Testing of Deep Reinforcement Learning Agents with Surrogate Models

MATTEO BIAGIOLA and PAOLO TONELLA, Università della Svizzera italiana, Switzerland

Deep Reinforcement Learning (DRL) has received a lot of attention from the research community in recent years. As the technology moves away from game playing to practical contexts, such as autonomous vehicles and robotics, it is crucial to evaluate the quality of DRL agents.

In this paper, we propose a search-based approach to test such agents. Our approach, implemented in a tool called Indago, trains a classifier on failure and non-failure environment configurations resulting from the DRL training process. The classifier is used at testing time as a surrogate model for the DRL agent execution in the environment, predicting the extent to which a given environment configuration induces a failure of the DRL agent under test. Indeed, the failure prediction acts as a fitness function, in order to guide the generation towards failure environment configurations, while saving computation time by deferring the execution of the DRL agent in the environment to those configurations that are more likely to expose failures.

Experimental results show that our search-based approach finds 50% more failures of the DRL agent than state-of-the-art techniques. Moreover, such failure environment configurations, as well as the behaviours of the DRL agent induced by them, are significantly more diverse.

CCS Concepts: • **Software and its engineering → Software verification and validation**.

Additional Key Words and Phrases: Software Testing, Reinforcement Learning

1 INTRODUCTION

Reinforcement Learning (RL) is a learning paradigm in which an agent interacts with the environment to complete a given task. Learning is driven by a reward signal returned to the agent by the environment. The ultimate goal of an agent is to learn a policy, i.e., a way to act in each environment state, that maximizes the amount of reward the agent earns in its lifetime. The first RL algorithms were tabular [60] and could solve toy tasks and play simple games. With the advent of Deep Learning (DL), new algorithms (called Deep RL, or DRL for short) were proposed, which could solve more difficult tasks and deal with complex environment states (e.g., images) [42].

DRL has recently been applied in many practical contexts. An example is personalization, i.e., the problem of customizing a service to the needs of a particular user. For instance, Netflix uses DRL to choose which movie artwork to show to a user in order to maximize engagement [28]. Similarly, Microsoft developed Personalizer [35], a service developers can use for content recommendation and ad placement. Meta proposed Horizon [29, 30] (also called ReAgent), an open source applied DRL platform, employed to deliver personalized notifications to their users, replacing the previous system based on supervised learning.

Another practical context in which the DRL paradigm is applied is continuous control. For instance, the automaker Audi [21] showcased that their 1:8 scale car could, using DRL, search for a parking place in an area of 9 square meters and park autonomously. Indeed, advancements in state representation learning and smooth exploration [36, 48, 70] made it possible to train DRL agents directly on real robots. Moreover, in recent years, simulators have become more realistic for a variety of tasks, besides achieving high parallelization thanks to GPU acceleration [26, 39]. In addition, domain randomization and learned actuator dynamics are reducing the sim-to-real gap in robotics research [25, 41, 53].

Despite the growing prevalence of DRL agents in the real world, methodologies for testing such agents are still largely unexplored. On the other hand, DRL agents present some peculiar characteristics. Indeed, DRL agents are

Authors' address: Matteo Biagiola, matteo.biagiola@usi.ch; Paolo Tonella, paolo.tonella@usi.ch, Università della Svizzera italiana, Lugano, Switzerland.

trained online since they interact with the environment to learn the optimal actions to perform the task. Specifically, in order to increase generalization, DRL agents are usually trained on randomized environment configurations [16, 65] to prevent agents from memorizing how to behave in a particular instance of the environment (e.g., a self-driving car that drives only on a specific track). Therefore, during training the DRL agent is presented with different environment configurations (e.g., different tracks) and it *fails* in some while it *succeeds* in others.

The current state-of-the-practice to test DRL agents is to run them on a set of environment configurations generated at random [43, 65]. However, testing a DRL agent on randomly generated environment configurations has two shortcomings. First, random generation is unlikely to expose failures. As a consequence, their absence might lead the developer to overestimate the capabilities of the DRL agent and to the deployment of an unsafe agent. Secondly, even finding *challenging* environment configurations by random exploration is difficult and computationally expensive, since many executions are needed and each execution requires running the DRL agent in a simulator or in the real world.

On the other hand, the *interactions* of the DRL agent with the environment during training provide clues about the weaknesses of the DRL agent that results from the training process. The intuition is that training failures are representative of *critical* environment configurations even for the DRL agent once it has been trained, and could be used as guidance for the generation of new environment configurations that will likely challenge it.

Our approach, implemented in a tool called Indago, considers the interaction data produced during the DRL training process as a labeled dataset to train a surrogate model — i.e., a classifier — on failure and non-failure environment configurations. Then, Indago uses such surrogate model as a proxy for the execution of the DRL agent in an environment with a newly generated configuration. In particular, Indago uses a search-based approach to maximize the failure prediction for an environment configuration given by the surrogate model. Moreover, Indago uses a mutation operator guided by saliency-based input attribution [56], in order to identify mutations that have the maximum influence on the failure prediction. In this way, the DRL agent under test is executed only on the most promising environment configurations, i.e., those with the highest failure predictions, hence saving computation time while maximizing failure exposures.

Our paper makes the following contributions:

1) Failure Search with Surrogate Models: in this paper we propose an approach that makes use of a surrogate model of the environment to guide failure search while automatically generating environment configurations for a DRL agent. In particular, we train a classifier on the training interaction data and use its output as a fitness function to maximize the failure prediction of a given environment configuration. Moreover, we use the saliency method to efficiently identify the most critical mutations for the given environment configurations.

2) The Indago Tool: a practical tool, that implements the aforementioned approach, which we make publicly available [7]. We also release three DRL agents trained on as many complex environments, as well as the required infrastructure to test them.

3) Experimental Evaluation: we systematically compare different configurations of Indago with the state-of-the-art sampling approach [65] that maximizes failure prediction by generating a large set of environment configurations. On three complex case studies, i.e., a parking task [37], a walking humanoid [64] and a self-driving car [63], our experiments show that, overall, Indago is able to find 50% more failure environment configurations than sampling. Moreover, we introduce a clustering-based technique to measure the diversity of failure environment configurations triggered by the competing approaches. Experimental results show that the failure environment configurations found by Indago are 77% more diverse than those generated by sampling. Moreover, the behaviours of the DRL agent induced by such environment configurations are 74% more diverse.

2 BACKGROUND AND MOTIVATION

2.1 Reinforcement Learning

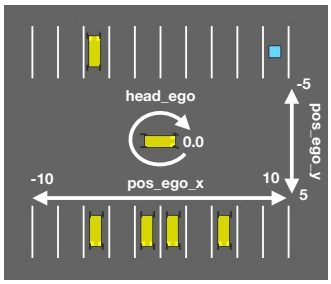
Fundamentals and Notations. Reinforcement Learning (RL) aims at learning a policy, which is a mapping from states to actions, in order to optimize a numerical reward signal [60]. The agent that acts in the environment needs to discover what actions result in a high reward thorough trial and error without the presence of a supervisor. The main assumption of this learning paradigm is the so called reward hypothesis [60], stating that training goals can be expressed as the maximization of the cumulative reward.

More formally, at each timestep t an RL agent receives a state s as input from the environment and it has to decide the action a to take. The executed action triggers a change of state and results in a reward value r given to the agent by the environment. Assuming that the task we formalize as an RL problem is episodic, i.e., it terminates once certain conditions hold, the goal of an RL agent is to maximize the cumulative reward (i.e., often called return) of the episode. In the general case, however, the RL objective is expressed as the maximization of the expected return since both the environment in which the agent operates and its policy can be stochastic. The main reason for the stochastic nature of a policy is due to a fundamental dilemma in RL, which is the exploration-exploitation dilemma. In fact, on the one hand, the agent has to exploit the actions already known to be rewarding but, on the other hand, it has to explore unknown actions that might result in even more reward.

The most important component of an RL agent is its policy $\pi : \mathcal{S} \rightarrow \mathcal{A}$, where $\mathcal{A}$ is a set of actions and $\mathcal{S}$ is a set of states. The optimal policy π^* tells the RL agent how to act in each state in order to maximize the expected return. The optimal policy can be learned directly or can be extracted from value functions, namely the state-value function $v_\pi(s)$ and the action-value function $q_\pi(s, a)$. The former quantifies the value of the state s, i.e., the expected return in s, whereas the latter quantifies the value of the state-action pair (s, a). The difference between the two functions is that v_π provides the value of a state s by considering each possible action the agent can take in s (in other words, the average of the expected return in s for all the actions), while q_π considers the value of a state s for a particular action a. Both functions satisfy the Bellman equations [60] which are recursive consistency equations relating the values of a state (or state-action pair) to the values of all the possible successor states (or state-action pairs). In particular, by solving the Bellman equation for q, we obtain q^* from which we can extract π^* by choosing in each state s the action a that maximizes q^*.

Deep RL Algorithms. Before deep learning, the RL problem was addressed using dynamic programming and approximate tabular methods, such as monte carlo methods and temporal difference learning [60]. However, such methods are not applicable to problems where the state dimensionality is high (e.g., images) and/or the action space is continuous (e.g., the throttle in a self-driving car). The advent of deep learning made it possible to create Deep RL (DRL) algorithms that work on such complex practical scenarios [42]. In particular Deep RL (DRL) algorithms use non-linear function approximators, such as neural networks, to approximate high dimensional state and action spaces besides modeling the dynamics of the environment. Therefore, instead of having exact representations of policies π and value functions (v and q), neural networks can be used to approximate such quantities as well as to model the environment in which the agent operates.

In our experiments we consider model-free DRL algorithms which do not use a model of the environment. There exist different categories of model-free algorithms, based on how the RL problem is addressed. Specifically, policy gradients algorithms, of which PPO is a notable example [55], directly solve the RL objective by representing the policy explicitly (i.e., with a neural network). Value-based algorithms, on the other hand, extract the policy by solving the




```
{
  "env_configuration": {
    "goal_lane": 20,
    "head_ego": 0.0,
    "pvehicles": {
       3, 5, 6, 8, 13
     },
    "pos_ego": (0.0, 0.0)
  }
}
```

Fig. 1. An initial configuration of the *Parking* environment in the *HighwayEnv* simulator [37].

Bellman equations, hence representing value functions explicitly. Examples of algorithms in such category are DQN and its improvements [15, 22, 23, 42, 54, 72]. Hybrid methods represent both policy and value functions to incorporate, the benefits of both policy gradients and value-based methods. SAC [20] and TQC [34] are the state-of-the-art algorithms in this category. Belonging to a special category is the algorithm HER [3], which was proposed as a wrapper on top of traditional DRL algorithms, to speed up learning in the context of goal-based tasks, e.g., parking a car from a starting position to a target parking spot.

2.2 Motivating Example

Figure 1 shows the *Parking* environment, created by Leurent et al. [37], in a particular configuration where the DRL agent needs to control the ego vehicle positioned at the center of the parking place. In such environment configuration the ego vehicle is at the center of the parking place, i.e., its position is $(0.0, 0.0)$, and it has a heading of 0.0 (this parameter ranges in the interval $[0.0, 1.0)$, representing a complete rotation). In the parking place there are 20 parking spots, 5 cars parked at lanes 3, 5, 6, 8, 13 and the target parking spot is at lane 20. The task of the DRL agent in this environment is to park the ego vehicle inside the target parking spot, with the proper heading.

The action space of the DRL agent is composed of two actions, namely throttle and steering, both of which are continuous. The DRL agent receives a negative reward at each timestep, proportional to the Euclidean distance of the ego vehicle from the target. Moreover, it receives a constant positive reward when the target is reached and a big constant negative reward when it collides with a parked vehicle. The task is episodic with an episode finishing when either the ego vehicle is parked in the right target spot with the right heading, or it collides with a parked vehicle, or the timeout, measured in number of timesteps, expires.

We made the environment *configurable*, such that the parameters of the configuration (i.e., the initial conditions at the beginning of each episode) can be changed programmatically. Correspondingly, an environment configuration needs to be *valid*, i.e., it has to respect the constraints imposed by the environment. Such constraints are designed by the developers of the environment to ensure that valid configurations are *solvable* by the DRL agent. In other words, any agent would be able solve the task when starting from a valid initial configuration, since the environment does not contain any physically insurmountable obstacle or impediment.

The constraints defined for the Parking environment are the following: there cannot be a parked vehicle in the goal lane (`goal_lane` $\notin$ `pvehicles`) since, otherwise, it would be impossible for the DRL agent to successfully park the vehicle in the target spot. Moreover, `goal_lane` and `pvehicles` elements can vary in the interval $[1, 20]$, i.e., the target cannot be out of the parking place and there cannot be vehicles outside of the parking spots. The `head_ego` parameter

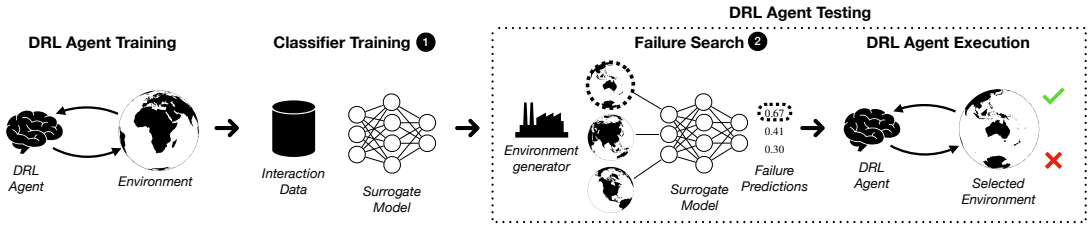


Fig. 2. Overall approach for testing DRL agents

can vary in the interval $[0.0, 1.0)$. The parameters `pos_ego.x` and `pos_ego.y` can vary in the intervals $[-10, 10]$ and $[-5, 5]$ respectively; in the former case the constraint ensures that the ego vehicle is not too far from the parking place while the latter constraint avoids the ego vehicle to be too close to parked vehicles that would make any maneuver impossible and, as a consequence, the task unsolvable.

3 APPROACH

The goal of our approach is to exploit the data resulting from the interaction between the DRL agent and the environment during training in order to discover the weaknesses of the DRL agent at testing time. The interaction data we consider is in the form of pairs (e_i, c_i) where e_i is the environment configuration at episode i during training and c_i is a class label, i.e., a boolean value indicating whether the DRL agent failed ($c_i = 1$) or not ($c_i = 0$) at the task in episode i.

Our approach exploits the information on the failures that happened during training with the objective of generating new *critical* test cases, i.e., environment configurations, in which the DRL agent under test (i.e., the DRL agent at the end of training) is likely to fail. Since the execution of the DRL agent in the environment is computationally expensive we avoid the execution of candidate new environment configurations that are not promising, i.e., that are unlikely to lead to the exposure of a failure. The training interaction data can be leveraged to predict which, among the newly generated environment configurations at testing time, are more likely to produce a failure and the DRL agent can be executed only in environments with such promising configurations.

The current state-of-the-practice to test DRL agents is to evaluate them for a certain number of episodes, each with an environment configuration generated at random [43, 65]. However, environment configurations generated at random are unlikely to expose failures although specific environment configurations may exist that are challenging for the DRL agent under test. On the other hand, the DRL training process offers a valuable source of information exploitable at testing time to efficiently expose failures of the DRL agent under test. The intuition is that the failures experienced during training by weaker versions of the DRL agent under test are *representative* of critical environment configurations of the DRL agent at the end of training, and can be used to guide the generation of new environment configurations that can challenge it.

Figure 2 summarizes the overall approach to use interaction data produced during the DRL training process to test a DRL agent. Given the output of the DRL training process (i.e., pairs of interaction data (e_i, c_i)), the first step ❶ is to train a classifier on failure prone ($c_i = 1$) environment configurations e_i and non-failing ones ($c_i = 0$). The classifier is used in the next step (DRL Agent Testing) to predict whether unseen environment configurations are likely to be failures. The failure search step ❷ uses the classifier to generate environment configurations in which the DRL agent is more likely to fail, by acting as proxy for the execution of the DRL agent in the environment with such configurations.

The output of step ❷ is the environment configuration that has the highest failure prediction among the candidates (in Figure 2 the selected environment configuration is encircled with a dashed line). Finally, the DRL agent is executed in an environment with such configuration, to check whether the execution results in an actual failure or not.

3.1 Classifier Training

The classifier is trained to predict whether, given an environment configuration, the DRL agent under test will fail in it or not. The interaction dataset $D = \{(e_1, c_1), \ldots, (e_N, c_N)\}$ is used to train such classifier. In particular, we train a *softmax* classifier (i.e., a neural network) to minimize the *cross-entropy* loss.

The classification problem to address in step ❶ presents some peculiar characteristics. In particular, the dataset D might be *unbalanced*, i.e., the number of environment configurations in which the agent fails ($c = 1$) is much lower than the number of environment configurations in which the agent does not fail ($c = 0$). The reason is that at the beginning of the training process the DRL agent fails in most of the environment configurations while, as training goes on, the DRL agent fails less and less until the training process converges and failures become rare. One of the strategies to train a classifier in situations of class unbalance is to introduce a weight vector W in the loss function [13]. Such vector has two components, i.e., one for each class, in order to scale the loss function for each i-th datapoint w.r.t. the class the datapoint belongs to. The idea is to give more weight to the datapoints of the underrepresented class (i.e., the failure class) such that the classifier can learn to classify datapoints of both classes equally. In the literature [14, 32], there are various proposals for the computation of W. Our implementation choice is described in Section 4.2.

3.2 Failure Prediction

The classifier is used to predict the class of any environment configuration that it is not present in the dataset D of interaction data. Therefore, it is a proxy for the actual execution of the DRL agent in the environment with a specific configuration. More formally, it is a function $f : E \rightarrow [0, 1] \in \mathbb{R}$ that takes as input an environment configuration $e \in E$ and outputs a failure prediction.

Given the classifier, our objective is to generate an environment configuration that maximizes the failure prediction, i.e., to solve $\hat{e} = \arg\max_e f(e)$, and then execute the DRL agent in an environment with configuration $\hat{e}$. It should be noticed that the real failure prediction function f^* is only approximated by our classifier, i.e., $f \approx f^*$. The reason for the approximation is two-fold: (1) the dataset D of interaction data might not be large and diverse enough to best represent all real-world conditions; (2) during training the DRL agent performs non-deterministic actions to better explore the state space. As a consequence, a failure that happens during training might be due to some non-deterministic actions carried out in a specific episode. Despite the mismatch between f and f^*, failure search (see Figure 2) can still effectively use the classifier that implements f, provided its feedback on the failure prediction of a new environment configuration can be reliably used to converge toward inputs that challenge the DRL agent. In fact, failure search does not require perfect guidance toward optimal inputs, it just needs a direction where the search should be directed [31].

3.3 Search-Based Failure Search with Surrogate Models

For the failure search step ❷ we use the output of the classifier as a *fitness function* to be optimized by a search-based algorithm. In particular, we consider two search-based algorithms, i.e., *Hill Climbing* and *Genetic Algorithm*, to generate the environment configurations where to execute the DRL agent.

Hill climbing is a local search algorithm that, starting from an arbitrary candidate solution to the problem (i.e., an environment configuration), incrementally changes such solution to create new ones. If a better solution is found

Algorithm 1: Hill climbing algorithm for the generation of environment configurations

Input : f, classifier;
NS, neighborhood size;
e_f, environment configuration in which the DRL agent failed during training.
Output : $\hat{e}$, environment configuration to execute the DRL agent on.

```
1  if e_f = null then
2  |  e ← generateRndEnvConfig()
3  else
4  |  e ← e_f
5  end
6  repeat
7  |  E ← {e}
8  |  FP ← {f(e)}
9  |  foreach i ∈ NS do
10 |  |  e_i ← mutateEnvConfig(e)
11 |  |  fp_i ← f(e_i)
12 |  |  E ← E ∪ {e_i}
13 |  |  FP ← FP ∪ {fp_i}
14 |  end
15 |  j ← arg max FP
16 |  e ← E[j]
17 until ¬ timeout()
18 ê ← e
19 return ê
```

the process is repeated until the current solution can no longer be improved or a timeout expires. Genetic algorithm is a population-based algorithm that combines global and local search to avoid getting stuck in local optima while improving existing solutions.

Algorithm 1 shows the pseudocode of the hill climbing algorithm for the generation of environment configurations. It takes as input the classifier f, the size of the neighborhood NS of the current solution e and an optional environment configuration in which the DRL agent failed during training e_f. If such environment configuration is not provided, the initial solution is generated at random (see *if* statement at Lines 1–5). The *for* loop at Lines 9–14 computes, at each i-th iteration, the *neighbors* of the current solution e. Indeed, function mutateEnvConfig at Line 10 takes care of *mutating* the current solution e and ensuring the validity of the result, i.e., the mutation is applied to e only if the new environment configuration e_i is valid. The mutated solution e_i is then evaluated by the classifier to compute its failure prediction at Line 11. At Lines 12–13 each mutation of the current solution, i.e., e_i, is stored as well as its failure prediction fp_i. At the end of the loop, the index j of the neighboring solution with the maximum failure prediction is computed (Line 15) which is used to retrieve the corresponding neighboring solution e (Line 16). The outermost loop at Lines 6–17 is repeated until there is search budget (i.e., the *timeout*) is not expired. Finally, the best solution e is assigned to $\hat{e}$ and returned.

The *Genetic Algorithm* shown in Algorithm 2 takes as input the classifier f, the population size PS, the crossover rate cr and an optional set of environment configurations in which the DRL agent failed during training E_f. Such set can be used to fill the initial population when it is available; otherwise the initial population is generated randomly

Algorithm 2: Genetic algorithm for the generation of environment configurations

Input : f, classifier;
PS, population size;
cr, crossover rate;
E_f, set of environment configurations in which the DRL agent failed during training/
Output : $\hat{e}$, environment configuration to execute the agent on.

```
1  population ← GENERATEPOPULATION(PS, E_f)
2  COMPUTEFITNESS(population, f)
3  currentIteration ← 0
4  repeat
5      newPop ← ELITISM(population)
6      while | newPop | < PS do
7          pe_1 ← SELECTION(population)
8          pe_2 ← SELECTION(population)
9          oe_1 ← COPY(pe_1)
10         oe_2 ← COPY(pe_1)
11         if GETRANDOMFLOAT() < cr then
12             oe_1, oe_2 ← CROSSOVER(oe_1, oe_2)
13         end
14         oe_1 ← MUTATEENVCONFIG(oe_1)
15         oe_2 ← MUTATEENVCONFIG(oe_2)
16         ADDBESTINDIVIDUALS(newPop, pe_1, pe_2, oe_1, oe_2)
17     end
18     population ← newPop
19     COMPUTEFITNESS(population, f)
20     RESEEDPOPULATION(population, currentIteration, E_f)
21     currentIteration ← currentIteration +1
22 until ¬ timeout()
23 ê ← GETINDIVIDUALWITHBESTFITNESS(population, f)
24 return ê
```

(Line 1). The COMPUTEFITNESS procedure at Line 2 computes the fitness value for each solution in the population, which is its failure prediction as computed by the classifier f. At Line 5 a new population with the best solutions from the current population is instantiated (*elitism*). The while loop at Lines 6–17 is the evolution part of the algorithm which terminates when the size of the new population reaches the target population size PS. At the beginning of the loop two solutions are selected based on their fitness (Lines 7–8) and are crossed over according to the crossover probability cr (Lines 11–13); the solutions oe_1 and oe_2 are modified only if the crossover results are two valid configurations. Afterwards, the solutions are mutated (Lines 14–15, also in this case oe_1 and oe_2 are modified only if the changes result in valid configurations) and finally the best solutions (either the parents pe_1, pe_2 or the offsprings oe_1, oe_2) are stored in the new population (Line 16). At the end of the while loop the current population is assigned the new population and the fitness value of each solution is computed (Lines 18–19). The RESEEDPOPULATION procedure at Line 20 takes care of avoiding stagnation by reseeding the population according to the *currentIteration* variable. If the set of failure environment configurations E_f is available, the solutions with the worst fitness in the current population are replaced by randomly sampled solutions from the set E_f; otherwise the worst solutions are replaced by randomly generated

individuals. The main loop (Lines 4–22) terminates when the search budget expires. Finally, the solution with the best fitness $\hat{e}$ is taken at Line 23 and returned.

Both search algorithms support seeding from existing environment configurations that caused a failure of the agent during training. In particular, let us suppose that the parameter e_f is not *null* in Algorithm 1 (similarly, E_f is not empty in Algorithm 2). The DRL agent may not fail in such environment configurations because it may have encountered similar environment configurations later during training and it might have adapted to them. However, despite adaptation, the DRL agent might not perform well and a *proper change* in such environment configuration has the potential to trigger a failure. For example, let us suppose that the environment configuration $e_f = [20, 0.0, \{3, 5, 6, 8, 13, 19\}, (0.0, 0.0)]$ causes a failure of the DRL agent during training in the Parking environment (it is the same environment configuration shown in Figure 1, except that there is a parked vehicle beside the target spot), but the DRL agent does not fail in e_f at testing time. However, changing the heading of the ego vehicle from 0.0 to 0.5, i.e., the opposite direction w.r.t. the target spot, might result in the DRL agent not being able to turn the vehicle and park it with the right heading, since the mutation has succeeded in making the environment more challenging, eventually exposing a failure.

3.3.1 Mutation Function. Knowing what to change in the environment configuration is important for finding failures. The function that makes this decision is the MUTATEENVCONFIG function (Line 13 in Algorithm 1 and Lines 14–15 in Algorithm 2). We propose two implementations of the MUTATEENVCONFIG function: the first one randomly changes a parameter of the given environment configuration, choosing among all parameters with equal probability. The second one only changes the parameter of an environment configuration that contributes the most to the failure prediction. The idea is that some parameters are more *critical* than others when considering how they affect failure prediction. Hence, changing them is more beneficial for generating failure environment configurations than changing the other parameters. In particular, we propose to use the *saliency* method to compute *input attribution* [56], i.e., to determine how much an input influences a prediction made by a neural network (our classifier). Given an environment configuration and the classifier, the saliency method computes the input gradient, i.e., the partial derivatives over all the individual parameters of the environment configuration. The absolute value of each gradient indicates which input parameter is more critical to the failure prediction and its sign indicates the direction (i.e., positive or negative) of change. In our saliency-guided implementation of the MUTATEENVCONFIG function we take the output computed by saliency and change the parameter in the environment configuration whose corresponding gradient is the highest one; the direction of change, either positive or negative, depends on the sign of the gradient.

For instance, let us consider the environment configuration: $e = [20, 0.0, \{3, 5, 6, 8, 13, 19\}, (0.0, 0.0)]$, i.e., an environment configuration of the Parking environment. The input attribution for this environment configuration is an array of the same size as the input, as it contains the partial derivatives over each input. Let us suppose that the highest value in the array is in second position, i.e., the position corresponding to the parameter `head_ego`, and that such value is positive. This means that the parameter `head_ego` is the most critical one affecting the failure prediction of the classifier and that changing it in the positive direction, i.e., increasing it, will also increase the failure prediction for the resulting environment configuration.

3.3.2 Crossover Function. The crossover function is specific to the genetic algorithm (Line 12 in Algorithm 2). We propose a single-point crossover implementation where, given two environment configurations, the cut point is determined randomly; then the elements in the two configurations before and after the cut point are swapped. We chose this simple implementation for crossover because it can be applied to configurations of any case study with little

to no modifications. Custom implementations that take into account the specific features of each case study remain possible in our implementation.

For instance, let us suppose that $e_1 = [20, 0.0, \{3, 5, 6, 8, 13, 19\}, (0.0, 0.0)]$ and $e_2 = [15, 0.5, \{1, 3, 9\}, (-1.0, 7.5)]$ are two Parking environment configurations. The cut point is computed based on the number of parameters in the environment configuration, which in the case of Parking is equal to four (i.e., `goal_lane`, `head_ego`, `pvehicles` and `pos_ego`). Let us suppose that the cut point is at first position. The two environment configurations after crossover will be as follows: $ce_1 = [20, 0.5, \{1, 3, 9\}, (-1.0, 7.5)]$ and $ce_2 = [15, 0.0, \{3, 5, 6, 8, 13, 19\}, (0.0, 0.0)]$.

3.4 Implementation

We implemented our approach in a Python tool called INDAGO (Latin for "search") which is publicly available [7]. The DRL agents are implemented by the *stable-baselines* [47] library and we use *Pytorch* [44] and *scikit-learn* [45] to implement the training and inference of the classifier. The *saliency* method is implemented by the *captum* library [33].

4 EMPIRICAL EVALUATION

We consider the following research questions:

RQ_1 (Effectiveness): *What is the effectiveness of the proposed approach? Can it generate failure environment configurations for the DRL agent under test?*

RQ_2 (Comparison): *How does INDAGO compare against the random baseline? How does it compare against the state-of-the-art sampling approach?*

RQ_3 (Hyperparameters): *What is the impact of the key hyperparameters of INDAGO?*

RQ_1 evaluates the effectiveness of INDAGO, i.e., its capability to generate failure environment configurations for the DRL agent under test.

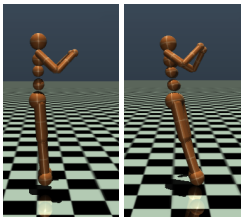
RQ_2 compares INDAGO with the random baseline and the state-of-the-art sampling approach [65], both w.r.t. the number of generated failure environment configurations and their diversity.

RQ_3 investigates the key hyperparameters of INDAGO both w.r.t. the number of failure environment configurations that are generated by each hyperparameter setting and their diversity. We, first of all, analyze the choice of the search algorithm, i.e., hill climbing vs genetic algorithm. Second, since our approach can work both with and without the provisioning of failure seeds (e_f and E_f are optional parameters in Algorithm 1 and Algorithm 2, respectively), we want to investigate which of the two seed strategies is more convenient to use. Third, when mutating the environment configurations INDAGO can either choose them randomly or it can focus on those that have the highest influence on the failure prediction, as determined by the saliency method. Hence, we want to evaluate the impact of the mutation strategy when using INDAGO.

4.1 Case Studies

Parking. We already introduced the first environment, i.e., Parking [37], in Section 2, where we also describe the representation of the environment configuration (see Figure 1). Such environment has been used in several studies, especially to evaluate the capabilities of DRL agents [10, 12, 17, 18, 59, 73, 76].

The encoding adopted to train a classifier for failure prediction consists of an array of 24 elements where the first two values are for the two single-value parameters of the environment configuration (i.e., `goal_lane` and `head_ego`), followed by 20 values corresponding to the one-hot encoding of the `pvehicles` parameter (i.e., each position has a




```
{
  "env_configuration": {
    "joints_pos": [-0.03, 0.02, 1.42, 0.98, .., 0.03],
    "joints_vel": [0.02, -0.01, 0.01, -0.03, .., -0.02]
  }
}
```

Fig. 3. Two initial configurations of the *Humanoid* environment in the *Mujoco* simulator [64]. The textual representation (on the right) corresponds to the environment configuration e_1.

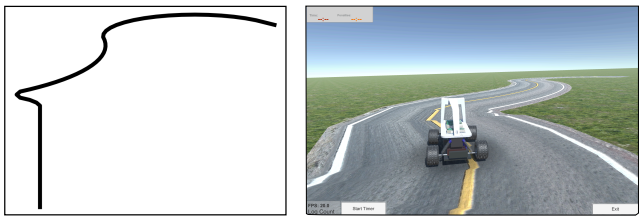
value of 1 only when there is a vehicle in the respective spot; a value of 0 otherwise) and finally the two values of the `pos_ego` parameter.

Regarding the mutation operators, the `goal_lane` parameter value is increased or decreased, with equal probability, and the change amount is an integer in the interval $[1, 20]$. The `heading_ego` parameter value is increased or decreased, with equal probability, and the change amount is a random floating point number in the interval $[0, 1)$. The coordinates of the `pos_ego` parameter tuple are increased or decreased, with equal probability, by a random float in the interval $[0, 1]$. For what concerns the `pvehicles` parameter, with equal probability parked cars in the `pvehicles` set are added/removed or the parked car occupancy indexes are mutated. In the former case a certain number of indices is selected at random for adding or removing parked cars with equal probability. In the latter case a certain number of parked car indices is selected at random to be mutated, i.e., increased or decreased by a random integer in the interval $[1, 20]$. Moreover, the crossover operator considers the parameters of the Parking environment configurations as unique features. An example of crossover for the Parking environment is in Section 3.3.2.

Humanoid. The second environment we consider is *Humanoid*, built using the *Mujoco* simulator [64]. Mujoco is a very popular simulator in the DRL community, especially to benchmark DRL algorithms in continuous control tasks. The 3D physics simulator has different pre-built environments, with Humanoid being one of the most challenging environments for DRL algorithms [71].

In Humanoid the DRL agent needs to control a bipedal robot in a 3D space and make it walk on a smooth surface. Each environment configuration is composed of two arrays, i.e., `joints_pos` and `joints_vel`. The former has size 24 and consists of the positions and rotations of the joints of the robot while the latter has size 23 and consists of the linear and angular velocities of those joints (more information are available in the OpenAI wiki [11]). All the angles are in radians, represented as floating point numbers. Figure 3 shows two configurations of the Humanoid environment; we can see that the initial joints positions can be altered, forming different initial configurations for the Humanoid robot.

The state space of the DRL agent is an array of size 376, composed of joint positions, angles and relative velocities, plus other components as center of mass inertia and velocity. The action space is composed of 17 elements that are the degrees of freedom of the robot, i.e., the actuated joints of the robot. The reward function encourages the DRL agent to walk as fast as possible plus a bonus for each timestep. An episode terminates when the abdomen y coordinate of the robot goes out of the range $(1, 2)$, which indicates that the robot has fallen down, or when a timeout expires (we set such timeout to 300 timesteps). When the robot *falls* we deem the episode unsuccessful; on the other hand, when the timeout expires we consider the episode successful. We changed the environment interface to be *configurable* such that the parameters `joints_pos` and `joints_vel` can be set programmatically at the beginning of each episode. The original implementation initializes `joints_pos` to $[0, 0, 1.4, 1, 0, \ldots, 0]$ and `joints_vel` to all zeros; then, to generate a



```
{
  "env_configuration": {
    "track": [
      (S,18), (S,12), (S,2), (DY,15.2),
      (L,3), (DY,30.5), (R,3), (DY,2.9),
      (L,18), (DY,8.4), (R,17), (S,2)
    ]
  }
}
```

Fig. 4. An initial configuration of the *Driving* environment in the *DonkeyCar* simulator [63].

new environment configuration, each value in both arrays can be changed by adding or subtracting a small quantity m (which is set to 0.03). We followed the original implementation to define *validity*, i.e., we consider an environment configuration to be valid only if the values of its parameters, i.e., `joints_pos` and `joints_vel`, are within the interval $[-m, m]$ w.r.t. the initial values of such parameters. For instance, a valid environment configuration can have the third value of its `joints_pos` array within the interval $[1.4 - m, 1.4 + m]$, since that value is initialized to 1.4.

The encoding we adopt as input to the classifier is the concatenation of the two arrays that define the environment configuration, i.e., `joints_pos` and `joints_vel`. Regarding the mutation operators, they are defined in the same way for both parameters, i.e., once an index of either `joints_pos` or `joints_vel` is chosen at random, the value at that index is either decreased or increased, with equal probability, by a random floating point quantity in the interval $[-m, m]$. Moreover, the crossover operator is based on the parameters of the Humanoid environment, as in Parking. In Humanoid there are only two parameters, i.e., `joints_pos` and `joints_vel`; therefore, during crossover, the entire `joints_pos` and `joints_vel` vectors are swapped between two different environment configurations.

Driving. The third environment we consider is *Driving*, built using the *DonkeyCar* simulator [63]. This platform has been used in previous works to train and test self-driving car software both based on supervised learning and reinforcement learning [58, 68, 69, 75, 77].

In this environment the DRL agent controls a car which drives along a track. The configuration determines the shape of the track in which the car drives (see Figure 4). The track is represented as a list of 12 pairs, where each pair consists of two elements, i.e., a command and a value (see the right side of Figure 4). The possible commands are `S` which indicates a straight line, `R` which indicates a right curve, `L` which indicates a left curve and `DY` which signals the beginning of a curve. The value associated to each command represents the number of road units, except for the `DY` command, where the associated value represents the individual angle of rotation for each road unit. For example, the sequence of commands `[(S,2), (DY,15.2), (L,3)]` (i.e., the first curve of the track represented by the environment configuration in Figure 4) instructs the road engine to build a straight road for two road units followed by a left curve which is three road units long, for a total of 45.6 degrees (i.e., 3×15.2).

The state space of the DRL agent is an RGB image of size (160, 120) taken by the front camera of the car, while the action space is two-dimensional, i.e., steering angle and throttle. The DRL agent receives a small negative reward per timestep plus a positive reward every time it crosses a waypoint in the track, minus a penalty related to the *cross track error*, i.e., the distance between the center of mass of the car and the center of the track. The first component of the reward encourages the DRL agent to go as fast as possible, the second privileges the progress on the track and the third forces the DRL agent to drive as close to the center of the track as possible. Moreover, the DRL agent receives a large

negative reward when it goes off road. In such cases the episode terminates unsuccessfully, whereas when the DRL agent crosses the end line of the track, the episode is deemed successful.

We modified the *DonkeyCar* simulator to make it *configurable*, such that at the beginning of each episode the track passed as input can be instantiated in the simulator. Regarding *validity*, there are a number of constraints that need to be respected in order for the track to be valid. First, each track should start and end with an `S` command. Moreover, after a `DY` command there must be either an `L` or `R` command. Afterwards, the track must not contain loops (i.e., self-intersections), it must not have very sharp turns (i.e., with a rotation angle $> 170°$) and it must have at least 3 curves, one of which must be with a rotation angle of at least $120°$. The last two constraints ensures that the generated tracks are non-trivial.

The encoding we adopt as input to the classifier is the concatenation of two arrays, i.e., the array of all commands, where each command is given a unique integer identifier, and the array of all values. Regarding the mutation operators, we define one for commands and one for values and, when analyzing a command-value pair, we either change the command or the value with equal probability. The change command operator can only change an `L` command to an `R` command or vice-versa. The change value operator first analyzes the associated command and, if it is a `DY` command, it either increases or decreases, with equal probability, the current value by a random floating point number in the interval $[0, 50]$. Otherwise it increases or decreases, with equal probability, the current value by a random integer number in the interval $[1, 20]$. Moreover, the crossover operator considers the Driving configuration as a list of command-value pairs. The cut point is a value in the interval $[1, 12]$ such that after crossover the resulting Driving environment configurations have command-value pairs coming from both parent environment configurations. We customized the crossover operator for this environment by retrying crossover a certain number of times, until either both resulting configurations are valid or the maximum counter is reached.

Table 1. Case studies training metrics

	DRL algorithm	# tr. timesteps	# tr. episodes	tr. time (min)	# tr. failures	% failure class
Parking	HER [3] + TQC [34]	$200k$	8789	395	529	6.0
Humanoid	TQC [34]	$1500k$	6965	888	3342	48.0
Driving	SAC [20]	$1000k$	11274	1257	661	5.8

4.2 Experimental Setup

4.2.1 Procedure. We trained the DRL agents in each environment using the hyperparameters recommended by Raffin et al. [46, 47]. Column 1 of Table 1 shows the algorithms we used for each environment. Column 2 shows the number of training timesteps which corresponds to the number of episodes shown in Column 3. We trained each DRL agent until convergence, i.e., until the success rate in the last 100 episodes was 100%. The average training time is $\approx$ 14 hours, with the DRL agent in the Parking environment being the cheapest to train ($\approx$ 6 hours) and the DRL agent in the Driving environment being the most expensive ($\approx$ 21 hours). Column 5 shows the number of unsuccessful episodes (i.e., failures) recorded during training and Column 6 shows the percentage of failures w.r.t. the number successful episodes. We can see that, considering all the episodes, the DRL agents in the Parking and Driving environments experienced few

failures during training (amounting to 5.9% of the total number of episodes), while the DRL agent in the Humanoid environment failed much more often. However, most of the failures (i.e., 1641) are within the first 10% of the episodes.

Failure Predictor Training. To compensate for the unbalanced dataset when training the failure classifier, we compute a weight vector W as follows: given N datapoints, the array of class targets Y, with C being the number of classes, $W = N/(C \cdot \texttt{hist}(Y))$ where $\texttt{hist}(Y)$ is a function that outputs an array with size C indicating the number of datapoints for each class. Such formula, gives a higher weight to the underrepresented classes [32].

We chose a multi-layer perceptron as the classifier architecture. The reason is twofold: first, the size of the available training data is small (at most $10k$ examples for both failure and non-failure classes, see Table 1). Secondly, our environment configurations are small size one-dimensional feature vectors (24 for Parking, 47 for Humanoid and 24 for Driving). Furthermore, since the output of the classifier is used as a fitness function during testing, adopting complex models to process simple inputs would be inefficient and prone to overfitting.

When training the failure predictor, there are two important hyperparameters to consider. The first one is the amount of initial interaction data to filter out. Indeed, at the beginning of training the agent carries out random actions and, as a consequence, it often fails regardless of the environment configuration. This means that the *earliest* failures are not useful to predict the failures of the DRL agent under test. The other hyperparameter that depends on the case study is the number of hidden layers of the multi-layer perceptron.

In our experiments we considered nine levels of filtering (i.e., 5, 10–80), where, for instance, 5% filtering means that the first 5% of the environment configurations are not considered for training the classifier. Moreover, we chose four different number of hidden layers, i.e., from 1 to 4, where each hidden layer is composed of 32 units followed by a batch normalization layer [27] and a dropout layer [57] with probability of dropout of 0.5 (except when the network has only one hidden layer). For each pair filter-layers we trained the classifier ten times (with all the other training hyperparameters, e.g., learning rate, fixed), every time with a different random seed. We used a validation set formed using 20% of the data to save the best model during training which we evaluated on a held-out test set. We built such test set by choosing a filtering level of 5% for each case study and by selecting 10% of the data at random. During hyperparameter tuning we measured precision and recall of the classifier considering the failure class as the positive class. We deem precision more important than recall since in order to guide failure search it is more important for the classifier to be as precise as possible (i.e., few false positives) even at the cost of missing some failures (i.e., false negatives). Specifically, we chose the best classifier model by looking at the results of the ten training runs for each filter-layers pair and we selected the model with the highest precision and with a recall of at least 10%, in order not to miss too many failures.

Baselines. We compared Indago with two approaches. The first approach is the *random* baseline, where environment configurations are generated at random. Such baseline is useful to understand whether the proposed approach is able to outperform the state-of-the-practice in testing DRL agents [43, 65]. The second approach is the state-of-the-art *sampling* approach by Uesato et al. [65]. This simple approach consists of generating a large initial set of M environment configurations at random and choosing the one that, according to the classifier f, has the highest failure prediction.

Indago. We considered hill climbing and genetic algorithm each with four different settings. In the first one, the seed environment configurations to evolve are generated at random (hc_{rnd} and ga_{rnd}). In the second, we used hill climbing and genetic algorithm to evolve failure environment configurations (hc_{fail} and ga_{fail}). Regarding the settings with the failure seeds, we always filtered out the initial 30% of the environment configurations, in order not to include failures that are likely not representative of the failures of the DRL agent under test. Furthermore, we considered hill climbing and genetic algorithm guided by saliency, evolving both random ($\text{hc}_{\text{sal+rnd}}$ and $\text{ga}_{\text{sal+rnd}}$) and failure ($\text{hc}_{\text{sal+fail}}$ and

$ga_{sal+fail}$) environment configurations. For INDAGO and the sampling approach we considered a fixed search budget B per environment configuration. In particular we chose $B = 5$ seconds for Parking and Humanoid, while $B = 30$ seconds for Driving, as from preliminary experiments we observed that such budget was enough to reach fitness convergence for all approaches.

During failure search, we generated $T = 100$ environment configurations for each case study and each approach, and we evaluated the respective DRL agent in each environment configuration. Since Humanoid and Driving simulators are non-deterministic we evaluated the respective DRL agents in each environment configuration ten times. Moreover, we repeated the experiments ten times for each failure search approach in order to cope with the intrinsic randomness of the approaches. Overall, we have 3 case studies, 10 techniques (8 settings of INDAGO, including hill climbing vs genetic algorithm, random vs failure seed, two mutation strategies (i.e., random and saliency), plus the random and the sampling approaches), each generating 100 environment configurations (each repeated 10 times in Humanoid and Driving). Finally, each technique is executed 10 times for a total of $210k$ simulations.

4.2.2 Metrics. In order to assess the *effectiveness* of our failure search approach (RQ_1) and to compare it with the baselines (RQ_2), we measured the number of failures each approach triggers given the number of environment configurations to generate (i.e., T) and the fixed search budget B per environment configuration. In non-deterministic environments, as Humanoid and Driving, we measured the failure probability of each environment configuration out of ten runs and we considered an environment configuration to cause a failure if its failure probability is > 0.5. As each failure search approach is executed ten times, we determined whether there is a statistical difference between the failures triggered by each pair of failure search approaches (including the 8 settings of INDAGO) by computing the Mann-Whitney U Test [4, 40]. To measure the effect size, we computed the Vargha Delaney metric $\hat{A}_{12}$ [67].

To compare the competing approaches (RQ_2) we also considered two types of diversity regarding the failures generated by each approach, namely input and output diversity. For *input diversity*, we first clustered all the environment configurations that caused a failure, across all considered failure search techniques, obtaining a single partition of all such failing environment configurations. Similarly, for *output diversity*, we clustered the *trajectories* (i.e., positions over time) of the DRL agents on the failure inducing environment configurations. In Parking and Driving we considered the trajectories of the vehicle while performing the task, i.e., parking the vehicle in the former case and driving along the track in the latter. In Humanoid, we considered the trajectory of the height of the robot, which determines whether the robot falls or not. Since trajectories can have different lengths, we extended them with zero-padding to the maximum observed length. For both input and output diversity we used the *k-means* clustering algorithm [5] and we determined the optimal number of clusters k^* by performing *silhouette analysis* [52], optimizing the balance between density and separation of the clusters. In our experiments we varied the number of clusters k between two and the number of inputs (i.e., environment configurations or trajectories) to be clustered, and computed the silhouette score for each candidate. We increased k^* to a higher value only if the new silhouette was at least 20% greater than the best silhouette score found so far, in order to filter out random fluctuations of the silhouette score.

After applying clustering, we computed two diversity metrics for each failure search approach, namely *coverage* and *entropy*. Given a failure search approach A and the optimal number of clusters k^*, the coverage of the clusters for the approach A is defined as:

$$C_A = \frac{\sum_{i=1}^{k^*} c_A(i)}{k^*} \tag{1}$$

where the function $c_A(i) : \mathbb{Z}^+ \rightarrow 0|1$ determines whether a certain cluster labeled by i is *covered* by the failure search approach A, i.e., whether at least one failure generated by the approach A belongs to the cluster with label i.

The second metric, *entropy*, measures how uniformly the different failures are distributed across the clusters. Given the number of failures triggered by the failure search approach A in the i-th cluster, $F_A(i)$, the probability of finding a failure generated by the approach A in cluster i is given by $p_i^A = F_A(i)/\sum_i F_A(i)$ and entropy is defined as:

$$H_A = \sum_{i=1}^{k^*} p_i^A \cdot \log_2(p_i^A) \tag{2}$$

In particular, entropy is zero when all failures are concentrated in one cluster, while it is maximum and equal to $\log_2(k^*)$ when failures are distributed uniformly across all the clusters. Hence, we can use $\overline{H}_A = H_A/\log_2(k^*)$ as a normalized measure of entropy ranging between 0 and 1. We ran clustering ten times to account for randomness and took the average of coverage and entropy across the ten runs. Then, we compared statistically the coverage and entropy averages for each failure search approach across the respective ten runs by computing the Mann-Whitney U Test and the Vargha-Delaney effect size.

To evaluate the impact of algorithm, seed and mutation strategies on INDAGO (RQ_3), we used the same metrics: number of failures and input/output diversity, the latter quantified with coverage and entropy.

4.3 Results

Table 2. Number of failures and input/output diversity measured by cluster coverage or entropy. Values represent the average among ten runs. Bold faced values indicate a statistically significant difference between INDAGO and sampling. Underlined values indicate a large effect size. Highlighted values summarize the best approach in terms of number of failures and in terms of diversity, across all case studies.

	PARKING					HUMANOID					DRIVING					AVERAGE				
		Diversity					Diversity					Diversity					Diversity			
		Input		Output			Input		Output			Input		Output			Input		Output	
	# Failures	Coverage (%)	Entropy (%)	Coverage (%)	Entropy (%)	# Failures	Coverage (%)	Entropy (%)	Coverage (%)	Entropy (%)	# Failures	Coverage (%)	Entropy (%)	Coverage (%)	Entropy (%)	# Failures	Coverage (%)	Entropy (%)	Coverage (%)	Entropy (%)
Baselines																				
random	1	55.00	18.37	31.94	12.13	1	16.08	0.00	30.00	0.00	1	10.00	0.00	10.00	0.00	1	27.03	6.12	23.98	4.04
sampling	13	50.00	0.00	41.10	22.91	1	14.72	12.62	25.00	0.00	5	80.00	54.97	75.00	37.50	6	48.24	22.53	47.03	20.14
INDAGO																				
hc_{rnd}	4	**75.00**	**41.54**	44.72	26.90	1	29.57	19.03	40.00	0.00	3	65.00	23.80	56.00	10.11	3	56.52	28.12	46.91	12.34
hc_{fail}	5	**85.00**	**64.87**	45.45	33.95	2	27.37	18.00	**50.00**	16.23	4	90.00	73.61	**95.00**	77.20	4	67.46	52.16	63.84	42.46
$hc_{sal+rnd}$	6	**90.00**	**57.37**	48.01	31.59	**4**	**59.73**	**61.32**	**95.00**	**77.75**	2	60.00	28.37	60.00	26.48	4	69.91	49.02	67.67	45.27
$hc_{sal+fail}$	13	**100.00**	**93.23**	66.03	**58.91**	**3**	**56.98**	**55.66**	**80.00**	**53.70**	**11**	**100.00**	83.80	**100.00**	**86.69**	9	85.66	77.56	82.01	66.43
ga_{rnd}	6	55.00	5.44	43.39	25.68	2	35.95	31.21	**55.00**	**29.18**	6	90.00	72.00	80.00	52.86	5	60.32	36.21	59.46	35.91
ga_{fail}	11	50.00	0.00	46.55	29.64	2	24.99	12.90	**63.50**	**43.48**	**22**	90.00	85.63	70.00	19.16	12	55.00	32.84	60.02	30.76
$ga_{sal+rnd}$	8	60.00	10.31	47.73	29.77	1	23.87	17.45	40.00	19.52	5	83.00	59.70	60.00	17.22	5	55.62	29.15	49.24	22.17
$ga_{sal+fail}$	**27**	55.00	2.01	51.95	29.22	**4**	**38.11**	24.99	**75.00**	**44.04**	**42**	**100.00**	**97.61**	70.00	12.73	24	64.37	41.53	65.65	28.66

Effectiveness (RQ_1). Column *# Failures* in Table 2 shows the average number of failure environment configurations triggered by each failure search approach out of ten runs for each case study. Considering INDAGO in all its settings, such number is between 4 and 27 in the Parking environment, between 1 and 4 in the Humanoid environment and between 2 and 42 in the Driving environment.

> **RQ_1**: Overall, INDAGO successfully challenged the DRL agent under test in all case studies, by generating a significant number of failure environment configurations.

Comparison (RQ_2). In Table 2, bold values indicate a statistically significant difference (at level $\alpha = 0.05$) between INDAGO and sampling; values are underlined when the effect size is large. In particular, in Parking the best approach is $ga_{sal+fail}$ that is able to generate more failures than other approaches (i.e., 27 on average) and the difference w.r.t. the sampling approach (i.e., 13 failures on average) is statistically significant with a large effect size. In Humanoid, several settings of INDAGO are significantly better than sampling (which exposes only 1 failure on average). In Driving, $ga_{sal+fail}$ and $hc_{sal+fail}$ are the best approaches with, on average, 42 and 11 failures respectively. Their difference w.r.t. the sampling approach (which exposes 5 failures on average) is statistically significant and the effect size is large.

Regarding the comparison with the random approach, in all case studies, classifier-based approaches (i.e., sampling and INDAGO) found significantly more failures with a large effect size. Hence, our empirical results show that the failures experienced during training are indeed related to and informative of the failures of the DRL agent under test.

The macro-columns *Diversity* in Table 2 show the average out of ten runs of the *Coverage* and *Entropy* metrics regarding input and output diversity for each failure search approach. Although in Parking the $ga_{sal+fail}$ approach generates more failures than sampling, the two approaches are comparable in terms of input and output diversity (both considering coverage and entropy). On the other hand, the $hc_{sal+fail}$ approach is able to generate inputs that are both significantly different than those of sampling (i.e., higher coverage) and better distributed among clusters (i.e., higher entropy). Output coverage and entropy values are higher for $hc_{sal+fail}$ than those of sampling (66.03% and 58.91% vs 41.10% and 22.91% respectively), with output coverage being comparable and output entropy being statistically better with a large effect size. This means that, although $hc_{sal+fail}$ and sampling generated the same number of failures (i.e., 13 on average), the failures produced by $hc_{sal+fail}$ exercise more diverse behaviours of the DRL agent under test.

For what concerns Humanoid, the two best approaches in terms of input and output diversity are settings of INDAGO ($hc_{sal+rnd}$ and $hc_{sal+fail}$) both considering coverage and entropy (the $ga_{sal+fail}$ setting is comparable to them). All of them are significantly better than sampling with a large effect size, with the exception of $ga_{sal+fail}$, whose input entropy (24.99%) is not significantly different from that of sampling (12.62%). In Driving, the best approaches in terms of input and output diversity are $hc_{sal+fail}$ and $ga_{sal+fail}$, the former significantly better than sampling on input coverage and output diversity and the latter on input diversity.

When comparing INDAGO with the random approach in terms of diversity, most of the INDAGO settings, especially those with the saliency-based mutations, are significantly better than random with a large effect size in all case studies.

> **RQ_2**: Overall, both considering the number of failures triggered and their input and output diversity, INDAGO generates significantly more failures than both the state-of-the-art sampling approach and the random approach in all case studies. Moreover, such failures are significantly more diverse.

Table 3. Impact of the seed strategy. In each row, the three case studies (Parking / Humanoid / Driving) are separated by a forward slash "/" symbol. An "F" symbol indicates a statistically significant difference in favor of the approach with the failure seed; a dash "-' symbol indicates that the two approaches are comparable and an "R" symbol (missing in the table) would indicate a statistically significant difference in favor of the approach with the random seed.

	Parking / Humanoid / Driving				
		Diversity			
		Input		Output	
	# Failures	Coverage	Entropy	Coverage	Entropy
hc_{rnd} *vs* hc_{fail}	- / - / F	- / - / F	- / - / F	- / - / F	- / - / F
ga_{rnd} *vs* ga_{fail}	F / - / F	- / - / -	- / - / -	- / - / -	- / - / -
$hc_{sal+rnd}$ *vs* $hc_{sal+fail}$	F / - / F	- / - / F	F / - / F	- / - / F	F / - / F
$ga_{sal+rnd}$ *vs* $ga_{sal+fail}$	F / F / F	- / - / F	- / - / F	- / F / -	- / - / -
rnd *vs* fail	0/12 *vs* **8/12**	0/12 *vs* **3/12**	0/12 *vs* **4/12**	0/12 *vs* **3/12**	0/12 *vs* **3/12**

Hyperparameters (RQ_3). The macro-columns *Average* in Table 2 show that, on average and considering all case studies, $hc_{sal+fail}$ generates failures that are the most diverse both in terms of input and output diversity (highlighted in the table). Specifically, $hc_{sal+fail}$ has the best input coverage (85.66% vs 69.91% of the second best $hc_{sal+rnd}$), the best input entropy (77.56% vs 52.16% of the second best hc_{fail}), the best output coverage (82.01% vs 67.67% of the second best $hc_{sal+rnd}$) and the best output entropy (66.43% vs 45.27% of the second best $hc_{sal+rnd}$).

On the other hand, $ga_{sal+fail}$ is the setting of Indago that generates the highest number of failures (i.e., 24 on average, highlighted in the table). However, such failures cover less clusters and have a lower entropy than the failures generated by $hc_{sal+fail}$. Therefore, considering the number of failures and their diversity, $hc_{sal+fail}$ is the preferable Indago setting.

Across all case studies, the settings of Indago that use the saliency-based mutations are more effective than their counterparts, i.e., the difference is always statistically significant with a large effect size, except for Humanoid, where $ga_{sal+fail}$ is comparable to ga_{rnd}. Overall, this shows that the guidance offered by the saliency-based mutation operator is effective at finding failure environment configurations.

In Table 3 we report a further comparison between the different settings of Indago, focused on the impact of the seed strategy (random, *rnd*, vs failure, *fail*). In each cell of the table the three case studies are separated by a forward slash "/" symbol. The symbol "F" (respectively "R") indicates that failure seeding (respectively random seeding) is statistically better than random (respectively failure) seeding. A dash symbol indicates no statistically significant difference. In terms of number of failures we can see that, in most cases (i.e., 8/12) the settings with the failure seeds are significantly better than their random seeds counterparts. In particular, the $ga_{sal+fail}$ setting is significantly better than the $ga_{sal+rnd}$ setting in all case studies (while, for instance, the $hc_{sal+fail}$ is significantly better than $hc_{sal+rnd}$ in Parking and Driving but not in Humanoid). From the point of view of diversity, the settings of Indago with the failure seeds are mostly comparable to the settings with the random seeds, except for Driving, where the failure seeds are critical to generate more diverse failures. Random seeding is never a better choice (indeed the "R" symbol is completely missing in the table), across all settings of Indago and across all three case studies.

RQ$_3$: Overall, the settings of INDAGO with the saliency mutation strategy and failure seeds, in all case studies, are either comparable or significantly better than their random counterparts. Between hill climbing and genetic algorithm, the former is preferable because it generates more diverse failure scenarios. The latter might be considered when the number of exposed failures is important, regardless of their diversity.

5 DISCUSSION

5.1 Solvability of the Failures

For each approach, we resumed training of the three DRL agents under test by feeding all the failure environment configurations found by each approach in every run of the environment generation process. In every case study and for each approach, we found that the given DRL agent could learn how to successfully terminate the respective episodes by performing some additional training. This shows that the failure environment configurations generated by all approaches, and indeed by INDAGO, are *solvable* by the DRL agents under test.

This also indicates that, although a generated environment configuration can be challenging for the DRL agent under test, there exists a sequence of actions that let the DRL agent solve the task successfully. For instance, after additional training, the DRL agent in the Parking environment is always able to find trajectories for the vehicle to reach the target spot, which is instead missed by the original DRL agent under test. Similarly, in the Driving environment, the generated failure environment configurations do not induce track shapes that are beyond the mechanical capabilities of the vehicle. Regarding Humanoid, the initial positions of the joints as well as their velocities, resulting from the generated failure environment configurations, do not prevent the DRL agent to control and regain the balance of the robot.

Solvability of the failure environment configurations generated by INDAGO shows that such environment configurations represent real weaknesses of the DRL agent under test, which could realistically occur during the operation of the DRL agent in production.

5.2 Training Failures

A simple way to test a DRL agent is to replay the training failures with the exception of the earliest ones (e.g., by filtering out the first 30% of the failure environment configurations). However, this has two major downsides: (1) the DRL agent under test may have adapted to failure environment configurations in which weaker versions of itself failed, despite the exclusion of early failures; (2) such an approach would not be generative and by design it can only replay existing failure environment configurations. Generative approaches like $\text{hc}_{\text{sal+fail}}$ produce diverse and potentially unlimited challenging inputs that expose the limitations of the DRL agent under test and that can be used for further training [16].

For the sake of completeness, we replayed at testing time the failure environment configurations that happened during the DRL training process and we compared them with the failures generated by $\text{hc}_{\text{sal+fail}}$. In particular, we clustered the DRL agents trajectories associated with those failures, following the same process described in Section 4.2.2. The top part of Figure 5 shows a 2D t-SNE [66] projection of the failure trajectories of the three DRL agents under all failure environment configurations in each case study. Cluster centroids are indicated with the letter **C**. The figure shows that in Humanoid the trajectories associated with the failure environment configurations discovered by INDAGO, cover more clusters than the trajectories associated with *training* failure environment configurations. In Parking and Driving the two classes of trajectories are complementary since they cover the same clusters but with a different

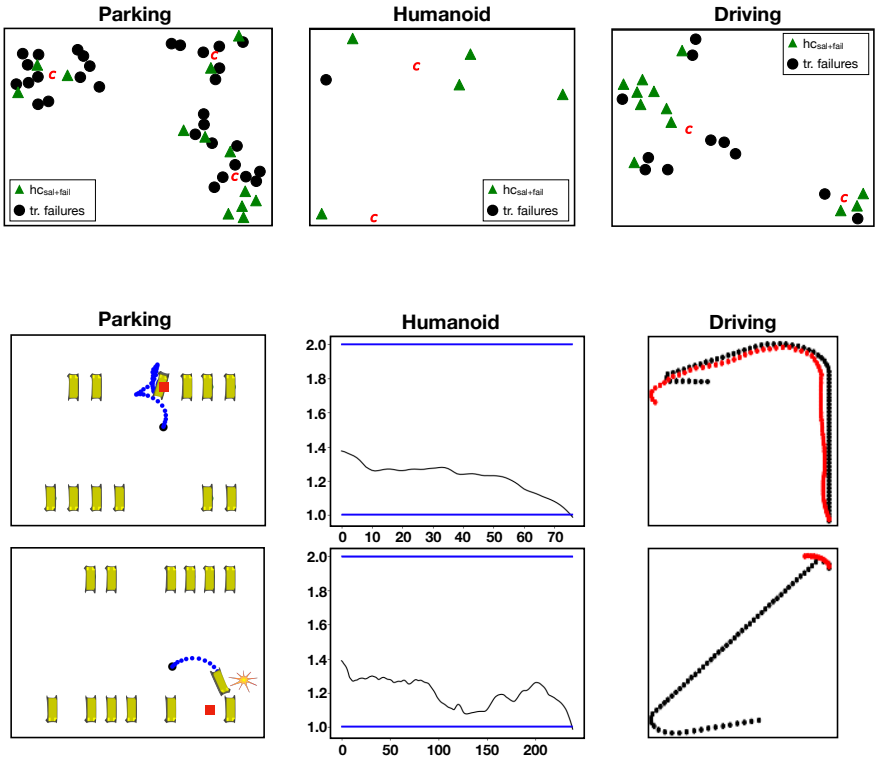


Fig. 5. Qualitative analysis: t-SNE projection in 2D of trajectories associated with $hc_{sal+fail}$ (△) and training failures (◯) configurations (top); representation of different trajectories associated with $hc_{sal+fail}$ failure configurations (bottom)

intra-cluster distribution, showing that the generative approach of INDAGO can explore new clusters or new regions within a cluster.

5.3 Qualitative Analysis of Failures

The bottom part of Figure 5 shows the failure trajectories associated to failure environment configurations generated by INDAGO. For all case studies we selected two failure trajectories belonging to different clusters representing two different failure modes.

In Parking, the first failure mode (at the top) consists of the DRL agent not being able to park the vehicle in the target spot with the right heading given a certain amount of time; the second failure mode (at the bottom) is when the DRL agent does not complete the parking maneuver due to a crash with another vehicle parked beside the target spot.

In Humanoid, the abdomen y coordinate of the robot should be among the two horizontal lines shown in Figure 5, otherwise the robot falls and the episode terminates unsuccessfully. The two failure trajectories associated with INDAGO failure environment configurations are different: the one at the top shows a trajectory that monotonically goes down in a relatively short amount of time (i.e., just above 70 timesteps). The trajectory at the bottom is more noisy indicating that the DRL agent under test is more uncertain. Indeed, the DRL agent is able to recover the partial fall around the middle of the episode (i.e., around 120 timesteps) while eventually failing after ≈ 200 timesteps.

In Driving, the first failure mode shows the DRL agent failing to drive the last curve of the track, while the trajectory at the bottom represents a track with a difficult left curve at the beginning.

5.4 Discrete vs Continuous Configurations

In the Parking environment, the number of failures found by the sampling approach are equivalent to the failures found by the best INDAGO setting, i.e., $hc_{sal+fail}$ (i.e., 13 on average), although the failures generated by $hc_{sal+fail}$ have a significantly higher input diversity and output entropy. Parking environment configurations are composed of several discrete and only a few continuous parameters, and they are subject to few constraints. As a consequence, sampling Parking environment configurations at random is efficient, as a large number of candidate environment configurations can be generated within the given search budget, among which it is more likely to find environment configurations where the failure prediction is high.

Also in the Humanoid environment there are few validity constraints, hence it is efficient to sample environment configurations at random. However, the space of environment configurations is larger than in Parking (almost twice as much, i.e., 47 vs 24 parameters) and all parameters are in the continuous domain. As a consequence, finding challenging environment configurations by sampling at random is not effective. Indeed, our results show that the sampling approach generates, on average, 1 failure, while $hc_{sal+fail}$ generates 3 failures (a 200% increase), with a significantly higher input and output diversity. This shows that INDAGO works much better than random sampling when environment configurations have continuous parameters.

The Driving environment offers yet another perspective. Indeed, similarly to Parking, Driving environment configurations are mostly composed of discrete parameters but, on the other hand, such environment configurations are subject to complex constraints. Therefore, sampling Driving environment configurations at random is not efficient, contrary to modifying existing environment configurations. Indeed, the sampling approach generates, on average, 5 failures, while $hc_{sal+fail}$ generates 11 failures (i.e., a 120% increase), with a significantly higher input and output diversity. Although, the Driving environment configurations have mostly discrete parameters our search-based approach is more effective than sampling, since it is able to use the search budget more efficiently by modifying existing environment configurations which satisfy the complex constraints that hold in this environment.

5.5 Threats to Validity

5.5.1 Internal Validity. A threat to internal validity may come from an unfair comparison of the considered approaches. We gave the same search budget to all approaches and we generated the same number of environment configurations. Moreover, we considered the same DRL agents for each approach and executed the tests on the same environments.

5.5.2 External Validity. Using a limited number of subjects poses an external validity threat, in terms of generalizability of our results. To mitigate such threat, we chose three environments which are widely used in the DRL community and have different characteristics that challenge the capabilities of each failure search approach.

5.5.3 Conclusion Validity. A conclusion validity threat may come from the wrong interpretation of the results due to random variations and inappropriate use of statistical tests. We mitigated this threat by executing each failure search approach multiple times, as well as repeating multiple times the execution of environment configurations on non-deterministic environments. When computing diversity, we executed clustering multiple times to account for the randomness of the *k*-means algorithm. Moreover, we compared the different approaches, both in terms of number of

failures and in terms of their diversity, using rigorous statistical tests such as the Mann-Whitney U Test for computing the p-value and the Vargha-Delaney metric to measure the effect size.

5.5.4 Reproducibility. In terms of reproducibility, we publish our replication package [7], making our evaluation repeatable and our results fully reproducible.

6 RELATED WORK

Testing DRL agents is a rather unexplored area of research. Works that generate adversarial attacks [61] have been proposed [24, 38], showing that such DRL agents can be vulnerable to adversarial attacks, similarly to DL agents (i.e., agents trained using supervised learning). However, our approach is substantially different since it is not focused on perturbing the raw inputs of the DRL agent sensors but rather on generating configurations for the whole environment the DRL agent runs into. The most similar work to ours is that by Uesato et al. [65], who proposed the sampling approach we used as baseline in our experiments. Our results show that our search-based approach outperforms it both in terms of number of failures triggered and their diversity in all case studies.

Biagiola et al. [8] proposed an approach to test the adaptation capabilities of DRL agents. In particular, the training of a DRL agent is resumed with environment configurations that are different from those experienced by the DRL agent during the previous training phase. Then, the proposed approach builds the adaptation frontier of the DRL agent, separating the configurations in which the DRL agent under test is able to adapt from those where adaptation is unsuccessful. Our approach is similar, in the sense that we also generate environment configurations. However, we are interested in testing the DRL agent to find its weaknesses at testing time rather than its adaptation capabilities (i.e., we do not resume training).

Also related to our work is that by Ruderman et al. [16], who studied how to train and test agents in procedurally generated environments. In particular, they trained DRL agents on a set of procedurally generated mazes for a 3D navigation task. Then, at testing time, they adopt a local search process to modify generated mazes guided by the performance of the DRL agent. This process generates out-of-distribution mazes, i.e., environment configurations that are not possible under the training distribution. In our work, we minimize the computational cost of the search by using a surrogate model of the environment (i.e., a failure predictor), rather than executing the DRL agent under test in the environment to measure its performance. Indeed, executing the DRL agent in the environment at each search iteration becomes prohibitively expensive in environments more complex than mazes. Moreover, our approach does not generate out-of-distribution environment configurations since all environment configurations generated at testing time are subject to the same validity constraints of the environment configurations generated during the DRL training process.

More recently, Tappler et al. [62] proposed a search approach to assess the quality of DRL agents. Their approach consists of searching for a reference trace that solves the RL task by sampling the environment. Such trace is built using a depth-first search algorithm and it is composed of all the states not part of the backtracking branches of the search. In particular, the search backtracks when it encounters an unsafe state, which is a state where the environment terminates the episode unsuccessfully. The states in the search graph preceding both an unsafe state and at least a successor non-terminal state, are called boundary states. The prefixes of the reference trace that end up in a boundary state are safety tests, since they are sequences of actions designed to bring the DRL agent under test into safety-critical situations. Our approach is complementary, since our goal is to generate new environment configurations to test the DRL agent, rather than evaluating it in the same environment configuration.

The literature in testing DL agents is quite rich and includes a multitude of works summarized in different surveys on the topic [9, 50, 74]. Particularly relevant to our work are those that use search-based methods to generate test inputs [1, 2, 6, 19, 49, 51, 78]. However, our work is different since it specifically targets DRL agents by exploiting the interaction data a DRL agent produces during training, which is not possible in DL testing since DL agents are trained offline.

7 CONCLUSION AND FUTURE WORK

Our approach to test DRL agents uses the interaction data produced by the DRL agents during training to train a surrogate model — i.e., a classifier — on failure and non-failure environment configurations. Then, it uses the failure prediction output of the surrogate model, as a fitness function to be maximized, so as to achieve high failure-exposure capabilities of the generated environment configurations while saving computation time. Our empirical results show that our search-based approach is able to generate more failures than the state-of-the-art sampling approach and that such failures are more diverse in all case studies. In our future work we plan to increase the diversity of the generated failures by incorporating it into the fitness function.

8 ACKNOWLEDGEMENTS

This work was partially supported by the H2020 project PRECRIME, funded under the ERC Advanced Grant 2017 Program (ERC Grant Agreement n. 787703).

REFERENCES

[1] Abdessalem, R. B., Nejati, S., Briand, L. C., and Stifter, T. Testing advanced driver assistance systems using multi-objective search and neural networks. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE* (2016), pp. 63–74.

[2] Abdessalem, R. B., Panichella, A., Nejati, S., Briand, L. C., and Stifter, T. Testing autonomous cars for feature interaction failures using many-objective search. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (New York, NY, USA, 2018), ASE 2018, ACM, pp. 143–154.

[3] Andrychowicz, M., Wolski, F., Ray, A., Schneider, J., Fong, R., Welinder, P., McGrew, B., Tobin, J., Abbeel, P., and Zaremba, W. Hindsight experience replay. *arXiv preprint arXiv:1707.01495* (2017).

[4] Arcuri, A., and Briand, L. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability 24*, 3 (2014), 219–250.

[5] Arthur, D., and Vassilvitskii, S. k-means++: The advantages of careful seeding. Tech. rep., Stanford, 2006.

[6] Ben Abdessalem, R., Nejati, S., C. Briand, L., and Stifter, T. Testing vision-based control systems using learnable evolutionary algorithms. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)* (May 2018), pp. 1016–1026.

[7] Biagiola, M. Replication package, 2022.

[8] Biagiola, M., and Tonella, P. Testing the plasticity of reinforcement learning based systems. *ACM Transactions on Software Engineering and Methodology* (2022).

[9] Braiek, H. B., and Khomh, F. On testing machine learning programs. *Journal of Systems and Software 164* (2020), 110542.

[10] Brito, B., Agarwal, A., and Alonso-Mora, J. Learning interaction-aware guidance policies for motion planning in dense traffic scenarios. *arXiv preprint arXiv:2107.04538* (2021).

[11] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. Humanoid-v1 openai gym. https://github.com/openai/gym/wiki/Humanoid-V1, 2021.

[12] Chen, D., Li, Z., Wang, Y., Jiang, L., and Wang, Y. Deep multi-agent reinforcement learning for highway on-ramp merging in mixed traffic. *arXiv preprint arXiv:2105.05701* (2021).

[13] Contributors, T. Torch crossentropyloss class documentation. https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html, 2021.

[14] Cui, Y., Jia, M., Lin, T.-Y., Song, Y., and Belongie, S. Class-balanced loss based on effective number of samples. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition* (2019), pp. 9268–9277.

[15] Dabney, W., Rowland, M., Bellemare, M. G., and Munos, R. Distributional reinforcement learning with quantile regression. In *Thirty-Second AAAI Conference on Artificial Intelligence* (2018).

[16] Everett, R. Strategically training and evaluating agents in procedurally generated environments. *Website* (2019).

[17] Eysenbach, B., Salakhutdinov, R. R., and Levine, S. Robust predictable control. *Advances in Neural Information Processing Systems 34* (2021), 27813–27825.

[18] Feng, S., Yan, X., Sun, H., Feng, Y., and Liu, H. X. Intelligent driving intelligence test for autonomous vehicles with naturalistic and adversarial environment. *Nature communications 12*, 1 (2021), 1–14.

[19] Gambi, A., Müller, M., and Fraser, G. Automatically testing self-driving cars with search-based procedural content generation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA* (2019), pp. 318–328.

[20] Haarnoja, T., Zhou, A., Abbeel, P., and Levine, S. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *arXiv preprint arXiv:1801.01290* (2018).

[21] Hartmann, C. Automatic intelligent parking: Audi at nips in barcelona. https://www.audi-mediacenter.com/en/press-releases/automatic-intelligent-parking-audi-at-nips-in-barcelona-7139, 2016.

[22] Hasselt, H. Double q-learning. *Advances in neural information processing systems 23* (2010), 2613–2621.

[23] Hessel, M., Modayil, J., Van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M., and Silver, D. Rainbow: Combining improvements in deep reinforcement learning. In *Thirty-second AAAI conference on artificial intelligence* (2018).

[24] Huang, S., Papernot, N., Goodfellow, I., Duan, Y., and Abbeel, P. Adversarial attacks on neural network policies. *arXiv preprint arXiv:1702.02284* (2017).

[25] Hwangbo, J., Lee, J., Dosovitskiy, A., Bellicoso, D., Tsounis, V., Koltun, V., and Hutter, M. Learning agile and dynamic motor skills for legged robots. *Science Robotics 4*, 26 (2019).

[26] Hwangbo, J., Lee, J., and Hutter, M. Per-contact iteration method for solving contact dynamics. *IEEE Robotics and Automation Letters 3*, 2 (2018), 895–902.

[27] Ioffe, S., and Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning* (2015), PMLR, pp. 448–456.

[28] J Ashok Chandrashekar, Fernando Amat, J. B., and Jebara, T. Artwork personalization at netflix. https://netflixtechblog.com/artwork-personalization-c589f074ad76, 2017.

[29] Jason Gauci, Edoardo Conti, K. V. Horizon: The first open source reinforcement learning platform for large-scale products and services. https://engineering.fb.com/2018/11/01/ml-applications/horizon/, 2018.

[30] Jason Gauci, Edoardo Conti, K. V. A platform for reasoning systems (reinforcement learning, contextual bandits, etc.). https://github.com/facebookresearch/ReAgent, 2021.

[31] Jin, Y. A comprehensive survey of fitness approximation in evolutionary computation. *Soft Comput. 9*, 1 (2005), 3–12.

[32] King, G., and Zeng, L. Logistic regression in rare events data. *Political analysis 9*, 2 (2001), 137–163.

[33] Kokhlikyan, N., Miglani, V., Martin, M., Wang, E., Alsallakh, B., Reynolds, J., Melnikov, A., Kliushkina, N., Araya, C., Yan, S., et al. Captum: A unified and generic model interpretability library for pytorch. *arXiv preprint arXiv:2009.07896* (2020).

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

[35] Langston, J. With reinforcement learning, microsoft brings a new class of ai solutions to customers. https://blogs.microsoft.com/ai/reinforcement-learning/, 2020.

[36] Lesort, T., Díaz-Rodríguez, N., Goudou, J.-F., and Filliat, D. State representation learning for control: An overview. *Neural Networks 108* (2018), 379–392.

[37] Leurent, E. An environment for autonomous driving decision-making. https://github.com/eleurent/highway-env, 2018.

[38] Lin, Y.-C., Hong, Z.-W., Liao, Y.-H., Shih, M.-L., Liu, M.-Y., and Sun, M. Tactics of adversarial attack on deep reinforcement learning agents. *arXiv preprint arXiv:1703.06748* (2017).

[39] Makoviychuk, V., Wawrzyniak, L., Guo, Y., Lu, M., Storey, K., Macklin, M., Hoeller, D., Rudin, N., Allshire, A., Handa, A., et al. Isaac gym: High performance gpu-based physics simulation for robot learning. *arXiv preprint arXiv:2108.10470* (2021).

[40] Mann, H. B., and Whitney, D. R. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics* (1947), 50–60.

[41] Miki, T., Lee, J., Hwangbo, J., Wellhausen, L., Koltun, V., and Hutter, M. Learning robust perceptive locomotion for quadrupedal robots in the wild. *Science Robotics 7*, 62 (2022), eabk2822.

[42] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. Human-level control through deep reinforcement learning. *nature 518*, 7540 (2015), 529–533.

[43] OpenAI, and collaborators. Gym leaderboard. https://github.com/openai/gym/wiki/Leaderboard, 2021.

[44] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems 32* (2019).

[45] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research 12* (2011), 2825–2830.

[46] Raffin, A. Rl baselines3 zoo. https://github.com/DLR-RM/rl-baselines3-zoo, 2020.

[47] Raffin, A., Hill, A., Gleave, A., Kanervisto, A., Ernestus, M., and Dormann, N. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research 22*, 268 (2021), 1–8.

[48] Raffin, A., Kober, J., and Stulp, F. Smooth exploration for robotic reinforcement learning. In *Conference on Robot Learning* (2022), PMLR, pp. 1634–1644.

[49] Riccio, V., Humbatova, N., Jahangirova, G., and Tonella, P. Deepmetis: Augmenting a deep learning test set to increase its mutation score. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2021), IEEE, pp. 355–367.

[50] Riccio, V., Jahangirova, G., Stocco, A., Humbatova, N., Weiss, M., and Tonella, P. Testing Machine Learning based Systems: A Systematic Mapping. *Empirical Software Engineering* (2020).

[51] Riccio, V., and Tonella, P. Model-Based Exploration of the Frontier of Behaviours for Deep Learning System Testing. In *Proceedings of ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2020), ESEC/FSE '20.

[52] Rousseeuw, P. J. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of computational and applied mathematics 20* (1987), 53–65.

[53] Rudin, N., Hoeller, D., Reist, P., and Hutter, M. Learning to walk in minutes using massively parallel deep reinforcement learning. In *Conference on Robot Learning* (2022), PMLR, pp. 91–100.

[54] Schaul, T., Quan, J., Antonoglou, I., and Silver, D. Prioritized experience replay. *arXiv preprint arXiv:1511.05952* (2015).

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

[56] Simonyan, K., Vedaldi, A., and Zisserman, A. Deep inside convolutional networks: Visualising image classification models and saliency maps. *arXiv preprint arXiv:1312.6034* (2013).

[57] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research 15*, 1 (2014), 1929–1958.

[58] Stocco, A., Pulfer, B., and Tonella, P. Mind the Gap! A Study on the Transferability of Virtual vs Physical-world Testing of Autonomous Driving Systems. *IEEE Transactions on Software Engineering* (2022).

[59] Sun, H., Feng, S., Yan, X., and Liu, H. X. Corner case generation and analysis for safety assessment of autonomous vehicles. *Transportation research record 2675*, 11 (2021), 587–600.

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

[61] Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I., and Fergus, R. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199* (2013).

[62] Tappler, M., Córdoba, F. C., Aichernig, B. K., and Könighofer, B. Search-based testing of reinforcement learning. *arXiv preprint arXiv:2205.04887* (2022).

[63] Tawn Kramer, M. E., and contributors. Self driving car sandbox. https://github.com/tawnkramer/sdsandbox, 2021.

[64] Todorov, E., Erez, T., and Tassa, Y. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems* (2012), IEEE, pp. 5026–5033.

[65] Uesato, J., Kumar, A., Szepesvari, C., Erez, T., Ruderman, A., Anderson, K., Heess, N., Kohli, P., et al. Rigorous agent evaluation: An adversarial approach to uncover catastrophic failures. *arXiv preprint arXiv:1812.01647* (2018).

[66] van der Maaten, L., and Hinton, G. Visualizing data using t-sne. *Journal of Machine Learning Research 9* (2008), 2579–2605.

[67] Vargha, A., and Delaney, H. D. A critique and improvement of the cl common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics 25*, 2 (2000), 101–132.

[68] Verma, A., Bagkar, S., Allam, N. V. S., Raman, A., Schmid, M., and Krovi, V. N. Implementation and Validation of Behavior Cloning Using Scaled Vehicles. In *SAE WCX Digital Summit* (2021), SAE International.

[69] Viitala, A., Boney, R., and Kannala, J. Learning to Drive Small Scale Cars from Scratch. *CoRR abs/2008.00715* (2020).

[70] Viitala, A., Boney, R., Zhao, Y., Ilin, A., and Kannala, J. Learning to drive (l2d) as a low-cost benchmark for real-world reinforcement learning. *arXiv e-prints* (2020), arXiv–2008.

[71] Wang, C., Wu, Y., Vuong, Q., and Ross, K. Towards simplicity in deep reinforcement learning: Streamlined off-policy learning.

[72] Wang, Z., Schaul, T., Hessel, M., Hasselt, H., Lanctot, M., and Freitas, N. Dueling network architectures for deep reinforcement learning. In *International conference on machine learning* (2016), PMLR, pp. 1995–2003.

[73] Xu, M., Huang, P., Li, F., Zhu, J., Qi, X., Oguchi, K., Huang, Z., Lam, H., and Zhao, D. Accelerated policy evaluation: Learning adversarial environments with adaptive importance sampling. *arXiv preprint arXiv:2106.10566* (2021).

[74] Zhang, J. M., Harman, M., Ma, L., and Liu, Y. Machine learning testing: Survey, landscapes and horizons. *IEEE Transactions on Software Engineering* (2020).

[75] Zhang, Q., Du, T., and Tian, C. Self-driving scale car trained by deep reinforcement learning. *arXiv preprint arXiv:1909.03467* (2019).

[76] Zhang, S., Wen, L., Peng, H., and Tseng, H. E. Quick learner automated vehicle adapting its roadmanship to varying traffic cultures with meta reinforcement learning. In *2021 IEEE International Intelligent Transportation Systems Conference (ITSC)* (2021), IEEE, pp. 1745–1752.

[77] Zhou, H., Chen, X., Zhang, G., and Zhou, W. Deep Reinforcement Learning for Autonomous Driving by Transferring Visual Features. In *2020 25th International Conference on Pattern Recognition (ICPR)* (2021).

[78] Zohdinasab, T., Riccio, V., Gambi, A., and Tonella, P. Deephyperion: exploring the feature space of deep learning-based systems through

illumination search. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (2021), pp. 79–90.