2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)

μPRL: A Mutation Testing Pipeline for Deep Reinforcement Learning based on Real Faults

Deepak-George Thomas *, Matteo Biagiola †, Nargiz Humbatova †, Mohammad Wardat ‡, Gunel Jahangirova §, Hridesh Rajan ¶, Paolo Tonella †

* Dept. of Computer Science, Iowa State University, Iowa, USA, dgthomas@iastate.edu
† Università della Svizzera italiana, Lugano, Switzerland, {matteo.biagiola, nargiz.humbatova, paolo.tonella}@usi.ch
‡ Dept. of Computer Science and Engineering, Oakland University, Michigan, USA, wardat@oakland.edu
§ King's College London, London, United Kingdom, gunel.jahangirova@kcl.ac.uk
¶ School of Science and Engineering, Tulane University, Louisiana, USA, hrajan@tulane.edu

***Abstract*—Reinforcement Learning (RL) is increasingly adopted to train agents that can deal with complex sequential tasks, such as driving an autonomous vehicle or controlling a humanoid robot. Correspondingly, novel approaches are needed to ensure that RL agents have been tested adequately before going to production. Among them, mutation testing is quite promising, especially under the assumption that the injected faults (mutations) mimic the real ones.**

In this paper, we first describe a taxonomy of real RL faults obtained by repository mining. Then, we present the mutation operators derived from such real faults and implemented in the tool μPRL. Finally, we discuss the experimental results, showing that μPRL is effective at discriminating strong from weak test generators, hence providing useful feedback to developers about the adequacy of the generated test scenarios.

***Index Terms*—reinforcement learning, mutation testing, real faults**

I. Introduction

Reinforcement Learning (RL) is being applied to various safety-critical systems such as traffic control, drone navigation, and power grids [1–3]. Due to its relevance in such systems, RL developers need to make sure that their RL agent is tested thoroughly. Mutation testing [4, 5] is a powerful technique to ensure that the test set exercises the system in an adequate way, but existing attempts to apply mutation testing to RL [6, 7] are limited, and do not cover the full spectrum of faults that may affect an RL agent. In this paper, we construct a comprehensive taxonomy of real faults identified by repository mining (we analysed 2,787 posts, 3.6 times more than previous work [8]) and we develop an RL mutation tool, named μPRL, which implements new mutation operators (MOs) mimicking the real faults in the taxonomy.

Our taxonomy of real RL faults targets RL developers who use well-known frameworks, such as StableBaselines3 [9] and OpenAI Gym [10], for their projects. We mined StackExchange and GitHub posts and then manually analysed the relevant artifacts to identify real bugs that developers experience. Then, we derived 15 mutation operators from the taxonomy and implemented 8 of them in the tool μPRL. These operators have been evaluated on four environments provided by OpenAI Gym [10] and HighwayEnv [11]: CartPole, Parking, LunarLander, and Humanoid [11–14]. The Humanoid environment is a particularly challenging robotics environment with a high dimensional observation space, thereby requiring extensive computational resources. In the evaluation, we applied μPRL to environments with both discrete and continuous action spaces, and we considered popular Deep RL (DRL) algorithms, including DQN, PPO, SAC, and TQC [15–18].

Experimental results indicate that our mutation tool μPRL is useful in discriminating strong from weak test scenario generators and achieves a high sensitivity to the test set quality, substantially higher than that of the state of the art tool RLMutation [7]. We also show that the new fault types introduced in our taxonomy are major contributors to the increased sensitivity of our mutation operators.

II. Related Work

We organise the related works into those analysing Deep Learning (DL)/RL faults and those mutating DL/RL models.

A. Fault Classification

DL Faults: Humbatova *et al.* [19] proposed a DL faults taxonomy using StackOverflow and GitHub artifacts. Islam *et al.* [20] studied the frequency, root cause, impact and pattern of bugs while using deep learning software.

RL Faults: Nikanjam *et al.* [8] developed a fault taxonomy for RL programs. They studied 761 RL artifacts obtained from StackOverflow and GitHub. While analysing GitHub they went over issues from the following frameworks - OpenAI Gym, Dopamine, Keras-rl and Tensorforce.

Their work focuses on mistakes developers make while writing an RL algorithm from scratch. However, RL algorithms are notoriously hard to implement [21], and even small implementation details have a dramatic effect on performance [22, 23]. Our work considers the perspective of software developers who use RL as a tool to address an engineering problem. Therefore, we focus on bugs that arise while using reliable RL implementations [8–10, 24–26] (or bugs that can be mapped

1558-1225/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSE55347.2025.00036

to those occurring in reliable implementations) . We compare with the taxonomy of Nikanjam *et al.* [8] in Section III-D.

Andrychowicz *et al.* [27] studied the effect of more than 50 design choices on an RL agent's performance by training around $250k$ agents. They used this study to recommend various hyperparameter choices.

B. Mutation Testing for AI-based systems

Mutation testing for DL: DeepMutation [28] and MuNN [29], were the pioneers in recognising the need for mutation operators specifically tailored to DL systems. Subsequently, DeepMutation was extended to a tool called DeepMutation++ [30], focusing on operators that can be applied to the weights or structure of an already trained model. Jahangirova and Tonella [31] performed an extensive empirical evaluation of the mutation operators proposed in DeepMutation++ and MuNN. DeepCrime [32] differs from DeepMutation++ in that it uses a set of mutation operators derived from *real faults* in DL systems [19]. Such operators are applied to a DL model before training.

Mutation testing for RL: There are currently two papers dedicated to mutation testing of RL systems. Lu *et al.* [6] introduce a set of RL-specific mutation operators, including *element-level* mutation operators and *agent-level* mutation operators. The element-level mutations consider the injection of perturbations into the states and rewards of an agent. Agent-level mutations introduce errors into an already trained agent. If a trained agent's policy is represented by a Q-table [33], an agent-level mutation would fuzz the state-action value estimations stored in the table. If the policy of a trained agent is implemented by a neural network, an agent-level mutation would remove a neuron from the input or output layer of the network. In addition, the authors propose operators that affect the exploration-exploitation balance by, for instance, mutating the exploration rate of the agent during training.

However, such mutation operators are not based on any real-world fault taxonomy or existing literature. Moreover, mutation killing is computed on a single repetition of the experiment, not accounting for randomness in the training process. Previous literature shows that RL algorithms are sensitive to the random seed [21], suggesting that statistical evaluations are needed to draw reliable conclusions [34].

Tambon *et al.* [7] explore the use of higher order mutants, i.e., the subsequent application of different mutations to a program under test, in the context of RL. The mutation operators they propose, implemented in the tool RLMutation, are based on a number of sources. As a basis, the authors have adopted the existing operators for RL [6] and DL systems [29, 30, 32] and complemented them with operators extracted from the state-of-the-art taxonomies of RL faults [35] and real DL faults [19]. They divided the obtained operators into three broad categories: *environment-level*, *agent-level*, and *policy-level* operators. Mutations at the *environment-level* include faults related to the observations the agent perceives in the environment, for instance due to faulty sensors or deliberate attacks. Operators at the *agent-level* stem from the faults that developers make while implementing RL agorithms, such as omitting the terminal state or selecting a wrong loss function. Finally, *policy-level* mutations focus on the agent's policy network, mutating activation functions or number of layers. These mutations are used to create first-order mutants. Among them, those that are not trivial, i.e., that are not killed by all the test environments, are considered for higher-order mutation.

Our mutation tool μPRL differs substantially from both Lu *et al.* [6] and Tambon *et al.* [7], because it is grounded on a novel taxonomy of real faults experienced by developers that rely on existing, mature frameworks for the creation of RL agents. This rules out the syntactic state/reward/policy perturbations introduced by Lu *et al.* [6] and the mistakes made by programmers when implementing RL algorithms from scratch that are instead considered by Tambon *et al.* [7]. Since RLMutation [7] includes also real faults that may occur when developers rely on existing RL frameworks, we conduct a detailed comparison with the existing taxonomy and tool in Section III-D and Section V-C, respectively.

III. Taxonomy of real RL faults

We constructed a taxonomy of real RL faults in a bottom-up way, starting from the collection of artefacts, obtained through software repository mining. We then labeled such artefacts, to eventually organise the labels into a taxonomy.

A. Mining of Software Artefacts

In our preliminary investigation, we observed that most discussions about faults reported by developers implementing an RL agent happen in Stack Exchange. We also noticed several commit messages about RL faults in GitHub. Hence, we mined these two repositories.

1) Mining GitHub: The foremost challenge while mining GitHub repositories was identifying popular RL frameworks. Nikanjam *et al.* used Keras-rl, Dopamine, Tensorforce, and OpenAI Gym [8]. However, OpenAI Gym is only used to simulate the interactions between an agent and the environment, and does not provide RL algorithms. While investigating the remaining frameworks we found that Tensorforce is no longer maintained and Keras-rl did not get updated since November 2019 [10, 24–26]. Therefore, to identify popular RL frameworks we checked top-tier Machine Learning and Software Engineering Conferences, such as ICML, ICSE, ESEC/FSE, and ASE. We manually inspected 89 papers, dropping all papers that focused on model-based RL, inverse RL, and multi-agent RL. This filtering step left us with 9 papers from SE conferences and 47 papers from ICML-22 that provided actionable insights. While many were custom implementations of RL algorithms, the majority of the papers that used frameworks used StableBaselines3 [9] (7 overall).

We followed Humbatova *et al.*'s [19] approach to mine GitHub repositories. We searched for files containing the string "stable_baselines3" using the GitHub Search API, and found 27,413 files. Since the API has a limit of $1k$ results per query, we searched for file sizes between 0 and $500k$ bytes, with a step size of 250 [19]. We identified 4,272 repositories

corresponding to these files. We then dropped all repositories that had less than 10 stars, 100 commits, 10 forks, and 5 contributors, which left us with 67 repositories. As the next step, we manually verified whether they were related to RL and dropped the repositories containing tutorials and code examples using StableBaselines3, obtaining the list of the final 43 repositories. These 43 repositories were then used to extract issues, pull requests (PRs) and commit messages. While extracting the issues, we only extracted those that contained the label "bug", "defect" or "error" in them. Following Islam *et al.* [20] we only selected commits that contain the term "fix".

To automate the process of extracting relevant artifacts from GitHub, we followed Humbatova *et al.* [19]: we combined all issues, PR titles, descriptions, and comments along with commit messages into a text dump. We did data-cleaning on the words within the text dump (dropped stop words, non-word characters, etc) and counted the frequency of each remaining word. Words that had a frequency lower than 20 (raised from 10 [19], to obtain a manageable list of words) were dropped, resulting in a list of 14,921 words. We divided the final list of words among 3 authors to identify relevant RL words and obtained 118. We then selected the corresponding issues, PRs, and commits, a total of 1,120 [19, 20].

2) Mining Stack Exchange: To include questions on Data Science and Artificial Intelligence, which might be relevant for our taxonomy, we mined both StackOverflow (SO) and Stack Exchange's (SE) Data Science (DS) and Artificial Intelligence (AI) Q&A websites (with SO falling under the umbrella of SE).

TABLE I: Number of unique tags and posts in SE and SO

	# Tags (All)	# Tags (RL)	# Posts
Artificial Intelligence (AI-SE)	974	104	783
Data Science (DS-SE)	668	9	245
StackOverflow (SO-SE)	63,653	19	3,682
Total	65,295	132	4,710

We used SE's Data Explorer to extract posts from SO and SE. We first extracted all tags from these websites; then, we filtered all tags without the term "reinforcement" in their respective name, excerpt (i.e., short description), or wikibody (i.e., detailed description). The resulting 132 tags were then used to select all posts from SE. Next, we excluded all posts without an accepted answer [19]. We also filtered out the posts whose title contained the terms "how", "install" or "build", to discard how-to questions. During manual inspection, we found that many posts were not RL specific, but rather related to Machine Learning in general. Therefore, we dropped all posts that had only one tag and it was "machine-learning". Following this procedure, we obtained 4,710 posts (see Table I).

Given the large number of posts, we performed various pilot studies to gauge the data quality of selected samples, and manually filtered out irrelevant posts. These pilot studies yielded a large number of false positives. Upon a closer examination of the dataset, we found that the posts from SO contained around 78% of the total posts and the tag "artifical-intelligence" was present in 2,779 SO posts (without dropping duplicates). A big chunk of the posts within this category contained posts concerning classical AI, such as the A* algorithm. Therefore, we dropped all posts that either contained the tag "artificial-intelligence" or a combination of "artificial-intelligence" and "machine-learning", without another RL tag. Lastly, following Islam *et al.* [20], we kept only posts that contained code. This brought our final dataset size down to 1,667.

3) Manual Labelling: One group of labellers manually analysed the artifacts and dropped false positives. Five authors participated in the labeling process for taxonomy construction. Each post in the dataset was randomly assigned to two authors. We used the procedure by Humbatova *et al.* [19] for the labeling process. The authors therein, developed a tool that helped them manage the labels. This tool allowed each assessor to pick a label generated by their colleagues. In case none of the labels matched the post description, they created their own label and added it to the tool, which then became accessible to others. We pre-loaded all labels created by Nikanjam *et al.* [8] into the labeling tool, to be consistent with, and build upon, the existing RL fault taxonomy [8]. Furthermore, to check the disagreement between various participants, we measured Krippendorf's Alpha [36, 37], which handles more than two raters, with each rater only labeling a subset of the posts. Krippendorf proposed to reject data where the confidence interval of the reliability falls below 0.667. Ideally the value of alpha should be 1.00 but variables with values greater than 0.800 could be relied upon [37, 38]. During labeling, the two raters of each post met together to resolve conflicts, when any such conflict arouse. When no resolution between two raters could be reached for a certain post, the overall group made the final decision through voting [19]. While the average agreement (Krippendorf's Alpha) was 0.546 before the consensus meeting, it raised to 0.926 after the meeting.

B. Taxonomy Construction

We followed Islam *et al.*'s [20] approach to build the taxonomy tree, wherein we built our tree on top of another existing RL fault tree by Nikanjam *et al.*'s [8] (marked in orange/blue in Figure 1). For new labels (marked in green in Figure 1), we followed Humbatova *et al.*'s [19] approach to group them into higher-level categories.

C. The Final Taxonomy

Reward Function. This category is related to faults affecting the reward function guiding the RL agent towards the learning objective.

❶ *Defining the reward function* - Designing the reward function is critical to achieve good performance in RL. We found the following faults associated with it. *Suboptimal reward function* – Defining a good reward function is challenging, especially for complex tasks involving multiple constraints. For instance, learning a robust policy for quadrupedal robots, requires a complex reward function encouraging linear and angular velocities, while penalising vibrations and energy consumption [39]. Manually setting weights for these components

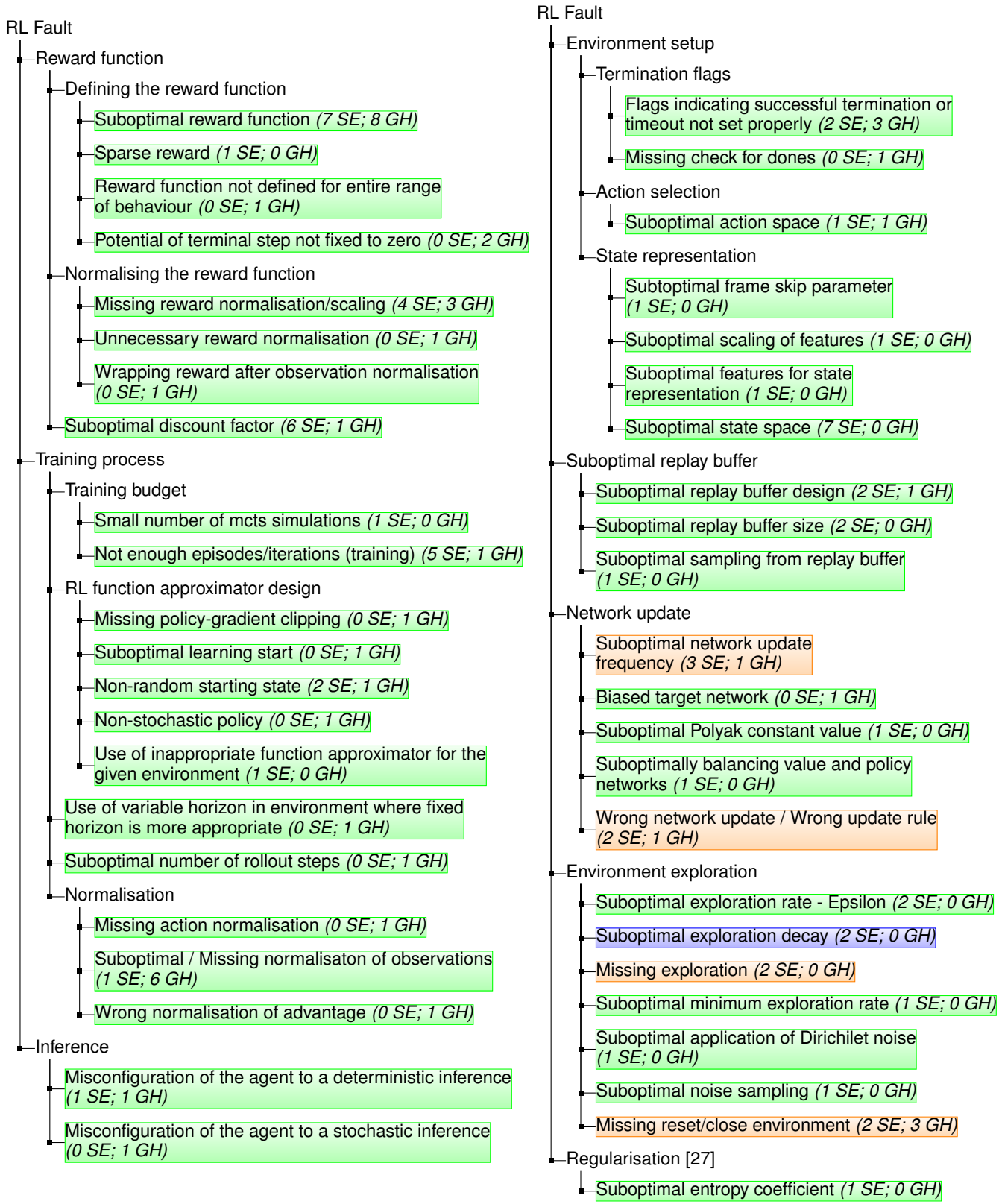

Fig. 1: Taxonomy of real RL faults: green indicates new fault types; orange and blue indicate fault types in common with the previous taxonomy [8]; blue indicates the ones that we renamed. SE/GH are preceded by the number of instances found in StackExchange/Github.

is not an easy task for developers [40], while they may greatly impact the learning effectiveness. *Sparse Reward* – This deals with cases where the reward is provided on rare occasions, for instance, at the end of each episode rather than at each timestep. In some environments, a sparse reward makes training ineffective or even impossible [41]. *Reward function not defined for the entire range of behavior* – This fault occurs when the reward function does not account for all of the trajectories that the agent might take. *Potential of terminal step not fixed to zero* – This fault is related to the process of reward shaping, wherein the agent is provided with supplemental rewards, to make the learning smoother. When the shaping function is based on a potential, the value of such potential should be zero at the terminal step.

❷ *Normalising the reward function* – A class of faults associated with reward functions is related to normalisation. *Missing reward normalisation/scaling* – Reward scaling typically involves taking the product of the environment rewards with a scalar ($\hat{r} = r\hat{\sigma}$) [42, 43]. In certain environments clipping is an alternative to scaling. Existing studies [21] show that the choice of reward scaling/clipping has a large impact on the output of the training process. *Unnecessary reward normalisation* – This fault occurs when reward function normalisation is not required and its use is actually detrimental to training. *Wrapping reward after observation normalisation* – In this fault, observations are normalised before a wrapper is applied to the reward function, making the wrapper suboptimal. *Suboptimal discount factor* – The discount factor γ is a critical parameter used to trade off future and immediate rewards. When γ is close to 0, the agent focuses on actions that maximise the short-term reward, whereas when γ is close to 1, the agent privileges actions that maximise future rewards.

Training Process. This category consists of faults that affect the training process of the RL agent.

❸ *Training budget* – This category concerns faults related to the number of iterations used to train the RL agent. *Small number of mcts simulations* – This fault was observed in the context of the AlphaGoZero algorithm [44]. This algorithm uses Monte Carlo Tree Search (MCTS) to learn optimal actions. Having a low number of MCTS simulations may lead to suboptimal actions being selected [44–46]. *Not enough episodes/iterations (training)* – This fault occurs when the number of training iterations for the RL algorithm is low. This prevents the RL algorithm from learning a good policy.

❹ *RL function approximator design* – A critical element in RL is the function approximator learnt during training. This category consists of faults that occur due to the design choices related to the selection of the function approximator. *Missing policy-gradient clipping* – Incorporating policy-gradient clipping improves the performance of actor-critic algorithms [27]. *Suboptimal learning starts* – When the training process starts, the agent is allowed to take a series of random actions without learning. The "learning starts" hyperparameter is a critical parameter that controls when the agent starts learning, after the training process has begun. *Non-random starting state* – Starting at the same state each time the agent is reset, prevents it from exploring the surrounding states and leads to overfitting. *Non-stochastic policy* – Certain RL algorithms require a stochastic policy and therefore the function approximator must be stochastic in nature. Implementing a deterministic function approximator could lead to a drop in performance. *Use of inappropriate function approximator for the given environment* – RL problems of different sizes, in terms of state and action spaces, require different approximation techniques. Relatively smaller problems might be solved using tabular methods whereas larger problems might require linear or non-linear function approximators (such as neural networks) [33].

Use of variable horizon in environment where fixed horizon is more appropriate – This fault occurs when reward learning is adopted during RL training. For effective reward learning (e.g., from human preferences), a fixed episode length was found to be often a better choice [47].

Suboptimal number of rollout steps – This fault occurs in the context of on-policy algorithms and refers to the number of rollout steps per environment used to update the policy. This hyperparameter significantly affects the algorithm's performance [27].

❺ *Normalisation* – This category is related to normalisation in the context of training. *Missing action normalisation* – Action normalisation has been found to be helpful, especially when the actions are continuous [48]. *Suboptimal / Missing normalisaton of observations* – Andrychowicz *et al.* [27] recommend to always normalise observations. As per their experiments, doing this was critical for achieving high performance in almost all the environments. *Wrong normalisation of advantage* – RL algorithms such as PPO [16] and A3C [49], compute the advantage function, i.e., an estimate of the value of a certain action in a given observation. Normalising this estimate with a single sample or a few samples may result in diverging computations (NaN), and to gradients with large variances.

Inference. This category deals with faults that occur at inference time, i.e., after the RL algorithm has been trained.

Misconfiguration of the agent to a deterministic inference – During inference, forcing an agent to take deterministic actions when it was trained with a stochastic policy, might lead to a drop in performance [33]. In fact, for problems where appreciable generalisation is required at inference time (e.g., when there is a substantial development-to-production shift), a better policy may be a stochastic one. *Misconfiguration of the agent to a stochastic inference* – Forcing an agent that was trained with a deterministic policy to become stochastic at inference time, thereby carrying out exploratory behavior, can also lead to a performance drop.

Environment setup. Faults related to the environment setup fall under this category.

❻ *Termination flags* – Flags that denote when an episode has ended might be set incorrectly. *Flags indicating successful termination or timeout not set properly* – Flags for termination and timeout should only be set in terminal states. Terminal states are critical for calculating state values, and these values are recursively used to compute the values for previous states.

Missing check for dones – During training the algorithm needs to correctly check whether a state is terminal (i.e., *done*), as this determines how the targets for the optimisation problem are computed.

❼ *Action selection* – This fault occurs when the user defines an action space that makes learning difficult or impossible (e.g., representing actions as discrete integers vs bit vectors).

❽ *State representation* – This category deals with faults associated with the definition of environment states. *Suboptimal frame skip parameter* – The frame skip parameter forces an action to be repeated for a specific number of frames. This parameter was found to have a significant impact, in terms of learning efficiency, in environments requiring high-frequency policies, such as Atari games and robotic applications [50, 51]. *Suboptimal scaling of features* – State features must be scaled appropriately, in order for an RL algorithm to learn efficiently. *Suboptimal features for state representation* – In order to speed up learning, rather than feed in raw state inputs and expect the learning algorithm to identify useful patterns, developers could use their domain knowledge and engineer the state to include relevant, possibly higher level, features. *Suboptimal state space* – The RL paradigm assumes that the environment the agent operates in, follows the Markov property, i.e., that the current state the agent perceives, summarises all the past interactions of the agent with the environment. In other words, all the information the agent needs to make optimal actions, need to be in the state space of the agent. If some crucial information are hidden from the agent, the Markov property does not hold, and the agent cannot learn optimally [52].

Suboptimal replay buffer. Off-policy RL algorithms typically use a replay buffer during training.

Suboptimal replay buffer design – Catastrophic forgetting [53] might be caused by the under-representation of data for specific tasks in the replay buffer. This fault is common in multi-tasks RL settings [54], i.e., when the RL agent has multiple objectives, but also in single environments that can be decomposed in sub-objectives (or levels) [55]. *Suboptimal replay buffer size* – The replay buffer size is a non-trivial tunable hyperparameter. While a smaller replay buffer may lead to relevant data getting replaced too quickly, a large buffer might lead to older and irrelevant data getting sampled, reducing learning efficiency [56]. *Suboptimal sampling from replay buffer* – It is crucial that the sampling process maintains the i.i.d. (identical and independently distributed) property of the data, and that the sampled data are not temporally correlated. If this property does not hold, the learning process might be negatively affected.

Network update. This category refers to updating the parameters of the neural networks implementing the function approximators.

Suboptimal network update frequency – The frequency of the target network updates is too low/high, impacting the learning effectiveness [8]. *Biased target network* – This fault occurs when the target network parameters are not independent from the online network's. The target network prevents the policy from exploring alternative solutions while the online network is being updated [15]. This fault prevents the target network from converging to the optimal one. *Suboptimal Polyak constant value* – An alternative to duplicating the online network weights as target network weights, is to perform soft updates by Polyak averaging. The critical hyperparameter controlling such soft updates is the Polyak update coefficient [15, 57]. *Suboptimally balancing value and policy networks* – This fault occurs while using the AlphaGo algorithm [58]. This algorithm uses MCTS to select actions by utilising value and policy networks. The parameter λ balances the decisions of these two networks. A fault occurs when a poor value of the hyperparameter λ is set [45, 46, 58]. *Wrong network update / Wrong update rule* – New data cannot be optimally learned by the RL algorithm (e.g., because the learning rate of the neural networks decays too quickly) [8].

Environment Exploration. We found a variety of critical exploration hyperparameters in various RL algorithms. Exploring too little may cause the RL algorithm to be unable to discover high reward states and actions; exploring too much prevents the agent from exploiting what it has learned.

Suboptimal exploration rate - Epsilon – This hyperparameter refers to the suboptimal setting of the epsilon parameter, present in various RL algorithms [15]. *Suboptimal exploration decay* – During the start of RL training, the algorithm is expected to explore states extensively, to identify promising states and actions. However, as the algorithm progresses, the amount of exploration is typically reduced so that the agent can exploit its existing knowledge. *Missing exploration* – This label refers to the case where the agent does not explore at all [8]. *Suboptimal minimum exploration rate* – Once the exploration parameter has been completely decayed, it should be left to a value that is still greater than zero. This ensures that the agent continues to explore for the remaining training budget. However, too large values will interfere with learning, while a value that's too low will prevent experiencing new states and actions. *Suboptimal application of Dirichilet noise* – Dirichilet noise is used by the AlphaGo algorithm for exploration. The noise sampled from the Dirichilet distribution, which requires careful setting, is added to the root node's prior probabilities [44]. *Suboptimal noise sampling* – This fault was found in the usage of the Deep Deterministic Policy Gradient (DDPG) algorithm. DDPG incorporates exploration during training by adding noise to actions. The choice of the distribution to sample the noise affects the exploration efficacy [57]. *Missing reset/close environment* – This fault deals with forgetting to reset or to close the environment during training or inference [8].

Regularisation – Policy regularisation improves the performance of RL algorithms [59].
Suboptimal entropy coefficient – The Asynchronous Actor Critic and PPO algorithms [16, 49] incorporate the policy's entropy to the loss function, to improve exploration. Therefore the entropy coefficient hyperparameter becomes critical to control the exploration rate of the agent.

D. Comparing Prior Work with our Taxonomy

Nikanjam *et al.* [8]'s final taxonomy has 11 fault categories. Our taxonomy contains five of these categories (with orange background in Figure 1), plus one which we renamed (with blue background in Figure 1). The remaining five categories do not match any category in our taxonomy for at least one of the following reasons: (1) the fault could not be mapped to an RL framework, i.e., it only affects re-implementations of RL algorithms; (2) the fault is not RL-specific, e.g., the fault is a generic DL fault; (3) there is no evidence for the fault in our mined posts, e.g., the associated posts contain a how-to question, instead of describing actual issues and discussing possible solutions; (4) the fault is a generic coding error; (5) the associated post does not refer to any code implementing the RL agent. For the matching fault types, we used the same labels as Nikanjam *et al.* [8], except for "Suboptimal exploration rate", which we renamed to "Suboptimal exploration decay", specifying more precisely that the fault is related to how fast/slow the exploration rate is decayed over time.

IV. Mutation Analysis

A. Mutation Operators

TABLE II: List of proposed mutation operators in μPRL.

Group	Operator	ID	IS
Reward function	Change Discount Factor	SDF	Y
	Make Reward Sparse	SPR	N
	Change Reward Scale	SRS	N
Training process	Change Number of Rollout Steps	SNR	Y
	Change Learning Start	SLS	Y
	Reduce Episodes/Iterations	NEI	Y
	Introduce Deterministic Start State	NRS	N
	Remove Normalisation of Actions	MNA	N
	Remove Normalisation of Observations	MNO	N
Regularisation	Change Entropy Coefficient	SEC	Y
Network update	Change Network Update Frequency	SNU	Y
	Change Polyak Constant Value	SPV	Y
Suboptimal Replay Buffer	Change Replay Buffer Size	SBS	N
Environment exploration	Change Minimum Exploration Rate	SMR	Y
	Change Exploration Rate	SER	N

To define a set of mutation operators, we analysed all of the 48 unique fault types in the RL taxonomy of real faults (see Section III-C). The extraction of mutation operators was organised into three stages. First, two of the authors independently went through the whole list of faults types and each derived potential mutation operators (MOs) from the inspected faults. Then, they have performed conflict resolution between themselves, and produced a list of proposed operators. At the next step, two other authors have separately inspected the set of candidate MOs and the faults that did not inspire any MO. Both of the authors have shown full agreement with the initial list of the operators, i.e. have not proposed any new MOs or rejected the existing ones. At the last stage, two authors, one from each stage, have gone through the list of MOs to document their feasibility and applicability scenarios. The MO extraction process, as well as the comments on the possible implementation approaches, are available in our replication package [60]. In total, we propose 15 mutation operators, with 8 of them implemented in our tool μPRL.

Table II enlists the final set of proposed MOs, which are divided according to the corresponding top category of the taxonomy (Column 1). Column 2 provides a short description of each MO, while Column 3 specifies a short abbreviated name. The last column "IS", which stands for "Implementation Status", shows whether an operator is implemented or not. The operators that are domain specific, i.e., that require a custom implementation for each case study, have not been implemented, as they are not generally applicable. For instance, the "Make Reward Sparse" operator requires knowledge of how the reward function is implemented in a given environment, while the "Missing Normalisation" operators are not applicable in environments where actions are discrete or observations are not normalised by default.

In total, the operators span six out of the eight top categories of the taxonomy. "Training process" is the most populated category with six of the proposed operators. Most of the operators stem from one fault type in the taxonomy, with the name of the MO corresponding to the name of the taxonomy leaf. "Change reward scale" is an exception to this rule as it corresponds both to the "Missing reward normalisation/scaling" and "Unnecessary reward normalisation" fault types.

B. Mutation Analysis Procedure

To ensure reliable and statistically sound evaluation of the quality of test sets, we adopt the definition of *statistical killing* [31]: a mutant is *killed* by a test set if the prediction accuracies of original and mutated model computed on such test set differ in a statistically significant way.

However, RL presents numerous differences w.r.t. DL models that we need to account for when evaluating an RL agent. In RL, since the agent is trained online, a test can be represented as an initial configuration of the environment where the agent operates [61, 62]. Let us consider the *CartPole* subject environment, consisting of a cart moving to keep a pole vertically aligned (this is the classical inverted pendulum problem). Its initial configuration e is a 4-dim vector $e = [x, \dot{x}, \theta, \dot{\theta}]$, where x is the initial position of the cart, $\dot{x}$ is the initial velocity of the cart, θ is the initial angle of the pole w.r.t. the vertical axis, and $\dot{\theta}$ is the initial angular velocity of the pole. Trained RL agents are typically evaluated using a set of randomly generated initial environment configurations [63], to test their generalisation capabilities. As there is no explicit test set to evaluate RL agents in a given environment, we refer to a test environment generator (or test generator *TG* for short), rather than a test set; a random *TG*, which randomly generates initial environment configurations, is one example of *TG*.

Let A be the RL agent under test. To support statistical analysis of mutation killing [31], we train n instances of A for N time steps each. For each instance, an arbitrary random seed is chosen for the generation of a reproducible sequence of *initial environment configurations* (environment

configurations, for short), which are used to train the agent, within the N time steps budget. Then, for any given mutation operator P and its configuration $P(j)$, $j \in J_P$, we train n mutant instances by reusing the same set of random seeds, and, as a result, the same sequence of initial environment configurations used to train the original agent instances. In this way, we create n pairs of original and mutant instances (o_i, m_i) that are trained on the same sequence (or sequence prefix, if a shorter sequence is used) of initial environments.

1) Killability: We first define the *killed* predicate for an MO parameter configuration $P(j)$ and then we use it to define the notion of *killable* (and its complement, *likely-equivalent*) MO configuration $P(j)$.

To decide whether a mutant is killed by a test generator *TG*, we execute each pair (o_i, m_i) of original and mutant agents in test mode on the sequences of environment configurations generated by *TG*. Since *TG* might generate different sequences depending on the agent under test, with no loss of generality we assume that two different test sequences $T_{o_i} = TG(o_i)$ and $T_{m_i} = TG(m_i)$ are obtained when applying *TG* to either the original agent o_i or the mutant m_i. When such a dependency does not hold (i.e., the dependency between *TG* and the agent under test), there is a single test sequence $T = T_{o_i} = T_{m_i}$. This happens for instance when using a random *TG* or a predefined sequence of environment configurations T as test set. It is also convenient to assume that the two test sequences have the same length $L = |T_{o_i}| = |T_{m_i}|$.

We represent the result of the execution of each pair (o_i, m_i) on the corresponding sequences (T_{o_i}, T_{m_i}) as a 4-tuple $(S_{o_i}, F_{o_i}, S_{m_i}, F_{m_i})$, where S_{o_i} and F_{o_i} are the number of successes and failures for the i-th original agent instance o_i, with $S_{o_i} + F_{o_i} = L$; the variables S_{m_i} and F_{m_i} store these measurements for the paired i-th mutant instance m_i, with $S_{m_i} + F_{m_i} = L$. Given the contingency table $[[S_{o_i}, F_{o_i}], [S_{m_i}, F_{m_i}]]$ for each pair (o_i, m_i) we apply the Fisher's test [64] to decide whether the mutant instance m_i is killed or not, the *killed* predicate K being defined as: $K(o_i, m_i) = 1 \Leftrightarrow p_{value} < 0.05$. Note that mutating the original agent may result in a mutant that improves over the performance of the original (*weaker*) agent [65]; in this case a certain pair can be killed because $F_{o'_i} > F_{m'_i}$, i.e., the original instance o'_i fails more often than the mutated instance m'_i. All such pairs (o'_i, m'_i) are discarded and for them the killed predicate is conventionally set to zero, i.e., $K(o'_i, m'_i) = 0$.

The killed predicate K of a given mutant configuration $P(j)$ is calculated for a given test generator *TG* based on the number of killed instance pairs over the total number of pairs $n - w$, where w is the number of pairs where the original instance is weaker (i.e., fails more often) than its mutated instance pair:

$$K(TG, P, j) = \begin{cases} \textit{1 (true)} & \text{if } K_R(TG, P, j) \geq 0.5 \\ \textit{0 (false)} & \text{otherwise} \end{cases}$$

where $K_R(TG, P, j) = 1/(n - w) \cdot \sum_{i=1}^{n} K(o_i, m_i)$ is the *killing rate* (i.e., proportion of killed mutant instances). A certain mutant configuration $P(j)$ is *killed* if at least half of its pairs is killed, excluding those pairs where the original instance is weaker than the mutant instance.

The notion of *likely-equivalent* (and its complement, *killable*) mutant that we use in our mutation procedure, is based on the one proposed in DeepCrime [32]: a mutant is *likely-equivalent* if the training data cannot capture the differences between the original and mutant model. Hence, to decide whether the MO parameter configuration $P(j)$ is *killable* we check if it is killed by the training data. Specifically, we set $TG = TRS_{o_i}$, i.e., we replay both o_i and m_i on the set of training environment configurations TRS_{o_i} and compute the killing predicate $K(TRS_{o_i}, P, j)$. As TRS_{o_i} was used to train o_i and at least a prefix of TRS_{o_i} was used to train m_i, we expect TRS_{o_i} to be highly discriminative between original and mutant [32].

A mutation operator P is *killable* if at least one of its mutant configurations $P(j)$ is killable. If a mutation operator is not killable, it is deemed likely-equivalent and discarded.

2) Triviality: We are also interested in checking whether a certain MO generates mutants that are trivial to kill. To evaluate triviality of each mutant, we reuse the results of the replay of each original agent and mutant pair (o_i, m_i) on their set of training environment configurations TRS_{o_i} from killability analysis. From TRS_{o_i}, we select the subset of environment configurations where the original agent instance o_i succeeds, and check in how many of these environment configurations the mutant instance m_i fails. We calculate the average proportion of failing environment configurations over all the pairs, and, if it exceeds 90%, we consider the mutant to be trivial. We exclude trivial mutants from our analysis, as they would inflate the mutation score without being discriminative.

3) Mutation Score: Once the likely-equivalent mutants are sorted out, for each pair (o_i, m_i) we generate L test environment configurations using the given test generator *TG*. The mutation score of a test generator *TG*, for a given mutation operator P, is the average killing rate K_R across all mutant configurations $P(j)$[1]. Given an RL mutation tool, the overall mutation score *MS* for a test generator *TG* is calculated as the average across the tool's MOs:

$$MS(TG, OP) = \frac{1}{|OP|} \sum_{P \in OP} \frac{1}{|J_P|} \sum_{j \in J_P} K_R(TG, P, j) \qquad (1)$$

V. Empirical evaluation

RQ_1 [Usefulness]: *Are μPRL's mutation operators useful? Do they discriminate between test environment generators of different qualities?*

In the first research question, we investigate whether the mutation testing pipeline in μPRL is able to generate *non-trivial, killable* and *discriminative* mutants, i.e., mutants that would tell apart test environment generators of different qualities.

Metrics. To answer RQ_1 we measure triviality and killability for each mutation operator in each subject environment and RL algorithm. We also measure the mutation scores of two

[1]Considering the killing rate K_R rather than the killed predicate K, ensures that the mutation score computation is more fine-grained.

test generators (details provided in Section V-B), namely *Weak* (TG_W) and *Strong* (TG_S). To evaluate the discriminative power of the mutants, we measure the *sensitivity* between the Weak and Strong test generators as defined in DeepCrime [32], when $MS(TG_S, OP) \geq MS(TG_W, OP)$:

$$Sensitivity = \frac{|MS(TG_S, OP) - MS(TG_W, OP)|}{MS(TG_S, OP)} \quad (2)$$

while we set it to zero when $MS(TG_S, OP) < MS(TG_W, OP)$.

RQ_2 [Comparison]: *How does μPRL compare with the state-of-the-art RLMutation approach?*

In this research question, we compare our tool with an existing mutation tool for RL, namely RLMutation [7].

Metrics. To answer RQ_2, we compare μPRL and RLMutation on the same subject environments and RL algorithms used for RQ_1, by measuring sensitivity of TG_W and TG_S on the mutants produced by both approaches.

RQ_3 [New Fault Types]: *What is the impact of the new fault types identified in our taxonomy and implemented in μPRL?*

In this research question, we evaluate the specific contribution of the new fault types that emerge from our taxonomy, w.r.t. existing RL fault taxonomies in the literature. In particular, we consider the impact of the five new mutation operators, namely SNR, SLS, NEI, SEC, and SPV.

Metrics. To answer RQ_3, we compute the sensitivity of the newly introduced fault types for each pair subject environment (*env*) – RL algorithm (A).

A. Subject Environments and RL Algorithms

Subject Environments. We evaluated our approach using two subject environments used in previous work [7], namely *CartPole* [12] and *LunarLander* [14] to be able to compare our approach with RLMutation, and we added two new subject environments, one concerning the driving domain, i.e., *Parking* [11], and a robotic environment, namely *Humanoid* [13], both of which are commonly used in the RL literature. Each environment has an initial configuration that is generated randomly at the beginning of each episode, according to the constraints determined by each environment.

RL Algorithms. We selected four RL algorithms that are widely used in the literature. DQN [15] is representative of value-based algorithms, while PPO [16] is a widely used policy-gradient algorithm. SAC [17] and TQC [18] are *hybrid* algorithms, i.e., blending value-based and policy-gradient techniques. DQN, SAC and TQC are off-policy algorithms, while PPO is an on-policy algorithm. The different characteristics of these four RL algorithms, allow for the application of all MOs of μPRL.

B. Procedure

RQ_1 [Usefulness]. For each subject environment we trained the original agents with the applicable RL algorithms using the hyperparameters provided by Raffin *et al.* [66]. DQN is only applicable to CartPole and LunarLander, as it only supports discrete action spaces, while SAC and TQC only support continous action spaces, hence they are only applicable on Parking and Humanoid. PPO cannot be applied on Parking as it is a goal-based environment that requires an off-policy algorithm. We discarded the PPO algorithm on Humanoid as with the default hyperparameters, the agent had a near zero success rate in repeated training instances.

We trained $n = 10$ original agents for each pair subject environment – RL algoritm (env, A), to account for the randomness of the training process. Then, for each applicable mutation operator (MO) P given the pair (env, A), we randomly sampled $j = 5$ mutant configurations.

When designing the sampling range for each MO, we started from the corresponding search space already defined by Raffin *et al.* [66] for hyperparameter tuning, but we adjusted it to increase the chance of generating challenging mutant configurations. For categorical search spaces, concerning six out of eight MOs, we decreased by 50% the upper and/or lower bounds of the original hyperparameter search space (for SDF we did not increase the upper bound as the original highest value 0.9999 is very close to the theoretical maximum 1.0). Three exceptions concern the SLS, NEI (not available in the original hyperparameter search space), and SNU operators, where we considered the corresponding mutation search space as relative to the training time steps budget.

After training the original agents and the corresponding mutant configurations for each pair (env, A), we replay the training environment configurations. For each mutation operator P, we select the configuration $P(j)$ that is killable, non-trivial, and closest to the original value.

The next step after replay is building the *Weak* (TG_W) and *Strong* (TG_S) test environment generators. To obtain TG_W, we first generated 200 test environment configurations at random and executed them on the original agent. During the execution we track the *quality of control* (*QOC*) of the trained agent, as a way to measure the *confidence* of the agent during a certain episode. For instance in CartPole, the agent can fail in two ways, i.e., either if it brings the cart too far from the center ($2.4m$) or if the pole it is controlling falls beyond a certain angle ($12°$). At each time step t the QOC_t is the minimum between two (normalised) distances, i.e., the absolute distance between the current position of the cart and 2.4, and the absolute difference between the current angle of the pole and $12°$. Likewise, the *QOC* metric can be defined for the other subject environments. Then, for each test environment configuration we take the minimum *QOC* value, and we rank the 200 test environment configurations in descending order of *QOC*. Finally, we select the first 50 test environment configurations, as those generated by TG_W.

To obtain TG_S, we resort to the approach by Uesato *et al.* [61] and Biagiola *et al.* [62], which consists of training a failure predictor on the training environment configurations, to learn failure patterns of the trained agent in order to generate challenging test environment configurations. Since our objective is to kill mutants, we train one neural network as failure predictor for each selected mutant configuration. Then, we use the trained neural network predictor in each mutant configuration to select 100 promising test environment config-

TABLE III: Results for RQ_1, RQ_2, RQ_3. Gray cells indicate that the specific mutation operator is not applicable to a certain (env, A) pair. Boldfaced values indicate that an operator is killable, while the symbol "–" stands for "not available". Underlined operators indicate new fault types w.r.t. existing taxonomies, and *Avg new* refers to the average computed only for underlined operators. The sensitivity of RLMutation is reported in the last row.

	CartPole										**LunarLander**										**Parking**									
	DQN					PPO					PPO					DQN					SAC					TQC				
			MS					MS					MS					MS					MS					MS		
	% trivial	% killable	Weak	Strong	Sensitivity	% trivial	% killable	Weak	Strong	Sensitivity	% trivial	% killable	Weak	Strong	Sensitivity	% trivial	% killable	Weak	Strong	Sensitivity	% trivial	% killable	Weak	Strong	Sensitivity	% trivial	% killable	Weak	Strong	Sensitivity
SEC						.00	.00	–	–	–	.00	**1.0**	.10	.78	.87															
SMR	.20	**.60**	.90	.90	.00											.00	**.80**	.14	1.0	.86									
SDF	.00	**.80**	.40	.44	.09	.00	**.20**	.40	.50	.20	.25	**1.0**	.10	.67	.85	.20	**1.0**	.67	.70	.04	.20	**1.0**	.80	1.0	.20	.00	**.80**	.10	.60	.83	.75	.75	–	–	–	.60	**.80**	.10	.20	.50
SLS	.40	**.60**	.50	.75	.33											.00	**.80**	.17	.60	.72	.00	**.80**	.00	1.0	1.0	.00	**.60**	.00	.30	1.0	.00	**1.0**	.00	.22	1.0	.00	.00	–	–	–
SNR						.00	.00	–	–	–	.00	**.75**	.10	.75	.87																	
NEI	.00	**.80**	.60	.67	.10	.00	**.20**	.50	.60	.17	.00	**.67**	.00	.70	1.0	.20	**.60**	.00	.67	1.0	.20	**1.0**	.00	.80	1.0	.00	**.80**	.00	.30	1.0	.20	**1.0**	.00	.20	1.0	.00	**.40**	.00	.00	–
SNU	.20	**.80**	.90	.80	.00											.00	**1.0**	.56	.88	.36																				
SPV																					.00	**.80**	.00	1.0	1.0	.00	**.40**	.00	.90	1.0	.00	**.50**	.00	.10	1.0	.00	.00	–	–	–
Avg	.16	.72	.66	.71	.11	.00	.10	.45	.55	.18	.06	.85	.08	.73	.90	.08	.84	.31	.77	.60	.10	.90	.20	.95	.80	.00	.65	.03	.53	.96	.24	.81	.00	.17	1.0	.15	.30	.05	.10	.50
Avg new	.20	.70	.55	.71	.22	.00	.07	.50	.60	.18	.00	.81	.07	.74	.91	.10	.70	.09	.64	.86	.07	.87	.00	.93	1.0	.00	.60	.00	.50	1.0	.07	.83	.00	.17	1.0	.00	.13	.00	.00	–
RLMut.			.90	.90	.00			.92	.92	.00			.83	.83	.04			1.0	.82	.00			–	–	–			–	–	–			–	–	–			–	–	–

urations, where each selected test environment configuration is chosen to maximise the probability of the failure predictor out of 500 candidates generated at random.

RQ_2 [Comparison]. To compare μPRL with RLMutation, we considered all the mutants produced by RLMutation which are publicly available. Then, for each killable mutant of RLMutation, we executed the test environment configurations generated by TG_W and TG_S. For each original and mutant pair, we used RLMutation to compute the killed predicate; we then computed the mutation score for TG_W and TG_S, as the ratio between the number of killed mutants and the total number of killable mutants.

C. Results

RQ_1 [Usefulness]. Table III shows the evaluation of μPRL for all subject environments and RL algorithms. Rows represent the MOs, while columns show the results of our mutation pipeline for each MO. For each pair (env, A), we report the percentage of trivial mutant configurations for each MO (*% trivial*), the percentage of killable configurations (*% killable*), the mutation scores (*MS*) for the TG_W (*Weak*) and TG_S (*Strong*) test generators, and the individual sensitivity (*Sensitivity*). For instance, in $(CartPole, DQN)$, the SLS mutation operator (4th row), has 40% of mutant configurations that are trivial, 60% of configurations that are killable, while the mutation score for $TG_W = 0.50$ and $TG_S = 0.75$, hence sensitivity is 0.33. We compute the mutation score for a given operator only if the operator is killable, and non-trivial. For instance, in $(CartPole, PPO)$, the SEC operator is non-killable (% killable $= 0.00$), while in $(Humanoid, SAC)$, the SDF operator is killable (% killable $= 0.75$), but all the killable configurations are trivial (% trivial $= 0.75$), hence we do not compute the mutation score for them.

We observe that for CartPole the sensitivity is quite low, i.e., 0.11 for DQN and 0.18 for PPO. Indeed, the DQN agent in CartPole is very weak, such that even TG_W is effective at killing mutant configurations (its mutation score is 0.66 on average, while the mutation score of TG_S is 0.71). On the other hand, the PPO agent on CartPole is very hard to kill for training environment configurations (on average the percentage of killable mutant configurations is 0.10), and, for killable configurations, TG_S has only a slight edge w.r.t. TG_W. However, for the remaining subject environments, which are more complex than CartPole (i.e., these environments are harder to learn for an RL agent), the average sensitivity ranges from a minimum of 0.50 in $(Humanoid, TQC)$ to a maximum of 1.00 in $(Humanoid, SAC)$.

RQ_1 [Usefulness]: Overall, the mutation operators of μPRL are effective at discriminating strong from weak test generators, especially in complex environments where the minimum sensitivity is 0.50 and the maximum is 1.0.

RQ_2 [Comparison]. The last row of Table III shows the average mutation score and sensitivity of RLMutation on the common pairs of subject environments and RL algorithms. We observe that, in all cases, the mutants created by μPRL are more sensitive than the mutants of RLMutation, whose maximum sensitivity is 0.04, for $(LunarLander, PPO)$.

RQ_2 [Comparison]: In all subject environments and RL algorithms, μPRL create mutants that are more sensitive than RLMutation's.

RQ$_3$ [New Fault Types]. In Table III we underline the mutation operators coming from the new fault types that are not present in existing RL fault taxonomies. The *Avg new* row shows the average metric values considering only the underlined mutation operators. For instance, for the pair $(CartPole, DQN)$, the average sensitivity across all mutation operators is 0.11, while the average sensitivity only considering the new fault types is 0.22. Overall, the mutation operators associated to new fault types have higher sensitivity than the total average in 5 cases; the same sensitivity in 2 cases (in one case sensitivity is not computable for the new operators).

RQ$_3$ [New Fault Types]: Mutation operators associated to new fault types contribute substantially to increase the sensitivity of μPRL.

VI. Threats to Validity

Internal Validity. An internal threat to the study's validity might come from the labeling of the artifacts. We addressed this threat by having at least two labelers independently label each post. We also fixed the disagreements within labelers and used Krippendorff's Alpha to quantify the disagreements. Furthermore, we initially conducted pilot studies to refine the labeling process. An additional internal validity threat concerns the subjective bias while constructing the taxonomy tree from the generated labels. This was alleviated by all the authors providing feedback on the final tree.

External Validity. An external validity threat is related to the generalizability of the bugs found on Stack Exchange and GitHub. While the sources of the bugs might be limited, we cross referenced top conferences and highly cited works, to ensure that such issues have been studied in the literature. The selection of conferences to identify a popular RL framework also poses an external validity threat. While we considered the top tier ML conference, i.e., ICML, considering other top ML conferences, could have given us a better picture of popular RL frameworks in the ML community. Generalization might also be affeced by our choice of mining Github only considering StableBaselines3. Although StableBaselines3 is the RL framework used by the majority of the papers in the selection of conferences we considered, by not investigating other RL frameworks we might lose out on a variety of important faults. Lastly, an additional external validity threat is related to the limited number of subject environments we considered in the evaluation. We selected *CartPole* and *LunarLander* to enable comparisons with previous work. Additionally, we considered *Parking* and *Humanoid*, which are also heavily used in DRL research. Overall, this selection of environments, supporting both discrete and continuous action spaces, allowed us to apply four foundational DRL algorithms, namely DQN, PPO, SAC and TQC.

Conclusion Validity. Conclusion validity threats are related to how random variations in the experiments are handled, and the inappropriate use of statistical tests. Since RL algorithms are notoriously sensitive to the random seed [21], we train original and mutated agents multiple times (i.e., $n = 10$), and we used rigorous statistical tests (i.e., the Fisher's test) to decide whether the mutated agent is killed. Recently Agarwal *et al.* [67] proposed bootstrap sampling to overcome the uncertainty given by a few-run RL training regime. We acknowledge that our experimental setting may benefit from bootstrap sampling, and by using the RLIABLE library [67], we re-computed the killability predicate on all $1.7k$ mutant instances using bootstrap sampling. In particular, we computed the probability of improvement, and estimated the confidence intervals using $2k$ samples. We found that the killed predicate computed using bootstrap sampling agrees with the killed predicate based on Fisher's test 88% of the time. Moreover, when in disagreement, 65% of the times the mutant instance is killed by the Fisher's test, indicating a higher statistical power than bootstrap sampling. Hence, these results suggest that bootstrap sampling would bring minimal benefit to our experimental setting, although it can be used as an alternative to the Fisher's test for the killed predicate.

VII. Conclusions and Future Work

We present a taxonomy of real RL faults. Using this taxonomy, we extracted mutation operators and implemented them in our tool μPRL. We evaluated its effectiveness in discriminating strong and weak test generators on a diverse set of environments using popular RL algorithms. Our tool also achieves higher sensitivity compared to the prior work, RLMutation, with a significant contribution from the operators that are derived from new taxonomy branches.

VIII. Data Availability

Our taxonomy labeling results are available on our replication package [68]. We also share the code of μPRL [60].

IX. Acknowledgments

We are grateful for the help provided by Breno Dantas Cruz. We also acknowledge the ICSE '25 reviewers for their valuable feedback. This work relied on the grants provided by the National Science Foundation: CCF-15-18897, CNS-15-13263, CNS-21-20448, CCF-19-34884, CCF-22-23812, and NRT-21-52117. This work used Explore ACCESS at Texas High Performance Research Computing through allocation CIS240181 from the Advanced Cyberinfrastructure Coordination Ecosystem: Services & Support (ACCESS) program, which is supported by National Science Foundation grants #2138259, #2138286, #2138307, #2137603, and #2138296 [69]. Matteo Biagiola was partially supported by the H2020 project PRECRIME, funded under the ERC Advanced Grant 2017 Program (ERC Grant Agreement n. 787703).

References

[1] D. Liu and L. Li, "A traffic light control method based on multi-agent deep reinforcement learning algorithm," *Scientific Reports*, vol. 13, no. 1, p. 9396, 2023.

[2] A. Chauhan, M. Baranwal, and A. Basumatary, "Powrl: A reinforcement learning framework for robust management of power networks," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 37, no. 12, 2023, pp. 14 757–14 764.

[3] E. Kaufmann, L. Bauersfeld, A. Loquercio, M. Müller, V. Koltun, and D. Scaramuzza, "Champion-level drone racing using deep reinforcement learning," *Nature*, vol. 620, no. 7976, pp. 982–987, 2023.

[4] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[5] R. G. Hamlet, "Testing programs with the aid of a compiler," *IEEE Transactions on Software Engineering*, vol. SE-3, pp. 279–290, 1977.

[6] Y. Lu, W. Sun, and M. Sun, "Towards mutation testing of reinforcement learning systems," *Journal of Systems Architecture*, vol. 131, p. 102701, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1383762122001977

[7] F. Tambon, V. Majdinasab, A. Nikanjam, F. Khomh, and G. Antoniol, "Mutation testing of deep reinforcement learning based on real faults," in *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*. Los Alamitos, CA, USA: IEEE Computer Society, apr 2023, pp. 188–198. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICST57152.2023.00026

[8] A. Nikanjam, M. M. Morovati, F. Khomh, and H. Ben Braiek, "Faults in deep reinforcement learning programs: a taxonomy and a detection approach," *Automated Software Engineering*, vol. 29, no. 1, p. 8, 2022.

[9] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann, "Stable-baselines3: Reliable reinforcement learning implementations," *Journal of Machine Learning Research*, vol. 22, no. 268, pp. 1–8, 2021. [Online]. Available: http://jmlr.org/papers/v22/20-1364.html

[10] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.

[11] E. Leurent, "An Environment for Autonomous Driving Decision-Making," May 2018. [Online]. Available: https://github.com/eleurent/highway-env

[12] A. G. Barto, R. S. Sutton, and C. W. Anderson, "Neuronlike adaptive elements that can solve difficult learning control problems," *IEEE Trans. Syst. Man Cybern.*, vol. 13, no. 5, pp. 834–846, 1983. [Online]. Available: https://doi.org/10.1109/TSMC.1983.6313077

[13] Y. Tassa, T. Erez, and E. Todorov, "Synthesis and stabilization of complex behaviors through online trajectory optimization," in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2012, Vilamoura, Algarve, Portugal, October 7-12, 2012*. IEEE, 2012, pp. 4906–4913. [Online]. Available: https://doi.org/10.1109/IROS.2012.6386025

[14] F. Foundation, "Lunarlander gymnasium documentation," https://gymnasium.farama.org/environments/box2d/lunar_lander/, 2024, online; accessed March 2024.

[15] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[16] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[17] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," in *International conference on machine learning*. PMLR, 2018, pp. 1861–1870.

[18] A. Kuznetsov, P. Shvechikov, A. Grishin, and D. Vetrov, "Controlling overestimation bias with truncated mixture of continuous distributional quantile critics," in *International Conference on Machine Learning*. PMLR, 2020, pp. 5556–5566.

[19] N. Humbatova, G. Jahangirova, G. Bavota, V. Riccio, A. Stocco, and P. Tonella, "Taxonomy of real faults in deep learning systems," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 1110–1121.

[20] M. J. Islam, G. Nguyen, R. Pan, and H. Rajan, "A comprehensive study on deep learning bug characteristics," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 510–520.

[21] P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger, "Deep reinforcement learning that matters," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 32, no. 1, 2018.

[22] L. Engstrom, A. Ilyas, S. Santurkar, D. Tsipras, F. Janoos, L. Rudolph, and A. Madry, "Implementation matters in deep RL: A case study on PPO and TRPO," in *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. [Online]. Available: https://openreview.net/forum?id=r1etN1rtPB

[23] Huang, Shengyi; Dossa, Rousslan Fernand Julien; Raffin, Antonin; Kanervisto, Anssi; Wang, Weixun, "The 37 Implementation Details of Proximal Policy Optimization," https://iclr-blog-track.github.io/2022/03/25/ppo-implementation-details/, 2022, online; accessed 16 March 2024.

[24] P. S. Castro, S. Moitra, C. Gelada, S. Kumar, and M. G. Bellemare, "Dopamine: A Research Framework for Deep Reinforcement Learning," 2018. [Online]. Available: http://arxiv.org/abs/1812.06110

[25] A. Kuhnle, M. Schaarschmidt, and K. Fricke, "Tensorforce: a tensorflow library for applied reinforcement learning," Web page, 2017. [Online]. Available: https://github.com/tensorforce/tensorforce

[26] M. Plappert, "keras-rl," https://github.com/keras-rl/keras-rl, 2016.

[27] M. Andrychowicz, A. Raichuk, P. Stańczyk, M. Orsini, S. Girgin, R. Marinier, L. Hussenot, M. Geist, O. Pietquin, M. Michalski *et al.*, "What matters in on-policy reinforcement learning? a large-scale empirical study," in *ICLR 2021-Ninth International Conference on Learning Representations*, 2021.

[28] L. Ma, F. Zhang, J. Sun, M. Xue, B. Li, F. Juefei-Xu, C. Xie, L. Li, Y. Liu, J. Zhao, and Y. Wang, "DeepMutation: Mutation testing of deep learning systems," in *29th IEEE International Symposium on Software Reliability Engineering, ISSRE 2018, Memphis, TN, USA, October 15-18, 2018*, 2018, pp. 100–111. [Online]. Available: https://doi.org/10.1109/ISSRE.2018.00021

[29] W. Shen, J. Wan, and Z. Chen, "MuNN: Mutation analysis of neural networks," in *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, July 2018, pp. 108–115.

[30] Q. Hu, L. Ma, X. Xie, B. Yu, Y. Liu, and J. Zhao, "DeepMutation++: A Mutation Testing Framework for Deep Learning Systems," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1158–1161. [Online]. Available: https://doi.org/10.1109/ASE.2019.00126

[31] G. Jahangirova and P. Tonella, "An empirical evaluation of mutation operators for deep learning systems," in *IEEE International Conference on Software Testing, Verification and Validation*, ser. ICST'20. IEEE, 2020, p. 12 pages. [Online]. Available: https://doi.org/10.1109/ICST46399.2020.00018

[32] N. Humbatova, G. Jahangirova, and P. Tonella, "Deepcrime: Mutation testing of deep learning systems based on real faults," ser. ISSTA 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 67–78. [Online]. Available: https://doi.org/10.1145/3460319.3464825

[33] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[34] R. Agarwal, M. Schwarzer, P. S. Castro, A. C. Courville, and M. G. Bellemare, "Deep reinforcement learning at the edge of the statistical precipice," in *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, M. Ranzato, A. Beygelzimer, Y. N. Dauphin, P. Liang, and J. W. Vaughan, Eds., 2021, pp. 29 304–29 320. [Online]. Available: https://proceedings.neurips.cc/paper/2021/hash/f514cec81cb148559cf475e7426eed5e-Abstract.html

[35] A. Nikanjam, M. M. Morovati, F. Khomh, and H. Ben Braiek, "Faults in deep reinforcement learning programs: A taxonomy and a detection approach," vol. 29, no. 1, may 2022. [Online]. Available: https://doi.org/10.1007/s10515-021-00313-x

[36] K. Krippendorff, "Estimating the reliability, systematic error and random error of interval data," *Educational and psychological measurement*, vol. 30, no. 1, pp. 61–70, 1970.

[37] ——, *Content analysis: An introduction to its methodology*. Sage publications, 2018.

[38] A. F. Hayes and K. Krippendorff, "Answering the call for a standard reliability measure for coding data," *Communication methods and measures*, vol. 1, no. 1, pp. 77–89, 2007.

[39] T. Miki, J. Lee, J. Hwangbo, L. Wellhausen, V. Koltun, and M. Hutter, "Learning robust perceptive locomotion for quadrupedal robots in the wild," *Sci. Robotics*, vol. 7, no. 62, 2022. [Online]. Available: https://doi.org/10.1126/scirobotics.abk2822

[40] P. Abbeel and A. Y. Ng, "Apprenticeship learning via inverse reinforcement learning," in *Proceedings of the twenty-first international conference on Machine learning*, 2004, p. 1.

[41] M. Riedmiller, R. Hafner, T. Lampe, M. Neunert, J. Degrave, T. Wiele, V. Mnih, N. Heess, and J. T. Springenberg, "Learning by playing solving sparse reward tasks from scratch," in *International conference on machine learning*. PMLR, 2018, pp. 4344–4353.

[42] Y. Duan, X. Chen, R. Houthooft, J. Schulman, and P. Abbeel, "Benchmarking deep reinforcement learning for continuous control," in *International conference on machine learning*. PMLR, 2016, pp. 1329–1338.
[43] S. Gu, T. Lillicrap, Z. Ghahramani, R. E. Turner, and S. Levine, "Q-prop: Sample-efficient policy gradient with an off-policy critic," *arXiv preprint arXiv:1611.02247*, 2016.
[44] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton *et al.*, "Mastering the game of go without human knowledge," *nature*, vol. 550, no. 7676, pp. 354–359, 2017.
[45] R. Coulom, "Efficient selectivity and backup operators in monte-carlo tree search," in *International conference on computers and games*. Springer, 2006, pp. 72–83.
[46] L. Kocsis and C. Szepesvári, "Bandit based monte-carlo planning," in *European conference on machine learning*. Springer, 2006, pp. 282–293.
[47] P. F. Christiano, J. Leike, T. Brown, M. Martic, S. Legg, and D. Amodei, "Deep reinforcement learning from human preferences," *Advances in neural information processing systems*, vol. 30, 2017.
[48] S. He, H.-S. Shin, and A. Tsourdos, "Computational missile guidance: A deep reinforcement learning approach," *Journal of Aerospace Information Systems*, vol. 18, no. 8, pp. 571–582, 2021.
[49] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *International conference on machine learning*. PMLR, 2016, pp. 1928–1937.
[50] A. Braylan, M. Hollenbeck, E. Meyerson, and R. Miikkulainen, "Frame skip is a powerful parameter for learning to play atari," in *Workshops at the twenty-ninth AAAI conference on artificial intelligence*, 2015.
[51] A. Srinivas, S. Sharma, and B. Ravindran, "Dynamic action repetition for deep reinforcement learning," in *Proc. AAAI*, 2017.
[52] S. P. Singh and R. S. Sutton, "Reinforcement learning with replacing eligibility traces," *Machine learning*, vol. 22, pp. 123–158, 1996.
[53] D. Rolnick, A. Ahuja, J. Schwarz, T. Lillicrap, and G. Wayne, "Experience replay for continual learning," *Advances in Neural Information Processing Systems*, vol. 32, 2019.
[54] J. Kirkpatrick, R. Pascanu, N. Rabinowitz, J. Veness, G. Desjardins, A. A. Rusu, K. Milan, J. Quan, T. Ramalho, A. Grabska-Barwinska, D. Hassabis, C. Clopath, D. Kumaran, and R. Hadsell, "Overcoming catastrophic forgetting in neural networks," *Proceedings of the National Academy of Sciences*, vol. 114, no. 13, pp. 3521–3526, 2017. [Online]. Available: https://www.pnas.org/doi/abs/10.1073/pnas.1611835114
[55] W. Fedus, D. Ghosh, J. D. Martin, M. G. Bellemare, Y. Bengio, and H. Larochelle, "On catastrophic interference in atari 2600 games," *CoRR*, vol. abs/2002.12499, 2020. [Online]. Available: https://arxiv.org/abs/2002.12499
[56] S. Zhang and R. S. Sutton, "A deeper look at experience replay," *arXiv preprint arXiv:1712.01275*, 2017.
[57] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.
[58] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, pp. 484–489, 2016.
[59] Z. Liu, X. Li, B. Kang, and T. Darrell, "Regularization matters in policy optimization-an empirical study on continuous control," in *International Conference on Learning Representations*, 2020.
[60] T. automated USI, 2024. [Online]. Available: https://github.com/testingautomated-usi/muPRL
[61] J. Uesato, A. Kumar, C. Szepesvári, T. Erez, A. Ruderman, K. Anderson, K. D. Dvijotham, N. Heess, and P. Kohli, "Rigorous agent evaluation: An adversarial approach to uncover catastrophic failures," in *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. [Online]. Available: https://openreview.net/forum?id=B1xhQhRcK7
[62] M. Biagiola and P. Tonella, "Testing of deep reinforcement learning agents with surrogate models," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 3, mar 2024. [Online]. Available: https://doi.org/10.1145/3631970
[63] OpenAI, "OpenAI Gym Leaderboard," https://github.com/openai/gym/wiki/Leaderboard/, 2024, online; accessed March 2024.
[64] R. A. Fisher, *Statistical Methods for Research Workers*. New York, NY: Springer New York, 1992, pp. 66–70. [Online]. Available: https://doi.org/10.1007/978-1-4612-4380-9_6
[65] J. Kim, N. Humbatova, G. Jahangirova, P. Tonella, and S. Yoo, "Repairing DNN architecture: Are we there yet?" in *IEEE Conference on Software Testing, Verification and Validation, ICST 2023, Dublin, Ireland, April 16-20, 2023*. IEEE, 2023, pp. 234–245. [Online]. Available: https://doi.org/10.1109/ICST57152.2023.00030
[66] A. Raffin, "Rl baselines3 zoo," https://github.com/DLR-RM/rl-baselines3-zoo, 2020, online; accessed March 2024.
[67] R. Agarwal, M. Schwarzer, P. S. Castro, A. C. Courville, and M. G. Bellemare, "Deep reinforcement learning at the edge of the statistical precipice," in *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, M. Ranzato, A. Beygelzimer, Y. N. Dauphin, P. Liang, and J. W. Vaughan, Eds., 2021, pp. 29 304–29 320. [Online]. Available: https://proceedings.neurips.cc/paper/2021/hash/f514cec81cb148559cf475e7426eed5e-Abstract.html
[68] [Online]. Available: https://github.com/Deepakgthomas/benchmarking_rlmut
[69] T. J. Boerner, S. Deems, T. R. Furlani, S. L. Knuth, and J. Towns, "ACCESS: Advancing Innovation: NSF's Advanced Cyberinfrastructure Coordination Ecosystem: Services & Support," in *Practice and Experience in Advanced Research Computing 2023: Computing for the Common Good*, ser. PEARC '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 173–176. [Online]. Available: https://doi.org/10.1145/3569951.3597559