The Journal of Systems and Software 210 (2024) 111963

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Coverage-guided fuzzing for deep reinforcement learning systems[☆], [☆☆]

Xiaohui Wan, Tiancheng Li, Weibin Lin, Yi Cai, Zheng Zheng [*]

School of Automation Science and Electrical Engineering, Beihang University, Beijing, 100191, China

ARTICLE INFO

Keywords:
Deep reinforcement learning system
Deep reinforcement learning testing
Fuzz testing
Coverage-guided fuzzing

ABSTRACT

While the past decade has witnessed a growing demand for employing deep reinforcement learning (DRL) in various domains to solve real-world problems, the reliability of DRL systems has become more of a concern. In particular, DRL agents are often trained on data from a potentially biased distribution over environmental settings, causing the trained agents to fail in certain cases despite high average-case performance. Hence, it is necessary and urgent to adequately test DRL agents to ensure the reliability of practical DRL systems. However, due to the fundamental difference in the programming paradigm and the development process, traditional software testing methodology cannot be applied directly to DRL systems. Given that, we introduce a novel testing framework for DRL systems, aiming to generate diverse test cases that can drive a DRL system to fail. Specifically, we design, implement and evaluate DRLFuzz, which is a coverage-guided fuzzing (CGF) framework for systematically testing DRL systems. Experimental results demonstrate that DRLFuzz can efficiently discover diverse failures in different DRL systems for various benchmark tasks. Compared with a random search baseline, DRLFuzz can generate 60% more failed cases in general. Additionally, the diversity of failed cases generated by DRLFuzz is increased by $4.6\% \sim 14.1\%$ in terms of mean pairwise distance (MPD). Furthermore, our experiments also indicate that the failed cases generated by DRLFuzz can be utilized to fine-tune the DRL agent to eliminate the failures resulting from inadequate exploration during training and thus improve the reliability of DRL systems.

1. Introduction

Reinforcement Learning (RL) is a subset of machine learning that enables an agent to learn an optimal policy in a dynamic environment to maximize its accumulated rewards. The policy is learned through interactions with the given environment and observations of how it responds. Over the past decade, deep reinforcement learning (DRL), which combines deep learning (DL) techniques with RL frameworks, has demonstrated impressive and even human-competitive performance in solving sequential decision-making (SDM) problems, such as DeepMind's Atari (Mnih et al., 2013) and AlphaGo (Silver et al., 2016). The encouraging accomplishments also inspired wide deployments of DRL techniques in safety-critical domains, such as autonomous driving (Sallab et al., 2017), intelligent robotics (Zhang et al., 2015), and military application (Yang et al., 2019). In this paper, such applications are termed as DRL systems, wherein the decision-making of a system is facilitated by the DRL module.

Although DRL systems have demonstrated impressive capabilities in various applications, defects in these systems can lead to agents exhibiting unexpected erroneous behaviors and performing poorly in particular cases, which are considered as failures of DRL systems (Uesato et al., 2018; Ruderman et al., 2018). While the likelihood of such failures is low, their consequences, such as vehicle damage or loss of human life, are unacceptable in safety-critical scenarios. In fact, one significant cause of such failures is the unfavorable sample distributions in the state space, triggered by the dynamics and constraints of RL environments. That is, the states explored during training are insufficient to achieve good coverage of the state space (Uesato et al., 2018; Schenke and Wallscheid, 2021). Worse yet, most common RL benchmarks still encourage training and evaluating on the same set of environments (Irpan, 2018; Cobbe et al., 2019). Many cases never appear in the training and testing phase, among which failed cases exists. Hence, it is urgent to promptly discover failures of DRL agents and fix defects in DRL systems, thereby eliminating potential safety risks arising from these failures.

Generating diverse failed cases to uncover potential faults of software systems is indeed one of the main goals of software testing (Myers

[☆] Editor: Aldeida Aleti.

[☆☆] This work was supported by the National Natural Science Foundation of China under grants 62372021.

[*] Corresponding author.
E-mail addresses: xhwan@buaa.edu.cn (X. Wan), li_tiancheng@buaa.edu.cn (T. Li), linwb@buaa.edu.cn (W. Lin), caiy@buaa.edu.cn (Y. Cai), zhengz@buaa.edu.cn (Z. Zheng).

https://doi.org/10.1016/j.jss.2024.111963
Received 11 March 2023; Received in revised form 29 November 2023; Accepted 3 January 2024
Available online 11 January 2024

et al., 2011; Ammann and Offutt, 2016). However, systematic testing of DRL systems presents unique challenges that differ from traditional software systems. In traditional software development practices, developers directly specify the application logic, conditions, and business rules of these systems. Once the software is deployed in production, these hard-coded rules are applied to the data to generate the output (Pei et al., 2017). The objective of testing is to ensure that these hard-coded rules are accurate and written without human mistakes. In contrast, DRL systems are typically developed to solve complex decision-making tasks that cannot be solved with explicit logic. DRL agents, the core component and behavioral subject of DRL systems, have to learn the decision-making logic by interacting with environments (Pei et al., 2017). Each DRL agent is dedicated to exploring how to respond to a given environment in order to maximize cumulative rewards. The fundamental differences in the development process make their testing mindsets different. In particular, the absence of explicit logic brings challenges to DRL testing.

Meanwhile, DRL systems are also different from DL systems. DL systems typically learn from a fixed dataset with ground-truth targets, known as a supervised learning problem, whereas DRL systems learn their decision logic by trial and error using feedback from their own actions and experiences in a given environment (Sutton and Barto, 2018). Specifically, DRL algorithms learn the behavior policy through interactions with the environment and observations of how it responds. The data-driven nature of DRL algorithms requires the agent to be thoroughly acquainted with the dynamics of the environment. Nevertheless, the internal system dynamics may lead to condensed sample accumulation in certain regions of state space, and the resulting insufficient state-space exploration will lead to biased agents and affect the reliability of DRL systems (Schenke and Wallscheid, 2021). For example, a well-trained agent can achieve excellent performance on a set of states that are already explored during training, but it performs poorly in certain regions of the state space where the states have never been visited, manifested in a sharp decline of the cumulative rewards per episode. Hence, this paper aims to propose a systematic testing framework to generate as many failed cases for DRL systems as possible. For one thing, the discovery of failed cases provides a feasible solution to assess the reliability of DRL systems. For another, these failed cases can help estimate which regions of the state space were underrepresented during the training phase and provide guidance for fault fixing and reliability enhancement.

Over the years, significant research efforts have been devoted to developing testing frameworks for traditional software. Among these approaches, coverage-guided fuzzing (CGF) has emerged as both an effective and efficient testing framework (Zalewski, 2014; Serebryany, 2015; Böhme et al., 2017a,b). Furthermore, CGF can be easily deployed and is highly extensible, requiring minimal knowledge of target applications. Therefore, researchers have begun extending CGF techniques to test deep neural networks (DNNs) (Pei et al., 2017; Du et al., 2018; Odena et al., 2019; Xie et al., 2019a). A standout in this domain is TensorFuzz (Odena et al., 2019), which proposed to check the coverage of test inputs based on their activation vectors and subsequently leveraging it to guide the fuzzing process. Inspired by the above facts, we explore to adopt the CGF technique to test DRL systems. The intuition of traditional CGF is that test cases traversing more program-running states are more likely to trigger failures. Many failures of DRL systems are caused by inadequate exploration during training. Hence, a underlying assumption here is that those test cases that adequately cover the state space are more likely to trigger the failure of DRL systems. Another significant advantage of CGF is that it does not require knowledge of the internal mechanisms of the system being tested, making it well-suited for DRL systems with black box characteristics.

However, existing CGF frameworks for DL systems cannot be directly applied to testing DRL systems due to their different characteristics. DL systems primarily solve prediction tasks such as classification and regression. In contrast, DRL systems mainly solve complex SDM tasks. For prediction tasks, each input is usually independently and identically distributed, which is one of the most common assumptions in supervised learning (Pan and Yang, 2009). Therefore, whether the DL system goes wrong or not can be distinguished by comparing the desired output with the one-step forecast output of the system. In contrast, a SDM problem consists of a sequence of state–action pairs. At each time step, an action is chosen and performed based on the current state, the current state and the action subsequently influence the state transitions. Thus we are more concerned with the overall performance of a sequence rather than the right and wrong of a single-step output. This distinction raises a new challenge for determining DRL system's failure and also makes DRL testing different from DL testing. Hence, it is necessary to design a novel CGF framework for DRL systems.

In fact, the essence of the CGF technique is to increase the probability of triggering failures by covering as many states as possible (Chen et al., 2018). However, the state space of RL tasks is usually huge. It is difficult to adequately cover the state space and trigger diverse failures, especially when facing finite testing resources or deadline pressures. Therefore, the key challenges of designing a CGF framework for testing DRL systems are (1) Generating test cases that trigger erroneous behaviors in huge state space and (2) Determining whether the current input exercises a new coverage. To address the above challenges, we design and build DRLFuzz, among which the seed selector, mutator, objective function, and coverage analyzer are designed emphatically. In addition, our experimental results indicate that the failed cases generated by DRLFuzz are also valuable for eliminating the original flaws and improving the performance of DRL systems. Specifically, those failed cases can be used to retrain the DRL model with a small learning rate, which is also known as fine-tuning in the field of machine learning (Finn et al., 2017; Ruder, 2017). To the best of our knowledge, this paper makes several contributions summarized as follows.

1. We design a gradient ascent-based seed mutation strategy for efficiently generating failed cases and an ANN (approximate nearest neighbor)-based coverage analyzer for quickly checking the coverage of a state in fuzzing.
2. We propose a novel CGF framework for DRL testing, namely DRLFuzz, and demonstrate its efficiency and effectiveness for discovering erroneous behaviors of DRL systems in various benchmark RL tasks.
3. We utilize the failed cases generated by DRLFuzz to fine-tune the DRL systems and investigate the feasibility of failure elimination with fine-tuning.
4. We provide a reproduction package[1] to facilitate further comparative studies on DRL testing.

The remainder of this paper is organized as follows: Section 2 provides backgrounds on DRL and CGF techniques. Section 3 uses an example to describe how a well-behaved DRL system fails in certain cases and why we apply the CGF framework for DRL testing. Section 4 provides a detailed description of our proposed CGF framework. The RL tasks and DRL models, experimental setup, and result analysis are presented in Section 5. Section 6 presents some discussions about our work. Section 7 discusses the threats to validity of our results. Section 8 briefly reviews the related work. Finally, the conclusion is provided in Section 9.

2. Background

2.1. Deep reinforcement learning

RL is one of three basic machine learning paradigms, alongside supervised learning and unsupervised learning (Morales, 2020). RL

[1] https://github.com/Wan-xiaohui/DRLFuzz.

differs from supervised learning in not needing labeled input–output pairs to be presented nor for sub-optimal actions to be explicitly corrected. Specifically, RL agents are required to learn optimal actions for specific environmental conditions by trial-and-error, which is required to maximize the cumulated reward (Sutton and Barto, 2018). A general RL problem can be formulated as a discrete-time stochastic control process: at each time step t, an agent perceives a state s_t from the finite state space $\mathcal{S}$, then it selects an action a_t from the finite set of actions $\mathcal{A}$. Upon taking action, the agent receives a reward r_t, and the state of the environment is changed to $s_{t+1} \in \mathcal{S}$. The state transition is determined by the environment dynamics P where $P(\cdot|s,a)$ is the distribution of the next state given the current state s and the selected action a.

Q-Learning is a widely-used traditional RL algorithm (Christopher, 1992), it is also the simplest and most popular value-based algorithm. Value-based algorithms aim to learn a value function, which subsequently makes it possible to define a policy. The value function is usually defined as the total amount of discounted rewards that an agent expects to accumulate over the future, starting from that state. Specifically, the Q-value function $Q^{\pi}(s,a) : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is defined as follows:

$$Q^{\pi}(s,a) = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k} \mid s_t = s, a_t = a, \pi\right].$$

The optimal Q-value function $Q^*(s,a)$ is the largest expected discounted rewards feasible for each state–action pair (s,a) with any policy π, which is defined as:

$$Q^*(s,a) = \underset{\pi}{\operatorname{argmax}} Q_{\pi}(s,a),$$

The optimal policy is one which results in optimal value function:

$$\pi^*(s) = \underset{a \in \mathcal{A}}{\operatorname{argmax}} Q^*(s,a).$$

Hence, if we know the optimal Q-value function, we can get an optimal policy from it. In fact, the goal of Q learning is to find the optimal policy by learning the optimal Q-values for each state–action pair. The basic version of Q-learning keeps a lookup table of values $Q^{\pi}(s,a)$ with one entry for every state–action pair, which is used to approximate the value function. To learn the optimal Q-value function, Q-learning uses the Bellman equation (Dreyfus and Bellman, 1962) for the Q-value function, whose unique solution is the optimal Q-value function $Q^*(s,a)$. In contexts where the state–action space is high-dimensional (potentially continuous), it becomes impractical to independently derive Q-value estimates for each state–action pair. Recognizing this challenge, Mnih et al. (2015) introduced an integration of Q-Learning with deep learning techniques, giving rise to deep Q-networks (DQNs). As an example, a DQN might deploy either a fully-connected network or a convolutional neural network to approximate the value function. Additionally, the structure of the Q network depends on the specific tasks. For simplicity, our paper uses a fully-connected network for value function approximation. The final goal is to learn estimates $Q_\theta(s,a)$ with a DNN with parameters θ, where a loss function has to be minimized:

$$\mathcal{L}(\theta) = \mathbb{E}_{\pi}\left[\left(r_t + \gamma \max_a Q_\theta\left(s_{t+1}, a\right) - Q_\theta(s_t, a_t)\right)^2\right].$$

The DNN reduces the mismatch between the estimate of the value of an action $Q_\theta\left(s_t, a_t\right)$ and the real return by updating its parameters θ. Here, the real return is approximated with TD target $r_t + \gamma \max_a Q_\theta\left(s_{t+1}, a\right)$, where r_t is the reward expectation of the state s_t, and s_{t+1} is a subsequent state of s_t. The difference between the TD target and the current Q value estimate is called the TD error.

In addition, DQNs use two main heuristics to address their instabilities: (1) The first heuristic involves introducing target networks for instantiating fitted Q-learning for some iterations and applying DNN parameter updates only periodically. (2) The second heuristic involves using replay memory (buffer) to store the agent's experience at several previous time steps and applying them as Q-updates during the training. In recent years, DRL techniques represented by DQN have achieved great success over the past few years, as well as achieving human-level performance on various real-world tasks (Mnih et al., 2015; Silver et al., 2016; Jaderberg et al., 2018). In such a context, this paper primarily focuses on testing techniques for DQN-based DRL systems. As the behavioral subject of the system is the DRL agent, the objective of DRL system testing is to discover the erroneous behaviors exhibited by the agent. Supposing that we have developed a DQN agent and suspect that certain test cases (each representing an initial state) may cause it to yield erroneous behaviors, how would we perform testing if those bad cases are hard to generate randomly?

To address this, we focus on testing DRL systems characterized by non-pixel state representations, discrete actions, and a deterministic policy within a stationary yet stochastic environment. Focusing on state-based RL indicates dealing with a state space of moderate size. This allows us to directly analyze the coverage of a test case using the Euclidean distance between state vectors in the state space. A discrete action space ensures each action is explicit and unambiguous, which contributes to achieving consistency and repeatability of tests. Given the nature of safety-critical applications, a deterministic policy is imperative, as they cannot accommodate probabilistic actions. The assumption of a stationary yet stochastic environment aligns well with practical scenarios, denoting a setting where environmental dynamics are stable over time, but their outcomes can be unpredictable (Sutton and Barto, 2018). Furthermore, we exclude the scenario of noisy rewards where an adversary might deceive the agent by altering the rewards (Zhang et al., 2020b; Rakhsha et al., 2021), given the pivotal role of reward information in guiding our fuzzing process.

2.2. Coverage-guided fuzzing

Fuzzing has gained popularity in academia and industry due to its scalability and effectiveness in generating test cases to discover defects in traditional software. In recent years, many efficient solutions or improvements have been proposed to improve the effectiveness and efficiency. Thereinto, feedback-driven fuzzing, especially CGF, has been widely used and proven effective for testing traditional software (Zalewski, 2014; Serebryany, 2015; Böhme et al., 2017a,b). CGF performs systematical random mutations on inputs and generates new inputs to drive the software into diverse states. During the fuzzing process, feedback information can be used to determine where and how to mutate the inputs in the previous iteration.

Given a target program, a typical CGF framework usually performs the following loop (Gan et al., 2018; Xie et al., 2018): (1) **Seed Selection**: It selects one or more interesting seeds from the seed pool. (2) **Seed Mutation**: The selected seeds are mutated a certain number of times to generate new tests with certain mutation strategies, such as bitwise/bytewise flips and block replacement. (3) **Testcase Execution**: The CGF drives the target program to execute the generated test cases and records the executed traces. (4) **Seed Pool Update**: The test cases that trigger crashes are collected and interesting seeds that cover new traces into the seed pool are added. The loop runs until the given computational resource exhausts. Two major components of the state-of-the-art CGF frameworks are *mutation* and *feedback guidance*, which make the fuzzing process more effective and efficient (Xie et al., 2018). They will be the focus of our DRLFuzz framework design.

3. Motivation

3.1. How a DRL agent fails

This subsection includes an interesting RL task, Flappy Bird, to describe how an DRL agent fails in certain cases. Flappy Bird is a side-scrolling game where the agent must successfully navigate through gaps between pipes, which has been integrated into PyGame Learning

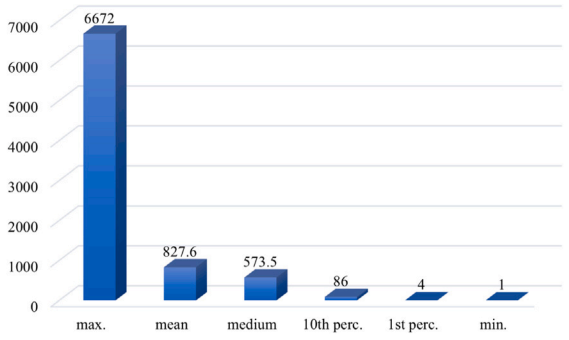


Fig. 1. Statistics of the episodic rewards distribution.

Environment (PLE)[2]. Specifically, the bird can perceive the current pipe until it flies through the pipe. The pipe that just passed can no longer impact the bird, and the bird moves its eyes to the next pipe. The bird receives a +1 reward for being alive for each step, or the game is over if it touches the floor or a pipe. The agent's state space is defined by four variables, including the y positions of the next pipe and next pipe after the next, horizontal distance to the next pipe, and vertical velocity of the bird. Here, the value range is determined by the gameplay and animation rendering, represented as follows.

$$\mathbb{S} = [25, 192] \times [25, 192] \times [-120, -75] \times [-56, 10].$$

A DQN agent was implemented to perform the RL task, more details are available in Section 5. Then, a run of 10 000 complete episodes is used to make a final evaluation of the trained DQN agent, and the performance of the agent is evaluated by cumulated rewards per episode. The distribution of episodic rewards over 10 000 episodes can be represented by descriptive statistics listed in Fig. 1. Thereinto, a percentile, abbreviated as perc. in Fig. 1, is a measure used in statistics indicating the value below which a given percentage of observations in a group of observations fall. The 1st percentile describes the episodic reward below which 1% of all episodic rewards in 10 000 episodes, while the 10th percentile is the episodic reward below which 10% of all episodic rewards in 10 000 episodes.

The results presented reveal that the DQN agent generally exhibits strong performance. Specifically, it achieves an average score exceeding 800, and in the most optimistic scenarios, it can surpass a remarkable 6600. However, it cannot be ignored that the agent performs poorly in some cases. For example, the score is lower than 86 with a probability of 10% and even lower than 4 with a probability of 1%. These cases show that the trained RL agents may perform badly in certain cases, which is not what we expect in practical applications. Much of this inconsistency can be attributed to inadequate exploration during the training phase. Given the continuity of the state space, it is virtually impossible to fully explore within finite time constraints. Compounding this challenge, unknown environmental dynamics and constraints (determined by task characteristics but unknown to the agent) often lead to unfavorable sample distributions in the state space during training (Schenke and Wallscheid, 2021), e.g., condensed sample accumulation in certain regions of state space.

Under the circumstances, the DRL agent may learn the biases in the training distribution in the process of learning its behavior policy, which eventually causes the agent to make a mistake in a part of the state space that the agent did not visit during training. In addition, training data is not independently and identically distributed, which is also different from a common assumption of supervised learning. Imagine that a DRL agent starts from a state never seen previously, it may make a wrong decision with a high probability, then the environment responds to the action and presents a new state to the agent, and the agent then goes on to make an irrational decision. If it continues, errors will start at certain states and accumulate over the length of an episode, and the agent finally obtain low accumulated rewards in a complete episode. Hence, this paper considers those initial states in an episode driving the DRL agent to obtain low episodic rewards as failed test cases.

3.2. Why utilize the CGF to generate failed cases

To uncover potential flaws or undesirable behaviors in the DRL model, we aim to generate a diverse tests to cover as many regions of state space under limited resources and time overhead. It is a major challenge, of course, because the state space of most RL tasks is usually huge. We intuitively grasp this, better coverage results in a higher probability of discovering failed cases. However, most visited cases only cover the same few regions due to the environment's dynamics, and most regions of state space could not be reached during training and testing (Ruderman et al., 2018; Uesato et al., 2018; Schenke and Wallscheid, 2021). As a result, it is not a wise choice to achieve high coverage through large amounts of test case generation and throwing into testing resources. As a baseline method, blind random testing is not only inefficient for discovering system failures but also fails to guarantee test coverage. Hence, it is necessary to develop a smart solution to adequately cover the input space.

CGF is one of the most popular solutions to discover security vulnerabilities in traditional software applications (Li et al., 2018). Specifically, it introduces the code coverage information as feedback in the test iteration to achieve a deep and thorough program fuzzing. That is to say, coverage-based fuzzers aim to generate test cases that cover as much code of programs as possible and expect a more thorough test and detect as many vulnerabilities as possible (Li et al., 2018). Their ideas are very coherent with ours since our goal is to generate diverse states which cover the state space to discover the faults of DRL systems. In addition, compared with other techniques, the prominent advantage of fuzzing is that it is performed in real execution and requires little knowledge of target applications. Therefore, the software under test can be regarded as a black box system in the fuzzing framework, which also fits our situation well. Most DRL systems are considered a black box because the DRL algorithms applied in the systems usually suffer from a lack of explainability (Heuillet et al., 2021). Considering these facts, we eventually apply the CGF framework to generate failed cases of DRL systems.

It is worth noting that this work is an extension of our prior study (Li et al., 2022). In that study, we proposed AgentFuzz, a random fuzz testing approach for DRL systems. AgentFuzz considered the design of both the seed mutator and seed selector, demonstrating notable performance when testing DRL agents in the game of Flappy Bird. However, without integrating state coverage information, it was unable to verify the coverage of test cases following random mutation. The absence of guidance from state coverage information can result in a lack of diversity in the generated test cases, and it also hinders the assessment of test adequacy. Recognizing these limitations, this paper proposes the DRLFuzz method inspirated from coverage-guided fuzz testing rather than blind fuzz testing.

4. Proposed approach

This section starts with an overview of DRLFuzz and then describes the design of each component in detail.

4.1. An overview of drlfuzz

A fuzzer iteratively and randomly generates inputs with which it tests a target program. Drawing inspiration from the fuzzers proposed in the previous studies, the architecture of DRLFuzz is similar to that of coverage-guided fuzzers for traditional software programs. Specifically, in DRLFuzz, the fuzzer starts with a set of initial states (batch inputs) generated by a random generator, and we restrict the inputs to valid

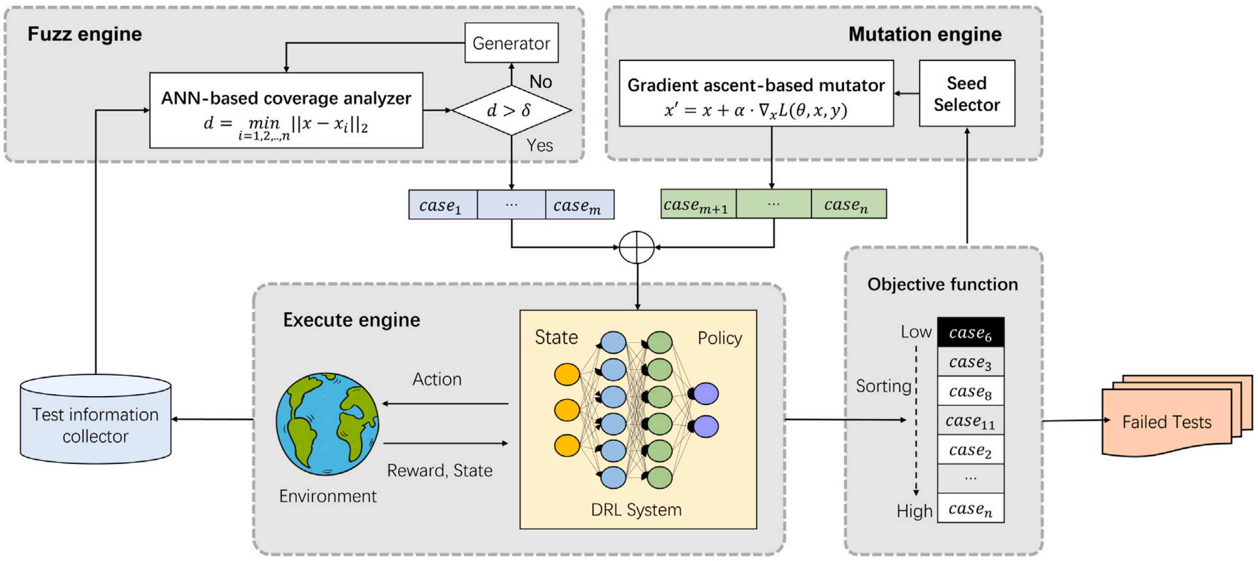

Fuzz engine
Generator
ANN-based coverage analyzer
$d = \min_{i=1,2,...,n} ||x - x_i||_2$
$d > \delta$
No
Yes
Mutation engine
Gradient ascent-based mutator
$x' = x + \alpha \cdot \nabla_x L(\theta, x, y)$
Seed Selector
$case_1$ ··· $case_m$
$case_{m+1}$ ··· $case_n$
Execute engine
Action
Reward, State
Environment
State
Policy
DRL System
Test information collector
Objective function
Low
Sorting
High
$case_6$ $case_3$ $case_8$ $case_{11}$ $case_2$ ··· $case_n$
Failed Tests

Fig. 2. The overall architecture of our proposed framework DRLFuzz.

model inputs with the correct shape and range according to specific RL tasks. In contrast, traditional fuzzers start with generating massive normal and abnormal inputs. These initial states form an initial seed pool. Following this, the target DRL system runs an episode using these batch inputs, the number of which is controlled by the *batch size*. The seed pool will be maintained in the fuzzing loop, each newly visited state that exercises new coverage will be added. This pool is crucial, not only for generating mutated inputs but also as the reference for coverage analysis.

Given the seed pool, the fuzzing proceeds as follows: for each iteration, **Seed Selector** is in charge of selecting some seeds to be mutated. The mutation strategy is defined by a **Seed Mutator**. Since random mutation may generate numerous seeds that fail to trigger erroneous behaviors, we mutate the selected seeds based on gradient ascent algorithm; note that the mutated inputs are still restricted to obeying valid value limits. Meanwhile, we generate more initial states using a random generator and combine those that exercise new coverage with the mutated inputs as initial states of DRL system execution in the next iteration. Two aspects are tracked during execution: the state information and the reward information. The former is utilized to determine whether to exercise new coverage by using a **Coverage Analyzer**, while the latter is utilized to calculate the objective function. **Objective Function** is a user-defined function that asserts a property that the input being fuzzed should obey, e.g., "the cumulative rewards during an episode starting by a certain initial state should not be too small". The mutated input is added to the list of test cases if it satisfies the objective function. Finally, the main fuzzing loop continues until failed test cases are enough or the maximum number of iterations is reached. Fig. 2 depicts the overview of DRLFuzz, and Alg. 3 specifies the main procedure.

4.2. Components of DRLFuzz

Seed Selector: At any given time, the fuzzer selects seeds from the seed pool at the beginning of a new iteration test in the main fuzzing loop. Obviously, how to select appropriate seeds from the pool is an important open problem in fuzzing. It has been proven that an ideal strategy could considerably improve the efficiency of the fuzzing process and help discover defects faster and more (Han and Cha, 2017; Böhme et al., 2017b; Xie et al., 2019a). In traditional CGF, an ideal strategy typically selects those seeds that cover new paths and more likely to trigger defects. However, for DRL testing, it is difficult to guide the generation of states that exercise new coverage and trigger erroneous behaviors simultaneously. Hence, we select those newly added seeds that drive the agent to obtain a low episodic reward in the last iteration. Actually, the above-selected seeds are part of batch inputs for the next iteration, another part of batch inputs are presented later.

Seed Mutator: Once the seed selector has selected a set of seeds to mutate, the mutations must be applied. The mutation strategy is another key problem in DRLFuzz, because it is responsible for generating new test cases based on the batch inputs obtained in the previous iteration. Blind mutation of test cases may lead to a waste of testing resources, and a suitable mutation strategy can improve the efficiency of the fuzzing process. Good test cases could target potentially vulnerable regions of state space and bring a faster discovery of system failures. Thus, we propose a gradient ascent-based mutator, inspired by adversarial attacks on DNN models (Goodfellow et al., 2014; Miyato et al., 2018). The selected inputs are mutated along the direction of the Q network's loss gradients with a tiny step. Alg. 1 provides a complementary depiction of the mutator.

Algorithm 1: Gradient ascent-based seed mutator

```
Input: Env: RL environment, model: Q network, $s_t$: state, $\gamma$:
       discount factor, $\alpha$: mutation size
Output: $s_t'$: mutated state
// Select action with highest Q-value
1 $Q(s_t, \cdot) \leftarrow$ model.predict($s_t$);
2 $a_t \leftarrow \arg\max_a Q(s, \cdot)$;
// Env returns the next state and reward
3 $s_{t+1}, r \leftarrow$ Env($s, a_t$);
// Calculate Q-value of next state and TD target
4 $Q(s_{t+1}, \cdot) \leftarrow$ model.predict($s_{t+1}$);
5 $Q_{target}(s_{t+1}, \cdot) \leftarrow r + \gamma \cdot \max_a Q(s_{t+1}, \cdot)$;
// Mutate the current state using gradient
   ascent
6 $s_t' \leftarrow s_t + \alpha \cdot \nabla_{s_t} \|Q_{target}(s_{t+1}, \cdot) - Q(s_t, \cdot)\|_2$;
```

The gradient of the Q function's loss provides us with a direction for seed mutation. The underlying idea is that a higher TD-error indicates a significant discrepancy between the agent's prediction and the actual reward, suggesting that the state might be novel or rarely visited. Such an intuitive perspective has been widely accepted and extensively applied in RL research (Andre et al., 1997; Schaul et al., 2016; Horgan et al., 2018). However, it may be inadequate to rely solely on TD-error for guiding mutation direction. On one hand, TD-error can be susceptible to the approximation errors inherent in the value function, particularly when function approximators like neural networks are employed. On the other hand, TD-error is a local measure and might not capture the global novelty of a state in relation to all other states. Conversely, examining the distribution of state vectors within the state space may offer a more comprehensive perspective on novelty (Bellemare et al., 2016; Tang et al., 2017).

In light of this, we also use the randomly generated initial states that exercise new coverage in the previous step as test cases. Both parts of the test cases should be truncated to ensure that each test case is a valid input for the DRL system being fuzzed. In other words, test cases in each iteration of DRLFuzz consist of two parts, mutated inputs and randomly generated inputs, the proportion of which is determined by a pre-defined parameter named *mutation ratio*. We denote the former as the “mutation engine” and the latter as the “fuzzing engine”. The combination of random generation and smart mutation techniques has been validated in several studies to enhance the efficiency of fuzzers (Liang et al., 2019; Lyu et al., 2018; Lee et al., 2023; Zhao et al., 2019). Some studies have termed this approach as “hybrid seed mutation”. The central principle behind hybrid seed mutation is that random generation and smart mutation strategies can be combined to maintain a dynamic balance between global exploration and deep search (Liang et al., 2019).

Objective Function: Generally, objective function is user-defined and it asserts a property that the input being fuzzed should obey. In this paper, we expect the DRL system to expect the DRL agent to perform well in any situation, i.e., to achieve high episodic rewards. Meanwhile, the goal of DRLFuzz is to generate test cases violating the property. When mutated inputs are fed into the DRL agent, both state and reward information are tracked. The episodic reward is then calculated and compared to a predetermined value, r_{min}. The objective function returns a boolean, where True indicates a violation of property and false indicates an unsuccessful violation. Moreover, in order to trigger the failures as soon as possible, we terminate the current episode and start a new one if the cumulated rewards exceed the upper bound r_{max}. Both r_{max} and r_{min} are set in advance.

Coverage Analyzer: The coverage analyzer is in charge of reading the runtime information, especially the state information, and checking whether the input exercises a new coverage. Here, the coverage analyzer is designed to check if the DRL system is in a state it has not been in before to explore new regions of state space constantly. Besides, this check has to be fast, and the calculation should be as simple as possible for fuzzing efficiency. Thus, a feasible solution for coverage checking is to determine whether the current state is close to one executed previously.

One way to achieve this is to use a kd-tree based ANN algorithm, a widely used solution to search nearest neighbors in a high-dimensional space (Maneewongvatana and Mount, 1999). Specifically, for the generated initial states, we look up its nearest neighbor in the seed pool, then check how far away the nearest neighbor is according to Euclidean distance. Next, the initial state will be added to the seed pool if their distance is greater than an adjustable threshold δ. However, it is time-consuming to build a kd-tree, and is impractical to build a kd-tree to check the coverage at each time step of an episode. Hence, we adopted a compromise that the kd-tree is built before each iteration and is no longer updated in the current iteration. Each iteration subsequently has a batch of test inputs as the initial states of an episode. During each episode, the current state will be added to the seed pool if it satisfies both of the following conditions: the distance to the last one added is greater than a inner threshold δ', while the distance to its nearest neighbor in the seed pool is greater than another adjustable threshold δ. Alg. 2 shows the details.

However, a question still remains: how can we determine the appropriate value for the threshold δ ? The answer affects the granularity of a coverage metric. The granularity should be appropriate, as neither too fine nor too coarse. If the granularity is too fine, most inputs would trivially yield new coverage. Then, we would just be doing a random search over the whole state space and then generating the mutated inputs by adding adversarial perturbations. Conversely, if the granularity is too coarse, inputs never exercise any new coverage, leading to an infinite loop containing coverage checking operations. In addition, as fuzzing continues, the seed pool increases, and randomly generated inputs have more difficulty exercising new coverage. Therefore, an adaptive adjustment strategy is introduced to check the coverage of generated inputs. Specifically, δ will exponentially decay if we generate the inputs continuously for a fixed number of times T but fail to exercise new coverage. The strategy enables our coverage metric to be refined from coarser to finer during the fuzzing process. It helps to avoid the problems of early stagnation of fuzzing due to a persistent inability to exercise new coverage at a coarse-grained level. More details are provided in lines 14–20 of Alg. 3.

Algorithm 2: ANN-based coverage analyzer

```
Input: Env: RL environment, model: Q network, $s_t$: state, $tree$:
       kd-tree, $r_{max}$: maximum reward, $\delta$: delta, $\delta'$: inner delta,
       $S$: seed pool
Output: $r_c$: episode reward, $S$: updated seed pool
1  Initialize $r_c \leftarrow 0$;
2  while ! GameOver() AND $r_c < r_{max}$ do
       // Select action with highest Q-value
3      $Q(s_t, \cdot) \leftarrow$ model.predict($s_t$);
4      $a_t \leftarrow \arg\max_a Q(s_t, \cdot)$;
       // Env returns the next state and reward
5      $s_{t+1}, r \leftarrow$ Env($s_t, a_t$);
       // Update the cumulative reward
6      $r_c \leftarrow r_c + r$;
       // Add new seeds into seed pool
7      if $Distance(s_{t+1}, s_t) > \delta'$ then
8          $s_t \leftarrow s_{t+1}$;
9          if tree.NearestNeighborDistance($s_{t+1}$)> $\delta$ then
10             $S$.append($s_{t+1}$);
```

5. Experiments

In this section, various experiments are conducted to investigate the effectiveness of our proposed testing approach and the effect of the system repair approach by fine-tuning.

5.1. RL tasks and DRL models

This paper uses the open-source RL environments from the “Pygame Learning Environment” (PLE) (Tasfi, 2016). A distinct advantage of PLE environments is that they can be manipulated while running because full source code for each task is easily accessible. Specifically, three tasks are chosen in our experiments, including Flappy Bird, Catcher, and Pong. They are commonly used in DRL studies to test the performance of DRL algorithms (Mnih et al., 2013; Liu et al., 2018; Kaplanis et al., 2018; Liu et al., 2019; Fu et al., 2019; Ramakrishnan et al., 2020; Zhu et al., 2020), indicating that they are representative as benchmark environments. A brief description is presented below, and more details are available on the official website of PLE.[2]

Flappy Bird: In this task, we need to train an agent to navigate through gaps between pipes. The state of the agent can be defined by

[2] https://github.com/ntasfi/PyGame-Learning-Environment.

Algorithm 3: DRLFuzz

```
Input: Env: RL environment, model: Q network, $\alpha$: mutation
       step, $\beta$: mutation ratio, $\gamma$: discount factor, $\delta$: delta, $\delta'$:
       inner delta, $\lambda$: decay ratio, $r_{min}, r_{max}$:
       maximum/minimum reward
Output: $F$: failed cases
   // Initialize the seed pool and batch inputs
1  $S, C \leftarrow$ Randomly generated test cases;
   // Build the kd-tree of seed pool
2  $tree \leftarrow buildKdTree(S)$;
   // The fuzzing process starts from here
3  for $i$ from 1 to $MaxIter$ do
4      for each $c$ in batch inputs $C$ do
           // Obtain the execution information
5          $r_c, S$ = CoverageAnalyzer(
             Env, model, $c$, $tree$, $\delta$, $\delta'$, $S$, $r_{max}$);
6          Record the episode reward $r_c$of $c$;
           // Store failed test cases
7          if $r_c < r_{min}$ then
8              $F$.append($c$);
       // Update the kd-tree of seed pool
9      $tree \leftarrow buildKdTree(S)$;
10     $S' \leftarrow$ the bottom $\beta$% of $S$ based on $r_c$;
       // Generate new batch inputs
11     $C \leftarrow \emptyset$;
12     for each $s'$ in $S'$ do
13         $C$.append(Mutator(Env, model, $s'$, $\gamma$, $\alpha$));
14     while $len(C) < BatchSize$ do
15         while True do
16             if No coverage for a long time then
17                 $\delta \leftarrow \delta \times \lambda$;
18             $c \leftarrow$ Randomly generated test cases;
19             if $tree.getNNDistance(c) > \delta$ then
20                 break;
```

four controllable variables. At every point in time, there are two possible actions — click or do nothing. More details have been presented in Section 3.1.

Catcher: In this task, there are four variables to define the agent's state. The agent must catch falling fruit with its paddle. It receives a +1 reward for each successful fruit catch, while the episode is over if the fruit is not caught.

Pong: This task simulates 2D table tennis. The state of the agent is defined by six controllable variables. The agent controls the left paddle to hit the ball back to the right side. It receives a positive +1 reward for each successful hit, while the episode is over when the agent fails to hit the ball.

These tasks represent a category of state-based SDM tasks, distinct from pixel-based tasks. They leverage abstract state representations, eliminating the need to process vast pixel data, also making coverage guidance based on state space feasible. Despite their simplicity, they faithfully capture the intricacies of real-world tasks. Notably, each dimension of the state in these games carries distinct physical meaning, ensuring that any modifications remain grounded in real-world contexts. These environments are configurable, allowing for programmatic modifications to various configuration parameters, such as initial conditions at the start of each episode. This mirrors real-world tasks, such as robot control, where configurable parameters might include the robot's initial position, speed, and distance from the nearest obstacle. We adjust these physical attributes to allow the agent to execute its policy from our specified starting state, completing an episode. Our goal is to strategically modify these attributes to introduce the agent to unfamiliar states, causing it to quickly (or with a higher probability) collide with obstacles, and consequently triggering failures.

Details for DRL models and training This paper focuses on testing techniques for DQNs. The theory of the DQN algorithm has been provided in Section 2.1. Three DQN agents with different model structures were implemented for different RL tasks to ensure that the experimental conclusions were more extensive. More specifically, for the Flappy Bird game, we use a 4-layer fully connected network (7, 128, 64, 2). For the Catcher, we use a 3-layer fully connected network (4, 128, 3). For the pong, we use a 4-layer fully connected network (6, 32, 64, 32, 3). In addition, all of them use ReLU as the activate function. During the training process, we use the Adam optimizer with learning rate $\alpha = 0.001$, batch size = 1024, and discount factor $\gamma = 0.9$. All of them are trained for 50 000 epochs, and meanwhile, the ϵ-greedy approach with $\epsilon = 0.1$ is applied to balance exploration and exploitation.

5.2. Experimental design

DRLFuzz aims to generate filed cases that lead DRL systems to perform poorly during an episode. Thus, the definition of DRL system failure is different from the previous work related to DRL testing (Al-Nima et al., 2021). In addition, the existing DL testing approaches cannot be applied to DRL testing. In this section, we aim to answer the following research questions:

1. **RQ1:** How effective is DRLFuzz for generating failed cases when comparing with two baseline methods?
2. **RQ2:** What is the effect of the proposed gradient ascent-based mutator and ANN-based coverage analyzer for DRL fuzzing?
3. **RQ3:** How to use the generated failed cases to repair DRL systems?

For **RQ1**, we evaluate the effectiveness of DRLfuzz by comparing it with two baseline methods, random search, and TensorFuzz. The random search baseline refers to randomly generating test inputs from the state space. Meanwhile, TensorFuzz is a coverage-guided fuzzing approach. The main difference between TensorFuzz and DRLFuzz is that DRLFuzz determines coverage on the raw state space, whereas TensorFuzz determines coverage using the activation vector from the final hidden layer of neural networks. DRLFuzz and the two baselines perform identical iterations, then we compare the efficiency and diversity of the generated failed test cases. Here, the diversity of test cases is measured by the mean pairwise distance (MPD), which is a commonly used diversity metric in the multi-dimensional space (Harrison and Klein, 2007; Biemann and Kearney, 2010; Tucker et al., 2017). MPD is the mean of all pairwise distances between individuals, which is defined as follows:

$$\frac{\sum_{i=0}^{|P|} \sum_{j=0,j/i}^{|P|} \operatorname{dist}\left(x_i, x_j\right)}{|P|(|P|-1)},$$

where P is the number of individuals and $\operatorname{dist}\left(x_i, x_j\right)$ calculates the Euclidean distance for each pair of the P individuals. A larger MPD value indicates a better diversity.

RQ2 investigates the effect of gradient ascent-based mutator and coverage analyzer in the DRLFuzz framework. For **RQ2**, we design two variants for the comparative study. **variant 1** only utilizes the gradient ascent-based search approach to mutate the initial seeds without utilizing a random generator to obtain new seeds. In contrast, **Variant 2**, which corresponds to AgentFuzz from our prior work (Li et al., 2022), is introduced as another variant of DRLFuzz for comparative analysis. **Variant 2** combines the gradient ascent-driven mutation approach with a random generator but does not consider state coverage in fuzzing. DRLFuzz and the two variants perform identical iterations, then we compare the efficiency and diversity of their test cases.

RQ3 aims to investigate the feasibility of failure elimination with fine-tuning. In traditional software testing, a fault of software system

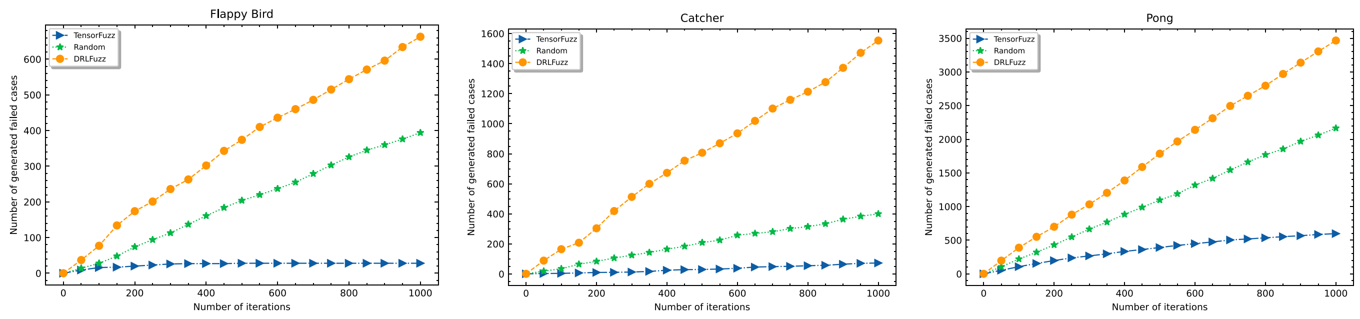


Fig. 3. Number of generated failed test cases vs. Number of iterations

is usually caused by human mistakes and defects. As long as these mistakes and defects are discovered and resolved, the failures of traditional software system can be eliminated effectively. When it comes to DRL systems and other intelligent systems, however, things become complicated. The decision logic of intelligent systems is learned from historical data and cannot be specified by developers. In other words, system failures can be caused by model training rather than implementation mistakes. Such faults are difficult to locate and eliminate due to the black box nature of DL/DRL algorithms, and fine-tuning is one of the most common solutions (Yan et al., 2018; Vaibhav et al., 2019; Chen et al., 2020; Wang and Su, 2020; Dong et al., 2021). Fine-tuning refers to training a model initialized with pre-trained weights (Erhan et al., 2010). Here, we utilize the failed test cases obtained by DRLFuzz to fine-tune the weights of the trained DRL agent by continuing the back-propagation. The test performance of DRL agents before and after fine-tuning will be analyzed.

All experiments were run on a Linux laptop running Ubuntu 16.04 (on an Intel Xeon Silver 4108@1.80 GHz Processor, a 256 GB DDR4 memory, and a NIVIDA GTX 1080Ti GPU).

5.3. Answer to RQ1

Fig. 3 shows the number of failed test cases generated along with the increasing of iteration numbers. Each iteration corresponds to a batch of test inputs. As shown in Figs. 3 and 4(a), the two baselines can also generate failed test cases, but their efficiency is relatively low. Notably, TensorFuzz performs significantly worse than random search, indicating that merely maximizing the coverage of the latent space is inadequate. For one, the latent space is a high-level abstraction of the raw state space, potentially lose crucial spatial information. Furthermore, achieving coverage in the latent space cannot provide directional guidance for generating failed cases. In contrast, DRLFuzz consistently generates more failed cases than the two baselines. Specifically, in the three tasks, DRLFuzz generates 663, 1554, and 3467 failed cases with 1000 iterations, marking improvements of 68.3%, 288.5%, and 60.1% compared to random search.

Furthermore, as observed in Fig. 4(b), the diversity of failed test cases generated by DRLFuzz is greater than that of the two baselines in terms of MPD. TensorFuzz performs still worse than random search, which is intuitive when you consider that the coverage of latent space does not necessarily imply the coverage of input space. In addition, the diversity of failed test cases generated by DRLFuzz is greater than that of the two baselines in terms of MPD. Specifically, the MPD values of failed cases generated by DRLFuzz are improved by 14.1%, 6.7%, and 4.6%, respectively, compared with random search. It goes against our initial intuitive judgment that random search can generate more diverse test cases. A plausible explanation for this could be that all failed cases generated by DRLFuzz are filtered through the coverage analyzer, ensuring that these cases are never too close in the high-dimensional Euclidean space. In contrast, randomly generating test cases in the state space can lead to a concentration of failed cases, resulting in relatively small pairwise distances. In conclusion, neither random search nor TensorFuzz can ensure the diversity among the failed cases generated within the state space.

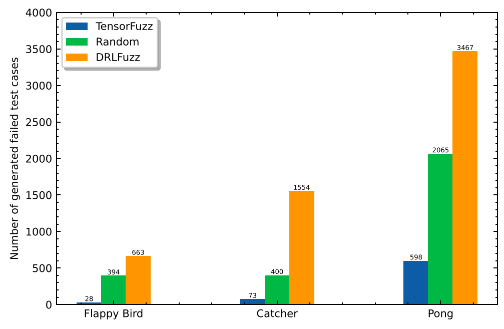
DRLFuzz generates failed cases more efficiently than random search and TensorFuzz. Moreover, the diversity of the failed cases generated by DRLFuzz is superior to those of the two baselines in terms of MPD.

5.4. Result analysis for RQ2

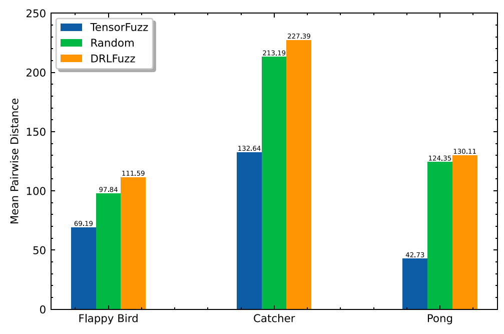
Two variants of DRLFuzz are implemented and then compared with DRLFuzz to analyze the effects of the gradient ascent-based mutator and the ANN-based coverage analyzer. As shown in Figs. 5 and 6(a), all three methods can generate failed test cases. However, their generation efficiency is different. Specifically, **variant 1** generates the least amount of failed test cases with 1000 iterations, which is even less than the random search baseline. In the three games, it only generates 73, 92, and 288 cases, respectively. Moreover, the changing curves of **variant 1** are almost stagnant during the later stage of iteration. This suggests that the mere use of a gradient ascent-based mutation strategy is not enough. It mutates a batch of inputs along the direction of loss gradients of Q networks, which may result in the lack of global exploration in the state space.

In addition, **variant 2** generates more failed test cases during 1000 iterations than **variant 1**. Specifically, it generates 534, 1405, 3268 cases in three tasks, which is improved by 6.3, 14.3, and 10.3 times, respectively. It indicates that combining the gradient ascent-based mutator with the random generator can improve the efficiency. However, even then, DRLFuzz is still superior to **variant 2** in terms of efficiency; the former is improved by 24.2%, 10.6%, and 6.1%, compared with the latter. Considering that their difference is that **variant 2** does not consider the state coverage, a coverage analyzer is helpful to improve the fuzzing efficiency.

Fig. 6(b) shows that the diversity of failed cases generated by **variant 1** is the worst in most cases. The reason may be that **variant 1** mutates the inputs along the direction of loss gradients, leading to a lack of global exploration in the state space. Attentive readers may have noticed an exceptional case in the game of Pong. The MPD value of **variant 1** is higher than that of its counterparts. After careful analysis, we found that the test input can be generated randomly if the muted input reaches the boundary. The implementation is to avoid the test input that is remaining stalled and never being updated caused by the mutated input reaching the boundary. However, the practice introduces the operation of random generation and thus exaggerates the diversity of **variant 1**. As a whole, the MPD value of **variant 2** is higher than that of **variant 1**, indicating that a gradient ascent-based mutator alone is not enough to ensure the diversity of the test cases. Furthermore, the MPD value of DRLFuzz is always higher than that of **variant 2**. This suggests that a coverage analyzer could help promote the diversity of the test cases.



(a) The total number of generated failed test cases with 1000 iterations (DRLFuzz and two baselines)



(b) The diversity of generated failed test cases with 1000 iterations (DRLFuzz and two baselines)

Fig. 4. The efficiency and diversity of generating failed test cases.

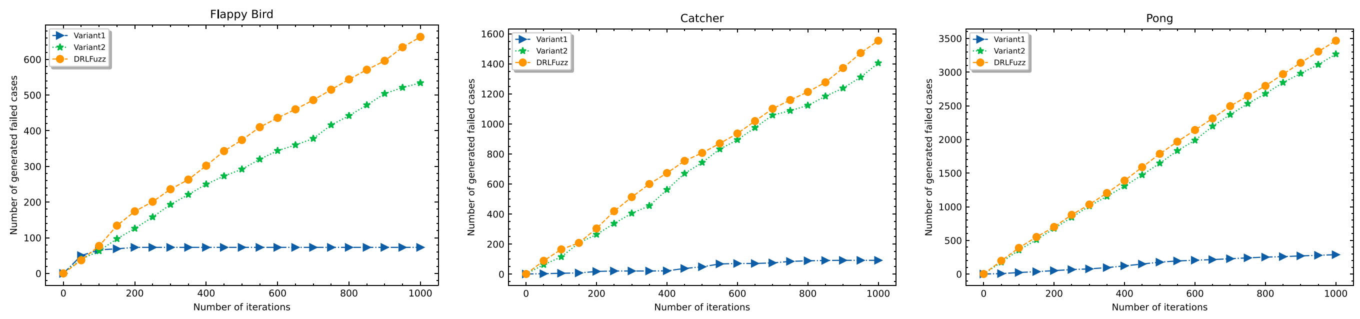
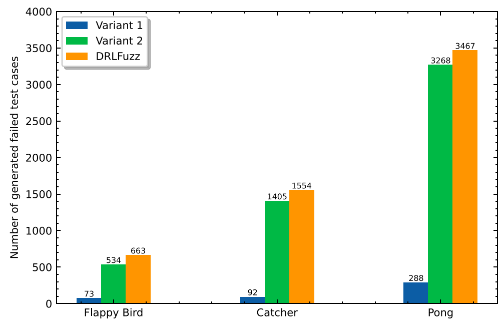
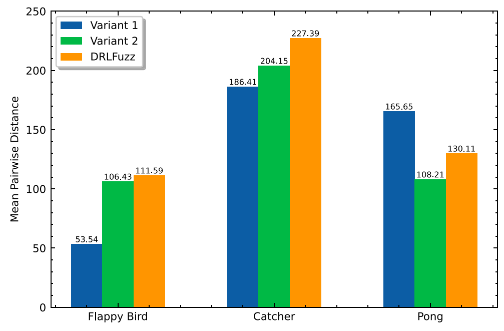


Fig. 5. The changing curves of failed test cases vs. iteration number (DRLFuzz and two variants).



(a) The total number of generated failed test cases after 1000 iterations (DRLFuzz and two variants)



(b) The diversity of generated failed test cases after 1000 iterations (DRLFuzz and two variants)

Fig. 6. The efficiency and diversity of generating failed test cases (DRLFuzz and two variants).

The mutator alone lacks global exploration, resulting in limited diversity. A combination of the seed mutator and the coverage analyzer can address this issue and thereby improve the efficiency and diversity of failed case generation.

5.5. Result analysis for RQ3

To answer the RQ3, we utilize the generated failed test cases to fine-tune the DRL system. Specifically, the DRL agent is executed in each episode starting with these cases, and then the agent's experience at each time step is stored to fine-tune the original DRL agent by one epoch with a small learning rate. Our experimental results, compared to the DRL system's performance before and after fine-tuning, are shown in Fig. 7. The results show that after fine-tuning, the DRL system obtains a substantial performance boost in failed cases, e.g., the average episodic reward is raised from 100.6 to 827.6, from 0 to 3410.4, and from 0 to 85.4, respectively. Note that the episodic rewards in some failed cases are not zero in Flappy Bird due to the presence of uncontrollable random factors. However, it does not affect our conclusion. Utilizing the failed cases to fine-tune the DRL agent can effectively repair the erroneous behaviors and improve its reliability. Moreover, model fine-tuning by using the failed cases generated by DRLFuzz can also improve the performance of the DRL system in random cases, e.g., the average episodic reward is raised from 827.6 to 990.6, from 3410.4 to 4854.9, and from 85.4 to 104.4. Thus, the DRL system's average score in random cases is improved by 19.7%, 42.4% and 22.2%.

Fine-tuning by utilizing the failed tests generated by DRLFuzz is a feasible strategy to improve the reliability and generalization performance of DRL systems.

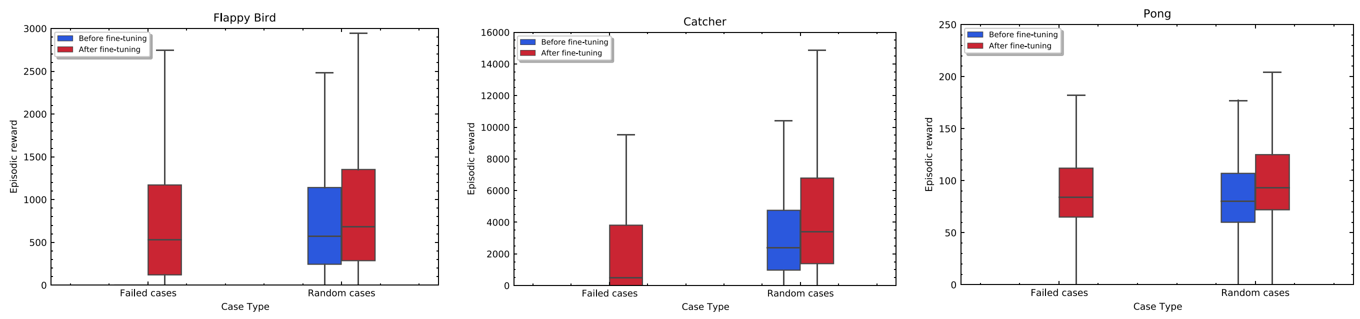


Fig. 7. The performance of the DRL system before and after fine-tuning.

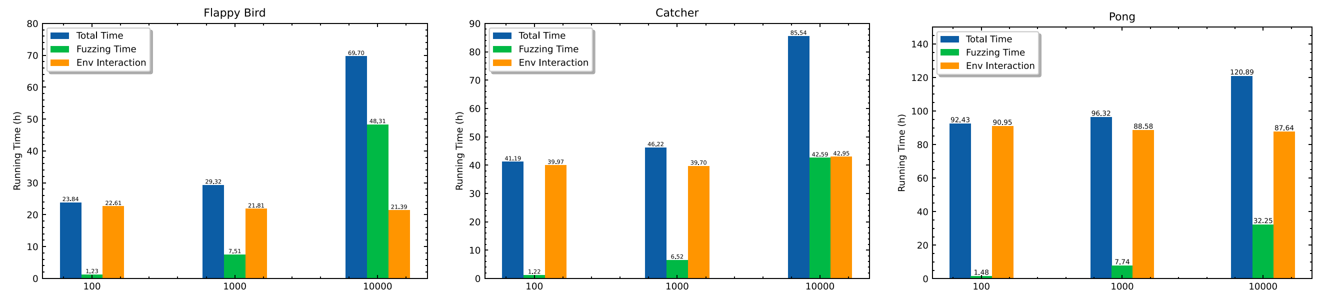


Fig. 8. The running time of DRLFuzz with different T values.

6. Discussions

We have investigated three research questions about DRLFuzz and shown the superiority of DRLFuzz. However, there still exists an important issue that can be discussed. As Section 4.2 mentions, DRLFuzz needs to generate new inputs that exercise new coverage in each iteration. However, as fuzzing continues, it becomes harder and harder to exercise new coverage. The current solution is to exponentially decay δ if it fails to exercise new coverage for a fixed number T. Hence, T is critical to the efficiency of test generation. Hence, we investigate the impact of T on the running time of DRLFuzz. Specifically, we set $T = 100$, 1000, and 10 000, while other experimental settings are the same as before. Here, the running time of fuzzing generation and interacting with the environment and the total time of DRLFuzz in the experiments are recorded and presented in Fig. 8.

Fig. 8 indicates that the T value is a major factor which influences the efficiency of DRLFuzz in terms of running time. The running time of fuzzing generation is very short when $T = 100$. However, if we take a larger T value, such as $T = 1000$, the running time of fuzzing generation is significantly increased from 1.23 to 7.51, from 1.22 to 6.52, and from 1.48 to 7.74, respectively. In contrast, The variation in the running time of interacting with the environment is tiny enough to be ignored. Moreover, if we take a larger T value, such as $T = 10\,000$, the running time of fuzzing generation will further increase.

The diversity of failed cases generated by **variant 1** is the worst in most cases. The reason may be that **variant 1** mutates the inputs along the direction of loss gradients, leading to a lack of global exploration in the state space. Attentive readers may have noticed an exceptional case in the game of Pong. The MPD value of **variant 1** is higher than that of its counterparts. After careful analysis, we found that the test input can be generated randomly if the muted input reaches the boundary. The implementation is to avoid the test input that is remaining stalled and never being updated caused by the mutated input reaching the boundary. However, the practice introduces the operation of random generation and thus exaggerates the diversity of **variant 1**. As a whole, the MPD value of **variant 2** is higher than that of **variant 1**, indicating that a gradient ascent-based mutator alone is not enough to ensure the diversity of the test cases. Furthermore, the MPD value of DRLFuzz is always higher than that of **variant 2**. This suggests that a coverage analyzer could help promote the diversity of the test cases.

7. Threats to validity

To minimize threats to internal validity, we have made sure that our implementations are correct. Specifically, we carefully read the article proposing the DQN algorithm (Mnih et al., 2013), then implement it by an open-source DL platform named PyTorch (Paszke et al., 2019). The RL environment used in this paper is also a popular open-source reinforcement learning platform named PLE (Tasfi, 2016). For most hyper-parameters of model structure and training process, we use the default settings whenever possible. In addition, for implementing our proposed approach, the gradient ascent-based adversarial attack method is reimplemented by following the description in the corresponding papers (Miyato et al., 2018). The kd-tree based ANN algorithm is implemented by Scipy (Virtanen et al., 2020), an open-source library of algorithms and mathematical tools.

Regarding threats to external validity, we use a limited number of RL tasks and DRL models to conduct experiments due to resource constraints. Hence, it is not sure that how well our proposed approach and experimental results will generalize to more DRL models and RL tasks. To mitigate this problem, we train the DQN agents with different model structures for different RL tasks and test all the trained DQN agents in 10 000 episodes starting by different initial states. Moreover, a detailed description of the experimental setup and our proposed approach is provided in this paper, which will be helpful to replicate our experiments.

As for limitations, DRLFuzz focuses on testing DRL systems characterized by non-pixel state representations, discrete actions, and a deterministic policy within a stationary yet stochastic environment. Given DRLFuzz depends on gradients during seed mutation, it functions as a grey-box fuzzing tool, making it less suitable for black-box testing contexts. Moreover, DRLFuzz emphasizes state coverage anchored in the raw state space, making it more applicable to state-based SDM tasks. To accommodate high-dimensional pixel-based SDM tasks, DRLFuzz may require a redefinition of state abstraction and state coverage. Also, with the average episode reward as its objective function, DRLFuzz primarily targets reward faults, situations where the agent fails to achieve the expected episode reward, rather than functional faults linked to unsafe behaviors. To detect other types of faults, such as functional faults, we would need to define the faulty state and design the objective function accordingly.

8. Related work

8.1. Testing techniques in DL systems

The reliability of DL systems has received considerable attention due to the tremendous progress of DL techniques. In such a context, researchers are beginning to pay more attention to the testing techniques for DL systems (Zhang et al., 2020a). Specifically, Pei et al. (2017) proposed the first testing framework for DL systems. The authors designed the first testing criteria, neuron coverage, to measure how many neurons are activated by the given test set. Since then, testing criteria (Ma et al., 2018a; Sun et al., 2018a; Kim et al., 2019; Ma et al., 2018b) and test input generation (Pei et al., 2017; Xie et al., 2019b; Tian et al., 2018; Sun et al., 2018b; Odena et al., 2019; Guo et al., 2018; Xie et al., 2019a) have become two main research directions.

For testing criteria, Pei et al. (2017) first proposed neuron coverage (NC) to guide test generation for DL systems. Ma et al. (2018a) extended the concept of NC and proposed five coverage criteria with different granularity for DL testing. Sun et al. (2018a) was inspired by the modified condition and decision coverage (MC/DC) coverage criteria in traditional software testing and proposed four coverage criteria. Kim et al. (2019) proposed surprise adequacy to measure the surprise of an input in terms of training data. Ma et al. (2018b) considered the interactions of neurons and proposed a set of combinatorial testing criteria for DL systems. Inspired by the above studies, Shi et al. (2021) utilized these metrics to prioritize test cases to improve the testing efficiency. While various structural coverage criteria have been proposed, their effectiveness is being doubted in recent research. Empirical studies indicate that structural coverage criteria could be useless even misleading for DL testing (Li et al., 2019; Dong et al., 2019; Harel-Canada et al., 2020).

For test input generation, Pei et al. (2017) proposed DeepXplore, the first white-box differential testing framework, to detect behavior inconsistencies among different DL systems. Xie et al. (2019b) proposed a black-box differential testing technique named DiffChaser to detect the disagreements. Tian et al. (2018) introduced metamorphic relations and proposed an NC-guided test input generation approach named DeepTest. Sun et al. (2018b) proposed a concolic testing framework for DL systems named DeepConclic, which generates test cases using constraint solving. Odena et al. (2019) proposed the first CGF framework named TensorFuzz to discover numerical errors, disagreements between DL models and their quantized versions. Guo et al. (2018) proposed a differential fuzzing framework named DLFuzz. In addition, Xie et al. (2019a) also proposed a CGF framework named DeepHunter to test DL systems, which provides more mutators and employs five testing criteria and four seed selection strategies.

8.2. Testing techniques in DRL systems

As DRL techniques are being widely used in safety-critical domains, DRL systems also need to be subjected to systematic testing processes. So fair, the research on DRL testing has rarely been investigated. To our knowledge, there are two lines of studies closely related to our work.

The first one is the adversarial attack study conducted by Huang et al. (2017), which utilized FGSM (Goodfellow et al., 2015) as a white-box attack to calculate adversarial perturbations for a trained neural network policy whose structure and parameters are available to the adversary. Both DRLFuzz and adversarial attack aim to drive the DRL agent to make erroneous behaviors. However, our work still has distinct differences from it. First, DRLFuzz aims to generate diverse initial states of an episode to drive the agent to fail, while adversarial attacks confuse the agent by adding tiny perturbations to state observations at every time step (or at least selected subset of time steps) (Chen et al., 2019). Second, most adversarial attacks suffer from two limitations: (1) they take no account of the state coverage of test cases; and (2) adversarial attack is inherently limited to only use the tiny perturbations on the state observations. Hence, the test cases generated by attack methods typically have low diversity (as shown in Fig. 6(b)). Actually, in **RQ2**, we designed two variants of the DRLFuzz, one of which, variant 1, utilizes the gradient ascent-based search approach to mutate the initial seeds and directly uses them as test cases. Hence, variant 1 is essentially the FGSM attack. As shown in Figs. 5 and 6, variant 1 is obviously inferior to the DRLFuzz in terms of the efficiency and diversity of generating failed test cases.

The second one involves adapting traditional software testing methods for DRL systems, highlighted by methods such as GANC (Al-Nima et al., 2021), AgentFuzz (Li et al., 2022), and STARLA (Zolfagharian et al., 2023). GANC utilized a genetic algorithm, guiding the generation of test cases to maximize the neuron coverage of DNNs. Its primary objective was to produce augmented inputs, enhancing the robustness and performance of DNN-based imitation learning systems under noisy environments. AgentFuzz was a test case generation method for DQN agents, inspired by the principles of fuzz testing. It primarily focused on the design of both the seed mutator and the seed selector. In contrast, STARLA was introduced as a search-based testing approach for DRL agents. Incorporating rewards, sequence failure probabilities, and model uncertainties, it redefined test case generation as a multi-objective search problem. Test cases were synthesized by utilizing operations from the genetic algorithm, such as crossover and mutation.

In contrast to DRLFuzz, STARLA requires the training of a classifier to determine the failure probability of sequences. This means that the effectiveness of STARLA is inherently influenced by the accuracy of the prediction model (Zolfagharian et al., 2023). Additionally, STARLA assumes access to interaction episodes during the model training phase. These factors present challenges in conducting a fair comparison between DRLFuzz and STARLA. To further analyze the characteristics of STARLA, we reproduced it to test DRL systems across three benchmark RL tasks, and included the preliminary experimental analysis in a separate file, along with the implementation code. All additional information is available in our open-source code repository.[3]

9. Conclusion and future work

In recent decades, With the rapid development of artificial intelligence techniques and their increasing application in safety-critical areas, intelligent systems' quality and reliability assurance have recently drawn extensive attention. So far, much effort has been focused on testing DL systems to ensure their reliability and safety. However, little work has been done for testing DRL systems. Unlike DL models commonly used to solve one-step prediction tasks (classification or regression), DRL models focus on SDM tasks. It is even more difficult to determine and generate erroneous behaviors during a multi-step decision process, which brings new challenges to DRL testing.

In such a context, this paper introduces a novel CGF framework for DRL systems, namely DRLFuzz, and describes the design of each component in detail. We have demonstrated the effectiveness and the efficiency of DRLFuzz when generating diverse failed test cases, as well as those initial states where DRL systems may perform poorly during an episode starting. Moreover, our experiments also indicate that the performance of DRL systems can be improved if we use the failed cases generated by DRLFuzz to fine-tune the trained DRL model. It provides us with a feasible way to repair the defects of DRL systems caused by inadequate exploration during training. Finally, we provide a reproduction package to facilitate further comparative studies on DRL testing. In the future, we plan to evaluate or improve our DRLFuzz framework on more diverse DRL systems and some practical application domains.

[3] https://github.com/Wan-xiaohui/DRLFuzz.

CRediT authorship contribution statement

Xiaohui Wan: Data curation, Investigation, Methodology, Software, Writing – original draft, Writing – review & editing. **Tiancheng Li:** Data curation, Formal analysis, Methodology, Resources, Software, Validation, Visualization. **Weibin Lin:** Data curation, Software, Writing – review & editing. **Yi Cai:** Data curation, Investigation, Visualization, Writing – review & editing. **Zheng Zheng:** Conceptualization, Funding acquisition, Project administration, Supervision, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

References

Al-Nima, R.R.O., Han, T., Al-Sumaidaee, S.A.M., et al., 2021. Robustness and performance of deep reinforcement learning. Appl. Soft Comput. 105, 107295.

Ammann, P., Offutt, J., 2016. Introduction to Software Testing. Cambridge University Press.

Andre, D., Friedman, N., Parr, R., 1997. Generalized prioritized sweeping. In: Proceedings of the 10th International Conference on Advances in Neural Information Processing Systems. MIT Press.

Bellemare, M., Srinivasan, S., Ostrovski, G., Schaul, T., Saxton, D., Munos, R., 2016. Unifying count-based exploration and intrinsic motivation. Adv. Neural Inf. Process. Syst. 29.

Biemann, T., Kearney, E., 2010. Size does matter: How varying group sizes in a sample affect the most common measures of group diversity. Organ. Res. Methods 13 (3), 582–599.

Böhme, M., Pham, V.-T., Nguyen, M.-D., et al., 2017a. Directed greybox fuzzing. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 2329–2344.

Böhme, M., Pham, V.-T., Roychoudhury, A., 2017b. Coverage-based greybox fuzzing as markov chain. IEEE Trans. Softw. Eng. 45 (5), 489–506.

Chen, C., Cui, B., Ma, J., et al., 2018. A systematic review of fuzzing techniques. Comput. Secur. 75, 118–137.

Chen, T., Liu, S., Chang, S., et al., 2020. Adversarial robustness: From self-supervised pre-training to fine-tuning. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. pp. 699–708.

Chen, T., Liu, J., Xiang, Y., et al., 2019. Adversarial attack and defense in reinforcement learning-from AI security view. Cybersecurity 2 (1), 1–22.

Christopher, J., 1992. Technical note q-learning. Mach. Learn. 8.

Cobbe, K., Klimov, O., Hesse, C., et al., 2019. Quantifying generalization in reinforcement learning. In: International Conference on Machine Learning. PMLR, pp. 1282–1289.

Dong, X., Luu, A.T., Lin, M., et al., 2021. How should pre-trained language models be fine-tuned towards adversarial robustness? Adv. Neural Inf. Process. Syst. 34, 4356–4369.

Dong, Y., Zhang, P., Wang, J., et al., 2019. There is limited correlation between coverage and robustness for deep neural networks. arXiv preprint arXiv:1911.05904.

Dreyfus, S.E., Bellman, R., 1962. Applied Dynamic Programming. Princeton University Press.

Du, X., Xie, X., Li, Y., et al., 2018. Deepcruiser: Automated guided testing for stateful deep learning systems. arXiv preprint arXiv:1812.05339.

Erhan, D., Courville, A., Bengio, Y., et al., 2010. Why does unsupervised pre-training help deep learning? In: Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics. JMLR Workshop and Conference Proceedings, pp. 201–208.

Finn, C., Abbeel, P., Levine, S., 2017. Model-agnostic meta-learning for fast adaptation of deep networks. In: International Conference on Machine Learning. PMLR, pp. 1126–1135.

Fu, Z.-Y., Zhan, D.-C., Li, X.-C., et al., 2019. Automatic successive reinforcement learning with multiple auxiliary rewards. In: IJCAI. pp. 2336–2342.

Gan, S., Zhang, C., Qin, X., et al., 2018. Collafl: Path sensitive fuzzing. In: 2018 IEEE Symposium on Security and Privacy. SP, IEEE, pp. 679–696.

Goodfellow, I.J., Shlens, J., Szegedy, C., 2014. Explaining and harnessing adversarial examples. arXiv preprint arXiv:1412.6572.

Goodfellow, I.J., Shlens, J., Szegedy, C., 2015. Explaining and harnessing adversarial examples.

Guo, J., Jiang, Y., Zhao, Y., et al., 2018. Dlfuzz: Differential fuzzing testing of deep learning systems. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 739–743.

Han, H., Cha, S.K., 2017. Imf: Inferred model-based fuzzer. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 2345–2358.

Harel-Canada, F., Wang, L., Gulzar, M.A., et al., 2020. Is neuron coverage a meaningful measure for testing deep neural networks? In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 851–862.

Harrison, D.A., Klein, K.J., 2007. What's the difference? Diversity constructs as separation, variety, or disparity in organizations. Acad. Manag. Rev. 32 (4), 1199–1228.

Heuillet, A., Couthouis, F., Díaz-Rodríguez, N., 2021. Explainability in deep reinforcement learning. Knowl.-Based Syst. 214, 106685.

Horgan, D., Quan, J., Budden, D., Barth-Maron, G., Hessel, M., van Hasselt, H., Silver, D., 2018. Distributed prioritized experience replay. In: Proceedings of the 6th International Conference on Learning Representations. OpenReview.net.

Huang, S., Papernot, N., Goodfellow, I., et al., 2017. Adversarial attacks on neural network policies. arXiv preprint arXiv:1702.02284.

Irpan, A., 2018. Deep reinforcement learning doesn't work yet. https://www.alexirpan.com/2018/02/14/rl-hard.html.

Jaderberg, M., Czarnecki, W., Dunning, I., et al., 2018. Human-level performance in first-person multiplayer games with population-based deep reinforcement learning. arxiv. arXiv preprint arXiv:1807.01281.

Kaplanis, C., Shanahan, M., Clopath, C., 2018. Continual reinforcement learning with complex synapses. In: International Conference on Machine Learning. PMLR, pp. 2497–2506.

Kim, J., Feldt, R., Yoo, S., 2019. Guiding deep learning system testing using surprise adequacy. In: 2019 IEEE/ACM 41st International Conference on Software Engineering. ICSE, IEEE, pp. 1039–1049.

Lee, M., Cha, S., Oh, H., 2023. Learning seed-adaptive mutation strategies for greybox fuzzing. In: 2023 IEEE/ACM 45th International Conference on Software Engineering. ICSE, IEEE, pp. 384–396.

Li, Z., Ma, X., Xu, C., et al., 2019. Structural coverage criteria for neural networks could be misleading. In: 2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results. ICSE-NIER, IEEE, pp. 89–92.

Li, T., Wan, X., Özbek, M.M., 2022. AgentFuzz: Fuzzing for deep reinforcement learning systems. In: 2022 IEEE International Symposium on Software Reliability Engineering Workshops. ISSREW, IEEE, pp. 110–113.

Li, J., Zhao, B., Zhang, C., 2018. Fuzzing: a survey. Cybersecurity 1 (1), 1–13.

Liang, J., Jiang, Y., Wang, M., Jiao, X., Chen, Y., Song, H., Choo, K.-K.R., 2019. Deepfuzzer: Accelerated deep greybox fuzzing. IEEE Trans. Dependable Secure Comput. 18 (6), 2675–2688.

Liu, V., Kumaraswamy, R., Le, L., et al., 2019. The utility of sparse representations for control in reinforcement learning. In: Proceedings of the AAAI Conference on Artificial Intelligence. Vol. 33, pp. 4384–4391.

Liu, G., Schulte, O., Zhu, W., et al., 2018. Toward interpretable deep reinforcement learning with linear model u-trees. In: Joint European Conference on Machine Learning and Knowledge Discovery in Databases. Springer, pp. 414–429.

Lyu, C., Ji, S., Li, Y., Zhou, J., Chen, J., Chen, J., 2018. Smartseed: Smart seed generation for efficient fuzzing. arXiv preprint arXiv:1807.02606.

Ma, L., Juefei-Xu, F., Zhang, F., et al., 2018a. Deepgauge: Multi-granularity testing criteria for deep learning systems. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. pp. 120–131.

Ma, L., Zhang, F., Xue, M., et al., 2018b. Combinatorial testing for deep learning systems. arXiv preprint arXiv:1806.07723.

Maneewongvatana, S., Mount, D.M., 1999. "Analysis of approximate nearest neighbor searching with clustered point sets." ALENEX 99.

Miyato, T., Maeda, S.-i., Koyama, M., et al., 2018. Virtual adversarial training: a regularization method for supervised and semi-supervised learning. IEEE Trans. Pattern Anal. Mach. Intell. 41 (8), 1979–1993.

Mnih, V., Kavukcuoglu, K., Silver, D., et al., 2013. Playing atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602.

Mnih, V., Kavukcuoglu, K., Silver, D., et al., 2015. Human-level control through deep reinforcement learning. Nature 518 (7540), 529–533.

Morales, M., 2020. Grokking Deep Reinforcement Learning. Simon and Schuster.

Myers, G.J., Sandler, C., Badgett, T., 2011. The Art of Software Testing. John Wiley & Sons.

Odena, A., Olsson, C., Andersen, D., et al., 2019. Tensorfuzz: Debugging neural networks with coverage-guided fuzzing. In: International Conference on Machine Learning. PMLR, pp. 4901–4911.

Pan, S.J., Yang, Q., 2009. A survey on transfer learning. IEEE Trans. Knowl. Data Eng. 22 (10), 1345–1359.

Paszke, A., Gross, S., Massa, F., et al., 2019. Pytorch: An imperative style, high-performance deep learning library. In: Advances in Neural Information Processing Systems 32. Curran Associates, Inc., pp. 8024–8035.

Pei, K., Cao, Y., Yang, J., et al., 2017. Deepxplore: Automated whitebox testing of deep learning systems. In: Proceedings of the 26th Symposium on Operating Systems Principles. pp. 1–18.

Rakhsha, A., Zhang, X., Zhu, X., Singla, A., 2021. Reward poisoning in reinforcement learning: Attacks against unknown learners in unknown environments. arXiv preprint arXiv:2102.08492.

Ramakrishnan, R., Kamar, E., Dey, D., et al., 2020. Blind spot detection for safe sim-to-real transfer. J. Artificial Intelligence Res. 67, 191–234.

Ruder, S., 2017. An overview of multi-task learning in deep neural networks. arXiv preprint arXiv:1706.05098.

Ruderman, A., Everett, R., Sikder, B., et al., 2018. Uncovering surprising behaviors in reinforcement learning via worst-case analysis.

Sallab, A.E., Abdou, M., Perot, E., et al., 2017. Deep reinforcement learning framework for autonomous driving. Electron. Imaging 2017 (19), 70–76.

Schaul, T., Quan, J., Antonoglou, I., Silver, D., 2016. Prioritized experience replay. In: Proceedings of the 4th International Conference on Learning Representations. OpenReview.net.

Schenke, M., Wallscheid, O., 2021. Improved exploring starts by kernel density estimation-based state-space coverage acceleration in reinforcement learning. arXiv preprint arXiv:2105.08990.

Serebryany, K., 2015. Libfuzzer–a library for coverage-guided fuzz testing. In: LLVM Project.

Shi, Y., Yin, B., Zheng, Z., Li, T., 2021. An empirical study on test case prioritization metrics for deep neural networks. In: 2021 IEEE 21st International Conference on Software Quality, Reliability and Security. QRS, IEEE, pp. 157–166.

Silver, D., Huang, A., Maddison, C.J., et al., 2016. Mastering the game of go with deep neural networks and tree search. Nature 529 (7587), 484–489.

Sun, Y., Huang, X., Kroening, D., et al., 2018a. Testing deep neural networks. arXiv preprint arXiv:1803.04792.

Sun, Y., Wu, M., Ruan, W., et al., 2018b. Concolic testing for deep neural networks. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. pp. 109–119.

Sutton, R.S., Barto, A.G., 2018. Reinforcement Learning: An Introduction. MIT Press.

Tang, H., Houthooft, R., Foote, D., Stooke, A., Xi Chen, O., Duan, Y., Schulman, J., DeTurck, F., Abbeel, P., 2017. # Exploration: A study of count-based exploration for deep reinforcement learning. Adv. Neural Inf. Process. Syst. 30.

Tasfi, N., 2016. Pygame learning environment. https://github.com/ntasfi/PyGame-Learning-Environment.

Tian, Y., Pei, K., Jana, S., et al., 2018. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In: Proceedings of the 40th International Conference on Software Engineering. pp. 303–314.

Tucker, C.M., Cadotte, M.W., Carvalho, S.B., et al., 2017. A guide to phylogenetic metrics for conservation, community ecology and macroecology. Biol. Rev. 92 (2), 698–715.

Uesato, J., Kumar, A., Szepesvari, C., et al., 2018. Rigorous agent evaluation: An adversarial approach to uncover catastrophic failures. In: International Conference on Learning Representations.

Vaibhav, V., Singh, S., Stewart, C., et al., 2019. Improving robustness of machine translation with synthetic noise. arXiv preprint arXiv:1902.09508.

Virtanen, P., Gommers, R., Oliphant, T.E., et al., 2020. SciPy 1.0: Fundamental algorithms for scientific computing in python. Nature Methods 17, 261–272.

Wang, S., Su, Z., 2020. Metamorphic object insertion for testing object detection systems. In: 2020 35th IEEE/ACM International Conference on Automated Software Engineering. ASE, IEEE, pp. 1053–1065.

Xie, X., Ma, L., Juefei-Xu, et al., 2018. Coverage-guided fuzzing for deep neural networks. arXiv preprint arXiv:1809.01266. 3.

Xie, X., Ma, L., Juefei-Xu, et al., 2019a. Deephunter: a coverage-guided fuzz testing framework for deep neural networks. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 146–157.

Xie, X., Ma, L., Wang, H., et al., 2019b. DiffChaser: Detecting disagreements for deep neural networks. In: IJCAI. pp. 5772–5778.

Yan, Z., Guo, Y., Zhang, C., 2018. Deep defense: Training dnns with improved adversarial robustness. Adv. Neural Inf. Process. Syst. 31.

Yang, Q., Zhang, J., Shi, G., et al., 2019. Maneuver decision of UAV in short-range air combat based on deep reinforcement learning. IEEE Access 8, 363–378.

Zalewski, M., 2014. American fuzzy lop (2017). p. 28, URL http://lcamtuf.coredump.cx/afl. 14.

Zhang, J.M., Harman, M., Ma, L., et al., 2020a. Machine learning testing: Survey, landscapes and horizons. IEEE Trans. Softw. Eng..

Zhang, F., Leitner, J., Milford, M., et al., 2015. Towards vision-based deep reinforcement learning for robotic motion control. arXiv preprint arXiv:1511.03791.

Zhang, X., Ma, Y., Singla, A., Zhu, X., 2020b. Adaptive reward-poisoning attacks against reinforcement learning. In: International Conference on Machine Learning. PMLR, pp. 11225–11234.

Zhao, L., Duan, Y., Yin, H., Xuan, J., 2019. Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing. In: NDSS.

Zhu, G., Wang, J., Ren, Z., et al., 2020. Object-oriented dynamics learning through multi-level abstraction. In: Proceedings of the AAAI Conference on Artificial Intelligence. Vol. 34, pp. 6989–6998.

Zolfagharian, A., Abdellatif, M., Briand, L.C., Bagherzadeh, M., Ramesh, S., 2023. A search-based testing approach for deep reinforcement learning agents. IEEE Trans. Softw. Eng..

Xiaohui Wan received his Bachelor's degree from School of Electrical and Information Engineering of Jiangsu University, Zhenjiang, China, in 2016. He is currently working toward the Ph.D. degree in control science and engineering from Beihang University, Beijing, China. His research interests include software defect prediction, machine learning testing, and intelligent software engineering.

Tiancheng Li received his Bachelor's degree in 2020 and his Master's degree in 2023 from School of Automation Science and Electronic Engineering at Beihang University, Beijing, China. His research interests include software testing and machine learning testing.

Weibin Lin received the B.S. degree with the School of Automation Science and Electrical Engineering from Beihang University, Beijing, China, in 2023. He is currently pursuing the Ph.D. degree in Beihang University. His research interests include machine learning and software reliability.

Yi Cai received his Bachelor's degree from School of Automation Science and Electrical Engineering of Beihang University, Beijing, China, in 2021. He is currently working toward the Master degree in control science and engineering from Beihang University, Beijing, China. His research interests include machine learning testing, and intelligent software engineering.

Zheng Zheng (Senior Member, IEEE) is a professor with Beihang University and the deputy dean with the School of Automation Science and Electrical Engineering. His research work is primarily concerned with software reliability and testing. Recently, He paid more attention on the intelligent software reliability engineering. He has co-authored more than 100 journal and conference publications, including IEEE Transactions on Dependable and Secure Computing, IEEE Transactions on Information Forensics and Security, IEEE Transactions on Software Engineering, IEEE Transactions on Reliability, IEEE Transactions on Services Computing, JSS and so on. He serves for IEEE PRDC2019, IEEE DASC 2019, IEEE ISSRE 2020, IEEE QRS 2021 as PC Co-Chairs, as well as WoSAR 2019, DeIS 2020 and DeIS 2021 as General Co-Chairs. He is the editor-in-chief of Atlantis Highlights in Engineering (Springer Nature), associate editor of IEEE Transactions on Reliability (2021-), Elesvier KBS (2018-) and Springer IJCIS (2012-) and guest editor of IEEE Transactions on Dependable and Secure Computing (2021). He is IEEE CIS Emerging Technologies TC member.