Learning and Repair of Deep Reinforcement Learning Policies from Fuzz-Testing Data

Martin Tappler
martin.tappler@ist.tugraz.at
Graz University of Technology, Institute of Software Technology
Graz, Austria
Silicon Austria Labs, TU Graz - SAL DES Lab
Graz, Austria

Andrea Pferscher
andrea.pferscher@ist.tugraz.at
Graz University of Technology, Institute of Software Technology
Graz, Austria

Bernhard K. Aichernig
aichernig@ist.tugraz.at
Graz University of Technology, Institute of Software Technology
Graz, Austria

Bettina Könighofer
bettina.koenighofer@iaik.tugraz.at
Graz University of Technology, Institute of Applied Information Processing and Communications
Graz, Austria

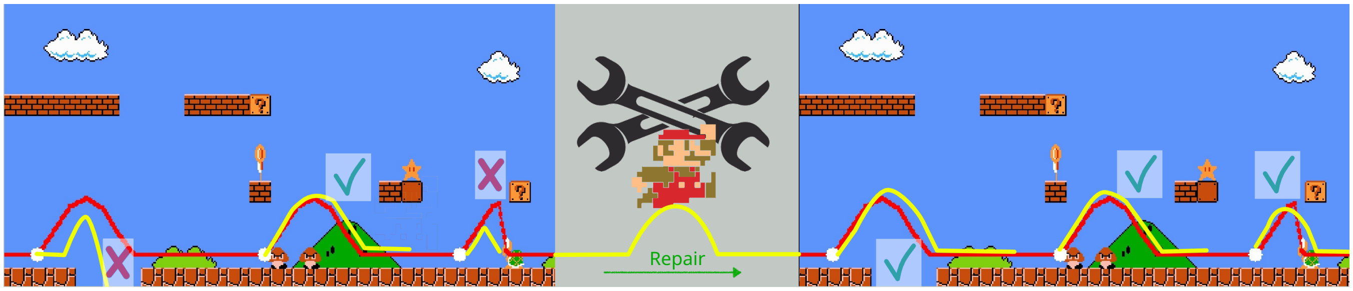


Figure 1: We find a reference trace for an RL task, depicted in red for one of our case studies of an agent playing the Nintendo game Super Mario Bros. Along the trace, we detect dangerous situations (depicted as white dots), from which we test the agent (depicted with yellow lines). We repair the deep RL policy w.r.t. safety violations; here these are losses of a life depicted with red crosses on the left. After testing the policy, we find that it passes all test cases, which is depicted with green check marks.

ABSTRACT

Reinforcement learning from demonstrations (RLfD) is a promising approach to improve the exploration efficiency of reinforcement learning (RL) by learning from expert demonstrations in addition to interactions with the environment. In this paper, we propose a framework that combines techniques from search-based testing with RLfD with the goal to raise the level of dependability of RL policies and to reduce human engineering effort. Within our framework, we provide methods for efficiently training, evaluating, and repairing RL policies. Instead of relying on the costly collection of demonstrations from (human) experts, we automatically compute a diverse set of demonstrations via search-based fuzzing methods and use the fuzz demonstrations for RLfD. To evaluate the safety and robustness of the trained RL agent, we search for safety-critical scenarios in the black-box environment. Finally, when unsafe behavior is detected, we compute demonstrations through fuzz testing that represent safe behavior and use them to repair the policy. Our experiments show that our framework is able to efficiently learn high-performing and safe policies without requiring any expert knowledge.

CCS CONCEPTS

• **Computing methodologies → Learning from demonstrations**; **Reinforcement learning**; • **Software and its engineering → Search-based software engineering**; *Software testing and debugging*.

KEYWORDS

Deep reinforcement learning, Reinforcement learning from demonstrations, Search-based software testing, Policy repair

ACM Reference Format:
Martin Tappler, Andrea Pferscher, Bernhard K. Aichernig, and Bettina Könighofer. 2024. Learning and Repair of Deep Reinforcement Learning Policies from Fuzz-Testing Data. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597503.3623311

ICSE '24, April 14–20, 2024, Lisbon, Portugal
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0217-4/24/04.
https://doi.org/10.1145/3597503.3623311

1 INTRODUCTION

Deep reinforcement learning has achieved stunning results in difficult decision-making problems, like in playing video games (e.g., AlphaStar [43]) and board games (e.g., AlphaGo [34]).

In RL [35], an RL agent learns an optimal behavior through trial and error via interactions with an unknown environment. A major challenge in RL is sample efficiency [42]. Many of the famous applications of deep RL required millions of interactions with the environment. Even though the numbers of environment interactions are very high and triggered a great amount of research and new approaches for sample efficient learning [33, 46], these numbers only report the training steps for the final learned controller and neglect the number of times the training process was executed.

The common procedure in RL is to train an agent and evaluate the final policy. In case the trained policy performs insufficiently well, a developer tunes hyperparameters or adjusts the reward function, and restarts the training process. Therefore, if we are interested in the total number of steps that were required to train the final controller, we should multiply the number of steps by the number of times the developer restarted the training.

Drawing the analogy to software development, this would map to writing code from scratch every time test cases fail. Reusing and retraining neural networks for visual tasks is common practice [6, 21], but to the best of our knowledge, this is not the case for RL.

Problem statement: The main goal of this paper is to propose a development framework for RL controllers that includes their training, their evaluation, and their policy repair. To achieve sample efficiency while reducing human engineering effort, our framework combines techniques from *search-based testing* with *deep reinforcement learning from demonstrations (RLfD).*

RLfD [31] combines learning from demonstrations with learning from exploring the environment, to be sample-efficient while generalizing globally. In RLfD, demonstration data are used to pre-train the agent so that it has an acceptable performance from the start of training. During training, the policy is further improved with newly collected data from exploration as well as from demonstration data.

In most work on RLfD, demonstrations originate either from a former agent or from a human expert. A common challenge in using the generated data to guide RL is that, in both cases, the data tends to have a limited variety. Such data with insufficient state coverage makes it hard to efficiently learn a robust policy.

After the training phase, the final policy of the RL agent needs to be evaluated, which is an extremely challenging task. The lack of determinism, combined with the very high complexity of the trained deep neural networks and the environment the agent operates in makes any kind of testing non-trivial. To the best of our knowledge, no work discusses how a policy of an RL agent can be repaired in case testing reveals any weaknesses in the policy.

Our development framework for RL agents via fuzz-testing data. We propose a framework that uses search-based testing techniques (1) to *train*, (2) to *evaluate*, and (3) to *repair the policy* of deep RL agents.

(1) Training RL agents from fuzz demonstrations. Figure 2 is a schematic overview of our proposed RLfD approach. First, we automatically generate demonstrations using the approach of [36]. This approach performs a simple backtracking-based depth-first search for an initial *reference demonstration*. Using this demonstration as a seed, a diverse set of demonstrations is sampled via fuzzing. The fuzzing is implemented using a meta-heuristic search with the aim to find demonstration trajectories that cover large parts of the state space and gain high rewards. Finally, the *fuzz demonstrations* are used to efficiently train an RL agent via RLfD.

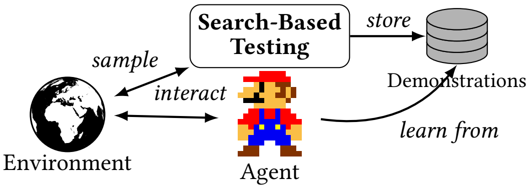


Figure 2: RL from demonstrations generated by search-based testing.

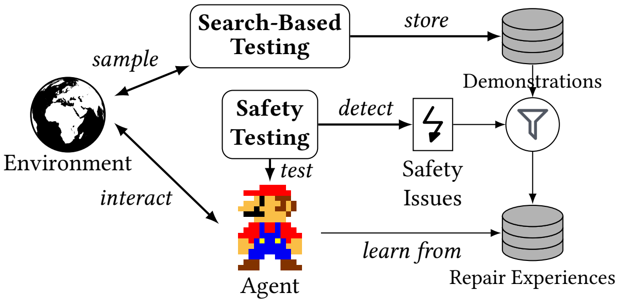


Figure 3: Repairing RL agents from demonstrations generated by search-based testing.

(2) Testing RL agents via search-based testing. Following training, a learned policy needs to be evaluated. We use the safety testing approach of [36] which tests trained agents in safety-critical situations that were detected in the search for the reference demonstration. Since the DFS algorithm backtracks from unsafe states, branches in the search reveal safety-critical situations. The safety of an agent is evaluated based on its ability to succeed in such situations.

(3) Policy repair of RL agents via fuzz-testing. If testing reveals unsafe behavior, we perform policy repair to correct detected unsafe behavior while seeking to retain the overall performance, and especially, to not induce new unsafe behavior. Figure 3 outlines our approach for repairing RL policies. We repair the agent's policy in the vicinity of states where testing reveals issues. To set up the repair, we collect examples of correct behavior near the detected faulty states from the fuzz demonstrations. We filter these demonstrations to keep only experiences relevant to the detected issues. If we do not have enough fuzz demonstrations in a particular area, we can apply another search to sample the environment. Using the collected experiences as additional demonstrations of safe behavior, we apply learning from demonstrations again to re-train the agent. Afterward, we go back to Step 2 to test the new policy. If the new policy still fails test cases, we perform another round of policy repair.

Properties of our RL development framework. *1. Applicable under partial observability.* The search and fuzzing approaches in our framework sample the environment as an oracle. Thus, even in the case of partial observability, our framework can be successfully applied. The only required information is the reward gained from executing a trace (did the trace partially or completely solve the task to be learned?) and if a trace violated safety. Exact state information is not required. *2. Usage of human experts demonstrations.* If human expert demonstrations are available, such demonstrations can be used (alone or additionally to automatically generated traces) as a basis for fuzzing for training and policy repair. *3. Improved dependability regarding safety violations due to fuzzing.* In software testing, fuzzing has been proven effective in discovering corner cases. We exploit this feature throughout our framework: by training with traces that solve the tasks in different and possibly not obvious manners, by finding corner cases in the state space that are safety-critical, and by repairing the agent with diverse traces.

Case Studies. We performed two sets of experiments, using our framework to develop deep RL agents for Super Mario Bros (level 1-4) and for solving navigation tasks in challenging gridworlds. Both case studies demonstrate that we are able to reduce the number of training steps without requiring any expert knowledge. The repair phase is able to successfully repair the policy to fix most detected unsafe behavior. We show that not doing repair but increasing the number of training steps does not result in safer policies.

Main Contributions. To summarize, our main contributions are: (1) We propose a development framework for RL that repairs a trained policy instead of starting training from scratch. (2) We reduce human effort in RLfD by automatically computing demonstrations via fuzzing. (3) To the best of our knowledge, we are the first who discuss policy repair for deep RL agents. (4) We show the potential and flexibility of our approach on solving a challenging computer game and navigation tasks.

2 RELATED WORK

Recently, there have been many impressive advances in RLfD. The survey [31] provides an overview of the most novel relevant algorithms to integrate demonstration data into deep RL. The approaches to solve the RL part of RLfD can be categorized into value-based [4, 14], policy-based [16, 17, 30], and actor-critic methods [11, 32]. We use the value-based approach Deep Q-learning from Demonstrations (DQfD) [14] since it is one of the most prominent algorithms. However, our framework can combined with any RLfD algorithm. Several works consider the problem of learning from imperfect or noisy demonstrations, or learning from fewer demonstrations [2, 9, 25, 48]. We assume to compute fuzz demonstrations under ideal conditions, i.e., we assume that fuzzing and testing have the exact same observation space as the agent. Closing domain gaps between fuzz demonstrations and the agent's action and reward trajectories is interesting future work. Techniques from transfer learning, like cross-domain transfer [38], may help to achieve this goal.

In the last year, several works proposed to apply software testing approaches to evaluate deep RL agents. In particular, [36] and [49] propose search-based testing methods, and [22] discussed mutation-based testing for the RL setting. Tappler et al. [36] were the first to propose a search-based testing framework to evaluate the safety of RL agents. Our paper directly reuses the techniques presented in [36] to (a) *search for reference demonstration and boundary states*, to (b) *test for safety* and to (c) *generate fuzz demonstrations* ((a)-(c) refer to Step 1-3 in the testing framework of [36]). Similar to [36], Zolfagharian et al. [49] search for demonstrations using genetic algorithms. However, instead of searching for boundary states (states in which some successors states are safety-critical and some are safe) and testing several agents near these boundary states, the approach directly searches for failing executions of an agent's policy.

For RLfD to be effective, it is necessary to have a diverse set of demonstrations, including traces that are successful. The demonstrations can come from search algorithms or from human experts. In this paper, we reused the approach and implementation of [36] to search for demonstrations. However, the algorithm from [49] could be used as well to search for demonstrations. We extend the work from [36] and [49] by using search-based testing methods not only to evaluate the agent but also for training and policy repair.

In addition to testing methods for DNNs that are used for control tasks (i.e., deep RL), the testing community was very active in developing testing frameworks and tools for DNNs used for visual tasks (e.g., for object detection). Tools, like DeepTest [40], TensorFuzz [27], and DeepHunter [44], have been designed to automatically discover errors in deep neural networks that occur only for rare inputs.

We showcase our approach in environments where several subgoals need to be completed to solve a task,like picking up a key to unlock a door. Environments with such sparse or non-Markovian rewards are often approached by learning automata as high-level task description [10, 13, 15, 45]. Our fuzzing-based demonstration generation provides a potentially more lightweight alternative to the inference of automata-based task descriptions.

Even though policy repair has not been investigated in the context of RL policies yet, repair techniques have been used in various application domains. In program repair, automatic analysis methods, like fuzzing, are used to detect and correct errors in software code [12, 26]. In this work, we use search-based testing to detect unsafe behavior of the agent and apply specialized re-training to correct its behavior which we refer to as repair.

3 PRELIMINARIES

A **Markov decision process** (MDP) $\mathcal{M} = (\mathcal{S}, s_0, \mathcal{A}, \mathcal{P}, \mathcal{R})$ is a tuple with a finite set $\mathcal{S}$ of states including initial state s_0, a finite set $\mathcal{A}$ of actions, and a *probabilistic transition function* $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$, and an *immediate reward function* $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$. We consider episodic RL tasks in which an agent fails by violating safety or succeeds by completing the task. Therefore, we introduce two sets of terminal states: *unsafe states* $\mathcal{S}_U \subseteq \mathcal{S}$ and *goal states* $\mathcal{S}_G \subseteq \mathcal{S}$. Visiting an unsafe state $s \in \mathcal{S}_U$ is a safety violation. If a *goal state* $s \in \mathcal{S}_G$ is visited, the task to be learned is completed successfully.

In **reinforcement learning** (RL), [35] an agent learns a task via interactions with an unknown environment modeled by an MDP $\mathcal{M} = (\mathcal{S}, s_0, \mathcal{A}, \mathcal{P}, \mathcal{R})$. At each state $s \in \mathcal{S}$, the agent chooses an action $a \in \mathcal{A}$, the environment then moves to a state s' with probability $\mathcal{P}(s, a, s')$. The reward is determined with $r = \mathcal{R}(s, a)$. A (deterministic) *policy* $\pi : \mathcal{S} \to \mathcal{A}$ is a function mapping states to actions. The set of all policies is denoted by Π. The return ret is the discounted cumulative reward defined by ret $= \Sigma_{t=0}^{\infty}\gamma^{t}\mathcal{R}(s_t, a_t)$, using the *discount factor* $\gamma \in [0, 1]$. The objective of the agent is to learn an *optimal policy* $\pi^\star$ that maximizes the expectation of the return, i.e., $\max_{\pi \in \Pi} \mathbb{E}_{\pi}(\texttt{ret})$.

Given a policy π, a state s and an action a, the value $Q^{\pi}(s, a)$ is defined as the *expected return* that can be obtained from (s, a) when following π. The optimal Q-value function $Q^\star(s, a)$ is computed by solving the Bellman equation:

$$Q^\star(s, a) = \mathbb{E}\Big[\mathcal{R}(s, a) + \gamma\Sigma_{s'}\mathcal{P}(s' \mid s, a) \max_{a'} Q^\star(s', a')\Big].$$

A policy $\pi^\star$ is optimal if $\pi^\star(s) = \max_a Q^\star(s, a)$. In deep RL, the value function $Q^\star(s, a)$ is approximated by a neural network. For a given state s, the network outputs a vector of action values $Q(s, \cdot; \theta)$ that approximates $Q^\star(s, a)$, where θ are the network parameters.

The DDQ Algorithm. The double Deep Q-Network algorithm (DDQ) [41] uses an *online network* with parameters θ and a *target network* with parameters θ^-. The policy is evaluated according to the online network, but the target network estimates the action values. The double DQN error is given by $J_{DQ}(Q) = \mathcal{R}(s, a) + \gamma Q(s_{t+1}, a_{t+1}^{\max}; \theta_t^-)$ with $a_{t+1}^{\max} = \texttt{argmax}_a Q(s_{t+1}, a; \theta_t)$. The online network parameters are continuously updated according to this error. Every T time steps, the target network is updated by overriding θ^- with the online network parameters θ. DDQ uses experience replay, where the agent adds all of its observed transitions to a replay buffer, which is sampled uniformly to update the online network.

The **n-step return** [28] is defined via $\mathcal{G}_t := \Sigma_{i=0}^{n-1}\gamma^{i} \cdot r_{t+i}$, for a constant n. Variations of DDQ, like DQfD, use n-step returns $\mathcal{G}_t$ to consider long-term dependencies instead or in addition to $\mathcal{R}_t$.

Demonstrations. An *experience* is a state-action-reward triple (s, a, r). A *demonstration* (or *trace*) is the experience sequence $\tau = \langle s_0, a_0, r_0, s_1, \dots, s_{n-1}, a_{n-1}, r_{n-1}\rangle$ induced by a policy π during an episode starting in the initial state s_0. It consists of the actions taken by an agent, the corresponding observed environment states, and the gained rewards. $\tau[i]$ is the i^{th} experience, i.e., $\tau[i] = (a_i, r_i, s_i)$ with $0 \leq i < n$. We denote sets of demonstrations with $\mathcal{T}$.

The action sequence with states and rewards omitted from τ is an *action demonstration* $\tau_{\mathcal{A}} = \langle a_0, a_1, \dots a_{n-1}\rangle$, and $\tau_{\mathcal{A}}[i] = a_i$ is the i^{th} action. Executing $\tau_{\mathcal{A}}$ from s_0 of $\mathcal{M}$ yields a trace $exec_{\tau}(\tau_{\mathcal{A}}, s_0) = \langle s_0, a_0, r_0, s_1, a_1, r_1 \dots, s_{n-1}, a_{n-1}, r_{n-1}\rangle$ with $n = |\tau_{\mathcal{A}}|$.

Visiting a goal state or an unsafe state terminates an episode. A demonstration τ is *successful* if it ends in a goal state $s \in \mathcal{S}_G$. If τ ends in an unsafe state $s \in \mathcal{S}_U$, we say that τ *failed* (or is unsafe). Otherwise, we say that τ is *safe*. We denote the set of successful demonstration with $\mathcal{T}_{\text{succ}} \subseteq \mathcal{T}$, and the set of failure traces with $\mathcal{T}_{\text{fail}} \subseteq \mathcal{T}$.

4 STEP 1 - TRAINING FROM FUZZ DEMONSTRATIONS

Our framework starts from the classical RL setting without having any expert demonstrations available. Figure 4 gives an overview of our framework for RL via fuzz demonstrations. In the first step of our framework, we search for a reference demonstration $\tau_{\text{ref}} \in \mathcal{T}_{\text{succ}}$ that solves the task to be learned, not necessarily in an optimal way. τ_{ref} is then used as a seed for the fuzzing algorithm to search for a diverse set of successful demonstrations $\mathcal{T}^d \subseteq \mathcal{T}_{\text{succ}}$. Finally, the fuzz demonstrations $\mathcal{T}^d$ are used for RLfD. We use the approach of [36] to search for a reference trace and to fuzz demonstrations. Thus, we only outline these algorithms in Section 4.1 and Section 4.2, and refer to [36] for details.

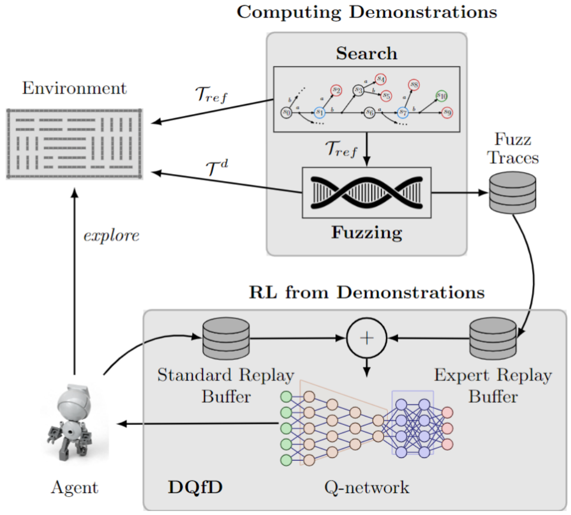


Figure 4: Overview of RL from fuzzed demonstrations.

4.1 Search for a Reference Demonstration

We search for a reference demonstration τ_{ref} using *backtracking-based, depth-first search* (DFS) by sampling the MDP $\mathcal{M}$. Every time the search visits an unsafe state in $\mathcal{S}_U$, the DFS algorithm backtracks. Already visited states along a search branch are stored to detect loops. If a state is visited again, the algorithm backtracks. When visiting a goal state in $\mathcal{S}_G$, the DFS terminates successfully.

Important for the search is that the stochastic behavior of $\mathcal{M}$ is abstracted away by repeating actions sufficiently often [19]. The search can be optimized by using proper abstractions of the state space to merge similar states.

Example. As an example, consider the problem of finding a reference demonstration solving the first level of Super Mario Bros, as illustrated in Figure 1. For the DFS, we only allow the actions "right" and "jump". Whenever an unsafe state is visited during the DFS, the search backtracks. White dots mark branching points. The search terminates if the demonstration reaches the end of the level.

4.2 Fuzzing of Demonstrations

Given the demonstration τ_{ref} as a seed, we use search-based fuzzing methods [47] to compute a set of demonstrations $\mathcal{T}^d \subseteq \mathcal{T}_{\text{succ}}$. Using a proper *fitness function* that assigns a value to a demonstration based on its gained reward and visited states, we guide the search to fuzz successful demonstrations that cover a large portion of the state space. Using the fitness function, we apply a genetic algorithm with a fixed population of action demonstrations for a fixed number of generations. The mutations insert, replace, remove, or append new actions into individuals. We combine individuals through a point-wise crossover that concatenates a prefix and a suffix of two individuals that are split at a random point. The fitness function is a weight sum of the normalized return gained from executing an individual action demonstration and the number of new states explored by the execution. The latter steers the search towards a diverse set of traces covering a large part of the state space, thus enabling learning robust policies via RLfD.

4.3 Learning from Fuzz Demonstrations

The fuzzing algorithm provides us with a set of fuzzed demonstrations $\mathcal{T}^d \subseteq \mathcal{T}_{succ}$ which can now be used for any RLfD algorithm. One such RLfD algorithm is deep Q-learning from demonstrations (DQfD) [14] which builds upon the DDQ algorithm.

Overview of DQfD. During training, the agent is equipped with two replay buffers:

- an *expert replay buffer* storing all experiences $\tau[i]$ with $0 \leq i < |\tau|$ for all demonstrations $\tau \in \mathcal{T}^d$, and
- a *standard replay buffer* of fixed size, where experiences gained by the agent are stored. Whenever the buffer size exceeds the fixed size, the oldest experiences are dropped.

DQfD implements training in two stages. In the first stage, the agent gets *pretrained* by replaying experiences observed during demonstrations from $\mathcal{T}^d$. For this purpose, we sample minibatches of size k and update the online Q-network by learning with the loss function $J(Q) = J_{DQ}(Q) + \lambda_1 J_n(Q) + \lambda_2 J_E(Q) + \lambda_3 J_{L2}(Q)$. $J_{DQ}(Q)$ is the standard one-step DQN loss, $J_n(Q)$ is an n-step return loss, $J_E(Q)$ is a large margin classification loss [29] that makes the fuzz-demonstration action values appear better in comparison to other actions, and $J_{L2}(Q)$ is a weight-regularization loss.

The second stage of DQfD is similar to standard DDQ, where the agent explores the environment and fills the standard replay buffer with new experiences. This buffer is a dequeue for *normal* experiences, i.e., it overrides old experiences upon reaching a capacity limit, whereas demonstrations from the expert replay buffer are never discarded. Whenever we update the online Q-network, we sample a minibatch of k experiences from both buffers. The same loss $J(Q)$ applies to both experience types, except that $J_E(Q)$ only applies to fuzz demonstrations. We use a slight adaptation of DQfD, where we sample a fixed share of $s_{exp} = \lceil k \cdot r_{\text{exp}} \rceil$ fuzz experiences and a fixed share of $\lfloor k \cdot (1 - r_{\text{exp}}) \rfloor$ normal experiences. This enables a straight-forward application to policy repair. We perform e_{train} episodes during the training stage.

5 STEP 2 - SAFETY TESTING OF RL AGENTS

The next development step after computing a policy is to evaluate it. Several recent testing approaches for deep RL agents could be applied [3, 36, 49]. In this paper, we evaluate the safety of the trained agent reusing the approach from [36], with the small modification of introducing probabilistic sampling of actions. In this section, we outline the approach for safety testing and refer to [36] for details.

Search for Boundary States. We first compute a test suite, which is a set of traces leading to safety-critical situations in which we want to evaluate the behavior of the agent. A safety test suite can be obtained as a side product of the search for the reference trace τ_{ref} as follows. The DFS backtracks when reaching an unsafe state in $\mathcal{S}_U$ in the MDP $\mathcal{M}$. Thus, the search reveals safety-critical situations. A *boundary state* is a state in a reference trace to which the DFS backtracked, i.e., a branching point. Therefore, it is a state from which the DFS first only explored failing traces before finding the successful trace τ_{ref}. While the failing branches of the search indicate that the boundary state is close to violating safety (visiting a state in $\mathcal{S}_U$), the reference trace suggests that it is still possible to circumvent a safety violation. The safety test suite is formed from the set of prefixes of τ_{ref} leading to the observed boundary states which we denote with $\mathcal{S}_B \subseteq \mathcal{S}$.

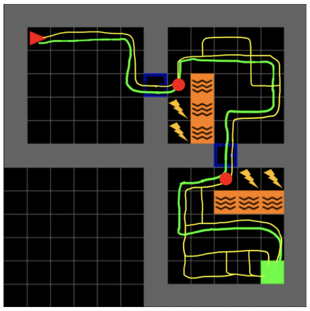


Figure 5: Gridworld showing a reference demonstration (green), boundary states (red dots), and fuzzed demonstrations (yellow). Fields with a lightning bolt are safety-critical.

Example. *Figure 5 illustrates the computation of a safety test suite. The green trace is an example of a reference trace τ_{ref}. The boundary states detected via searching for τ_{ref} are shown as red dots.*

Test-Case Execution. The test-case execution is parameterized by a trace τ_{tc} leading to a boundary state and a test length tl. To execute a test case, we first execute the actions of τ_{tc}. After that, we let the policy under test π_{UT} take over. For tl many test steps, we choose an action a according to π_{UT}. We execute a in the environment and observe the next state s. If $s \in \mathcal{S}_U$, the test-case execution is stopped with a **fail** verdict. A test-case execution **passes** if we perform tl steps without visiting an unsafe state. Note that each test case should be executed several times to take the probabilistic behavior of the environment $\mathcal{M}$ into account.

We propose a slight modification of the test-case execution of [36]. Additionally to τ_{tc} and tl, we parameterize test-case execution with a low probability p_{change} for action perturbations. During the execution of a test case, we choose a random action a with probability p_{change} instead of picking the action from π_{UT}. Having a probability $p_{change} > 0$ has two advantages. First, it increases test coverage since a randomized policy likely visits more states than a deterministic policy. Second, it takes action robustness [39] in the safety testing into account. Since an action might fail during execution, this evaluation gives a measure of how close agents are to violating safety.

Example. *In Fig. 5, states labeled with a lightning bolt are safety-critical states which are not boundary states and thus are not in the safety test suite. However, a test-case execution with $p_{change} > 0$ might also visit such states such that the agent's policy in these cases can be evaluated. Additionally, a policy that avoids situations where a perturbed action selection would violate safety will have higher scores in the safety evaluation.*

Algorithm 1 Policy repair.

```
Input: Policy under repair $\pi_{UR}$, repair states $\mathcal{S}_R$, fuzz demonstrations $\mathcal{T}^d$, maximum repair environment distance $\Delta$, DQfD parameters
Output: Repaired policy $\pi_{UR}$
 1: $RE \leftarrow \{\}$                                   ▷ Initialize repair experiences
 2: for $\tau \in \mathcal{T}^d$ do
 3:     for $(s_i, a_i, r_i) \in \tau$ do
 4:         if $\exists s_R \in \mathcal{S}_R : d(s_i, s_R) \leq \Delta$ then
 5:             $RE \leftarrow RE \cup \{(s_i, a_i, r_i)\}$
 6: $ExpBuffer \leftarrow RE$, $ReplayBuffer \leftarrow \{\}$
 7: while $|ReplayBuffer| < bufferSize$ do
 8:     $\tau \leftarrow$ RunEpisode($\pi_{UR}$)
 9:     for $(s_i, a_i, r_i) \in \tau$ do
10:         $ReplayBuffer \leftarrow ReplayBuffer \cup \{(s_i, a_i, r_i)\}$
11: for $i \leftarrow 0$ to $pretrainSteps$ do
12:     $r_{\text{exp}} \leftarrow \frac{i \cdot r_{\text{exp-rep}}}{pretrainSteps}$
13:     $s_{\text{exp}} = \lceil k \cdot r_{\text{exp}} \rceil$
14:     $\pi_{UR} \leftarrow$ DQfDpre($\pi_{UR}, s_{\text{exp}}, ReplayBuffer, ExpBuffer$)
15: $\pi_{UR} \leftarrow$ DQfD($\pi_{UR}, s_{\text{exp}}, ReplayBuffer, ExpBuffer$)
16: return $\pi_{UR}$
```

6 STEP 3 - POLICY REPAIR VIA FUZZ DATA

The second step of our framework tests the policy of a trained RL agent. For a given policy under repair π_{UR}, testing returns a set of states $\mathcal{S}_R \subseteq \mathcal{S}$, called *repair states*, in which the behavior of the policy is unsafe. The goal of policy repair is to robustly correct the behavior of the agent's policy around the states in $\mathcal{S}_R$ while retaining the overall performance of the agent.

Overview of Policy Repair. We are given the policy under repair π_{UR}, the set of fuzz demonstrations $\mathcal{T}^d$ as described in Sec. 4.2, and the set of repair states $\mathcal{S}_R$. For any state $s_R \in \mathcal{S}_R$, we assemble a set of *repair experiences E* that represent correct behavior in the vicinity of s_R, i.e., experiences that we use to repair policies. We extract these experiences from $\mathcal{T}^d$. In case that the demonstrations in $\mathcal{T}^d$ do not contain enough experiences near a state $s_R \in \mathcal{S}_R$, the environment can be sampled for additional demonstrations that represent correct behavior near s_R. Finally, we repair the policy using the repair experiences and obtain a new policy π'_{UR}. With π'_{UR}, we return to the testing step of our framework. In case that testing reveals undesired behavior in π'_{UR}, another iteration of repair can be performed until the performance and safety of the final policy is satisfying. Algorithm 1 implements the individual of steps of policy repair, which we discuss in detail below. Parameters, like the number of training episodes for repair e_{repair}, are summarized in the input *DQfD parameters*.

Assembling of Repair Experiences. The repair algorithm starts by collecting repair experiences in the lines 1 to 5. Four components form the basis for the computation of the repair experiences *RE*: (1) the set of repair state $\mathcal{S}_R$, (2) the fuzz traces $\mathcal{T}^d$ demonstrating safe behavior, (3) a problem-dependent distance function d, and (4) a Δ defining the size of repair environment around individual repair states. We check every experience $e = (s_i, a_i, r_i)$ of every fuzz demonstration $\tau \in \mathcal{T}^d$. If the state s_i lies in the neighborhood of any repair state, we add e to the set of repair experiences. To define a distance function, we generally apply an abstraction over the states and we choose Δ based on the test-case execution length tl and the length of the reference trace. Δ can be seen as a distance to be traveled by the agent during a successful test. To this end, *RE* contains the experiences prior to reaching the repair state and as many experiences as would be required to pass the test case. Note that alternatively to abstraction-based distance functions, simple index-based distance functions may be viable. If reference trace and fuzz demonstrations have similar length, the s_i in an experience e can be abstracted to its position in the fuzz demonstration.

Policy Repair. The remainder of Algorithm 1 implements our proposed DQfD variation. Using the repair experiences *RE*, we repair the policy π_{UR} that comes in the form of a neural network. We implement this through DQfD while ensuring a non-abrupt distribution shift. In Line 6, we populate the expert replay buffer with the repair experiences *RE*. Lines 7 to 10 execute the non-repaired π_{UR} to sample demonstrations that we use to fill the standard replay buffer. In the lines 11 to 14, we perform the *pre-training* stage of DQfD with the important modification that we linearly increase the share of expert demonstration experiences from 0 (using no experiences from the expert buffer) to a specified target ratio $r_{\text{exp-rep}}$ of expert and normal experiences. The function DQfDpre performs minibatch updates of the neural network to update the policy π_{UR}, like in the pretraining stage of DQfD. In contrast to standard DQfD, we use the experiences of the trained unsafe agent also during pre-training. With that and the gradually increasing expert share, we ensure that the distribution of experiences does not shift drastically as compared to the initial training of π_{UR}. This DQfD-like pretraining adjusts the policy under repair π_{UR} so that it follows the repair experiences *RE*, rather than training a new policy from scratch. By doing so, we repair the agent so that it learns to safely escape from experienced safety-critical states. Finally, in Line 15, we perform e_{repair} training episodes, where we update the Q-network with minibatches containing expert experiences and newly gained experiences as in the second stage of DQfD outlined in Sec. 4.3.

Evaluation & Iterated Repair. Repairing the policy π_{UR} for the repair states $\mathcal{S}_R$ gives us a new policy π'_{UR}. In the next step, we test π'_{UR} for safety according to Sec. 5. This step may reveal new repair states $\mathcal{S}'_R$ for π'_{UR}. If $\mathcal{S}'_R$ is nonempty, we perform another repair step for π'_{UR} using the repair states $\mathcal{S}'_R \cup \mathcal{S}_R$. That is, we perform Algorithm 1 again. The nature of training may induce new unsafe or non-optimal behavior. In our experiments, however, we found that the number of repair states generally decreases and a few repair iterations (one or two) are mostly sufficient.

7 EXPERIMENTS

We train RL agents to complete the first four levels of Super Mario Bros. (SMB) with RL-development framework and four different gridworlds. The experiments were conducted on nodes of a compute cluster equipped with an Nvidia Tesla T4™ graphics card.

Common Parameters – Training. In both types of case studies, we set the following parameters for DDQ and DQfD: $\gamma = 0.95$, a buffer size of $2 \cdot 10^4$, $n = 10$ for n-step return losses, a minibatch size of $k = 32$, and we use the Adam optimizer [20] with a learning rate of 10^{-4}, a weight decay of $5 \cdot 10^{-5}$. We start training with an exploration rate of 0.1 and decrease it with a multiplicative decay rate of 0.999999 to 0.01 (which is the fixed exploration of DQfD for ATARI games [14]). The DQfD-specific parameters are as follows: We perform $t_{\text{pre}} = 2 \cdot 10^5$ pre-training steps and use a fixed expert demonstration share of $r_{\text{exp}} = \frac{1}{2}$. Finally, the weights for the losses are $\lambda_1 = \lambda_2 = \lambda_3 = 1$.

Common Parameters – Testing. For testing, we set the test length such that we check the environment around a boundary state without reaching the next boundary state, thus individual tests are independent from each other. We set $p_{change} = 0.01$ to test robustness to small perturbations of the agent's actions.

7.1 Super Mario Bros. Experiments

Training from Fuzz Demonstrations – Setup. All SMB experiments where performed in the SMB environment [18], where we build upon the implementation of [36] that provides a DDQ agent for comparison and a search-based testing framework. For each level, we first computed a single reference demonstration τ_{ref} that completes the level. For fuzz testing, we used a population size of 100 and 50 iterations, i.e., we simulate 5000 episodes during fuzz testing. All successful demonstrations of each generation form the demonstration set $\mathcal{T}^d$. To evaluate the quality of demonstrations from fuzz testing, one of the authors manually collected 50 successful demonstrations $\mathcal{T}^e$ for each of the four considered levels. We denote experiments with such expert demonstrations with DQfED and experiments with DQfD from fuzz demonstrations with DQfD. We trained both types of DQfD agents with the respective demonstrations in $\mathcal{T}^d$ and $\mathcal{T}^e$, and we compare to double DQN agents trained with n-step return, which we abbreviate DDQ. Note that computing n-step return losses for DDQ enables a fair comparison with DQfD in terms of how much information is extracted from gained experiences. Using n-step return losses in learning ensures that experiences are propagated further in time faster. We compare all agents in terms of gained cumulative rewards.

We perform $e_{\text{train}} = 15000$ episodes in the training phase of DQfD and 20000 training episodes with DDQ. Hence, the sampling budget is the same for both approaches, as we fuzz for 5000 episodes. The cost of the initial search for a reference trace is negligible as we show in Table 1. The agents' task is to finish a level. They get more reward the further they get in the level and get negative reward for a game over and for the amount of time spent in the level. Concretely, the reward in a step is given by $\frac{(x-x')+(t-t')+d}{100}$, where x is the agent's x-coordinate in a level, t is the time left to complete the level, primed values denote the corresponding values from the last time step, and $d = -25$ if an unsafe state is reached and $d = 0$ otherwise. The scaling factor of 100 improves convergence when training uses n-step return losses. The MDP states seen by the RL agent are stacks of four greyscale images resized to 84 by 84 pixels, where stacking helps to track movement speed and direction. The actions are running and jumping to the right. Since levels are of different lengths, the maximal cumulative reward varies from level to level.

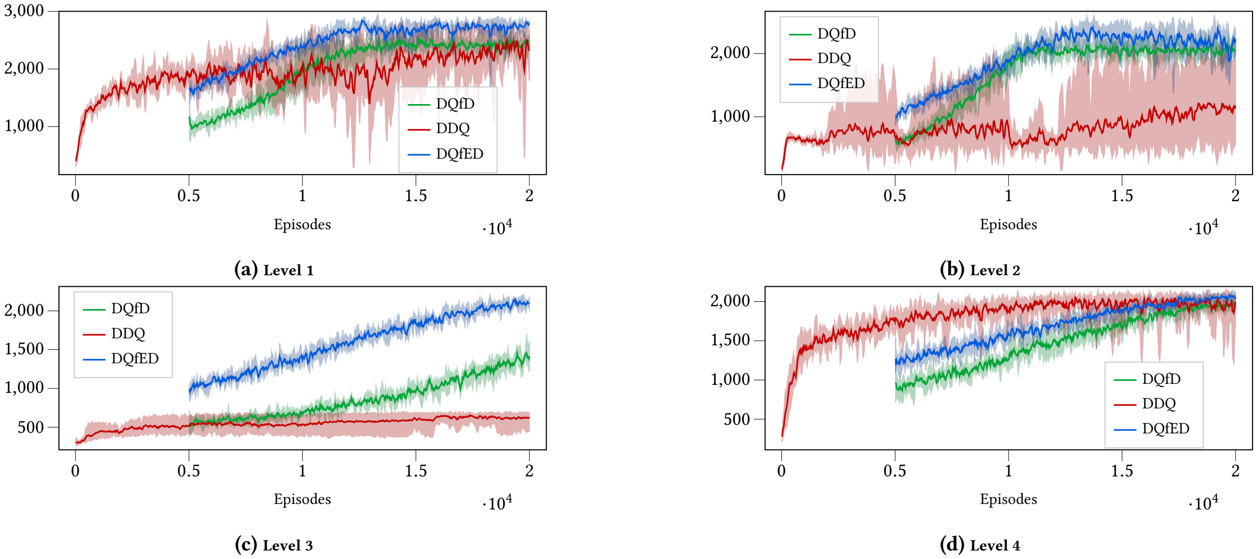
Training from Fuzz Demonstrations – Results. We plot the average cumulative reward gained during training in Fig. 6. The thick line is averaged over five runs of each training type, whereas the shaded area depicts the range between minimum and maximum of the average cumulative rewards. The green lines represents the results for our DQfD implementation using the demonstrations $\mathcal{T}^d$, the blue lines depict the DQfED results with expert demonstrations, and the red line represents the DDQ agent. We offset the DQfD plots by 5000 episodes, as this is the sampling budget for fuzzing.

Table 1: Average runtime in seconds of the individual steps in DQfD and runtime of DDQ.

	DQfD				DDQ
	search	fuzzing	pretraining	Σ	Σ
1-1	7.89	5566.08	1567.67	76218.08	90696.12
1-2	152.69	5554.94	1567.87	81671.72	195373.51
1-3	75.43	3016.59	1566.37	24892.87	15332.85
1-4	18.47	7026.72	1556.60	39836.69	49451.16

We can see that DQfD only had a minor effect in the learning performance for levels 1 and 4, but it allowed to successfully train agents for levels 2 and 3 in which DDQ hardly progressed. The reason, most likely, is that the levels have varying difficulties: Level 1 and Level 4 are relatively basic with only a few obstacles scattered across the levels. In contrast, Level 2 features several pits and obstacles in close proximity to each other, and Level 3 features many pits. Thus, Level 2 and Level 3 require tight maneuvers that are difficult to learn only via exploration, such that demonstrations give the agent a head start. The derived *test suites* also suggest a difference in difficulty between the levels. There are 31 test cases for Level 3, despite it being relatively short, whereas there are only 16 test cases for Level 4. Note that for all four levels, the DQfD agent shows a very stable learning curve. Comparing DQfD and DQfED, we can see that expert demonstrations lead to slightly better performance in the levels 1, 2, and 4 and to faster convergence and more effective pretraining in Level 3. In contrast, DQfD shows more stable progression; the minimal cumulative return of DQfED shows some spikes, where it goes below the DQfD return range, most prominently in Level 2. Hence, DQfD with completely automatic demonstration generation is mostly on par with DQfED, which requires an expensive human oracle. DQfD with fuzz demonstration would require more training to reach the performance of DQfED in Level 3, which is possible without human intervention. The plots of the second repair iteration for Level 3 in Fig. 7 show that additional training with fuzz demonstrations indeed further improves performance.

We balanced the sampling budget between DQfD and DDQ. Table 1 additionally provides an overview of the runtime of the individual steps. The columns contain the stage, the runtime of the DFS for the reference trace, the runtime of fuzzing, and the runtime of the pretraining steps, i.e., neural network updates without sampling. The summed runtime for DQfD includes all these steps and the time required by training, whereas the summed DDQ runtime solely includes the training runtime. The search is generally fast, taking less than three minutes. Fuzzing takes at most 2.7 hours and the pretraining time is typically about half an hour since we set a constant number of minibatch updates of the neural network. Comparing the overall runtimes, we can observe that the time invested in preparation for DQfD pays off, as the overall runtime of DDQ is generally higher. DDQ is faster only in Level 3, where the DDQ agents fail early, such that episodes are short. The difference between DDQ and DQfD is especially large in Level 2, where the agent may get stuck due to obstacles blocking the way.



(a) Level 1

(b) Level 2

(c) Level 3

(d) Level 4

Figure 6: The cumulative rewards gained during training of SMB with a sliding average of 100 episodes.

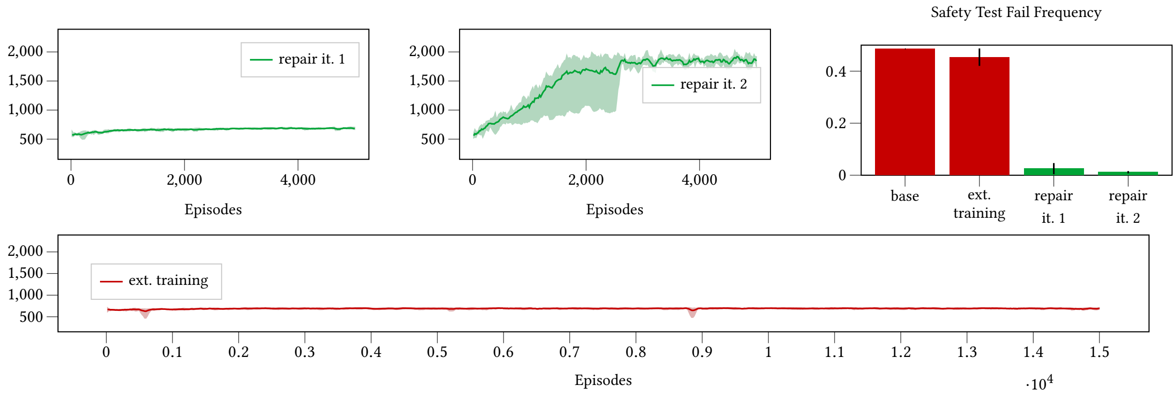


Figure 7: Cumulative rewards gained during training performed for repair and extended training for Level 3, and test results before and after repair, and after extended training.

Repairing DDQ Agents – Setup. To evaluate policy repair, we test the DDQ agent (trained for 15000 episodes) for safety issues and repair these issues using the fuzz demonstrations. We set the target expert share to $r_{\text{exp-rep}} = 0.5$ and use the same parameters as for DQfD with the following exceptions: We set the weight for the supervised loss to $\lambda_2 = 0.1$, we decrease the exploration rate to 0.99999, and we perform 5000 training episodes. With the lower λ_2, we aim to change the policy carefully. The reason for less exploration is that the policies under repair already performs well. For the experience selection, we define the distance d via the difference in the x-coordinates based on the underlying coordinate system of SMB, i.e., we abstract states to their x-coordinates. We set $\Delta = tl \cdot \overline{xs}$, where tl is the length of a test, which is the number of steps performed during a test after visiting the corresponding boundary state, and $\overline{xs}$ is the average distance traveled in the x-direction within a single step.

Finally, we compare repaired policies to policies resulting from *extended training* for another 15000 episodes starting from the initial policy under repair. That is, after extended training, the agent has been trained for a total of 30000 episodes. After two repair iterations, the repaired agent has been trained for a total of 25000 episodes and the repair makes use of demonstrations collected during 5000 fuzzing episodes. Hence, the comparison is fair in terms of sampling budget. Moreover, repair and extended training start from the same initial policy. We perform each run of repair and extended training five times for one of the base DDQ agents trained for 15000 episodes.

Repair Results. In Fig. 7 and Fig. 8, we show results from repairing and extended training for Levels 3 and 4. The results highlight different aspects: repair improves safety as well as performance for the third level. In Level 4, standard DDQ already performs well w.r.t. reward, thus repair improves only safety. The line plots at the top show the cumulative reward gained during training for repair as in

Fig. 6, while the plot at the bottom shows the cumulative reward gained during extended training. The bar plots depict average safety testing results that are ratios of failing test cases to all executed test cases. The red bars correspond to test results from DDQ training for 15000 episodes and extended training for another 15000 episodes, respectively. The green bars correspond to test results after repair iterations. Black bars show the respective standard deviation.

DDQ did not find a good policy for Level 3, even with extended training. Analogously to performance, extended training did not increase safety. With the first repair iteration, we generally achieved close-to-zero test fails, albeit with low reward. The policies failed at a point that is not covered by the generated test suite. We combated that by extending the repair experiences to include fuzz demonstrations near the state where the agent fails, as determined by episode length. Since this requires a larger change of the policy under repair, we changed two parameters for the second repair iteration of Level 3 agents, increasing the weight for the supervised loss to $\lambda_2 = 1$ and the pre-training steps to $t_{\text{pre}} = 5 \cdot 10^5$. Fig. 7 shows that this indeed improves the reward, even beyond the reward of DQfD shown in Fig. 6c. At the same time, the safety test-case fail rate stays low after the second repair iteration.

For Level 4 in Fig. 8, we see that extended training can hurt safety, despite negative rewards for safety issues. The first repair iteration likewise even reduced the overall safety. A closer investigation showed that it mostly fixed issues corresponding to failing test cases of the base DDQ agent, but it introduced issues elsewhere. The policy was changed considerably in this first repair iteration, causing other test cases to fail and unstable performance in terms of reward. The second iteration fixed most safety issues. Only one test case still failed in three of five runs after the second iteration.

7.2 Minigrid Experiments

Setup. The gridworld experiments were performed in the Minigrid environment [5]. For this gridworld environment, we implemented a DDQ agent and a fuzz-testing framework. The reward in these environments is given by $g - 0.01 \cdot d - s + r$, where $g = 1$ if the goal is reached, d is the Manhattan distance to the goal, $s = 0.01$ if the maximum number of steps is reached, and $r = 0.025$ if a new room was entered. For Minigrids with only one room, we penalize each step with -0.01. The states seen by the RL agent are 84 by 84 images of the environment and the available actions are: turn left, turn right, move forward, pick up, put down, and unlock. We reduce the action set to a subset that is sufficient to reach the goal. Additionally, the agent knows if it is holding an item.

Training from Fuzz Demonstrations. For our evaluation, we selected four gridworlds: (a) `ThreeRoomsLava`, (b) `DistShift1`, (c) `LavaCrossingS9N1`, and (d) `UnlockPickup`. Gridworld (a) depicted in Fig. 5 is a manually designed multi-room gridworld with doors and lava fields, while gridworlds (b)-(d) are from [5]. The agent's task is to enter the goal field or pick up a box. To do this, the agent must unlock and open doors, avoid lava fields, and complete the task before reaching the maximum number of steps. Note that the agent must complete subtasks sequentially, which may temporarily decrease the cumulative reward by increasing the distance to the goal. However, completing the series of subtasks ultimately leads to a large cumulative reward via completion of the entire task. For example, in `UnlockPickup` (d), shown in Fig. 9, the agent must perform a fairly complex sequence of actions including picking up the key, unlocking the door, discarding the key, and picking up the box. The learning setup is similar to the SMB case study, except that we set an increased initial exploration rate of 0.4 for Gridworld (a) to guarantee the DDQ agent sufficient exploration in all rooms. In comparison to SMB, we reduced the training episodes to $e_{\text{train}} = 1500$ episodes in the training phase of DQfD and to 2000 training episodes with DDQ.

Table 2: Average runtime in seconds of the individual steps in DQfD and runtime of DDQ for the Minigrid experiments.

	DQfD				DDQ
	search	fuzzing	pretraining	Σ	Σ
`DistShift1`	0.04	149.38	1859.24	4108.30	9780.74
`LavaCrossingS9N1`	0.06	211.06	1735.18	3754.74	6628.17
`UnlockPickup`	0.06	386.33	1869.72	6363.39	72462.02
`ThreeRoomsLava`	0.13	1221.98	1844.85	16582.26	48147.06

Results. Figure 9 illustrates the cumulative rewards gained during training for the gridworlds. DDQ and DQfD could learn policies to complete the task of the gridworlds (b) and (c). Similar to SMB, we see that DDQ yields more unstable and inferior results. Therefore, DDQ could not reliably learn a policy that can achieve the goal. The DQfD results show that search-based demonstrations can also support learning of navigation problems. DDQ was not able to learn the Gridworlds (a) and (d) due to sparse rewards and tasks that require a certain sequence of actions for completion. We assume that DDQ requires a different reward function to accomplish this task, while DQfD is sufficient without further reward shaping. Hence, our demonstration generation provides an alternative to the inference of high-level subgoal descriptions that is often applied in environments with sparse or non-Markovian rewards [13, 45]. Moreover, Table 2 shows that DQfD is on average 4.45 faster in training than DDQ even when 5000 episodes are fuzzed in advance.
Repair Results. Figure 10 presents the results of the testing and repair of agents for Gridworld (a). Note that we aim to test a policy for robustness. Similar to SMB, the test and repair results highlight different aspects. First, testing of the trained DDQ agent showed failed test cases. A closer look at the trajectory of the agent revealed that the DDQ-learned policy is more likely to run into safety-critical states, which are marked with a lightning bolt in Fig 5. Second, by repairing the policy with fuzz demonstrations, the agent not only avoids these safety-critical states but improves performance by achieving the goal in fewer steps. We executed 5000 test cases for the DDQ policy, which resulted in a proportion of 0.16 failing test cases. By repairing, we can reduce the proportion to 0.0062 at the mean. This is a significant reduction with $p < 0.01$. Extended training could not reduce the proportion of failing test cases and the standard deviation (0.013) was exceptionally high between the different test executions. Furthermore, unlike repair, extended training did not improve the agent's performance. We omit repair for gridworlds (b)-(d) since in gridworlds (b) and (c) the agents could achieve the task within a fair amount of episodes, and Gridworld (d) has no safety-critical states (no lava).

Threats to Validity. Our findings compare DQfD from fuzzing data and standard DDQ, where DQfD shows favorable performance.

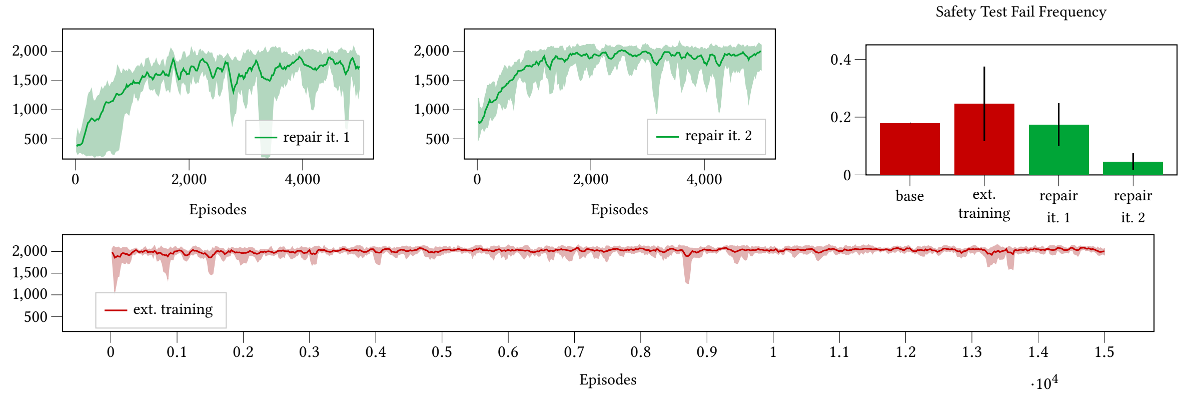
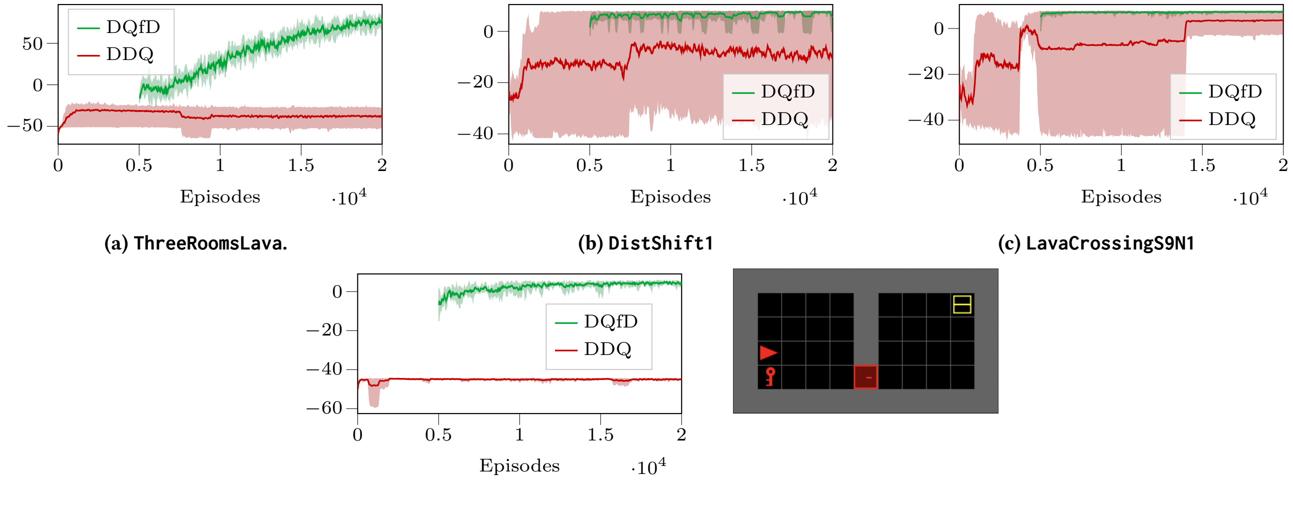


Figure 8: Cumulative rewards gained during training performed for repair and extended training for Level 4, and test results before and after repair, and after extended training.



(a) ThreeRoomsLava.

(b) DistShift1

(c) LavaCrossingS9N1

(d) UnlockPickup

Figure 9: Cumulative rewards gained during training of gridworlds with a sliding average of 100 episodes.

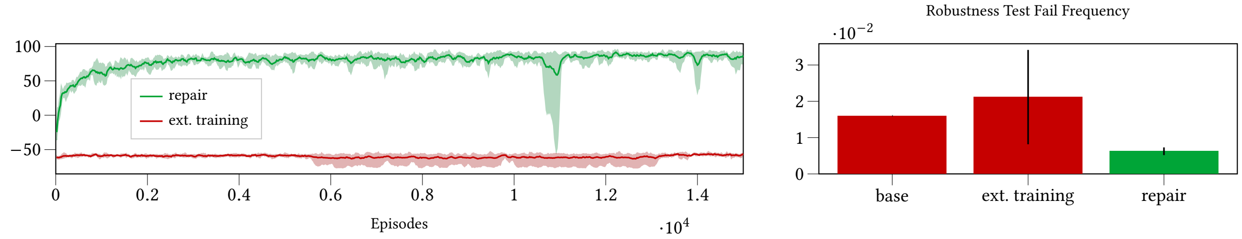


Figure 10: Cumulative rewards gained during training performed for repair and extended training for (d) ThreeRoomsLava gridworld, and test results before and after repair, and after extended training.

The results may be influenced by the hyperparameter selection, i.e., different hyperparameters may favor DDQ. However, we aimed to decrease the dependency on the hyperparameter selection. For this, we tuned the hyperparameters on the first SMB level so that both approaches can learn well within about 20000 episodes. We left the hyperparameters unchanged for all other levels, similar to [14] who used the same parameters for various games.

For demonstrating the general feasibility of our approach, we considered two environments that are different in nature. Nevertheless, there is a possibility that our results may not translate to other environments. However, the environments chosen are representative of RL problems considered in the literature. Computer games are popular benchmarks since they involve long, complex action sequences, require image processing, and feature adversarial environments [14, 24, 43]. Navigation in gridworlds, while appearing simple, is often challenging for RL, as rewards are sparse and only gained when sequences of subtasks have been completed; cf. `UnlockPickup` in Fig. 9 and work on inferring subtask structures [8, 13, 45]. In order to apply our approach, we require two assumptions to be fulfilled by an environment. First, we rely on RLfD to work, where others [31] have shown that it works well in various RL environments. Second, we require a testing approach for the environment under consideration. In this regard, we show that search-based testing can be adapted to navigation in gridworld environments. RL testing is in its early stages, but recent work shows its potential to increase trust in RL [22, 49], a very relevant research topic. In particular, Zolfagharian et al. [49] propose search-based testing for control tasks, another popular type of RL task.

Another potential threat to validity stems from uncertainty. Due to their long runtime, we performed only five runs of each experiment. In these experiments, the DQfD performance shows a clear trend and low variance as depicted by Fig. 6 and Fig. 9, whereas DDQ shows more variance. Considering maximum cumulative reward, we can see that learning from fuzz-trace experiences especially helps in solving challenging tasks, like levels 2 and 3 of SMB, and complex navigation tasks like in Gridworld (d). However, the search performed for fuzzing is heavily influenced by randomness, which potentially affects DQfD performance. If the search fails to find good demonstrations, DQfD may show worse performance than in our experiments.

Supplementary Material. Source code, setup files, and raw result data are available online [37].

8 CONCLUSION

We propose a development approach for RL from demonstrations obtained via search-based testing. The approach includes learning from demonstrations, testing of a learned policy, and repairing of policies using safe demonstrations. While previous work on RL from demonstrations relies on data collected by (human) experts, we generate demonstrations fully automatically using fuzz testing. We showcase deep Q-learning from fuzz demonstrations with two case studies comprising levels from the computer game Super Mario Bros. and navigation tasks in gridworlds. Our experiments demonstrate that DQfD using fuzz demonstrations helps to solve tasks more efficiently, especially if they are difficult. To repair a policy, we first identify safety issues and safe alternative behavior through search-based testing that includes fuzz testing. Via iterated DQfD, we repair a policy w.r.t. safety while retaining its performance in terms of cumulative reward. In particular, we show that we can reliably improve safety, whereas extended training does not necessarily improve safety.

In future work, we will explore other RLfD approaches and combinations with testing approaches, like learning-based testing [1, 23] using appropriate abstractions, e.g., through clustering [7].

ACKNOWLEDGMENTS

This work has been supported by the "University SAL Lab" initiative of Silicon Austria Labs (SAL) and its Austrian partner universities for applied fundamental research for electronic based systems. Further, this work has been supported by the State Government of Styria, Austria – Department Zukunftsfonds Steiermark. Additionally, this work has been funded by the AIDOaRt project (grant agreement No 101007350) from the ECSEL Joint Undertaking (JU). The JU receives support from the European Union's Horizon 2020 research and innovation programme and Sweden, Austria, Czech Republic, Finland, France, Italy, and Spain. Finally, the authors would like to acknowledge the use of HPC resources provided by the ZID of Graz University of Technology

REFERENCES

[1] Bernhard K. Aichernig, Roderick Bloem, Masoud Ebrahimi, Martin Horn, Franz Pernkopf, Wolfgang Roth, Astrid Rupp, Martin Tappler, and Markus Tranninger. 2019. Learning a Behavior Model of Hybrid Systems Through Combining Model-Based Testing and Machine Learning. In *Testing Software and Systems - 31st IFIP WG 6.1 International Conference, ICTSS 2019, Paris, France, October 15-17, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11812)*. Springer, 3–21. https://doi.org/10.1007/978-3-030-31280-0_1

[2] Yusuf Aytar, Tobias Pfaff, David Budden, Tom Le Paine, Ziyu Wang, and Nando de Freitas. 2018. Playing Hard Exploration Games by Watching YouTube. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*. 2935–2945. https://proceedings.neurips.cc/paper/2018/hash/35309226eb45ec366ca86a4329a2b7c3-Abstract.html

[3] Matteo Biagiola and Paolo Tonella. 2023. Testing of Deep Reinforcement Learning Agents with Surrogate Models. arXiv:2305.12751 [cs.SE]

[4] Si-An Chen, Voot Tangkaratt, Hsuan-Tien Lin, and Masashi Sugiyama. 2020. Active Deep Q-Learning with Demonstration. *Mach. Learn.* 109, 9-10 (2020), 1699–1725. https://doi.org/10.1007/s10994-019-05849-4

[5] Maxime Chevalier-Boisvert, Bolun Dai, Mark Towers, Rodrigo de Lazcano, Lucas Willems, Salem Lahlou, Suman Pal, Pablo Samuel Castro, and Jordan Terry. 2023. Minigrid & Miniworld: Modular & Customizable Reinforcement Learning Environments for Goal-Oriented Tasks. *CoRR* abs/2306.13831 (2023).

[6] Anamika Dhillon and Gyanendra K. Verma. 2020. Convolutional Neural Network: A Review of Models, Methodologies and Applications to Object Detection. *Prog. Artif. Intell.* 9, 2 (2020), 85–112. https://doi.org/10.1007/s13748-019-00203-0

[7] Guoliang Dong, Jingyi Wang, Jun Sun, Yang Zhang, Xinyu Wang, Ting Dai, Jin Song Dong, and Xingen Wang. 2020. Towards Interpreting Recurrent Neural Networks through Probabilistic Abstraction. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 499–510. https://doi.org/10.1145/3324884.3416592

[8] Daniel Furelos-Blanco, Mark Law, Anders Jonsson, Krysia Broda, and Alessandra Russo. 2021. Induction and Exploitation of Subgoal Automata for Reinforcement Learning. *J. Artif. Intell. Res.* 70 (2021), 1031–1116. https://doi.org/10.1613/jair.1.12372

[9] Yang Gao, Huazhe Xu, Ji Lin, Fisher Yu, Sergey Levine, and Trevor Darrell. 2018. Reinforcement Learning from Imperfect Demonstrations. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Workshop Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=HytbCQG8z

[10] Maor Gaon and Ronen I. Brafman. 2020. Reinforcement Learning with Non-Markovian Rewards. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*. AAAI Press, 3980–3987. https://ojs.aaai.org/index.php/AAAI/article/view/5814

[11] Vinicius G. Goecks, Gregory M. Gremillion, Vernon J. Lawhern, John Valasek, and Nicholas R. Waytowich. 2020. Integrating Behavior Cloning and Reinforcement Learning for Improved Performance in Dense and Sparse Reward Environments. In *Proceedings of the 19th International Conference on Autonomous Agents and Multiagent Systems, AAMAS '20, Auckland, New Zealand, May 9-13, 2020*. International Foundation for Autonomous Agents and Multiagent Systems, 465–473. https://doi.org/10.5555/3398761.3398819

[12] Claire Le Goues, Michael Pradel, Abhik Roychoudhury, and Satish Chandra. 2021. Automatic Program Repair. *IEEE Softw.* 38, 4 (2021), 22–27. https://doi.org/10.1109/MS.2021.3072577

[13] Mohammadhosein Hasanbeig, Natasha Yogananda Jeppu, Alessandro Abate, Tom Melham, and Daniel Kroening. 2021. DeepSynth: Automata Synthesis for Automatic Task Segmentation in Deep Reinforcement Learning. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021.* AAAI Press, 7647–7656. https://ojs.aaai.org/index.php/AAAI/article/view/16935

[14] Todd Hester, Matej Vecerík, Olivier Pietquin, Marc Lanctot, Tom Schaul, Bilal Piot, Dan Horgan, John Quan, Andrew Sendonaris, Ian Osband, Gabriel Dulac-Arnold, John P. Agapiou, Joel Z. Leibo, and Audrunas Gruslys. 2018. Deep Q-Learning From Demonstrations. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018.* AAAI Press, 3223–3230. https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16976

[15] Rodrigo Toro Icarte, Ethan Waldie, Toryn Q. Klassen, Richard Anthony Valenzano, Margarita P. Castro, and Sheila A. McIlraith. 2019. Learning Reward Machines for Partially Observable Reinforcement Learning. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 15497–15508. https://proceedings.neurips.cc/paper/2019/hash/532435c44bec236b471a47a88d63513d-Abstract.html

[16] Mingxuan Jing, Xiaojian Ma, Wenbing Huang, Fuchun Sun, Chao Yang, Bin Fang, and Huaping Liu. 2020. Reinforcement Learning from Imperfect Demonstrations under Soft Expert Guidance. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020.* AAAI Press, 5109–5116. https://ojs.aaai.org/index.php/AAAI/article/view/5953

[17] Bingyi Kang, Zequn Jie, and Jiashi Feng. 2018. Policy Optimization with Demonstrations. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018 (Proceedings of Machine Learning Research, Vol. 80).* PMLR, 2474–2483. http://proceedings.mlr.press/v80/kang18a.html

[18] Christian Kauten. 2018. Super Mario Bros for OpenAI Gym. GitHub. https://github.com/Kautenja/gym-super-mario-bros

[19] Ali Khalili and Armando Tacchella. 2014. Learning Nondeterministic Mealy Machines. In *ICGI 2014 (JMLR Workshop and Conference Proceedings, Vol. 34).* 109–123. http://proceedings.mlr.press/v34/khalili14a.html

[20] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings.* http://arxiv.org/abs/1412.6980

[21] Zhizhong Li and Derek Hoiem. 2018. Learning without Forgetting. *IEEE Trans. Pattern Anal. Mach. Intell.* 40, 12 (2018), 2935–2947. https://doi.org/10.1109/TPAMI.2017.2773081

[22] Yuteng Lu, Weidi Sun, and Meng Sun. 2022. Towards Mutation Testing of Reinforcement Learning systems. *J. Syst. Archit.* 131 (2022), 102701. https://doi.org/10.1016/j.sysarc.2022.102701

[23] Karl Meinke. 2018. Learning-Based Testing: Recent Progress and Future Prospects. In *Machine Learning for Dynamic Software Analysis: Potentials and Limits - International Dagstuhl Seminar 16172, Dagstuhl Castle, Germany, April 24-27, 2016, Revised Papers (Lecture Notes in Computer Science, Vol. 11026).* Springer, 53–73. https://doi.org/10.1007/978-3-319-96562-8_2

[24] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin A. Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. 2015. Human-level control through deep reinforcement learning. *Nat.* 518, 7540 (2015), 529–533. https://doi.org/10.1038/nature14236

[25] Tong Mu, Georgios Theocharous, David Arbour, and Emma Brunskill. 2022. Constraint Sampling Reinforcement Learning: Incorporating Expertise for Faster Learning. In *Thirty-Sixth AAAI Conference on Artificial Intelligence, AAAI 2022, Thirty-Fourth Conference on Innovative Applications of Artificial Intelligence, IAAI 2022, The Twelveth Symposium on Educational Advances in Artificial Intelligence, EAAI 2022 Virtual Event, February 22 - March 1, 2022.* AAAI Press, 7841–7849. https://ojs.aaai.org/index.php/AAAI/article/view/20753

[26] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program Repair via Semantic Analysis. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, David Notkin, Betty H. C. Cheng, and Klaus Pohl (Eds.). IEEE Computer Society, 772–781. https://doi.org/10.1109/ICSE.2013.6606623

[27] Augustus Odena, Catherine Olsson, David Andersen, and Ian Goodfellow. 2019. TensorFuzz: Debugging Neural Networks with Coverage-Guided Fuzzing. In *Proceedings of the 36th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 97)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, 4901–4911.

[28] Jing Peng and Ronald J. Williams. 1996. Incremental Multi-Step Q-Learning. *Mach. Learn.* 22, 1-3 (1996), 283–290. https://doi.org/10.1023/A:1018076709321

[29] Bilal Piot, Matthieu Geist, and Olivier Pietquin. 2014. Boosted Bellman Residual Minimization Handling Expert Demonstrations. In *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD 2014, Nancy, France, September 15-19, 2014. Proceedings, Part II (Lecture Notes in Computer Science, Vol. 8725).* Springer, 549–564. https://doi.org/10.1007/978-3-662-44851-9_35

[30] Aravind Rajeswaran, Vikash Kumar, Abhishek Gupta, Giulia Vezzani, John Schulman, Emanuel Todorov, and Sergey Levine. 2018. Learning Complex Dexterous Manipulation with Deep Reinforcement Learning and Demonstrations. In *Robotics: Science and Systems XIV, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, June 26-30, 2018.* https://doi.org/10.15607/RSS.2018.XIV.049

[31] Jorge Ramírez, Wen Yu, and Adolfo Perrusquía. 2022. Model-Free Reinforcement Learning from Expert Demonstrations: A Survey. *Artif. Intell. Rev.* 55, 4 (2022), 3213–3241. https://doi.org/10.1007/s10462-021-10085-1

[32] Marc Rigter, Bruno Lacerda, and Nick Hawes. 2020. A Framework for Learning From Demonstration With Minimal Human Effort. *IEEE Robotics Autom. Lett.* 5, 2 (2020), 2023–2030. https://doi.org/10.1109/LRA.2020.2970619

[33] Max Schwarzer, Nitarshan Rajkumar, Michael Noukhovitch, Ankesh Anand, Laurent Charlin, R. Devon Hjelm, Philip Bachman, and Aaron C. Courville. 2021. Pretraining Representations for Data-Efficient Reinforcement Learning. *CoRR* abs/2106.04799 (2021). arXiv:2106.04799 https://arxiv.org/abs/2106.04799

[34] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Vedavyas Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy P. Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. 2016. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nat.* 529, 7587 (2016), 484–489. https://doi.org/10.1038/nature16961

[35] Richard S. Sutton and Andrew G. Barto. 1998. *Reinforcement Learning - An Introduction.* MIT Press. https://www.worldcat.org/oclc/37293240

[36] Martin Tappler, Filip Cano Córdoba, Bernhard K. Aichernig, and Bettina Könighofer. 2022. Search-Based Testing of Reinforcement Learning. In *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI 2022, Vienna, Austria, 23-29 July 2022.* ijcai.org, 503–510. https://doi.org/10.24963/ijcai.2022/72

[37] Martin Tappler and Andrea Pferscher. 2023. Supplementary Material for "Learning and Repair of Deep Reinforcement Learning Policies from Fuzz-Testing Data". (August 2023). https://doi.org/10.6084/m9.figshare.22353712

[38] Matthew E. Taylor and Peter Stone. 2007. Cross-Domain Transfer for Reinforcement Learning. In *Machine Learning, Proceedings of the Twenty-Fourth International Conference (ICML 2007), Corvallis, Oregon, USA, June 20-24, 2007 (ACM International Conference Proceeding Series, Vol. 227)*, Zoubin Ghahramani (Ed.). ACM, 879–886. https://doi.org/10.1145/1273496.1273607

[39] Chen Tessler, Yonathan Efroni, and Shie Mannor. 2019. Action Robust Reinforcement Learning and Applications in Continuous Control. In *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA (Proceedings of Machine Learning Research, Vol. 97)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, 6215–6224. http://proceedings.mlr.press/v97/tessler19a.html

[40] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. DeepTest: Automated Testing of Deep-Neural-Network-Driven Autonomous Cars. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 303–314. https://doi.org/10.1145/3180155.3180220

[41] Hado van Hasselt, Arthur Guez, and David Silver. 2016. Deep Reinforcement Learning with Double Q-Learning. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA.* AAAI Press, 2094–2100. http://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/view/12389

[42] Ricardo Vinuesa, Hossein Azizpour, Iolanda Leite, Madeline Balaam, Virginia Dignum, Sami Domisch, Anna Felländer, Simone Daniela Langhans, Max Tegmark, and Francesco Fuso Nerini. 2020. The Role of Artificial Intelligence in Achieving the Sustainable Development Goals. *Nature communications* 11, 1 (2020), 233.

[43] Oriol Vinyals, Igor Babuschkin, Wojciech M. Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H. Choi, Richard Powell, Timo Ewalds, Petko Georgiev, Junhyuk Oh, Dan Horgan, Manuel Kroiss, Ivo Danihelka, Aja Huang, Laurent Sifre, Trevor Cai, John P. Agapiou, Max Jaderberg, Alexander Sasha Vezhnevets, Rémi Leblond, Tobias Pohlen, Valentin Dalibard, David Budden, Yury Sulsky, James Molloy, Tom Le Paine, Çaglar Gülçehre, Ziyu Wang, Tobias Pfaff, Yuhuai Wu, Roman Ring, Dani Yogatama, Dario Wünsch, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy P. Lillicrap, Koray Kavukcuoglu, Demis Hassabis, Chris Apps, and David Silver. 2019. Grandmaster Level in StarCraft

II using Multi-Agent Reinforcement Learning. *Nat.* 575, 7782 (2019), 350–354. https://doi.org/10.1038/s41586-019-1724-z

[44] Xiaofei Xie, Lei Ma, Felix Juefei-Xu, Minhui Xue, Hongxu Chen, Yang Liu, Jianjun Zhao, Bo Li, Jianxiong Yin, and Simon See. 2019. 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, ISSTA 2019, Beijing, China, July 15-19, 2019*, Dongmei Zhang and Anders Møller (Eds.). ACM, 146–157. https://doi.org/10.1145/3293882.3330579

[45] Zhe Xu, Ivan Gavran, Yousef Ahmad, Rupak Majumdar, Daniel Neider, Ufuk Topcu, and Bo Wu. 2020. Joint Inference of Reward Machines and Policies for Reinforcement Learning. In *Proceedings of the Thirtieth International Conference on Automated Planning and Scheduling, Nancy, France, October 26-30, 2020*. AAAI Press, 590–598. https://ojs.aaai.org/index.php/ICAPS/article/view/6756

[46] Yang Yu. 2018. Towards Sample Efficient Reinforcement Learning. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, Jérôme Lang (Ed.). ijcai.org, 5739–5743. https://doi.org/10.24963/ijcai.2018/820

[47] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. 2021. The Fuzzing Book. https://www.fuzzingbook.org/. accessed: 2023, August 31.

[48] Zhuangdi Zhu, Kaixiang Lin, Bo Dai, and Jiayu Zhou. 2022. Self-Adaptive Imitation Learning: Learning Tasks with Delayed Rewards from Sub-Optimal Demonstrations. In *Thirty-Sixth AAAI Conference on Artificial Intelligence, AAAI 2022, Thirty-Fourth Conference on Innovative Applications of Artificial Intelligence, IAAI 2022, The Twelveth Symposium on Educational Advances in Artificial Intelligence, EAAI 2022 Virtual Event, February 22 - March 1, 2022*. AAAI Press, 9269–9277. https://ojs.aaai.org/index.php/AAAI/article/view/20914

[49] Amirhossein Zolfagharian, Manel Abdellatif, Lionel C. Briand, Mojtaba Bagherzadeh, and Ramesh S. 2022. Search-Based Testing Approach for Deep Reinforcement Learning Agents. *CoRR* abs/2206.07813 (2022). https://doi.org/10.48550/arXiv.2206.07813 arXiv:2206.07813