv preprint arXiv:1706.10295, 2017.

[165] H. Zhu, P. A. Hall, and J. H. May, “Software unit test coverage and adequacy,” Acm computing surveys (csur), vol. 29, no. 4, pp. 366–427, 1997.

[166] M. Harman, Y. Jia, and Y. Zhang, “Achievements, open problems and challenges for search based software testing,” in 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST). IEEE, 2015, pp. 1–12.

[167] M. J. Munro, “Product metrics for automatic identification of" bad smell" design problems in java source-code,” in 11th IEEE International Software Metrics Symposium (METRICS’05). IEEE, 2005, pp. 15–15.

[168] S. Singh and K. S. Kahlon, “Effectiveness of encapsulation and object-oriented metrics to refactor code and identify error prone classes using bad smells,” ACM SIGSOFT Software Engineering Notes, vol. 36, no. 5, pp. 1–10, 2011.

[169] J. Gao, M. Gen, L. Sun, and X. Zhao, “A hybrid of genetic algorithm and bottleneck shifting for multiobjective flexible job shop scheduling problems,” Computers & Industrial Engineering, vol. 53, no. 1, pp. 149–162, 2007.

[170] S. Kaur and A. Verma, “An efficient approach to genetic algorithm for task scheduling in cloud computing environment,” International Journal of Information Technology and Computer Science (IJITCS), vol. 4, no. 10, pp. 74–79, 2012.

[171] H. Wang, S. Zheng, C. Xiong, and R. Socher, “On the generalization gap in reparameterizable reinforcement learning,” in International Conference on Machine Learning. PMLR, 2019, pp. 6648–6658.

[172] A. Lazaric, “Transfer in reinforcement learning: a framework and a survey,” in Reinforcement Learning: State-of-the-Art. Springer, 2012, pp. 143–173.

[173] Z. Zhu, K. Lin, A. K. Jain, and J. Zhou, “Transfer learning in deep reinforcement learning: A survey,” IEEE Transactions on Pattern Analysis and Machine Intelligence, 2023.

[174] J. McCarthy, M. L. Minsky, N. Rochester, and C. E. Shannon, “A proposal for the dartmouth summer research project on artificial intelligence, august 31, 1955,” AI magazine, vol. 27, no. 4, pp. 12–12, 2006.

[175] A. Arnab, M. Dehghani, G. Heigold, C. Sun, M. Lučić, and C. Schmid, "Vivit: A video vision transformer," in Proceedings of the IEEE/CVF international conference on computer vision, 2021, pp. 6836–6846.

[176] K. O. McGraw and S. P. Wong, "A common language effect size statistic." Psychological bulletin, vol. 111, no. 2, p. 361, 1992.

[177] A. Arcuri and L. Briand, "A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," Software Testing, Verification and Reliability, vol. 24, no. 3, pp. 219–250, 2014.

[178] S. Hochreiter and J. Schmidhuber, "Long short-term memory," Neural computation, vol. 9, no. 8, pp. 1735–1780, 1997.

[179] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, "The arcade learning environment: An evaluation platform for general agents," Journal of Artificial Intelligence Research, vol. 47, pp. 253–279, 2013.

[180] R. Agarwal, D. Schuurmans, and M. Norouzi, "An optimistic perspective on offline reinforcement learning," in International Conference on Machine Learning. PMLR, 2020, pp. 104–114.

[181] V. Mnih et al., "Human-level control through deep reinforcement learning," nature, vol. 518, no. 7540, pp. 529–533, 2015.

[182] "Cartpole," https://gym.openai.com/envs/CartPole-v0/, 2016.

[183] "Supertuxkart," https://github.com/supertuxkart/stk-code.

[184] "Replication package generalist agents for se," https://github.com/npaulinastevia/Harnessing_generalist_agents_on_SE, 2023.

[185] E. Taillard, "Benchmarks for basic scheduling problems," european journal of operational research, vol. 64, no. 2, pp. 278–285, 1993.

[186] X. Chang, Z. Liang, Y. Zhang, L. Cui, Z. Long, G. Wu, Y. Gao, W. Chen, J. Wei, and T. Huang, "A reinforcement learning approach to generating test cases for web applications," in 2023 IEEE/ACM International Conference on Automation of Software Test (AST). IEEE, 2023, pp. 13–23.

[187] A. Khamaj and A. M. Ali, "Adapting user experience with reinforcement learning: Personalizing interfaces based on user behavior analysis in real-time," Alexandria Engineering Journal, vol. 95, pp. 164–173, 2024.

[188] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, “Exploring the limits of transfer learning with a unified text-to-text transformer,” Journal of machine learning research, vol. 21, no. 140, pp. 1–67, 2020.

[189] L. Xue, N. Constant, A. Roberts, M. Kale, R. Al-Rfou, A. Siddhant, A. Barua, and C. Raffel, “mT5: A massively multilingual pre-trained text-to-text transformer,” in Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, K. Toutanova, A. Rumshisky, L. Zettlemoyer, D. Hakkani-Tur, I. Beltagy, S. Bethard, R. Cotterell, T. Chakraborty, and Y. Zhou, Eds. Online: Association for Computational Linguistics, Jun. 2021, pp. 483–498. [Online]. Available: https://aclanthology.org/2021.naacl-main.41

[190] A. M. Ismail, S. H. Ab Hamid, A. A. Sani, and N. N. M. Daud, “Toward reduction in false positives just-in-time software defect prediction using deep reinforcement learning,” IEEE Access, 2024.

[191] C. S. Corley, K. Damevski, and N. A. Kraft, “Changeset-based topic modeling of software repositories,” IEEE Transactions on Software Engineering, vol. 46, no. 10, pp. 1068–1080, 2018.

[192] D. Kim, Y. Tao, S. Kim, and A. Zeller, “Where should we fix this bug? a two-phase recommendation model,” IEEE transactions on software Engineering, vol. 39, no. 11, pp. 1597–1610, 2013.

[193] A. T. Nguyen, T. T. Nguyen, J. Al-Kofahi, H. V. Nguyen, and T. N. Nguyen, “A topic-based approach for narrowing the search space of buggy files from a bug report,” in 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011). IEEE, 2011, pp. 263–272.

[194] V. Murali, L. Gross, R. Qian, and S. Chandra, “Industry-scale ir-based bug localization: A perspective from facebook,” in 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP). IEEE, 2021, pp. 188–197.

[195] W. Zou, D. Lo, Z. Chen, X. Xia, Y. Feng, and B. Xu, “How practitioners perceive automated bug report management techniques,” IEEE Transactions on Software Engineering, vol. 46, no. 8, pp. 836–862, 2018.

[196] D. Rolnick, A. Ahuja, J. Schwarz, T. Lillicrap, and G. Wayne, “Experience replay for continual learning,” Advances in neural information processing systems, vol. 32, 2019.

[197] S. Powers, E. Xing, E. Kolve, R. Mottaghi, and A. Gupta, “Cora: Benchmarks, baselines, and metrics as a platform for continual reinforcement learning agents,” in Conference on Lifelong Learning Agents. PMLR, 2022, pp. 705–743.

[198] S. E. S. Taba, F. Khomh, Y. Zou, A. E. Hassan, and M. Nagappan, “Predicting bugs using antipatterns,” in 2013 IEEE International Conference on Software Maintenance. IEEE, 2013, pp. 270–279.

[199] L. Da Silva, P. Borba, T. Maciel, W. Mahmood, T. Berger, J. Moisakis, A. Gomes, and V. Leite, “Detecting semantic conflicts with unit tests,” Journal of Systems and Software, vol. 214, p. 112070, 2024.

[200] L. Da Silva, P. Borba, and A. Pires, “Build conflicts in the wild,” Journal of Software: Evolution and Process, vol. 34, no. 4, p. e2441, 2022.

[201] “Link for bug - 384108,” https://bugs.eclipse.org/bugs/show_bug.cgi?id=384108, 2012.

[202] T. Zimmermann, R. Premraj, and A. Zeller, “Predicting defects for eclipse,” in Third International Workshop on Predictor Models in Software Engineering (PROMISE’07: ICSE Workshops 2007). IEEE, 2007, pp. 9–9.

[203] T. Mccabe, “Cyclomatic complexity and the year 2000,” IEEE Software, vol. 13, no. 3, pp. 115–117, 1996.

[204] “scc tool,” https://github.com/boyter/scc, 2018.

[205] P. S. N. Mindom, A. Nikanjam, and F. Khomh, “A comparison of reinforcement learning frameworks for software testing tasks,” arXiv preprint arXiv:2208.12136, 2022.

[206] X. Chen et al., “An adaptive deep rl method for non-stationary environments with piecewise stable context,” Advances in Neural Information Processing Systems, vol. 35, pp. 35 449–35 461, 2022.

[207] J. Johnson, M. Douze, and H. Jégou, “Billion-scale similarity search with gpus,” IEEE Transactions on Big Data, vol. 7, no. 3, pp. 535–547, 2019.

[208] A. K. Gangwar and S. Kumar, “Concept drift in software defect prediction: A method for detecting and handling the drift,” ACM Transactions on Internet Technology, vol. 23, no. 2, pp. 1–28, 2023.

[209] A. Singh, A. Walenstein, and A. Lakhotia, "Tracking concept drift in malware families," in Proceedings of the 5th ACM workshop on Security and artificial intelligence, 2012, pp. 81–92.

[210] J. Ekanayake, J. Tappolet, H. C. Gall, and A. Bernstein, "Tracking concept drift of software projects using defect prediction quality," in 2009 6th IEEE International Working Conference on Mining Software Repositories. IEEE, 2009, pp. 51–60.

[211] M. Ciniselli, N. Cooper, L. Pascarella, A. Mastropaolo, E. Aghajani, D. Poshyvanyk, M. Di Penta, and G. Bavota, "An empirical study on the usage of transformer models for code completion," IEEE Transactions on Software Engineering, vol. 48, no. 12, pp. 4818–4837, 2021.

[212] A. M. Dakhel, A. Nikanjam, V. Majdinasab, F. Khomh, and M. C. Desmarais, "Effective test generation using pre-trained large language models and mutation testing," Information and Software Technology, vol. 171, p. 107468, 2024.

[213] S. Fakhoury, S. Chakraborty, M. Musuvathi, and S. K. Lahiri, "Towards generating functionally correct code edits from natural language issue descriptions," arXiv preprint arXiv:2304.03816, 2023.

[214] S. Fatima, T. A. Ghaleb, and L. Briand, "Flakify: A black-box, language model-based predictor for flaky tests," IEEE Transactions on Software Engineering, vol. 49, no. 4, pp. 1912–1927, 2022.

[215] K. Jesse, P. T. Devanbu, and A. Sawant, "Learning to predict user-defined types," IEEE Transactions on Software Engineering, vol. 49, no. 4, pp. 1508–1522, 2022.

[216] N. T. Islam, J. Khoury, A. Seong, M. B. Karkevandi, G. D. L. T. Parra, E. Bou-Harb, and P. Najafirad, "Llm-powered code vulnerability repair with reinforcement learning and semantic reward," arXiv preprint arXiv:2401.03374, 2024.

[217] B. Steenhoek, M. Tufano, N. Sundaresan, and A. Svyatkovskiy, "Reinforcement learning from automatic feedback for high-quality unit test generation. preprint (2023)," arXiv preprint arXiv:2310.02368.

[218] G. Dong, H. Yuan, K. Lu, C. Li, M. Xue, D. Liu, W. Wang, Z. Yuan, C. Zhou, and J. Zhou, "How abilities in large language models are affected by supervised fine-tuning data composition," arXiv preprint arXiv:2310.05492, 2023.

[219] 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.

[220] M. J. Mataric, "Reward functions for accelerated learning," in Machine learning proceedings 1994. Elsevier, 1994, pp. 181–189.

[221] F. Dai and M. Walter, "Maximum expected hitting cost of a markov decision process and informativeness of rewards," Advances in Neural Information Processing Systems, vol. 32, 2019.

[222] J. A. Arjona-Medina, M. Gillhofer, M. Widrich, T. Unterthiner, J. Brandstetter, and S. Hochreiter, "Rudder: Return decomposition for delayed rewards," Advances in Neural Information Processing Systems, vol. 32, 2019.

[223] R. Devidze, P. Kamalaruban, and A. Singla, "Exploration-guided reward shaping for reinforcement learning under sparse rewards," Advances in Neural Information Processing Systems, vol. 35, pp. 5829–5842, 2022.

APPENDIX A SUPPLEMENTARY MATERIAL FOR CHAPTER 4

Table A.1 reports the general characteristics of the studied DRL environments.

Supplementary Results: Playtesting in the Blockmaze game

- Table A.2 show the results of Welch's ANOVA post hoc tests regarding the bugs detected by the DRL algorithms on 10,000 steps.
- Tables A.3, A.4, and A.5 show the results of CLES between the best configuration of each framework and selected baselines for all datasets, to assess the effect size of differences.

Table A.1 General characteristics of the studied DRL environments.

	Game Testing	Test case prioritization	
		Pairwise strategy	Pointwise strategy
action space	Discrete	Discrete	Continuous-1D
observation space	Continuous-2D	Continuous-2D	Continuous-2D
Reward distribution	Negative and positive values	Positive values	Positive values
Environment type	Deterministic	Deterministic	Deterministic

Table A.2 Results of Welch's ANOVA and Games-Howell post-hoc test regarding the number of bugs detected by DRL algorithms on a 10k steps budget.

A	B	mean(A)	mean(B)	pval	CLES
A2C_SB	**A2C_TF**	4.24	5.05	0.0	0.38
A2C_SB	DQN_KR	4.24	0.22	0.0	0.99
A2C_SB	DQN_SB	4.24	1.74	0.0	0.92
A2C_SB	DQN_TF	4.24	0.20	0.0	0.99
A2C_SB	**PPO_SB**	4.24	6.61	0.0	0.18
A2C_SB	**PPO_TF**	4.24	5.23	0.0	0.36
A2C_TF	DQN_KR	5.05	0.22	0.0	0.99
A2C_TF	DQN_SB	5.05	1.74	0.0	0.92
A2C_TF	DQN_TF	5.05	0.20	0.0	0.99
A2C_TF	**PPO_SB**	5.05	6.61	0.0	0.30
A2C_TF	**PPO_TF**	5.05	5.23	0.0	0.48
DQN_KR	**DQN_SB**	0.22	1.74	0.0	0.04
DQN_KR	DQN_TF	0.22	0.20	0.07	0.51
DQN_KR	**PPO_SB**	0.22	6.61	0.0	0.0
DQN_KR	**PPO_TF**	0.22	5.23	0.0	0.01
DQN_SB	DQN_TF	1.74	0.20	0.0	0.96
DQN_SB	**PPO_SB**	1.74	6.61	0.0	0.01
DQN_SB	**PPO_TF**	1.74	5.23	0.0	0.07
DQN_TF	**PPO_SB**	0.20	6.61	0.0	0.0
DQN_TF	**PPO_TF**	0.20	5.23	0.0	0.01
PPO_SB	PPO_TF	6.61	5.23	0.0	0.67

Table A.3 Common Language Effect Size between Pairwise-A2C-SB and selected baselines.

	RL-BS1	RL-BS2	MART
	CLES	CLES	CLES
IO	NA	.792	.762
CODEC	NA	.743	**.857**
IMAG	NA	.717	.724
COMP	NA	**.910**	.639
LANG	NA	.766	.699
MATH	NA	.773	.588
PAINT.	**.607**	NA	NA
IOFROL	.344	NA	NA

Table A.4 Common Language Effect Size between Pairwise-DQN-KR and selected baselines.

	RL-BS1	RL-BS2	MART
	CLES	CLES	CLES
IO	NA	.238	.627
CODEC	NA	.244	**.761**
IMAG	NA	.242	.599
COMP	NA	**.348**	.458
LANG	NA	.321	.495
MATH	NA	.272	.568
PAINT.	**.282**	NA	NA
IOFROL	.268	NA	NA

Table A.5 Common Language Effect Size between Pairwise-A2C-TF and selected baselines.

	RL-BS1	RL-BS2	MART
	CLES	CLES	CLES
IO	NA	.238	.619
CODEC	NA	.237	**.766**
IMAG	NA	.224	.583
COMP	NA	**.335**	.451
LANG	NA	.264	.451
MATH	NA	.163	.429
PAINT.	.409	NA	NA
IOFROL	**.559**	NA	NA