Journal of Computer Science and Electrical Engineering
Print ISSN: 2663-1938
Online ISSN: 2663-1946
DOI: https://doi.org/10.61784/jcsee3133

DEEP REINFORCEMENT LEARNING FOR TEST CASE GENERATION AND PRIORITIZATION

JingXuan Guo, JingJing Liu*
School of Railway Intelligent Engineering, Dalian Jiaotong University, Dalian 116045, Liaoning, China.
**Corresponding Author: JingJing Liu*

Abstract: With the increasing scale and complexity of software systems, improving testing efficiency and fault detection capability under limited testing resources has become a critical issue in software testing research. To address the limitations of traditional approaches, which are often static, separately designed, and unable to make full use of dynamic execution feedback in test case generation and prioritization, this paper proposes a unified testing optimization framework based on deep reinforcement learning. The proposed method formulates both test case generation and test case prioritization as a sequential decision-making problem aimed at maximizing testing utility, and develops a feedback-driven strategy learning mechanism through state representation, action selection, and reward design, enabling the model to dynamically adjust testing behavior according to coverage information, fault detection outcomes, and budget constraints. Experimental results show that the proposed method effectively improves statement coverage, branch coverage, and the number of detected faults in test case generation, while also achieving higher APFD and earlier fault detection under limited budgets in test case prioritization. Further ablation analysis demonstrates that coverage reward, fault-related reward, and budget constraint play complementary roles in optimizing the testing strategy. Overall, this study shows that deep reinforcement learning provides an adaptive and unified solution for automated decision-making in software testing and offers a promising direction for the development of intelligent software testing methods.
Keywords: Software testing; Deep reinforcement learning; Test case generation; Test case prioritization

1 INTRODUCTION

With the growing scale and increasing complexity of modern software systems, software quality assurance has become a central issue in software engineering research [1,2]. As a fundamental means of ensuring software reliability and stability, software testing plays a critical role throughout the entire software development lifecycle [3,4]. Among the many challenges in testing, test case generation and test case prioritization are two of the most essential factors affecting both testing efficiency and effectiveness [5]. High-quality test cases help cover program behaviors more thoroughly and expose potential defects more effectively, while an appropriate prioritization strategy makes it possible to detect faults earlier under limited testing budgets. This issue has become particularly important in the context of continuous integration, continuous delivery, and rapid release cycles, where improving testing return at lower cost is now a pressing concern in both research and industrial practice.

To address these challenges, a wide range of approaches have been proposed for test case generation and prioritization. For test case generation, commonly used methods include random testing, symbolic execution, search-based software testing, and evolutionary or genetic algorithms [6]. For test case prioritization, existing studies have explored various strategies based on code coverage, historical fault information, execution cost, and requirement risk [7]. Although these approaches have shown effectiveness in specific settings, they still suffer from several limitations. On the one hand, many of them rely heavily on manually designed heuristics and therefore have limited adaptability in complex and dynamic testing environments. On the other hand, test case generation and prioritization are often treated as two separate tasks, lacking a unified optimization perspective and making it difficult to fully exploit feedback collected during the testing process. In recent years, deep reinforcement learning has demonstrated strong capabilities in sequential decision-making, adaptive control, and intelligent optimization, offering a promising new direction for addressing software testing problems in a more adaptive and data-driven manner.

Motivated by this, this paper investigates a deep reinforcement learning-based approach for the unified modeling and optimization of test case generation and prioritization in software testing. Specifically, the testing process is first formulated as a sequential decision-making problem aimed at maximizing testing utility, where the core elements of reinforcement learning, including state representation, action selection, and reward design, are carefully constructed. On this basis, a DRL-based testing framework is developed to support both test case generation and prioritization, enabling the testing strategy to be dynamically improved through execution feedback. Finally, experiments are conducted to evaluate the proposed approach in terms of test coverage, fault detection capability, and prioritization effectiveness. This work aims to provide a more intelligent and integrated solution for automated decision-making in software testing, thereby improving testing efficiency and quality under limited resource constraints.

2 FRAMEWORK DESIGH

Test case generation and test case prioritization in software testing can both be viewed as decision-making problems aimed at maximizing testing utility [8]. Given a program under test P, an input space I, a candidate test case set T, and a limited testing budget B, the central objective is to improve testing effectiveness as much as possible under resource constraints. In general, testing effectiveness can be reflected in several aspects, such as higher code coverage, stronger fault detection capability, and lower execution cost. Traditional studies often treat test case generation and test case prioritization as two separate problems: the former focuses on constructing effective test inputs from the input space, while the latter aims to determine an execution order that enables earlier fault detection from an existing test suite. However, from the perspective of the overall testing process, both tasks serve the same optimization goal, namely maximizing testing utility under a constrained budget. This makes it necessary to formulate them within a unified framework.

To this end, this paper models the problem as a sequential decision-making process. At each stage of testing, an agent makes decisions based on the current testing state so as to progressively improve subsequent testing outcomes. More specifically, the state $s_t$represents the information in the current testing environment that is relevant to decision-making, which may include current coverage status, the number or distribution of detected faults, historical execution feedback, and the remaining testing budget. The action $a_t$denotes the testing operation taken by the agent at the current step, such as generating a new test case, selecting one candidate test case for prioritized execution, or modifying and extending an existing input. The reward r_t measures the utility brought by the current action and is typically determined by a combination of coverage gain, fault detection performance, and execution cost. Under this formulation, both test case generation and prioritization can be regarded as processes of continuously improving a testing policy through dynamic feedback, which naturally motivates the use of deep reinforcement learning.

Based on the above formulation, the objective of this study is to learn an optimal testing policy πsuch that the agent can continuously adapt its decisions according to environmental feedback and maximize cumulative reward within a limited budget. This objective can be expressed as maximizing the expected return over time steps:

$$\max_{\pi} E\left[\sum_{t=0}^{T} \gamma^{t} r_{t}\right] \tag{1}$$

where $\gamma \in [0,1]$ is the discount factor used to balance immediate reward and long-term utility. This formalization transforms test case generation and prioritization from isolated and static procedures into a unified decision framework oriented toward overall testing benefit. Based on this problem definition, the following section will further introduce the DRL-based testing framework adopted in this work and explain how it enables dynamic optimization of testing strategies through execution feedback.

3 DRL-based Testing Framework

To achieve unified optimization of test case generation and test case prioritization, this paper develops a deep reinforcement learning-based testing framework. The central idea of the framework is to view software testing as a closed-loop optimization process consisting of state perception, decision execution, and feedback-driven policy update, so that the agent can continuously refine its testing strategy through interaction with the testing environment. Unlike traditional approaches that depend on fixed rules or static heuristics, the proposed framework is able to adaptively learn, from execution feedback, which testing actions are more likely to improve coverage, expose latent faults, or yield higher testing utility under limited budget. In this way, test case generation and prioritization are no longer treated as isolated tasks; instead, they are modeled as two closely related components under a shared decision objective, both contributing to the overall improvement of testing effectiveness.

Within this framework, the current testing environment is first characterized by a state representation module, which encodes relevant information such as current coverage status, observed execution outcomes, historical fault feedback, and the remaining testing budget. Based on this state information, the DRL agent selects the next action. For the task of test case generation, actions may correspond to exploring the input space, constructing new test inputs, or mutating and extending existing test cases. For the task of test case prioritization, actions correspond to selecting the next most valuable test case from the candidate test suite for execution. After an action is executed, the testing environment returns feedback signals, such as coverage gain, whether a new fault has been detected, and the associated execution cost. The framework then computes a reward and updates the policy network accordingly. Through this interaction mechanism, the agent gradually approaches a more effective testing strategy by continuously learning from trial and feedback.

From a procedural perspective, the framework can be summarized into three main stages. The first stage is testing state modeling, whose goal is to extract decision-relevant information from the testing process and transform it into a state representation suitable for reinforcement learning. The second stage is policy learning, in which the agent repeatedly interacts with the testing environment, accumulates experience, and optimizes its decision policy, thereby learning how to generate more valuable test cases and how to determine more effective execution orders. The third stage is testing utility optimization, where the learned policy is applied to dynamically select better actions during actual testing so as to balance coverage improvement, early fault detection, and testing cost control. Under this framework, test case generation is treated as the front-end process for improving the quality of the candidate test suite, while prioritization is regarded as the back-end process for improving the efficiency of testing resource utilization. Together, they form a unified feedback-driven testing optimization mechanism.

4 EXPERIMENTS AND ANALYSIS

4.1 Experimental Setup

To evaluate the effectiveness of the proposed DRL-based software testing approach, experiments were conducted from two perspectives: test case generation and test case prioritization. The evaluation focused on three research questions. First, can the proposed method generate higher-quality test cases that improve coverage and enhance fault detection capability? Second, can it produce a more effective prioritization order under limited testing budgets so that faults can be revealed earlier? Third, do the key design components of the unified framework make a substantial contribution to the overall performance improvement? To answer these questions, several representative baselines were considered, including random testing, genetic algorithm-based generation, coverage-based prioritization, and history-based prioritization. The proposed approach was then compared against these baselines in terms of coverage, number of detected faults, APFD, and average execution cost.

In the experimental design, the generation task and the prioritization task were evaluated separately, followed by an ablation study to investigate the impact of reward design on model behavior. For the generation task, the main metrics included statement coverage, branch coverage, and the number of detected faults. For the prioritization task, the analysis focused on APFD, the number of faults detected within the first 30% of the testing budget, and average execution time. All methods were tested under the same budget setting and execution environment in order to ensure a fair and consistent comparison.

4.2 Effectiveness of Test Case Generation

Table 1 presents the comparison results of different test case generation methods in terms of coverage and fault detection capability. As shown in the table, the proposed method outperformed the baseline approaches across all metrics, indicating a clear advantage in overall testing utility.

Table 1 Comparison of Test Case Generation Performance

Method	Statement Coverage (%)	Branch Coverage (%)	Faults Detected
Random Testing	68.4	59.2	21
Genetic Algorithm	74.9	66.8	26
DRL-based Method (Ours)	83.7	76.4	34

The results in Table 1 show that random testing, although simple to implement, performs poorly in both coverage and fault detection, suggesting that it lacks an effective search mechanism in complex input spaces. The genetic algorithm improves test input quality through evolutionary search, increasing statement coverage and branch coverage to 74.9% and 66.8%, respectively, while detecting 26 faults. In contrast, the proposed DRL-based method further raises statement coverage to 83.7%, branch coverage to 76.4%, and the number of detected faults to 34. These improvements suggest that deep reinforcement learning can make more effective use of execution feedback and continuously refine the generation strategy toward input regions that are more valuable for improving coverage and exposing latent defects.

To provide a more intuitive view of the coverage performance, Figure 1 illustrates the comparison of statement coverage and branch coverage across different generation methods. It can be clearly observed that the proposed method maintains a consistent advantage on both metrics. This indicates that the method not only improves the breadth of testing but also strengthens the exploration of critical branch behaviors. Such an advantage is particularly important for triggering deeper program paths in complex software systems, and it also supports the rationale of modeling test generation as a feedback-driven sequential decision-making problem.

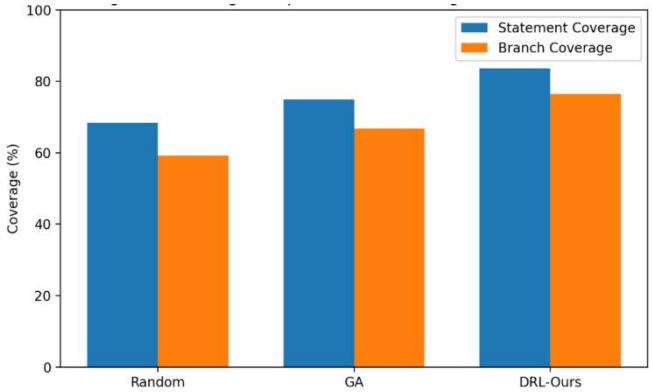

Figure 1 Coverage Comparison of Test Case Generation Methods

4.3 Effectiveness of Test Case Prioritization

In addition to test generation, the proposed framework was further evaluated on the task of test case prioritization. Table 2 reports the comparison results of different prioritization strategies in terms of APFD, the number of faults detected within the first 30% of the testing budget, and average execution time.

Table 2 Comparison of Test Case Prioritization Performance

Method	APFD	Faults Detected in First 30% Budget	Avg. Execution Time (s)
Random Prioritization	0.71	11	418
Coverage-based Prioritization	0.79	16	401
History-based Prioritization	0.82	18	395
DRL-based Method (Ours)	0.89	24	388

As shown in Table 2, random prioritization yields the weakest performance, with an APFD of only 0.71, indicating that it is unable to schedule high-value test cases effectively under a limited budget. Coverage-based and history-based prioritization methods achieve APFD values of 0.79 and 0.82, respectively, demonstrating that static heuristic strategies can improve early fault detection to some extent. However, the proposed method reaches an APFD of 0.89, significantly outperforming all baselines. Moreover, within the first 30% of the testing budget, the proposed method detects 24 faults, whereas the history-based method detects only 18. This result suggests that the DRL framework can more effectively integrate coverage gain, historical feedback, and execution cost to learn a dynamic prioritization strategy that is better suited to the current testing environment.

Figure 2 further visualizes the APFD performance of different prioritization methods. The proposed approach consistently remains the best-performing strategy, indicating its ability to schedule more fault-prone test cases earlier in the execution sequence. This advantage is especially valuable in continuous integration and regression testing scenarios, where testing time is often limited and early fault exposure is of high practical importance. In addition, the proposed method also shows a slight advantage in average execution time, indicating that the improvement in prioritization effectiveness is not achieved at the cost of substantially increased testing overhead. Instead, it reflects a better balance between testing benefit and execution cost.

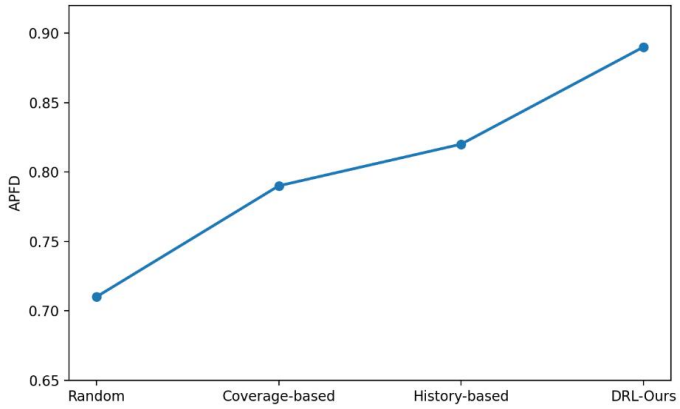

Figure 2 Prioritization effectiveness across different methods

4.4 Ablation Analysis and Discussion

To further examine the contribution of different design factors in the proposed framework, an ablation study was conducted on the reward design. Specifically, the coverage reward, fault detection reward, and budget constraint term were removed one at a time, and the resulting variants were compared with the full model. The results are shown in Table 3.

Table 3 Ablation Study of Reward Design

Setting	Statement Coverage (%)	Faults Detected	APFD
w/o Coverage Reward	78.1	27	0.83
w/o Fault Reward	80.3	25	0.78
w/o Budget Term	84.2	31	0.85
Full Model	83.7	34	0.89

As Table 3 shows, removing the coverage reward leads to a noticeable decrease in statement coverage, while fault detection capability and APFD also decline, indicating that coverage gain plays an important role in guiding the agent to

explore the program state space. When the fault reward is removed, the model still maintains moderate coverage performance, but the number of detected faults drops more substantially, and APFD decreases from 0.89 to 0.78. This suggests that the fault-related reward is crucial for driving the model toward genuinely high-value testing behaviors. By contrast, when the budget term is removed, the model achieves a slightly higher coverage value, but neither fault detection nor APFD reaches the level of the full model. This indicates that maximizing coverage alone does not necessarily lead to the best testing utility, and that a reasonable cost constraint helps the model learn a more practically meaningful strategy.

Figure 3 visualizes the number of detected faults under different ablation settings. The full model consistently achieves the best result, further demonstrating the effectiveness of the proposed multi-objective reward design. Overall, this finding highlights two important characteristics of the framework. First, testing utility should not be driven by a single metric alone; instead, coverage, fault detection, and cost should be considered jointly. Second, the advantage of deep reinforcement learning in software testing comes not only from the learning model itself, but also from a state representation and reward formulation that are well aligned with the practical objectives of software testing.

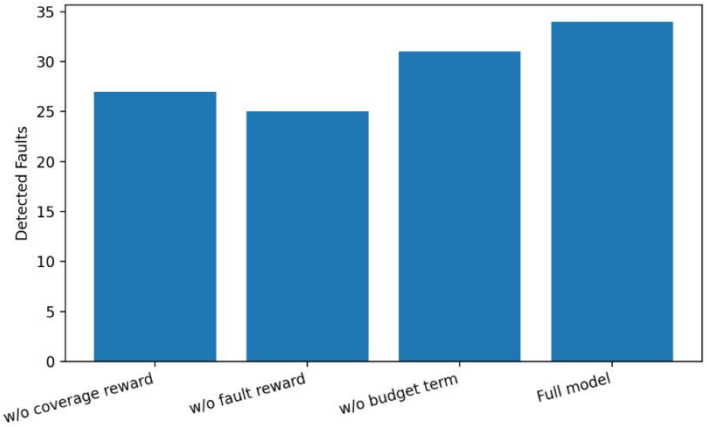

Figure 3 Ablation Study on Reward Design

Taken together, the above results indicate that the proposed DRL-based testing framework is effective in both test case generation and test case prioritization. Compared with random and static heuristic methods, the proposed approach makes better use of dynamic execution feedback and achieves superior performance in terms of coverage, fault detection, and early fault exposure. The ablation study further confirms that unified modeling and reward design are key factors behind the observed improvement. Overall, the proposed method provides an adaptive and extensible solution to automated decision-making in software testing.

5 CONCLUSION

This paper investigated the problem of test case generation and test case prioritization in software testing and proposed a unified testing optimization framework based on deep reinforcement learning. Unlike traditional approaches that treat generation and prioritization as separate tasks, the proposed method formulates both of them as a sequential decision-making process aimed at maximizing testing utility. A corresponding DRL mechanism was constructed through state representation, action selection, and reward design. On this basis, a feedback-driven testing framework was developed, enabling the model to dynamically adjust its strategy according to coverage information, fault detection outcomes, and budget constraints observed during test execution, thereby continuously improving the testing process.

Experimental results demonstrated that the proposed method is effective in both test case generation and test case prioritization. For the generation task, the method was able to produce more valuable test inputs and outperformed representative baselines such as random testing and genetic algorithms in terms of statement coverage, branch coverage, and the number of detected faults. For the prioritization task, the method achieved earlier fault detection under limited testing budgets and obtained superior results on key indicators such as APFD. In addition, the ablation study showed that coverage reward, fault-related reward, and budget constraint play complementary roles in policy learning, indicating that reward design is a critical factor in improving the overall performance of the framework. Overall, the findings of this study suggest that deep reinforcement learning provides an adaptive and unified solution for automated decision-making in software testing.

Although encouraging results have been obtained, there remains considerable room for further improvement. Future work may proceed in several directions. First, the state representation can be further enhanced by incorporating richer program information, such as structural features, control flow, and data flow, so as to improve the model's understanding of complex software behaviors. Second, more efficient and stable reinforcement learning algorithms can be explored to reduce training cost and improve scalability in large-scale testing scenarios. Third, the proposed framework may be extended to a broader range of testing tasks, including regression testing, GUI testing, API testing, and security-oriented testing. Finally, it would also be promising to combine deep reinforcement learning with large language models and program analysis techniques to build more generalizable and intelligent testing systems.

Advancing along these directions may not only further improve testing efficiency and effectiveness, but also promote the development of intelligent software testing toward more practical and in-depth applications.

COMPETING INTERESTS

The authors have no relevant financial or non-financial interests to disclose.

REFERENCES

[1] Pargaonkar S. Synergizing requirements engineering and quality assurance: A comprehensive exploration in software quality engineering. International Journal of Science and Research (IJSR), 2023, 12(8): 2003-2007.
[2] Axelsson J, Skoglund M. Quality assurance in software ecosystems: A systematic literature mapping and research agenda. Journal of Systems and Software, 2016, 114: 69-81.
[3] Homès B. Fundamentals of software testing. John Wiley & Sons, 2024. DOI:10.1002/9781394298976.
[4] Jia X. The Role and Importance of Software Testing in Software Quality Management. Journal of Industry and Engineering Management, 2023, 1(4): 39-44.
[5] Bajaj A, Sangwan OP. A systematic literature review of test case prioritization using genetic algorithms. Ieee Access, 2019, 7: 126355-126375.
[6] Verma AS, Choudhary A, Tiwari S. Software test case generation tools and techniques: A review. International Journal of Mathematical, Engineering and Management Sciences, 2023, 8(2): 293.
[7] Mukherjee R, Patnaik KS. A survey on different approaches for software test case prioritization. Journal of King Saud University-Computer and Information Sciences, 2021, 33(9): 1041-1054.
[8] Singh A, Singhrova A, Bhatia R, et al. A systematic literature review on test case prioritization techniques. Agile Software Development: Trends, Challenges and Applications, 2023: 101-159.