ISSN Number: 2454-2024(Online)

International Journal of Technical Research & Science

AUTOMATED SOFTWARE TESTING USING REINFORCEMENT LEARNING APPROACH

Deepak Upadhyay, Komal Paliwal
E-Mail Id: brightwaysimmigration98@gmail.com
Department of Computer Science Engineering, SITE Nathdwara, Rajasthan, India

Abstract- Software testing plays a critical role in ensuring and evaluating the quality of software by verifying that it functions as intended and does not produce unintended behaviors. Despite advancements in software development methodologies and programming languages, testing remains a crucial component of the development process. While numerous techniques have been proposed to automate software testing, many fail to achieve satisfactory performance in terms of accuracy. This research introduces a reinforcement learning-based method for software testing, which demonstrates an impressive accuracy of 96%, offering a significant improvement over traditional testing approach.

Keywords: Software testing, automated testing, reinforcement learning.

1. INTRODUCTION

Software testing refers to a quality assurance methodology that focuses on assessing the system under test (SUT) by conducting an observation of its operational performance to unmask possible failures [1]. A fault is discovered when the SUT's outer behaviour differs from the one it should have, according to either the requirements or some other description of preferred behaviour. Since this process involves executing the SUT, it is commonly known as dynamic analysis. Conversely, some quality assurance activities, known as static analysis, do not require the SUT's execution. A crucial part of testing is the test case, which specifies the conditions under which the SUT should be executed in order to detect failures. When a test case exposes a failure, it is deemed successful or effective [2]. In order to overcome the impracticality of the exhaustive testing strategy, a number of techniques have been presented, which aid the developers and testers in creating reduced but efficient test suites. Each of these approaches focuses on different parts of a program by means of particular criteria, which specify the test requirements that a test suite should satisfy. Such requirements can be obtained from the software's different sectors, like specifications or implementation. In this context, the SUT can be instrumented to report how well a test suite satisfies these requirements.

1.1 Functional Testing (or black-box testing)

This type of testing relies solely on the SUT's specifications, ignoring internal structure, to generate test cases. Popular criteria within this technique include equivalence partitioning and boundary-value analysis.

1.2 Structural Testing (or white-box testing)

This testing approach is grounded on the SUT's implementation and aims to come up with the test cases that cover all structures (paths, instructions, branches). Structural criteria are normally classified into the two, namely control-flow, and data-flow, categories. The control-flow criteria mainly concentrate on the execution of all the paths or branches, while the data-flow criteria emphasize the flows of the data values such as variable definitions and usages and so on. The all-uses criterion is one of the commonly used data-flow criteria which ensures each data definition and associated use is covered without redefinitions on the path.

1.3 Mutation Testing

Mutation Testing is the process of modifying the SUT so as to simulate common programming errors. These changes, called mutation operators, produce different SUT versions, known as mutants [3]. Testers then design cases to detect these seeded errors. A test case that demonstrates an execution path of a mutant that is different from the original is said to "kill" the mutant. Mutation testing relies on the assumption that a well-designed test suite will kill all non-equivalent mutants, which motivates the testers to readjust the suite if necessary. This technique is mostly used in academic settings.

Software testing is important for ensuring and evaluating the quality of software. Despite advancements in software development methodologies and programming languages, testing is still an important part of the process. Its main purpose is to verify that the software functions as intended and does not produce unintended behaviours,

thus improving the overall quality of the software [4]. However, testing is a resource-intensive and costly activity, often consuming more than 50% of the total software development costs. Also, like any human-driven task, it is prone to errors, and developing reliable software systems continues to be a significant challenge. To solve these problems, researchers and practitioners are actively exploring more efficient and effective software testing methods.A practical method to solving software testing problems is to automate the testing process [5]. As a result, significant efforts are being dedicated to automating these activities. AI (Artificial intelligence), especially ML (machine learning), proves effective in reducing the effort involved in many software engineering tasks. ML, a field that intersects AI, computer science, and statistics, is being applied to automate different aspects of software engineering. Many software testing problems can be framed as learning tasks, making them well-suited to machine learning algorithms. This leads to increasing interest in using ML to streamline and enhance the software testing process [6]. Additionally, as software systems continue to grow in complexity, traditional testing methods often struggle to scale, making ML-based techniques even more appealing for modern software systems.

2. LITERATURE SURVEY

M. Esnaashari, et al. (2021) presented a method for automating test data generation using a standardized approach that focused on covering all finite paths. In this process, the problem was created as a search problem and then solved using meta-heuristic algorithms [7]. The scheme employed a memetic algorithm where reinforcement learning was used as a local search technique within a genetic algorithm. Experimental results showed that this method outperformed many existing evolutionary and meta-heuristic algorithms in terms of speed and test data generation efficiency, achieving better coverage with fewer evaluations. The comparison included conventional genetic algorithms, various genetic algorithm improvements, random search, particle swarm optimization, bees algorithm, ant colony optimization, simulated annealing, hill climbing, and tabu search. M. Waltz, et al. (2024) proposed the T-Estimator (TE) for two-sample testing of the mean, which flexibly adjusted the significance level of the underlying hypothesis tests to interpolate between over- and underestimation [8]. Furthermore, a generalization called the K-Estimator (KE) was introduced, which maintained the same bias and variance bounds as the TE while allowing the use of a nearly arbitrary kernel function. The TE and KE were applied to modify Q-Learning and the Bootstrapped Deep Q-Network (BDQN), and their convergence was demonstrated in the tabular setting. Additionally, a modification of the TE-based BDQN was presented, which dynamically adjusted the significance level to minimize the absolute estimation bias. All the proposed estimators and algorithms were rigorously evaluated and applied to many tasks and environments, showcasing the bias control and performance improvements enabled by the TE and KE. L. Cai, et al. (2022) proposed a lightweight deep reinforcement learning-based application automation testing method called MARTesting (Multi-Attribute Fusion Reinforcement Testing) [9]. This method started by eliminating invalid widgets through a difference operation between the attribute sets of the current and previous states. Next, it abstracted the attributes of all widget elements on a page into a state representation, which served as the input to the neural network. This state was determined by combining the position and text information of the page elements. Finally, this method defined a reward function that incorporated both the novelty of the state and the execution frequency of actions. Experimental results on six open-source applications demonstrated that the MARTesting approach significantly improved code coverage and branch coverage compared to existing methods. L. Cai, et al. (2021) constructed ICCD (Interactive Control Feature Diagram) to improve the DDQN (Deep Double Q-Network) by incorporating a residual network [10]. This improvement allowed the algorithm to take images as input. Furthermore, the original single output action (n*w*h) was split into two successive outputs: one representing the interaction (1, n) and the other representing the position (1, w * h). A new reward function was proposed, combining the interaction frequency with image similarity in the ICCD, aimed to explore different UIs (user interfaces) and ensuring that multiple actions were executed under the same UI. Experiments conducted on five open-source applications showed that the presented method outperformed existing techniques in terms of both code coverage and branch coverage.

S. Saber, et al. (2021) explored automating GUI testing was a complex task due to the need for human involvement in determining actions and evaluating outcomes [11]. This study presented a new method to fully automate GUI testing using deep reinforcement learning. This deep reinforcement learning model autonomously explored the system's states and identifies potential testing sequences. The automated testing agent starts by exploring the environment, learning the most efficient paths to maximize coverage while simultaneously detecting GUI bugs. This method allowed testers to focus more on functional testing, ultimately improving the overall software quality. This work evaluated the model on various industry products, and the results showed a significant improvement in coverage compared to random testing. T. Cao, et al. (2021) explored continuous integration testing introduced the

challenge of sparse rewards in reinforcement learning, as frequent integrations typically result in few test failures. This problem was solved by increasing the number of rewarded test cases [12]. This work proposed a reinforcement learning reward strategy based on TCSD (Test Case Synchronization and Diversity), which rewards both failed test cases and, additionally, selects passed test cases that have the potential to fail. Experiments conducted on six real-world industrial datasets showed that the TCSD method improved the learning efficiency and fault detection capabilities of reinforcement learning, achieving an average 6.35% increase in NAPFD compared to traditional strategies.

M. H. Moghadam, et al. (2020) provided a method for autonomous performance assessment that makes use of self-adaptive techniques and fuzzy logic in addition to model-free reinforcement learning [13]. Without access to the system model and source code, it could determine the best course of action for creating workload- and platform-based test conditions that meet the desired testing goal. In order to solve this problem of uncertainty and improve the precision and flexibility of the suggested learning, fuzzy logic and self-adaptive technique are employed. This study evaluation experiments showed that the suggested autonomous performance testing framework may effectively and adaptably produce test circumstances for a range of testing cases.

3. RESEARCH METHODOLOGY

The technique of reinforcement learning is applied for the automated software testing. Reinforcement learning is a machine learning method that allows agents to gain knowledge by interacting directly with an environment, rather than relying on a pre-existing dataset. Unlike other learning methods, RL focuses on finding the optimal decision-making strategy through trial and error. In this process, an agent performs an action, receives feedback from the environment in the form of a reward, and transitions to a new state based on the environment's response. The agent's goal is to maximize the cumulative reward, which is the total reward accumulated over a series of actions. Q-Learning is a popular RL technique that draws inspiration from behaviourist psychology, and it is grounded in the concept of a Markov Decision Process (MDP), which provides the mathematical framework for modelling decision-making in such dynamic environments [5]. It is expressed as:

- S: Set of possible states
- A: Set of possible actions
- R: Distribution of reward given (state, action) pair
- P: Transition probability i.e distribution over the next state given (state, action) pair
- γ: Discount factor

Fig. 3.1 illustrates the reinforcement learning process and the interaction dynamics between the agent and the environment.

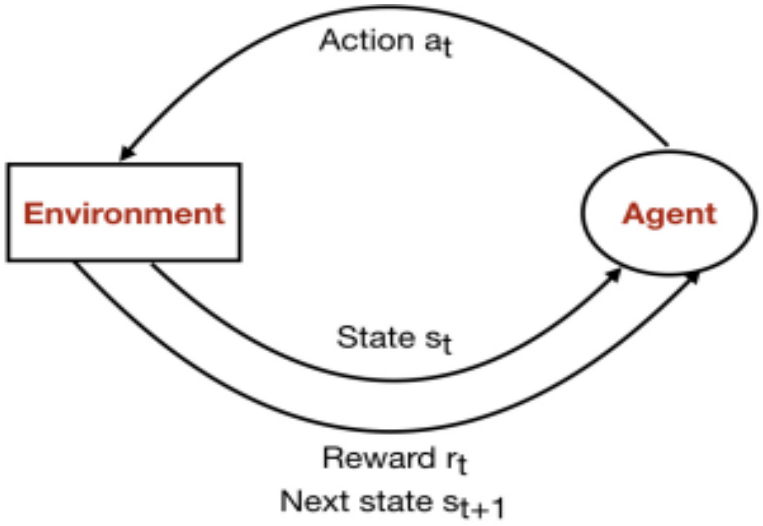

Fig. 3.1 Reinforcement Learning Mechanism

In Q-learning, an agent, at each discrete time step $t = 0, 1, 2, 3.\ldots.$ makes interaction with the environment [6]. Initially, at $t = 0$, the environment begins in a starting state $s_0 = S$. Then at every time step from $t = 0$ until the process is complete:

- The agent chooses an action $a_t \in A(s_t)$ where $A(s_t)$ is represents the set of possible actions in state s_t.

- The environment provides a mathematical reward r_t, sampled according to the reward distribution $R(.|s_t, a_t)$.
- As per the transition probability $P(.|s_t, a_t)$, the environment determines and returns the subsequent state s_{t+1}
- After receiving the reward r_t, agent moves to the new state s_{t+1}.

The agent perceives the environment's current state s_t at every time step t and chooses an action a_t as per a policy π. This policy defines the agent's behaviour in response to the environment. Since the objective is for the agent to act optimally to maximize cumulative rewards, the goal of reinforcement learning is to identify the optimal policy π^* that archives the highest $\sum_{t>0} \gamma^t r_t$, also known as cumulative discounted reward.

In Q-learning, the Q-value function (or Q-function) is defined for a given policy π. The aim here is to evaluate the quality of a state-action pair. This function provides the expected cumulative reward for any state s and action a, assuming the agent starts at s, takes action a, and then follows policy π. The optimum Q-value function, denoted by Q^* represents the highest anticipated cumulative reward that can be achieved from a given state-action pair, considering all probable policies.

$$Q^{\pi}(s_t, a_t) = \pi max \sum_{t>0} (\gamma^t r_t | s = s_0, a = a_t, \pi) \qquad (1)$$

If the optimum state-action values for the subsequent step $Q^*(s_{t+1}, a_{t+1})$ are identified, the best approach is to select the action that makes the expected reward $r + \gamma Q^*(s_{t+1}, a_{t+1})$ maximum. In this case, r is the immediate reward of the current step. Q^* meets the Bellman equation which is:

$$Q^*(s_t, a_t) = R(s_t, a_t) + \gamma a_{t+1} max Q(s_{t+1}, a_{t+1}) \qquad (2)$$

where γ represents the discount-rate parameter ranging between 0 and 1 which determines the balance between immediate and cumulative rewards. A value of γ nearer to 0 gives more importance to immediate rewards, while a value closer to 1 prioritizes cumulative rewards. The optimal policy π^* is thus the one that chooses the action with the highest Q-value, as determined by Q^*. The Q-learning algorithm iteratively estimates the Q-function using equation (2). Initially, the Q-function is set to a default value. After each action a_t taken by the agent from state s_t to s_{t+1}, with the reward r_t, the Q-function is updated using the following formula:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\big(r_t + \gamma a max Q(s_{t+1}, a) - Q(s_t, a_t)\big) \quad (3)$$

Here, α represents the learning rate $(0 \leq \alpha \leq 1)$. The learning rate controls how much the new observation influences the updated estimate of the Q-function. The Q-learning algorithm is proven to converge to the true Q-function when applied to a Markovian environment, with bounded immediate rewards and continuous updates of state-action pairs.

4. RESULTS AND DISCUSSION

This research work is based on the software testing based on agile testing. The heart disease dataset is collected from kaggle. The proposed model is implemented and results of the method is compared in terms of accuracy, precision and recall. The detail description is the parameters is given below:

Accuracy: Accuracy is defined as the number of points correctly classified divided by total number of points multiplied by 100.

$$\text{Accuracy} = \frac{Number\ of\ points\ correctly\ classified}{Total\ Number\ of\ points} * 100 \text{ -(1)}$$

Precision: In pattern recognition, information retrieval and binary classification, precision (also called positive predictive value) is the fraction of relevant instances among the retrieved instances.

$$Precision = \frac{True\ Positive}{True\ Positive + False\ Positive} - (2)$$

Recall: Recall is the fraction of relevant instances that have been retrieved over the total amount of relevant instances.

$$Recall = \frac{True\ Positive}{True\ Positive + False\ Negative} - (3)$$

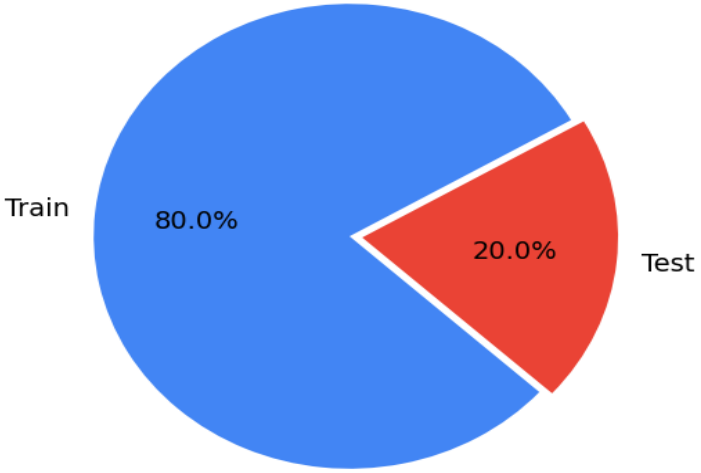

Fig. 4.1 Training and Test Data

As shown in fig. 4.1, the percentage of training and test data is illustrated. The training data is 80 percent and test data is 20 percent

Fig. 3.2 Model Training Information

As shown in fig.3.2, the model training and loss is illustrated in the figure. It is analyzed that training accuracy is achieved upto 96 percent

Table-3.1 Performance Analysis

Model	Accuracy	Precision	Recall
Random Forest	66 Percent	56 Percent	66 Percent
SVM	77.59 Percent	78 Percent	78 Percent
KNN	69.88 Percent	70 Percent	70 Percent
Proposed Model	91 Percent	91.2 Percent	92 Percent

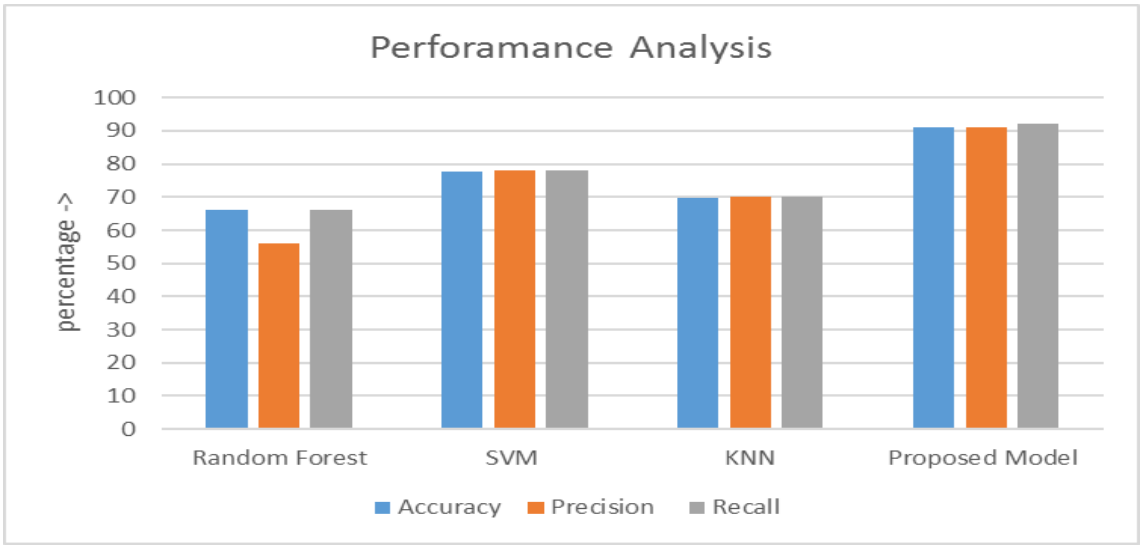

Fig. 3.3 Performance Analysis

The results of the contrast among the proposed methodology, the proposed algorithm, and the current method, KNN classification, is shown in Fig. 3.3. According to the analysis, the proposed method performed better for forecasting defect detection than the current method with respect to of precision, recall, and accuracy.

CONCLUSION

Software testing is important for ensuring and evaluating the quality of software. Despite advancements in software development methodologies and programming languages, testing is still an important part of the process. Its main purpose is to verify that the software functions as intended and does not produce unintended behaviours, thus improving the overall quality of the software. However, testing is a resource-intensive and costly activity, often consuming more than 50% of the total software development costs. To automate the software testing many techniques are proposed in the previous times but those techniques are unable to achieve good performance in terms of accuracy. In this research work reinforcement learning based method is proposed which achieves 96 percent accuracy for software testing.

REFERENCES

[1] C. S. Spahiu, L. Stanescu, R. Marinescu and M. Brezovan, "Machine Learning System For Automated Testing," 2022 23rd International Carpathian Control Conference (ICCC), Sinaia, Romania, 2022, pp. 142-146, doi: 10.1109/ICCC54292.2022.9805972.

[2] N. Jha and R. Popli, "Artificial Intelligence For Software Testing-Perspectives And Practices," 2021 Fourth International Conference on Computational Intelligence and Communication Technologies (CCICT), Sonepat, India, 2021, pp. 377-382, doi: 10.1109/CCICT53244.2021.00075.

[3] L. M. Alrawais, M. Alenezi, and M. Akour, “Security Testing Framework for Web Applications,” International Journal of Software Innovation, vol. 6, no. 3, pp. 93–117, Jul. 2018.

[4] C. Tao, J. Gao and T. Wang, "Testing and Quality Validation for AI Software–Perspectives, Issues, and Practices," in IEEE Access, vol. 7, pp. 120164-120175, 2019, doi: 10.1109/ACCESS.2019.2937107

[5] I. S. W. B. Prasetya, S. Shirzadehhajimahmood, S. G. Ansari, P. Fernandes and R. Prada, "An Agent-based Architecture for AI-Enhanced Automated Testing for XR Systems, a Short Paper," 2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Porto de Galinhas, Brazil, 2021, pp. 213-217, doi: 10.1109/ICSTW52544.2021.00044.

[6] C. Nitin Rajadhyaksha and J. R. Saini, "Robotic Process Automation for Software Project Management," 2022 IEEE 7th International conference for Convergence in Technology (I2CT), Mumbai, India, 2022, pp. 1-5, doi: 10.1109/I2CT54291.2022.9823972.

[7] M. Esnaashari and A. H. Damia, “Automation of software test data generation using genetic algorithm and reinforcement learning,” Expert Systems with Applications, vol. 183, p. 115446, Nov. 2021, doi: https://doi.org/10.1016/j.eswa.2021.115446.

[8] M. Waltz and O. Okhrin, “Addressing maximization bias in reinforcement learning with two-sample testing,” Artificial Intelligence, vol. 336, p. 104204, Nov. 2024, doi:

https://doi.org/10.1016/j.artint.2024.104204. and O. Okhrin, “Addressing maximization bias in reinforcement learning with two-sample testing,” Artificial Intelligence, vol. 336, p. 104204, Nov. 2024, doi: https://doi.org/10.1016/j.artint.2024.104204.

[9] L. Cai, J. Wang, M. Chen and J. Wang, "Reinforcement Learning Application Testing Method based on Multi-attribute Fusion," 2022 9th International Conference on Dependable Systems and Their Applications (DSA), Wulumuqi, China, 2022, pp. 24-33, doi: 10.1109/DSA56465.2022.00013.

[10] L. Cai, J. Wang, M. Cheng and J. Wang, "Automated Testing of Android Applications Integrating Residual Network and Deep Reinforcement Learning," 2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS), Hainan, China, 2021, pp. 189-196, doi: 10.1109/QRS54544.2021.00030.

[11] S. Saber et al., "Autonomous GUI Testing using Deep Reinforcement Learning," 2021 17th International Computer Engineering Conference (ICENCO), Cairo, Egypt, 2021, pp. 94-100, doi: 10.1109/ICENCO49852.2021.9715282.

[12] T. Cao, Z. Li, R. Zhao and Y. Yang, "Historical Information Stability based Reward for Reinforcement Learning in Continuous Integration Testing," 2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS), Hainan, China, 2021, pp. 231-242, doi: 10.1109/QRS54544.2021.00034.

[13] M. H. Moghadam, M. Saadatmand, M. Borg, M. Bohlin and B. Lisper, "Poster: Performance Testing Driven by Reinforcement Learning," 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST), Porto, Portugal, 2020, pp. 402-405, doi: 10.1109/ICST46399.2020.00048.