Advancements in Software Engineering through Reinforcement Learning: A Pathway to Autonomous AI Systems

Dr. Rohan K. Desai and Dr. Leila A. Jensen

Dr. Rohan K. Desai, Department of Computer Science, University of Michigan, Ann Arbor, MI, USA; Dr. Leila A. Jensen, Artificial Intelligence Research Laboratory, Carnegie Mellon University, Pittsburgh, PA, USA

Abstract: Reinforcement learning (RL) has risen as a robust framework for facilitating independent decisionmaking in software engineering, altering the ways systems adjust, enhance, and recuperate from failures. This paper examines how RL contributes to the enhancement of software engineering methods through the facilitation of self-learning and adaptive AI-based systems. The research explores the incorporation of RL across different phases of software development, encompassing design, testing, optimization, and maintenance, enabling autonomous systems that can make intricate decisions with little human involvement. The capability of RL to adapt from immediate feedback enables software to progress according to shifting needs and settings, enhancing system effectiveness, scalability, and robustness. We explore essential applications such as autonomous bug identification, system enhancement, code creation, and dynamic resource allocation in cloudbased settings. The article offers a comparative examination of conventional software engineering methods against RL-based strategies, emphasizing the possible advantages and difficulties of implementing RL in practical software environments. Important performance metrics like system stability, recovery durations, and resource usage are examined regarding RL's efficiency in independently handling tasks that normally need human supervision. We also discuss the challenges and limitations, including the complexities of training RL models, issues of interpretability, and security risks, which must be taken into account when deploying RL-based solutions in operational systems. By means of experimental case studies and simulations, we illustrate the realistic effectiveness of RL in automating routine tasks, speeding up the software lifecycle, and facilitating more resilient and adaptable systems. This paper presents a thorough framework for utilizing RL to extend the limits of autonomous AI in software engineering, setting the stage for smarter, more efficient, and robust software systems.

Keywords: reinforcement learning, autonomous systems, software engineering, AI-driven systems.

I. INTRODUCTION

The potential of reinforcement learning (RL), a branch of machine learning, to automatically learn optimal behaviors through interaction with the environment has attracted a lot of interest. An agent-environment interaction framework serves as the foundation for reinforcement learning (RL), in which the agent gains knowledsge by acting and getting feedback in the form of rewards or penalties. RL's capacity to learn from realtime data, adjust to changing conditions, and make decisions on its own makes it a game-changer in a number of fields, including software engineering. For responsibilities like system design, testing, debugging, optimization, and maintenance, software engineering approaches have historically placed a significant emphasis on human knowledge. However, manual procedures are becoming less and less capable of meeting the demands of scalability, adaptability, and rapid development cycles as the complexity and scope of contemporary software systems continue to increase. By automating decision-making procedures, minimizing human intervention, and enhancing system performance in real-time, the use of RL in software engineering is a viable way to overcome these difficulties.

In recent years, RL has discovered real-world uses in various fields of software engineering, including autonomous bug detection, code optimization, system monitoring, and dynamic resource allocation. RL-driven systems can consistently improve their strategies by utilizing feedback loops that allow them to make realtime modifications based on performance metrics. For example, RL algorithms can be utilized to automatically fine-tune the setup of software systems, develop effective bug-fixing methods, or dynamically distribute computational resources in cloud settings according to varying workloads. The adaptive characteristics of RL allow these systems to evolve continuously, adjusting to emerging challenges and settings, thereby boosting their resilience and scalability. Studies on RL in software engineering have shown that these self-governing systems can surpass conventional techniques in essential performance metrics like fault detection rates, recovery durations, and resource utilization efficiency. Research conducted by Liu et al. (2020) and Wang et al. (2021) indicates that reinforcement learning (RL) can greatly decrease the necessity for human involvement in domains like system optimization and upkeep, resulting in more reliable and efficient software

A primary reason for incorporating RL into software engineering is the transition to increasingly intricate, distributed, and real-time systems. Specifically, due to the emergence of cloud-based platforms, edge computing, and microservices, software systems are becoming more decentralized and dynamic. These systems need to be able to adjust themselves to varying conditions, like changing network traffic, hardware malfunctions, and limited resources. Conventional approaches to system optimization and upkeep, which frequently depend on fixed settings and human interventions, are inadequate to address these requirements. RL, due to its natural flexibility and capacity to enhance choices using real-time information, provides an innovative answer. By engaging in ongoing learning and iterative enhancements, RL-based

systems can independently regulate system behaviors, minimize latency, optimize resource usage, and boost fault tolerance, thus advancing overall system performance.

Although the possible advantages of RL in software engineering are considerable, effectively incorporating RL into production systems also poses difficulties. Training RL models for software tasks typically demands considerable computational power and a significant volume of domain-specific data. Moreover, the interpretability of RL models, which complicates comprehension of their decision-making processes, presents an obstacle to their broad adoption in safetycritical or regulated settings. Additionally, RL's dependence on exploration throughout training may occasionally result in suboptimal or erratic behavior, complicating its implementation in practical applications. Tackling these challenges is essential for the smooth incorporation of RL into software engineering methodologies, and current research seeks to create more reliable, understandable, and effective RL models.

This study examines the revolutionary potential of reinforcement learning (RL) in software engineering, emphasizing its uses, difficulties, and potential to enable self-optimization and self-improvement autonomous systems. We offer a thorough study of the state of RL in this field by examining the body of existing research, giving actual data from experimental case studies, and contrasting RL-driven approaches with conventional software engineering processes. Our goal is to demonstrate how RL has the power to transform software development procedures, improve system performance, and increase the scalability and adaptability of contemporary software systems. Through this investigation, we also talk about potential avenues for further study and pinpoint the major elements that will influence how RL is integrated into software engineering going forward.

II. LITERATURE REVIEW

Software engineering is only one of the many fields where reinforcement learning (RL) is finding growing use. The complexity of contemporary software systems has increased, making traditional approaches to tasks like resource management, performance optimization, and debugging less effective. Autonomous decisionmaking is possible with the incorporation of reinforcement learning (RL) into software engineering, where systems may interact with their surroundings to continually improve. Key contributions to the topic are summarized in this overview of the literature, which also highlights developments in software engineering automation, fault detection, and optimization while contrasting RL-based techniques with traditional methods.

A major application of RL in software engineering is selfsufficient bug identification and troubleshooting. Historically, debugging has been a time-consuming and error-laden task, depending on developers to discover and resolve problems in the code. Nonetheless, RL has demonstrated promise in easing these difficulties. For instance, in their 2018 research, Zhang et al. suggested a deep reinforcement learning method for bug localization that independently detects issue-prone sections in extensive codebases. By presenting the debugging process as a decision-making challenge, the RL agent learns to investigate various sections of the code, earning rewards for accurately detecting bugs and facing penalties for incorrect attempts. Their findings showed that RL surpasses conventional heuristic techniques by attaining greater accuracy in bug identification, substantially diminishing the time developers devote to this activity. A comparable method was investigated by Liu et al. (2020), in which a system based on RL was utilized to automatically create patches for identified bugs. This RL-driven approach showed significant advancement in correcting mistakes with little developer involvement, indicating a move towards increased autonomous software maintenance.

RL has also been utilized in dynamic decision-making and problem-solving within software development processes, especially concerning code generation. A significant contribution is made by Chen et al. (2019), who presented a reinforcement learning-based method to automate code generation. Through the use of RL, the system learned from earlier code-writing instances and user input, independently producing both functional and optimized code. This technique was evaluated on different programming assignments, such as function creation and algorithm development, and demonstrated better results compared to conventional template-based methods. The RL model showed adaptability by accommodating various programming languages and coding styles, indicating its capacity to improve productivity and lessen the manual work needed in code creation. This marks a substantial advancement in utilizing RL to automate creative processes in software engineering.

The application of RL in system upkeep and fault resilience has been examined in various research studies. For instance, Xie et al. (2020) created a system based on RL that independently identified and rectified errors in software systems. Their model employed a mix of reinforcement learning and deep learning to anticipate system failures prior to their occurrence and triggered remedial actions like software updates or resource adjustments. The RL agent in this framework acquired knowledge from system performance metrics, modifying its recovery approaches for various fault types and settings. Their findings indicated that the RL-based method decreased system downtime by more than 30% in comparison to conventional fault tolerance techniques, which typically depend on established recovery protocols or human involvement. In a similar manner, Singh et al. (2021) utilized RL to enhance fault tolerance in cloud systems, employing the agent to independently recover from hardware malfunctions. The system's capability to adjust cloud resources dynamically using real-time data assisted in sustaining system availability, leading to notable enhancements in reliability and performance.

The problem of stability and convergence is another difficulty when using RL in software engineering. To converge to an ideal solution, RL agents frequently need a

lot of training iterations, which can be time-consuming and unrealistic in software development settings that move quickly. Recent work by Zhang et al. (2020) addressed this by utilizing cutting-edge methods like curriculum learning and experience replay to increase the convergence speed of RL algorithms. By using these techniques, RL agents can gain knowledge from a wider range of experiences, which enhances their capacity to navigate intricate and changing settings. Their study's findings demonstrated that the RL model's convergence time was much shortened, increasing the viability of RLbased systems for software engineering real-time applications.

By automating many facets of software development, optimization, maintenance, and fault recovery, reinforcement learning (RL) has the potential to completely transform the area of software engineering, according to the literature on the subject. The aforementioned research show that RL can increase software systems' adaptability in dynamic contexts, save manual labor, and increase efficiency. But there are still a lot of obstacles to overcome, especially in the areas of real-time deployment, model interpretability, and training efficiency. As the subject develops, more study is required to improve RL methods and create answers to these problems so that software engineering processes can use them more widely.

III. METHODOLOGY

This section describes the approach used to look at how reinforcement learning (RL) might improve software engineering procedures, namely in the areas of autonomous system recovery, performance optimization, and bug detection. Three main parts comprise the methodology: (1) formulation of the problem and design of the RL agent; (2) experimental setup, which includes datasets, system configurations, and evaluation metrics; and (3) training, testing, and performance analysis of the model. The study uses an experimental methodology to evaluate the efficacy and viability of RL-driven solutions, utilizing both simulated settings and real-world software systems.

1. Formulating Problems and Designing RL Agents In this work, we concentrate on using RL in three important software engineering domains: fault recovery, performance optimization, and bug identification. We specify a particular RL agent for every application, whose job it is to learn the best course of action based on real-time feedback.

- Bug Detection: A Markov Decision Process (MDP) is used to describe the bug detection agent, and features taken from source code (such as code complexity, prior bug reports, and code structure) make up the state space. The agent's actions include pointing out possible code locations that might have errors. The purpose of the reward function is to give the agent positive feedback when it correctly detects a bug and negative feedback when it doesn't. The agent is trained to maximize bug localization accuracy while minimizing detection time.
- Performance Optimization: In cloud-based systems, the performance optimization agent seeks to dynamically modify system resources (such as CPU, memory, and network bandwidth). While actions reflect different resource allocation decisions (e.g., scaling up/down or load balancing), the state space contains system performance measures like response time, CPU utilization, and memory usage. The reward function penalizes inefficient resource consumption and pays the agent for enhancing system performance (such as maximizing throughput or lowering delay).
- Fault Recovery: By reallocating resources or initiating self-healing actions (such as resuming failing services), the fault recovery agent is intended to independently identify and mitigate system problems. System status indicators (such as failure logs and system health) make up the state space, and the agent's actions entail choosing from a predetermined range of recovery strategies. System downtime or inability to recover results in fines, whereas successful system recovery results in positive benefits.

A Deep Q-Network (DQN) architecture is used by each agent to translate state-action pairs to a Q-value, which calculates the action's long-term reward. Using the Bellman equation, the Q-values are updated iteratively to direct the agent toward the best course of action for every job.

2. Experimental Setup

The experimental setup consists of both simulated environments and real-world applications, enabling a comprehensive evaluation of reinforcement learning (RL) models under different conditions.

- Simulated Environments: Controlled experiments are conducted using simulated environments for tasks such as bug detection and performance optimization. For bug detection, the RL agent is trained on a dataset of software projects with known bugs, where each project is represented by its source code, and bug reports are generated based on historical data. For performance optimization, cloud-based systems with dynamic workloads and resource allocation strategies are simulated, allowing the RL agent to explore various optimization approaches under changing conditions.
- Real-World Applications: To assess the practical applicability of RL, experiments are conducted on a cloud-based microservices architecture and an open-source software project. The cloud-based system is modeled after an e-commerce platform, where RL agents handle resource allocation and failure management. In the opensource software project, the RL agent is responsible for identifying and suggesting fixes for known bugs in the codebase.
- Datasets and System Configurations: Bug detection datasets are sourced from public repositories like GitHub, where issues and bugs are well-documented. For performance optimization, synthetic workloads are generated using load-

testing tools such as Apache JMeter to simulate different levels of user traffic and resource demands. For fault recovery, system logs and failure datasets from real-world cloud environments serve as training and evaluation data for RL agents.

- Evaluation Metrics: The performance of RL agents is measured using task-specific evaluation metrics. For bug detection, precision, recall, F1score, and detection time assess the accuracy and efficiency of the RL agent. In performance optimization, metrics such as throughput, latency, and resource utilization efficiency determine the impact of RL-based optimizations. For fault recovery, key success indicators include system uptime, recovery time, and overall system availability.

3. Model Training and Testing

The training process for reinforcement learning (RL) agents utilizes the Q-learning algorithm with function approximation through Deep Q-Networks (DQN). This approach is particularly effective for handling large state spaces, such as those encountered in software engineering applications. The DQN model consists of an input layer, multiple fully connected hidden layers, and an output layer that predicts Q-values for different actions. Training occurs in episodes, where the agent interacts with the environment, observes states, selects actions, and receives rewards based on the outcomes. The training process follows these key steps:

1. State Representation: In bug detection, the state is defined by extracted code features, while in performance optimization and fault recovery, it consists of system performance metrics or failure indicators.
2. Action Selection: The RL agent chooses actions based on Q-values, employing an epsilon-greedy strategy to balance exploration and exploitation. Initially, the agent explores actions randomly, but as training progresses, it increasingly selects actions with higher Q-values.
3. Reward Function: The reward function is designed to steer the agent toward optimal decisions. Positive rewards are assigned for actions that enhance system performance—such as detecting bugs, optimizing resource usage, or recovering from failures—while penalties are applied for ineffective actions, such as missing bugs, inefficient resource allocation, or failed recovery attempts.
4. Experience Replay: To enhance training stability and efficiency, experience replay is implemented. This involves storing past state-action-reward transitions in a buffer and sampling them randomly to update Q-values. This technique prevents the model from overfitting to recent experiences and allows the agent to learn from a diverse range of scenarios.
5. Model Evaluation: Once training is complete, the RL agent is tested in a new environment or on fresh data to assess its generalizability and robustness. This evaluation ensures that the trained model can effectively handle previously unseen situations.

IV. RESULTS AND ANALYSIS

This section contains the findings from our experiments that assess how well the reinforcement learning (RL) agents perform in tasks related to fault recovery, performance optimization, and bug identification. Comparisons between baseline approaches and the RLbased approach are shown, along with a discussion of the performance criteria used to evaluate each task. To guarantee the reliability and importance of the results, statistical analysis is also included, including p-values and confidence ranges.

1. Bug Detection Capability

In order to detect defects in a sizable open-source codebase, we trained an RL agent in the first round of tests. A baseline approach, a heuristic-based issue detection tool that depends on pre-established patterns like static analysis rules and recognized bug kinds, was The effectiveness of

u

Table 1: Bug Detection Metrics Con

Method	Precision (%)	Recall (%)	
RL-Based Approach	90.6	85.3	
Heuristic-Based Method	78.3	72.1	

the heuristic-based approach and the RL-based agent in bug discovery is contrasted in Table 1 using a number of criteria. The RL agent shows a significant increase in recall and precision, suggesting that it not only finds defects more precisely but also extracts a larger percentage of real bugs from the codebase. The RL agent performs better overall in bug discovery, as evidenced by its higher F1-score, which is a combined measure of precision and recall. Furthermore, the RL agent greatly shortens the detection time, increasing its effectiveness in practical applications.

Analysis of Statistics

A paired t-test was used to compare the heuristic-based method and the RL-based strategy in order to determine the statistical significance of these findings. With a pvalue of 0.01, the results show that the RL agent performs better than the heuristic approach, indicating that the performance difference is statistically significant at the 95% confidence level. This demonstrates that the RL model provides a significant edge in terms of precision and effectiveness for detecting bugs.

2. Enhancement of Performance

The second experiment assesses the RL agent's capacity for performance optimization in a cloud-based system. Based on system performance parameters like throughput, latency, and resource consumption, the task entails dynamically modifying the distribution of resources (CPU, memory, and network bandwidth). A static resource allocation technique, in which resource constraints are pre-established based on workload estimations, is contrasted with the RL-based optimization method.

The throughput, latency, CPU, and memory utilization data are shown in Table 2. The throughput of the RL agent is significantly higher than that of the static allocation approach, reaching 150.2 Mbps as opposed to 120.5 Mbps. This suggests that the RL agent uses the system's resources more effectively, which raises throughput. Simultaneously, the RL agent speeds up system responsiveness by reducing latency to 50.3 ms, which is significantly less than the 75.2 ms latency seen with static allocation. Additionally, compared to the static allocation technique, the RL agent shows more balanced resource consumption, with both CPU and memory utilization being noticeably lower, suggesting increased system efficiency.

Table 2: Performance Optimization Metrics Comparison

Method	Throughput (Mbps)	Latency (ms)	CPU Utilization (%)	Memory Utilization (%)
RL-Based Optimization	150.2	50.3	72.5	68.4
Static Resource Allocation	120.5	75.2	85.3	79.6

The comparison between predefined failover mechanisms and the RL-based recovery strategy is displayed in Table 3. With an average recovery time of 18.5 seconds as opposed to 45.2 seconds for the configured failover technique, the RL agent dramatically cuts down on recovery time. Furthermore, the RL strategy maintains 99.7% uptime and 99.4% availability, which is significantly higher than the 96.5% uptime and 97.8% availability brought about by the failover method. This suggests that RL can recover systems faster and more effectively, reducing downtime and enhancing system dependability.

Analysis of Statistics

A paired t-test was used for the statistical investigation of fault recovery. With a p-value of 0.001, the results demonstrate that the RL-based recovery strategy dramatically shortens recovery time. Additionally, significant p-values of 0.004 and 0.02, respectively, are obtained from the improvement in uptime and system availability. These results demonstrate that RL-based fault recovery is a very successful way to guarantee high system uptime and minimize downtime.

3. Performance of Fault Recovery

In the third experiment, the RL agent's fault recovery skills in a cloud-based setting are examined. The agent's job is to identify and fix system malfunctions by rearranging resources or starting self-healing processes. A baseline approach that makes use of manual intervention and predetermined failover mechanisms is contrasted with the RL-based fault recovery method.

Table 3: Fault Recovery Metrics Com

Method	Recovery Time (s)	Upt (%
RL-Based Recovery	18.5	99
Predefined Failover	45.2	96

V. DISCUSSION

The findings in the preceding section show how reinforcement learning (RL) has a great deal of promise for enhancing software engineering activities in a number of areas, such as fault recovery, performance optimization, and bug identification. These results have important ramifications for software engineering's future of autonomous systems, where RL can be used to increase system responsiveness, efficiency, and dependability. This part covers the main takeaways from the experiments, offers a thorough analysis of the findings, and contrasts them with previous research on RL applications in software engineering.

Bug Identification Efficiency

The RL-based bug detection system showed better performance than the heuristic-based method, as indicated by the increased precision, recall, and F1-score metrics. The RL model's precision of 90.6% and recall of 85.3% indicate that it is both efficient at detecting bugs and

successfully captures a significant share of the bugs in the codebase. The F1-score of 87.8% further emphasizes the well-rounded performance of the RL agent, as it enhances both precision and recall. This signifies a notable advancement compared to conventional bug detection. techniques that depend on fixed rules or pattern-oriented methods, which frequently overlook subtle mistakes in the code (Li et al., 2020; Zhang et al., 2019).

A significant benefit of employing RL for bug detection is its capacity to learn from the surroundings and adjust over time, thus enhancing its detection skills with increased exposure to fresh data. Conversely, heuristic approaches are often constrained by established guidelines that might not consider changing patterns or unknown bug categories (Jones & Heninger, 2021). The RL agent's capacity to adjust to various programming paradigms and changing codebases is a key element that enhances its performance. The statistically significant enhancement in bug detection efficacy, indicated by a pvalue of 0.01, highlights the strength and dependability of the RL model in practical situations.

Enhancing Performance

In the experiments for performance optimization, the RL agent showed an impressive capability to enhance resource allocation in a cloud-based system, attaining greater throughput (150.2 Mbps) and reduced latency (50.3 ms) in comparison to traditional static resource allocation techniques. These findings align with earlier research that has demonstrated the effectiveness of RL in enhancing dynamic systems, where resources must be regularly modified in reaction to fluctuating workloads and system circumstances (Hassani et al., 2021; Lee et al., 2022). The advantage of the RL approach lies in its capability to independently adjust resource allocation according to real-time system performance, unlike traditional static methods that rely on fixed configurations and cannot adapt to changes in resource demand. The reduced CPU and memory usage noted in the RL model (72.5% and 68.4%, respectively) underscores its effectiveness. Static resource allocation techniques frequently result in either over-provisioning or under-provisioning, leading to resource waste or performance issues in the system.

decomposition (Zhao et al., 2021). On the other hand, the RL agent maximizes the utilization of available resources, guaranteeing that the system stays responsive and efficient while avoiding any component overload. The capacity to enhance resource use in realtime is essential for achieving peak performance in cloud settings, where resources are distributed among various users and applications.

The statistical relevance of the performance enhancements, as shown by the p-values for throughput, latency, CPU, and memory usage, underscores the efficacy of the RL-based optimization method. The findings indicate that RL can be successfully utilized to enhance resource management in intricate cloud settings, providing a solution that is both more scalable and dynamic compared to conventional static approaches.

Fault Recovery Efficiency

The RL-based fault recovery system showed better performance than existing failover methods, achieving notably quicker recovery times (18.5 seconds compared to 45.2 seconds) and enhanced system availability (99.4% against 97.8%). These outcomes align with earlier research that emphasizes the capability of RL to improve system resilience and fault tolerance (Ming et al., 2021; Liu et al., 2022). RL's capacity to learn from previous mistakes and foresee the best recovery measures enables it to swiftly and independently address system faults, reducing downtime and guaranteeing high availability.

The reduced recovery time noted in the RL agent indicates that it can better pinpoint the underlying causes of failures and implement corrective measures more effectively than traditional methods, which typically need human involvement or depend on predetermined recovery routes. The capability to independently adjust to various failure situations is a significant benefit of RL in recovery from faults. Additionally, the increased uptime and availability metrics indicate that the RL agent is more dependable and able to uphold system stability during failures.

These findings highlight the promise of RL in missioncritical settings where system outages can lead to considerable financial losses or operational interruptions. By utilizing RL to enhance fault recovery, organizations can guarantee that their systems stay functional despite unforeseen failures, resulting in improved system reliability and business continuity.

VI. CONCLUSION

The findings from our experiments provide compelling evidence that reinforcement learning (RL) can significantly enhance software engineering practices by improving bug detection, performance optimization, and fault recovery. Compared to conventional approaches, RL-based models demonstrate superior accuracy, efficiency, and adaptability, making them a promising solution for dynamic and complex software environments.

A key advantage of RL is its ability to learn and refine decision-making strategies through continuous interaction with the system. Unlike rule-based or heuristic-driven techniques, RL models adapt to evolving software conditions, allowing them to identify patterns and optimize solutions without requiring predefined rules. This adaptability proves particularly beneficial in real-time failure management, where RL can proactively adjust system parameters, allocate resources, and implement corrective actions with minimal human intervention.

The experimental results further indicate that RL-based models can reduce system downtime, enhance fault tolerance, and improve overall software resilience. These improvements are particularly valuable in distributed computing, cloud environments, and largescale applications, where maintaining stability and efficiency is critical. Additionally, RL's ability to generalize across different failure scenarios makes it a versatile tool for

enhancing software reliability in diverse operational contexts.

Despite these advantages, challenges remain in scaling RL models to highly complex systems. The computational demands of training RL agents, the need for extensive and high-quality training data, and ensuring model interpretability are areas that require further exploration. Addressing these challenges will be crucial for deploying RL-based solutions in production environments effectively.

Future work will focus on improving the scalability of RL algorithms, refining their interpretability, and integrating them with complementary AI techniques such as deep learning and evolutionary computing. By leveraging hybrid AI approaches, researchers can further enhance RL's effectiveness in software engineering, enabling more robust, autonomous, and optimized systems across various domains.

REFERENCES

[1]. Tesauro, G. (1995). Temporal difference learning and TD-Gammon. Communications of the ACM, 38(3), 58–68.
[2]. Kaelbling, L. P., Littman, M. L., & Moore, A. W. (1996). Reinforcement learning: A survey. Journal of Artificial Intelligence Research, 4, 237– 285.
[3]. Mnih, V., Kavukcuoglu, K., Silver, D., et al. (2015). Human-level control through deep reinforcement learning. Nature, 518, 529–533.
[4]. Bellemare, M. G., Dabney, W., & Munos, R. (2017). A distributional perspective on reinforcement learning. Proceedings of the 34th International Conference on Machine Learning (ICML).
[5]. Sutton, R. S., Precup, D., & Singh, S. (1999). Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. Artificial Intelligence, 112(1–2), 181– 211.
[6]. Dietterich, T. G. (2000). Hierarchical reinforcement learning with the MAXQ value function decomposition. Journal of Artificial Intelligence Research, 13, 227–303.
[7]. Bertsekas, D. P. (1996). Neuro-Dynamic Programming. Athena Scientific.
[8]. Zambonelli, F., Jennings, N. R., & Wooldridge, M. (2003). Developing multiagent systems: The Gaia methodology. ACM Transactions on Software Engineering and Methodology (TOSEM), 12(3), 317–370.
[9]. Kober, J., Bagnell, J. A., & Peters, J. (2013). Reinforcement learning in robotics: A survey. The International Journal of Robotics Research, 32(11), 1238–1274.
[10]. Silver, D., Huang, A., Maddison, C. J., et al. (2016). Mastering the game of Go with deep neural networks and tree search. Nature, 529, 484–489.
[11]. Schmidhuber, J. (2015). Deep learning in neural networks: An overview. Neural Networks, 61, 85–117.
[12]. Lillicrap, T. P., Hunt, J. J., Pritzel, A., et al. (2016). Continuous control with deep reinforcement learning. International Conference on Learning Representations (ICLR).
[13]. Levine, S., Pastor, P., Krizhevsky, A., et al. (2016). Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection. International Journal of Robotics Research, 37(4–5), 421–436.
[14]. Chandy, M. K., & Schulte, W. R. (2009). Event Processing: Designing IT Systems for Agile Companies. McGraw-Hill.
[15]. Lewis, G., & Wrage, L. (2005). A framework for evaluating self-healing systems. Proceedings of the 3rd International Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS).