Testing the Plasticity of Reinforcement Learning Based Systems

MATTEO BIAGIOLA, Università della Svizzera italiana, Switzerland

PAOLO TONELLA, Università della Svizzera italiana, Switzerland

The data set available for pre-release training of a machine learning based system is often not representative of all possible execution contexts that the system will encounter in the field. Reinforcement Learning (RL) is a prominent approach among those that support continual learning, i.e., learning continually in the field, in the post-release phase. No study has so far investigated any method to test the plasticity of RL based systems, i.e., their capability to adapt to an execution context that may deviate from the training one.

We propose an approach to test the plasticity of RL based systems. The output of our approach is a quantification of the adaptation and anti-regression capabilities of the system, obtained by computing the adaptation frontier of the system in a changed environment. We visualize such frontier as an adaptation/anti-regression heatmap in two dimensions, or as a clustered projection when more than two dimensions are involved. In this way, we provide developers with information on the amount of changes that can be accommodated by the continual learning component of the system, which is key to decide if online, in-the-field learning can be safely enabled or not.

CCS Concepts: • **Software and its engineering** → **Software testing and debugging**.

Additional Key Words and Phrases: software testing, reinforcement learning, empirical software engineering

ACM Reference Format:

Matteo Biagiola and Paolo Tonella. 2022. Testing the Plasticity of Reinforcement Learning Based Systems. *ACM Trans. Softw. Eng. Methodol.* 1, 1, Article 1 (January 2022), 53 pages. https://doi.org/10.1145/3511701

1 INTRODUCTION

Reinforcement Learning [80] (RL) is a learning paradigm in which an *agent* interacts with an *environment* and chooses its *actions* in order to maximize a learning objective framed in a *reward function*. The combination of deep learning and RL, referred to as *Deep Reinforcement Learning* (DRL), has achieved numerous successes in recent years. Notably, the first DRL algorithm called *Deep Q Network* (DQN) [50] published in 2013 was trained to play Atari games [7] from pixels. Only three years later, the same team developed *AlphaGo* that defeated the best human player on the Go game [74]. More recent breakthroughs are the *OpenAI Five* [9] and the *AlphaStar* agents, which learn to master difficult strategy games such as Dota 2 and StarCraft 2 and beat the respective best human players. OpenAI Five and AlphaStar showed that DRL can effectively tackle challenges such as long term planning, partial observability, and complex, continuous state-action spaces, that are key challenges that are present in real world scenarios.

One of the critics that is directed towards DRL is that it needs a huge amount of experience in order to learn reasonable behaviors (*sample inefficiency*), i.e. something that can be accomplished only in simulation [32]. Moreover, in order to establish what is the best action in a particular situation, a DRL agent needs to *explore* the environment

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2022 Association for Computing Machinery.

Manuscript submitted to ACM

and try different actions to see which ones yield the highest rewards, although exploring the environment can be dangerous or expensive in many real world scenarios. On the other hand, DRL offers major advantages over alternative supervised approaches to machine learning (e.g., deep neural networks). First of all, it does not require any labeled training dataset. Indeed, manual creation of a representative and large training dataset is a bottleneck in many domains, where it may be very difficult to collect data for all possible environment configurations and conditions (e.g., all weather and lighting conditions possibly encountered by a self-driving car) and their manual labeling can be very expensive. On the contrary, DRL algorithms are guided just by the reward function and do not require any further ground truth labeling that accompany the data. This means that training is achieved by just letting the DL algorithm run unattended and unsupervised for a long period of time in the environment. Another benefit associated with its unsupervised nature is that DRL algorithms can adapt to changes in the environment quite naturally. By enabling some residual learning capability when the DRL algorithm is executed in the field, the learned strategy can be adapted to the shifts of the environment configurations and conditions. Such continual learning capability (i.e. plasticity), together with its peculiar unsupervised training mode, make RL extremely appealing to the industry as an alternative to supervised approaches in many complex and rapidly evolving domains.

DRL has being applied in more *practical* contexts than board games and video games. One such context is *personalization*, in which the DRL agent interacts with humans to let a digital system make relevant recommendations to individual users. In this domain, Netflix uses a simplified version of the RL problem, called *contextual bandits*, to choose which artworks to show in a movie that is recommended to a user [33]; the objective is to tailor the choice of the artwork to the particular user in order to maximize engagement. Microsoft's *Personalizer* [41] also uses this technology both for Microsoft products, e.g. to select the right offers and content across Windows, Edge browser and Xbox, and externally as a service, e.g. to deliver tailored recommendations in an online marketplace. Moreover, Facebook developed and open-sourced *Horizon*, described as the first open source RL platform for production [35] (now called *ReAgent* [36]). So far Horizon was used to deliver more relevant notifications in Facebook products, optimizing streaming video bit rates, and improving the virtual assistant suggestions in Messenger.

Beyond personalization, DRL has been applied also to *continuous control* tasks. One such example is a robot used to sort out objects in a warehouse [68]. *Covariant.AI*, the company that developed the software for the robot, used DRL to address the challenge of picking objects with a robotic arm from a bin of random items. The problem is difficult because the shapes and the surfaces of the objects are not uniform and a more traditional robotic arm controller cannot be programmed to perform the task always with the same predefined motion. On the contrary, a DRL agent can learn a strategy to pick up objects rather than being tailored to a specific one and it can *continue* to learn as it sees objects of unknown shapes. Another example of DRL applied in the real world is the intelligent parking system being developed by *Audi* [27]. The auto-maker demonstrated at NIPS 2016 (an important machine learning conference) that their 1:8 scale model car could learn how to properly find a suitable parking space, i.e. a metal frame, on an area measuring 3 x 3 meters, and park autonomously there. The parking problem is modeled as an RL problem and while the demonstration is only a proof of concept the next step announced by Audi is transferring the parking-space search process to a real car.

Despite the evidence that DRL is used in the real world, few works focus on testing DRL based systems [61, 93]. Some works study how DRL based systems behave in the presence of *adversarial attacks* [31, 44], i.e. techniques that automatically craft inputs in which the trained agent performs very poorly. Uesato et al. [88] focus, instead, on finding initial configurations of a DRL based system in order to find catastrophic failures. Ruderman et al. [18, 65], on the other hand, use *procedurally generated environments* and a search process to find specific configurations of the environment in which the trained agent fails. However, none of those works focus on testing the *plasticity* of DRL based systems,

i.e. the extent to which a DRL agent under test is able to adapt to a changing environment that is different from the environment it was initially trained on. In fact, Uesato et al. and Ruderman et al. only *evaluate* a trained agent on a changed environment rather than letting the agent continually learn on a changing environment to assess its adaptation capabilities. In fact, one of the characteristics of RL that differentiates it from supervised learning is that an RL agent can potentially learn continuously and autonomously even when the initial environment it was trained on evolves, since it does not require the presence of a supervisor during the training process. Therefore, testing the plasticity of an RL agent before deployment is important to understand its strengths and weaknesses in specific environment configurations that can arise in the real world.

In this paper we propose an approach to characterize the adaptation capabilities of the DRL agent under test in its environment. In particular, our approach takes as input a DRL agent trained in a parameterized environment. Then, it *samples* the parameter space defined by the environment parameters and it trains, in a continual learning mode, the agent on the resulting environment configurations, with the objective of characterizing the *adaptation frontier* of the agent. The adaptation frontier consists of two environment configurations that are *close* to each other in the parameter space and that trigger opposite adaptation behaviors of the agent (i.e., successful vs failed adaptation). After such *search* process, the parameter space is approximated to interpolate the missing values in order to compute the *adaptation volume* that quantifies the adaptation capabilities of the agent in the environment configurations defined by the environment parameters. In addition to the adaptation volume metric, our approach provides the users with a visualization of the adaptation capabilities of the agent. Specifically, when the environment has two parameters, or alternatively two parameters of interests are chosen at a time, we provide the users with two *heatmaps*: the *adaptation heatmap*, which visualizes the adaptation frontier of the agent, and the *anti-regression heatmap*, which shows whether the agent has any regression in the regions of the parameter space where the agent was able to behave correctly after adaptation. The adaptation (respectively anti-regression) heatmap shows in green the regions of the parameter space where the agent adapts (respectively does not regress), in red the regions where the agent does not adapt (respectively regresses) and different shades of colors between green and red to indicate the adaptation (respectively anti-regression) probability. If the user chooses to analyze the behavior of the given DRL agent considering more that two environment parameters, the visualization of the adaptation frontier is made possible through a combination of dimensionality reduction and clustering techniques. Afterwards, our approach employs decision trees such that the user can understand what are the crucial parameters of the environment that characterize the clusters of frontier points.

Our paper makes the following contributions:

- The first approach that tests the adaptation and regression capabilities of a DRL based system. Our approach characterizes the adaptation frontier of the agent in the parameter space defined by the environment and provides a visualization of such frontier;
- An implementation of our approach in a tool named AlphaTest, which is publicly available as open source software [47];
- An empirical evaluation of AlphaTest considering three DRL algorithms, four continuous control environments and up to three different combinations of environment parameters for each environment.

2 BACKGROUND

This section contains a brief introduction to reinforcement learning as well as to the main deep reinforcement learning algorithms in the state of the art (namely, DQN, PPO, SAC). The reader already familiar with these concepts can safely

skip this section entirely or partially. The interested reader can refer to Sutton and Barto [80] and Joshua et al. [3] for a more detailed discussion of the topic.

2.1 Introduction to Reinforcement Learning

Reinforcement Learning (RL) consists of learning a *policy*, i.e. how to act in each state the environment, in order to maximize a numerical reward signal [80]. The learner (or agent) is not told by any supervisor what actions are good (i.e., lead to the highest reward) but it has to learn it on its own, through trial-and-error interaction with the environment. The fundamental assumption of this learning paradigm is the so called *reward hypothesis* which states that training goals can be expressed as the maximization of the total (or cumulative) reward that the agent will get in its lifetime (also called *return*).

One of the challenges that arise in RL is the trade-off between *exploration* and *exploitation*. The agent needs to *exploit* those actions it already knows they are rewarding but it also needs to *explore* because there could be actions that are even more rewarding. The dilemma is that exploration and exploitation are not mutually exclusive and that the right trade-off needs to be found by the agent in order to succeed at the given task. Moreover, another issue relevant to trial-and-error learning is the so called *credit-assignment* problem [49]. In particular the question to answer is how to select and give credit to the actions that are more relevant for the success of the agent. In fact, in the most interesting and challenging cases, the reward signal is often sparse, noisy (in general due to the non-determinism of the environment and the policy) and delayed. For example if the agent is learning how to play a game it might get a positive reward only when it wins the game. Afterwards, it has to figure out what are the good actions that led to the victory, making them more likely.

More formally the RL agent takes as input a state s at each time-step t of its interaction with the environment and it has to decide which action a to take. Such action is determined by its policy π which is a mapping from states to actions. The agent receives a reward from the environment at each time-step and, for simplicity, let us assume that the task the agent needs to solve is episodic, i.e. the interaction of the agent with the environment ends when certain conditions hold. Ultimately the goal of a RL agent is to find a policy π which maximizes the *expected* return (the expectation operator is needed because in the general case both the environment and the policy are stochastic). In this case the policy π is the *optimal* policy and it is indicated as π^*.

The optimal policy can also be indirectly extracted from *value* functions, namely the *state-value* function $v_\pi(s)$ and the *action-value* function $q_\pi(s, a)$. The state-value function is defined as the expected return the agent will get if it starts from state s and then follows the policy π (i.e. it takes actions according to π) forever since then. Similarly, the action-value function is defined as the expected return the agent will get if it starts from state s, takes action a and then follows the policy π forever. Such functions must satisfy the recursive consistency relationships expressed by the *Bellman equations* [80], which relate the values of a state (or state-action pair) to the values of all the possible successor states (or state-action pairs). The *optimal* action-value function q^* can be computed by *solving* the Bellman equation for the action-value function. Consequently, from q^* it is possible to extract π^*, by choosing in each state s the action a that maximizes q^*.

2.2 Reinforcement Learning Algorithms

There are several design choices that regard RL algorithms. In the following we describe the trade-off between these choices and position the RL algorithms considered in our experiments within such trade-off.

Even though the RL problem was addressed without deep learning (through dynamic programming techniques, monte carlo methods and temporal difference learning [80]), it was the fusion with deep learning that made RL applicable to complex practical problems [50]. In particular deep learning made it possible to scale RL algorithms to complex high dimensional state and action spaces (e.g. continuous spaces or high dimensional discrete spaces like images). In deep RL (DRL) non-linear function approximators (namely, neural networks) are used to estimate quantities that depend on state and/or actions. Actually, neural networks can be used to estimate a policy π (either stochastic or deterministic), a value function V (or an action-value function Q) and/or a *model* of the environment [1].

One of the fundamental design choices for RL algorithms is the usage of a model of the environment, i.e. a quantity that predicts how the environment evolves when an action is taken and the reward it gives. If the algorithm uses a model of the environment, either available or by learning it, then it is said to be *model-based*; otherwise it is said to be *model-free*. A model of the environment could be beneficial for the agent since it can be used to look ahead by predicting the possible outcomes of its decisions. On the other hand model-based algorithms rely on a model of the environment that could be biased. For example a model may represent very well a specific part of the environment but it might fail in capturing other parts of it (e.g. because those other parts are difficult to explore). When the model is biased it does not accurately represent the real conditions of the environment and, as a consequence, the resulting agent might behave poorly on it.

Often the choice between using model-free or model-based algorithms depends on the task we want to solve by using RL methods. If, for example, the environment has very complex dynamics but the pattern for optimal behavior (i.e. the policy) is simple, then model-free algorithms may be the best choice. On the other hand, if the environment is simple to represent but the strategy to solve the task is complex, then model-based algorithms are more appropriate. Moreover, despite model-free methods might be inferior w.r.t. model-based algorithms regarding their *sampling efficiency* (i.e., the amount of data the algorithm needs in order to get a new policy), they are applicable in more situations (e.g. also in those cases in which the environment is difficult to model) and, as a consequence, they are more popular. For this reason we decided to focus our study on model-free algorithms, leaving the study of model-based algorithms for future work.

Model-free algorithms can be divided in three categories, namely *policy gradients*, *value-based* and *hybrid* methods. We discuss them in the context of Deep Reinforcement Learning (DRL) since in this paper we use DRL algorithms belonging to such categories.

2.2.1 Policy Gradients. In *policy gradient* methods the policy π is represented by a neural network whose weights are updated by maximizing the expected return. This optimization is performed *on-policy*, which means that each update is carried out with data (i.e. states, actions and rewards resulting from the interaction of the agent with the environment) coming only from the policy that produced that data. The state-of-the-art policy gradient algorithm is *Proximal Policy Optimization* (PPO) [71] whose focus is, at each step, to improve the policy as much as possible without causing performance collapse. In fact a large policy update may lead to a bad policy; such policy will be used for learning, which in turn may lead to even worse subsequent policies. On the other hand, if the improvement steps are too small then learning will be slow. In order to achieve this goal PPO has a mechanism called *clipping* that prevents the policy from changing too much and the hyperparameter ϵ controls how much the new policy can change w.r.t. the previous version of it. Moreover, previous policy gradient methods learn from current experience and discard past experiences after gradient updates, which makes them sample inefficient. PPO, instead, is designed to reuse even experiences coming from old versions of the policy being updated.

[1]When a state dependent function (value function v or action-value function q) is approximated we indicate it with a capital letter (respectively V and Q)

Regarding the exploration-exploitation trade-off, PPO trains a stochastic policy that lets the agent explore the environment in the initial phases of training. As training progresses, the policy becomes more and more deterministic and the agent is more encouraged to exploit than to explore.

2.2.2 Value-Based. In value-based methods it is the the action-value function that is represented with a neural network (i.e. Q). The weights are not updated by maximizing the expected return but rather the optimization is based on the Bellman equation. Moreover, the optimization is performed *off-policy*, which means that it can be done with data coming from any policy, not only from the policy that produced it. Once the action-value function Q is trained, the policy is obtained by choosing the action that maximizes it for each state. The most popular example of value-based method is *Deep Q Network* (or DQN) [50], which actually started the field of DRL and was improved over the years with various optimizations [28, 69, 90]. The basic idea behind the DQN algorithm is to estimate the action-value function by solving the Bellman equation iteratively. When a non-linear function approximator is used to represent the action-value function (e.g. a neural network), the iterative algorithm is not guaranteed to converge [50, 86]. The sources of instabilities leading to divergence are addressed by using *experience replay*, i.e. a buffer of agent's experiences (sequence of states, actions and rewards) that are replayed randomly during the optimization process, and the use of a separate neural network to represent the action-value function Q. Adding a separate neural network makes the optimization process much more similar to supervised learning and more stable.

Since DQN does not represent the policy explicitly, it needs a way to interact with the environment (often called *behavior* policy). Specifically, DQN uses an ϵ-greedy policy specifying that the agent selects a *greedy* action with probability $1 - \epsilon$ (i.e. it selects the optimal action in a certain state according to the current Q function) and selects a random action with probability ϵ (with ϵ annealed linearly from 1.0 to 0.1 over the first K time-steps, and fixed at 0.1 thereafter). This strategy implies that the agent mostly explores in the early training phase and mostly exploits at the end of training (a minimum amount of exploration is beneficial also in the final stages of training).

2.2.3 Hybrid between Policy Gradients and Value-Based. Algorithms belonging to this category use both policy gradients and value-based methods. One notable example of algorithm from this category is *Soft Actor Critic* (or SAC) [26]. SAC is based on the maximum entropy RL framework in which the objective is to both maximize the expected return and to maximize the policy entropy (i.e., the degree of "randomness" of the policy). This is desirable because policies optimized for maximum entropy will be more robust to unexpected environmental changes that are common in the real-world (i.e., at test time). Moreover, maximum entropy policies promote exploration, hence acquiring diverse behaviors of the agent.

SAC has a mechanism to control the entropy of the policy through a coefficient α. Depending on the implementation α is either set manually or automatically adjusted during training (the implementation of SAC we used [30] supports both). Moreover, similarly to DQN, SAC also performs its update steps off-policy.

3 APPROACH

Let E be an environment, which can be defined as the set of n parameters $\{p_1, \ldots, p_n\}$, and A_E be a reinforcement learning (RL) agent trained on E until the environment is *solved*, or alternatively until the desired performance level is reached. Moreover, let E' be a new environment with parameters $\{p'_1, \ldots, p'_n\}$ and $A_{E'}$ be a RL agent that is obtained by training A_E *continually* on E' for a certain period of time T. More precisely the task the agent needs to solve remains fixed (e.g. stabilizing the pole in a cartpole environment) but the environment parameters change (e.g. the mass of the pole increases) and the agent is asked to sequentially learn how to adapt to the changes. Such *adaptation* process can be

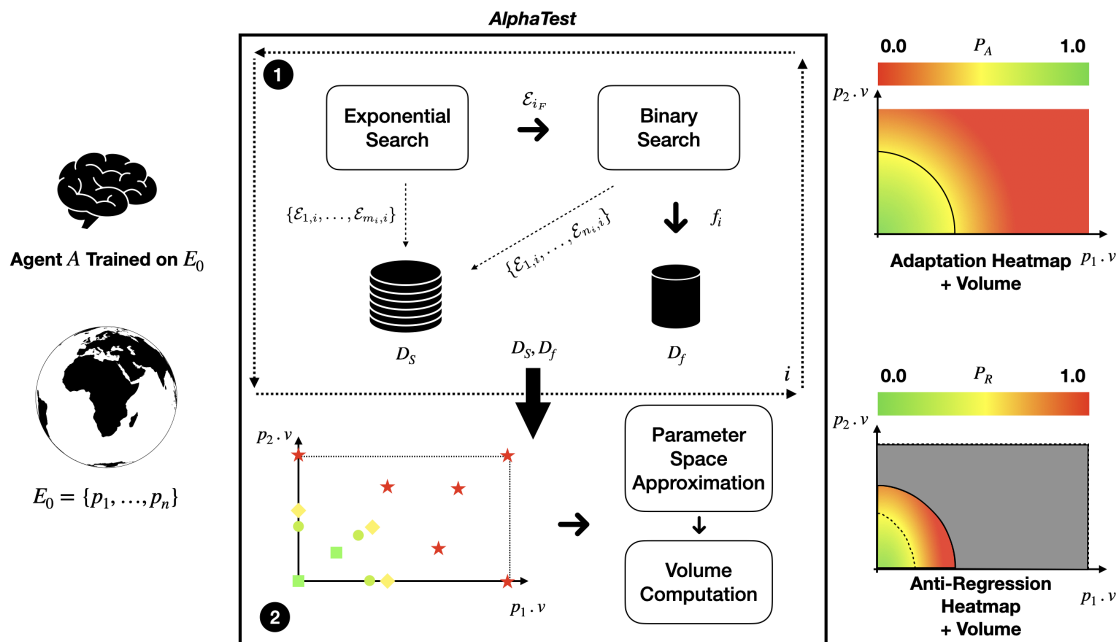

Fig. 1. Overview of the approach implemented in AlphaTest

successful if the agent $A_{E'}$ solves the environment E' (or it reaches a desired level of performance on E', when there is no binary notion of success); unsuccessful otherwise. Differently, techniques for *Transfer Reinforcement Learning* are not focused on the issues related to sequential (or continual) learning but rather on finding ways to transfer knowledge acquired by an agent in one domain to another domain, in order to accelerate the learning process [42, 56, 82, 96]. In the context of continual learning the problem we want to address is to understand the *frontier* between the environments E' in which the agent A_E can successfully *adapt* and the closest environments E'' in which the agent A_E cannot adapt. Intuitively, the *adaptation* frontier is defined as the set of all environment pairs (E'_T, E'_F) such that on E'_T adaptation is successful and on E'_F adaptation is not successful, and E'_T and E'_F are ϵ-*close* (i.e., have a distance lower than a user defined threshold ϵ) in the parameter space. When adaptation on a new environment E' is successful we are also interested in studying whether adaptation affects the capabilities of the agent on the original environment E it was trained on (i.e. whether there are regressions). The goal of our approach is to find the adaptation frontier of the agent A_E efficiently (as a consequence, regressions will be also found efficiently), as exhaustive exploration of the environment parameter space is infeasible due to its size and to the need to conduct a complete continual training session in each parameter configuration. The adaptation and anti-regression frontiers help developers understand the strengths and weaknesses of the agent in variable environment configurations before deployment.

Figure 1 shows an overview of our approach, composed of two phases. The first phase is the *search* phase (see ❶ in Figure 1) where we iteratively modify the environment parameters and let the agent *continually* learn on the modified environments to evaluate whether it can successfully adapt to the changes or not. Since the assessment of a single environment configuration involves full continual learning of the agent in the changed environment, we adopt efficient search strategies that minimize the number of steps needed to find the "frontier" of the agent's adaptation capabilities.

Then, in the second phase (see ❷ in Figure 1) we interpolate the missing points in the *adaptation heatmap* (see the *parameter space approximation* box in Figure 1), so as to make the computation of the adaptation volume possible.

The search phase (❶ in Figure 1) is divided into two sub-phases, namely *exponential search* and *binary search*. For the search phase we chose exponential and binary search for the following reasons. First of all, each search iteration requires a continual learning run of the agent in the changed environment. The high computational cost associated with a full continual learning session does not make it compatible with the use of population based, evolutionary algorithms already proposed in the literature for frontier exploration [62]. Moreover, exponential and binary search have a logarithmic computational complexity and, hence, they are very efficient in terms of number of steps required to achieve their respective objectives. Specifically, exponential search and binary search share some common operations. In particular both of them generate *search points* where a search point is defined as a 3-tuple $\mathcal{E}_{j,i} = \langle E_{j,i}, P_A, P_R \rangle$. The first component of a search point is the *environment configuration* $E_{j,i}$ at a particular iteration of the search. Specifically, the first subscript j indicates either the exponential search or the binary search iteration, whereas the second subscript i indicates the global search iteration. The second component is the *adaptation probability* P_A which tells us what is the probability that the agent trained on the original environment E_0 is able to adapt to the environment $E_{j,i}$. The choice of representing adaptation as a probability instead of a boolean predicate is due to the inherent non-determinism of deep reinforcement learning (DRL) algorithms [29]. For each environment configuration, we train the agent multiple times with different random seeds and we deem the adaptation successful if the agent adapts the majority of the times (i.e. $P_A > 0.5$, but the threshold is conventional and can be modified). Finally, the last component is the *regression probability* $P_R|_{P_A > 0.5}$, which tells us what is the probability that an agent which adapted successfully to the environment $E_{j,i}$ forgets how to behave in the original environment E_0 (i.e. it regresses). To simplify the notation, we write $\mathcal{E}_{k_{T|F}}$ to indicate that, at a generic iteration k, either the adaptation is successful (i.e. $P_A > 0.5$, hence the adaptation predicate $adapted(\mathcal{E}_k) = T$) or unsuccessful (i.e. $P_A \leq 0.5$, hence the adaptation predicate $adapted(\mathcal{E}_k) = F$).

The objective of the exponential search is to find an environment configuration in which the agent is not able to adapt. In other words the output of the exponential search is a search point where $P_A \leq 0.5$, namely $\mathcal{E}_{i_F}$. Then, the binary search component takes as input $\mathcal{E}_{i_F}$ and looks for a *frontier pair*, defined as follows:

Definition 1 (**Frontier Pair**). *A frontier pair is a pair of search points, namely $\mathcal{E}_{k_F}$ and $\mathcal{E}_{h_T}$, such that:*

$$dist(\mathcal{E}_{k_F}.E, \mathcal{E}_{h_T}.E) \leq \epsilon$$

Hence, the objective of binary search is finding two environment configurations that are *close* to each other (according to distance *dist*) and that trigger different behaviors of the agent, i.e. the agent is able to adapt to one environment configuration and it is not able to adapt to the other environment configuration. In practice, for a pair of frontier points $\mathcal{E}_{k_F}.E = \{p_1, \dots, p_n\}$ and $\mathcal{E}_{h_T}.E = \{q_1, \dots, q_n\}$ we can define a distance function *dist* that computes the average relative parameter change:

$$dist(\mathcal{E}_{k_F}.E, \mathcal{E}_{h_T}.E) = \frac{1}{n} \sum_{i=1}^{n} \frac{|p_i.v - q_i.v|}{(|p_i.v| + |q_i.v|)/2} \quad (1)$$

The underlying *assumption* we make in Definition 1 is that the values that the parameters defining the environment E can assume are real numbers, i.e. $p_i.v \in \mathbb{R}, q_i.v \in \mathbb{R}, \forall i \in [1, n]$. When *dist* measures a relative change, as in the formula above, we can choose a value of ϵ which ensures a small percentage change, such as $\epsilon = 0.05$.

During the search phase, search points are stored in a dataset D_S whereas the frontier pairs produced by the binary search are stored in the dataset D_f. Such datasets are the input to the second phase of our approach (❷ in Figure 1), namely the *volume computation* phase. In fact, in order to quantify the adaptation capabilities of a DRL algorithm and how it regresses when the adaptation is successful, we want to compute the volume underlying the adaptation (respectively the regression) *frontier*. To this aim we first map each environment configuration in the search points onto an n-dimensional grid ($n = |E_0|$), where the i-th dimension represents the range of values of each parameter p_i and the value of each cell in the grid represents the adaptation probability (respectively the regression probability) of the agent in that environment configuration. Then, for each grid cell that does not have a probability value, since it was not covered during exponential/binary search, we iteratively approximate its value based on the probability values of its *neighbors* (e.g., by majority voting), in order to get a completely filled grid (we call this sub-phase *parameter space approximation*). Then, the *grid counting* sub-phase consists of just counting the number of cells with $P_A > 0.5$ for what regards the adaptation capabilities and counting the number of cells with $P_{R|P_A>0.5} \leq 0.5$ for what regards the anti-regression capabilities of the agent.

If the environment E given as input is described by two parameters (i.e. p_1 and p_2, with $n = 2$), which might be two parameters of interest selected among all possible parameters, the grid with the mapped search points can be represented in a two-dimensional plot (see the bottom left corner in the middle of Figure 1). The points in the plot represent environment configurations ($E_{ij} = (p_1.v_i, p_2.v_j)$) and their shapes and colors indicate the adaptation probability (the same applies to the regression probability plot as well). In the example, red stars are environment configuration in which the adaptation probability is zero, whereas green squares are environment configurations in which the adaptation probability is 1.0 (e.g. the original environment the agent was trained on). Frontier pairs are represented by yellow diamonds and greenish circles where the adaptation probability is respectively a bit below 0.5 and a bit above it.

The output of our approach is, in general, (1) the adaptation volume that tells the users of our approach how much the given DRL algorithm (or agent) is able to adapt when the initial environment changes, together with (2) a visualization the frontier pairs sampled by the search (when $n > 2$ the visualization of the frontier pairs is built using a dimensionality reduction technique that maps n-dimensional vectors to two-dimensional ones). Our approach also outputs (3) the *anti*-regression volume which quantifies how much the DRL algorithm is able to remember how to behave in the original environment (i.e. does not have regressions), when adaptation is successful. When $n = 2$, or alternatively two parameters of the environment are chosen at a time, the approach outputs two two-dimensional heatmaps that visually show the behaviors of the DRL algorithm across the parameter space, regarding both adaptation and regression, as shown on the right hand side of Figure 1. While the adaptation heatmap (see the top right corner of Figure 1) has usually a *continuous* frontier, indicated with a black solid line, the anti-regression heatmap, defined only within the adaptation frontier (the gray region indicates the part of the parameter space where the anti-regression heatmap is not defined), is often *discontinuous*. On the bottom right corner of Figure 1 such anti-regression frontier is indicated as a continuous dashed line only for illustration purposes.

3.1 Motivating Example

Figure 2 shows the *CartPole* environment [6], which is one of the environments we used in our evaluation (see Section 4). CartPole, first described by Barto et al. [5], is an inverted pendulum which is attached through an un-actuated joint to a cart. The cart moves on a track which is frictionless in the default configuration and the whole system is controlled by a force either pushing it to the right direction (+1) or to the left (−1) with the same magnitude. The pole starts upright

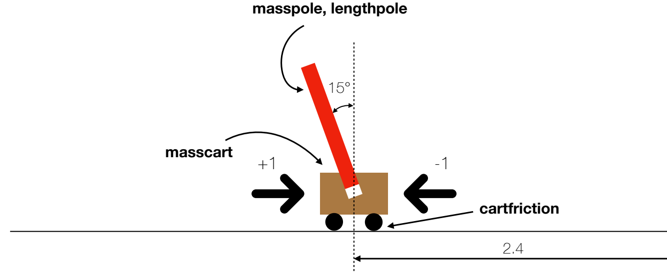

Fig. 2. Representation of the Cartpole environment [6, 57], described by Barto et al. [5]

and the objective is to move the cart in a way that prevents the pole from falling over. At every time-step when the pole stays upright the agent gets a +1 reward. The task of controlling the cart is episodic and there are three conditions that determine the end of an episode: (1) the pole falls over more than 15° from the vertical, (2) the cart moves more than 2.4 units from the center or (3) 500 time-steps pass.

In this environment the observation space of the agent is composed of four variables, namely the cart position (in the interval $[-4.8, 4.8]$) the cart velocity (in the interval $[-\infty, \infty]$), the pole angle (in the interval $[-24°, 24°]$) and the pole velocity at tip (in the interval $[-\infty, \infty]$). The action space of the agent is discrete and the agent can only decide to either move the cart left or right. Moreover, this task (or environment) is *solvable*, meaning that the developers of the environment specified a condition for which the task is considered solved. In particular, for the CartPole environment, if an agent gets an average reward over 100 testing episodes that is greater or equal to 475, then the task is deemed solved.

One possible way of parameterizing the CartPole environment is by defining it in terms of 4 parameters, namely *masspole*, *lengthpole*, *masscart* and *cartfriction*. We define a parameter p as a 4-tuple $\langle v_0, v, L, m \rangle$ where v_0 is the default or initial value for that parameter, v is the current value, L is a tuple $\langle l, h \rangle$ representing the valid range for the values of the parameter (i.e. l stands for the low limit and h stands for the high limit), and $m > 0$ is a constant value that stands for *multiplier*. The multiplier is computed before the exponential search and guides it on how to modify the initial value v_0 in order to make the environment more *challenging* for the agent.

The default or initial value v_0 for each parameter is already defined by the developers of the environment whereas the current value v is computed by the search phase at each iteration of the approach. The only variable of each parameter that needs to be specified by the users of our approach is L. The limits of a parameter L can be constrained by the environment itself (e.g. physical limits of the simulator) or alternatively they can be specified based on the desired values the user wants to analyze its agent on. Moreover, the limits determine the *direction*, and hence the value of the multiplier m, following which the environment becomes more challenging for the agent. Assuming that there is only one direction that makes the environment challenging, either towards increasing $|v_0|$ or decreasing it towards 0 (if both directions make the environment challenging, we just instantiate our approach twice, once per direction), one of the values in L is equal to v_0. If $L.l = v_0$ then $m > 1$, otherwise $L.h = v_0$ and $m \in (0, 1)$. For example, the parameter *masscart* has $v_0 = 1.0$ in the CartPole environment and, by increasing it, the cart becomes more difficult to control since the magnitude of the force that is applied to the cart remains the same. In this case, as for the other parameters of the CartPole environment, the multiplier $m > 1$ and $l = v_0$.

The users of our approach need to specify both the adaptation and the regression conditions (respectively Ac and Rc) for the environment, which determine when adaptation (respectively regression testing) is deemed successful

Algorithm 1: Pseudocode of AlphaTest

Input : E_0: initial environment $\{p_1, \dots, p_n\}$, $p_i = \langle v_{0,i}, v_i = v_{0,i}, L_i = \langle l_i, h_i \rangle, m_i = -1 \rangle$
A: agent trained on E_0 until desired performance is reached
Ac $(T_r) \rightarrow T|F$: adaptation condition function for the environment, T_r = training or evaluation trace
Rc $(T_r) \rightarrow T|F$: regression condition function for the environment, T_r = training or evaluation trace
ϵ: constant that determines the stopping condition for binary search
n_{tr}: number of training runs

Output: A_v: adaptation volume
R_v: anti-regression volume
F: frontier visualization if $n > 2$
A_{map}: adaptation heatmap if $n = 2$
R_{map}: anti-regression heatmap if $n = 2$

1. /* Search Phase */
2. determineMultipliers(E_0, A, Ac)
3. $D_S \leftarrow \emptyset$
4. $D_f \leftarrow \emptyset$
5. $envs \leftarrow$ computeEnvConfigurations(E_0)
6. **do**
7. $\quad \mathcal{E}_F, D_{S_e} \leftarrow$ expSearch($envs$, E_0, A, Ac, Rc, D_S, n_{tr})
8. $\quad D_S \leftarrow D_S \cup D_{S_e}$
9. $\quad f, D_{S_b} \leftarrow$ binarySearch($\mathcal{E}_F$, E_0, A, Ac, Rc, D_S, ϵ, n_{tr})
10. $\quad D_S \leftarrow D_S \cup D_{S_b}$
11. $\quad D_f \leftarrow D_f \cup f$
12. **while** $\exists\, env \in envs : env.exec = F$
13. /* Volume Computation Phase */
14. $A_{gr}, R_{gr} \leftarrow$ nearestNeighbor(D_S) ▷ *Parameter Space Approximation*
15. $A_v, R_v \leftarrow$ computeVolume(A_{gr}, R_{gr}) ▷ *Grid Counting*
16. **if** $|E_0| == 2$ **then**
17. $\quad A_{map}, R_{map} \leftarrow$ buildMaps(A_{gr}, R_{gr})
18. $\quad$ **return** $A_v, R_v, A_{map}, R_{map}$
19. **end**
20. $F \leftarrow$ buildFrontViz(D_f)
21. **return** A_v, R_v, F

(respectively failed). In practice, one can introduce a percentage performance degradation (e.g., 20% for Ac and 5% for Rc) that is considered acceptable respectively during adaptation and regression. For instance, for the CartPole environment Ac returns true if the agent gets at least an average reward of 380, corresponding to 475 (solved environment) minus 20%. Regarding the regression condition Rc, it returns false (i.e., there are no regressions) if the agent gets at least an average reward of 450, corresponding to 475 (solved environment) minus 5%. More generally, the users of our approach are allowed to define such conditions based on specific knowledge of the environment of interest, for example, based on safety conditions that must not be violated.

3.2 Algorithm

Algorithm 1 shows the pseudocode of our approach, as implemented in AlphaTest. The algorithm takes as input the initial environment E_0 where all the limits L_i for each parameter p_i have already been set, the current values v_i are

initialized to the respective original values $v_{0,i}$ and all the multipliers are set to -1. The agent A needs to be trained on E_0 until the desired performance is reached (when we refer to an agent trained on E_0 we simply use A instead of A_{E_0}, whereas when the agent is trained on another environment $E \neq E_0$ we explicitly indicate that by using A_E) and the user needs to specify both the constant ϵ, that determines how close two search points in the frontier pair should be (see Definition 1), and the number of training runs n_{tr} used to compute P_A and P_R. Moreover, the user needs to specify two functions, Ac and Rc, to check the adaptation and regression conditions. Such functions take as input a *trace* and output a boolean value indicating whether the respective condition is satisfied. The trace T_r refers to the output of the training or the evaluation of an agent (in Algorithm 2 and Algorithm 3 indicated as TRAIN and EVAL functions respectively). In our empirical evaluation, the training and evaluation traces contain the rewards the agent gets over a certain number of episodes.

3.3 Search Phase

The search phase (see Algorithm 1) starts by calling the procedure DETERMINEMULTIPLIERS, shown in Algorithm 2, which sets the multipliers for all the parameters in E_0. The objective of such procedure is, for each parameter p, to find a value $p.m$ such that $p.m \times p.v$ gives a new environment ($\neq E_0$) in which the agent trained on E_0 fails to *adapt* (Lines 8–11). In order to make this phase efficient, we skip continual learning and apply (i.e. evaluate) the pre-trained agent on the new environment. In fact, for the computation of the multipliers it is enough to approximate the behavior of the agent after continual learning with the evaluation of the initially trained agent. Correspondingly, at Line 11 we *improperly* use the adaptation condition, as at Line 10 we evaluate the agent A without training it on the new environment. Although evaluation alone is not enough to deem the adaptation of the agent on a new environment successful or unsuccessful (since adaptation would involve also training and learning how to behave in the changed environment), it is much cheaper computationally and it proved to be sufficient to find good multipliers, which is the overall goal of this procedure. In other words, we are using evaluation as a proxy for training to quickly find environment configurations that potentially challenge the capabilities of adaptation of the agent.

Let us now consider all the steps of Algorithm 2 in detail. At Line 3 we set the initial value of the i-th parameter to its default value v_0. Then, at Lines 4–7, we infer the *challenging* direction by looking at the limits of the parameter p, i.e. $\langle l, h \rangle$, and we double $p.v$ at every step (Line 9) if such direction is towards h ($c = 2$ at Line 4) while we halve it otherwise ($c = 0.5$ at Line 5). The loop at Lines 8–11 stops when either $p.v$ is beyond the given limits or the adaptation condition Ac is false. Finally, the multiplier for the i-th parameter is set at Line 12. For example, let us suppose that for our running example CartPole we have two parameters $p_1 = masscart = \langle v_0 = 1.0, v = 1.0, L = \langle l = 1.0, h = 10.0 \rangle, m = -1 \rangle$ and $p_2 = masspole = \langle v_0 = 0.1, v = 0.1, L = \langle l = 0.1, h = 4.0 \rangle, m = -1 \rangle$. In both cases $v_0 = p.L.l$, hence $c = 2$. To simplify the notation we indicate an environment configuration as $(p_1.v, p_2.v)$. Let us suppose that the configuration $E = (2.0, 0.1)$ is such that a given DRL agent A does not satisfy Ac during evaluation (i.e. by evaluating the agent A on $E \neq E_0$ over a certain number of episodes n_{ea} the average reward is less than 380). Then, $p_1.m = \frac{2.0}{1.0} = 2.0$. Instead, for what regards p_2, let us suppose that the first environment configuration for which Ac is not satisfied is $(1.0, 0.4)$, hence $p_2.m = \frac{0.4}{0.1} = 4.0$.

For brevity we omitted two edge cases in Algorithm 2. The first one regards the situation in which $p.v_0 < 0$; in such case we still work with positive numbers ($|p.v|$) and change the sign for the specific parameter when creating the environment with the specific environment configuration. Second, if the limits are such that it is not possible to falsify the adaptation condition, we automatically decrease/increase the limits $\langle l, h \rangle$ until Ac is falsifiable.

The multipliers computed by DETERMINEMULTIPLIERS are used to calculate the environment configurations at Line 5 of Algorithm 1. Specifically, for each parameter p, we multiply the current value $p.v$ by the corresponding multiplier $p.m$

Algorithm 2: Procedure DETERMINEMULTIPLIERS

Input : E_0: initial environment $\{p_1, \ldots, p_n\}$, $p_i = \langle v_{0,i}, v_i = v_{0,i}, L_i = \langle l_i, h_i \rangle, m_i = -1 \rangle$
A: agent trained on E_0
$\text{Ac}\ (T_r) \rightarrow T|F$: adaptation condition function for the environment, T_r = training or evaluation trace

Output: E_0: updated multipliers $\forall p \in E_0$

```
1  foreach i ∈ |E0| do
2      p ← E0[i]
3      p.v ← p.v0                                   ▷ Set initial value at each iteration
4      c ← 2
5      if p.L.h == p.v0 then
6          c ← 0.5
7      end
8      do
9          p.v ← p.v × c
10         Tr ← EVAL(A,E0)
11     while AC(Tr) and p.v ∈ [p.L.l, p.L.h]
12     p.m ← p.v / p.v0
13     p.v ← p.v0                                   ▷ Reset current value for next iteration
14 end
```

until one of the limits (either $p.L.l$ or $p.L.h$) is reached. By combining those values across all parameters we construct the environment configurations to be explored during exponential search and save them into *envs*. Each element env_i is equal to $\langle E_i, exec = T|F \rangle$ and *exec* tells us whether the environment configuration E_i was *executed* or not during exponential search. Considering our running example, we have the following values for p_1: $\{1.0, 2.0, 4.0, 8.0, 10.0\}$ and the following values for p_2: $\{0.1, 0.4, 1.6, 4.0\}$ including the low limits $p_1.L.l = 1.0$, $p_2.L.h = 0.1$ and the high limits $p_1.L.h = 10.0$ and $p_2.L.h = 4.0$. We have 5 values for p_1 and 4 values for p_2, therefore all the possible environment configurations are 19, i.e. $5 \times 4 - 1$ because one environment configuration is the original environment $E_0 = (1.0, 0.1)$ that is already executed.

3.3.1 Exponential Search. Algorithm 3 shows the exponential search function called by the main program (Algorithm 1, Line 7). The main loop at Lines 2–22 executes at least once, since the function assumes that there is at least one environment to execute, until a search point with $P_A \leq 0.5$ is found. At Line 3 by calling the function CHOOSENOTEXEC we choose (e.g., randomly) one environment configuration that was not executed. Then, at Line 4 DOMINANCEANALYSIS is performed to look for an already executed environment (i.e. one whose corresponding search point belongs to the set D_S, computed by Algorithm 1) that *dominates* or is *dominated* by the candidate environment to be executed, *env.E*. More specifically, DOMINANCEANALYSIS looks for any failing environment $\mathcal{E}_F \in D_S$ that is dominated by the current environment *env.E*, or alternatively for any succeeding environment $\mathcal{E}_T \in D_S$ that dominates the current environment *env.E*. In fact, in both cases the current environment does not need to be executed.

Formally, we can define dominance as follows. Let us introduce the inequality symbol $>_c$ that points to the more challenging direction of a parameter (i.e., $x_1 >_c x_2$ reads as $x_1 > x_2$ if the parameter makes the environment more challenging when its value increases; it reads as $x_1 < x_2$ otherwise). An environment E_a *dominates* another environment E_b *iff* $\exists i : (E_a[i].v >_c E_b[i].v) \wedge (E_a[k].v \geq_c E_b[k].v\, \forall k \neq i)$.

Algorithm 3: Exponential search function

Input : *envs*: environments to execute s.t. $\forall$ *env* $\in$ *envs*, *env* $= \langle E, exec \rangle$, $E \neq E_0$, *exec* $= T|F$
E_0: initial environment $\{p_1, \ldots, p_n\}$, $p_i = \langle v_{0,i}, v_i = v_{0,i}, L_i = \langle l_i, h_i \rangle, m_i = -1 \rangle$
A: agent trained on E_0
Ac $(T_r) \rightarrow T|F$: adaptation condition function for the environment, T_r = training or evaluation trace
Rc $(T_r) \rightarrow T|F$: regression condition function for the environment, T_r = training or evaluation trace
D_S: dataset of search points previously executed by the main algorithm
n_{tr}: number of training runs
Output : $\mathcal{E}_F$: search point $\langle E, P_A > 0.5, P_R \rangle$
D_{S_e}: dataset of search points for exponential search
Require : $\exists$ *env* s.t. *env.exec* $= F$

```
1  D_Se ← ∅
2  do
3      env ← chooseNotExec(envs)
4      E_d ← dominanceAnalysis(env.E, D_S)
5      P_R ← null
6      if E_d ≠ null then
7          P_A ← E_d.P_A
8          P_R ← E_d.P_R
9      else
10         As_env.E ← train(A,env.E,Ac,n_tr)
11         P_A ← |As_env.E| / n_tr
12         if P_A > 0.5 then
13             R_ps ← ∅
14             foreach A_env.E ∈ As_env.E do
15                 T_r ← eval(A_env.E,E_0)
16                 R_ps ← R_ps ∪ Rc(T_r)
17             end
18             P_R ← |R_ps|_{R_p=T} / |As_env.E|
19         end
20     end
21     D_Se ← D_Se ∪ ⟨env.E, P_A, P_R⟩
22 while P_A > 0.5
23 return E_d, D_Se
```

Figure 3 shows the dominance relationship between two environments. On the x axis we have the values of the parameter p_1 whereas in the y axis we have the values of the parameter p_2. For both parameters the direction that makes the environment more challenging for the agent is the positive direction. In Figure 3.A the environment $E = (x_2, y_2)$ *dominates* the already executed environment $\mathcal{E}_F.E = (x_1, y_1)$, where the agent does not adapt (i.e. $P_A \leq 0.5$). Given the dominance relation between the two environments we can say that the agent will not be able to adapt on E without carrying out the training process. The reason is that E will be more challenging for the agent than $\mathcal{E}_F.E$ since $x_2 > x_1$ and $y_2 > y_1$. As a consequence, if the agent is not able to adapt to $\mathcal{E}_F.E$ it will not be able to adapt to the more challenging environment E. On the other hand, Figure 3.B shows the case in which the environment $E = (x_1, y_1)$ is *dominated* by an already executed environment $\mathcal{E}_T.E = (x_2, y_2)$ where the agent adapts (i.e. $P_A > 0.5$). For a similar reason, we can conclude that the agent will be able to adapt to E without actually training it on E, since E is less

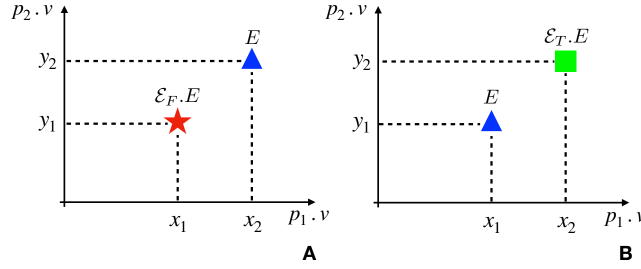

Fig. 3. Environment dominance in two dimensions; E is the environment under analysis

challenging than $\mathcal{E}_T.E$. Such dominance analysis is performed both in exponential search and binary search and it reduces the computation time of the search process by skipping unnecessary executions.

The DOMINANCEANALYSIS function returns the first search point $\mathcal{E}_d$ that satisfies the dominance relation described above; otherwise it returns *null*. If $\mathcal{E}_d$ exists then the adaptation/regression probabilities of *env.E* are approximated as the probabilities of the already executed search point (Lines 7–8). Otherwise, at Line 10 the TRAIN function is called. It performs n_{tr} training runs of the agent A on the environment *env.E*. Upon each training run, the function AC is called on the training trace produced by the run and if AC returns true then the agent is saved in the list $As_{env.E}$, the list of *successfully trained* agents. The adaptation probability is computed at Line 11 as the ratio between the number of times the agent is able to adapt and the number of training runs n_{tr}. If the adaptation probability is > 0.5, to measure the regression probability P_R we perform an evaluation run in the original environment of each successful agent ($A_{env.E} \in As_{env.E}$; see for loop at Lines 14–17). The regression probability is computed at Line 18 as the number of times the regression predicate is true (namely $|R_{p_s}|_{R_p=T}$) and the number of evaluation runs (i.e. $|As_{env.E}|$).

Considering our CartPole environment and assuming $n_{tr} = 3$, let us suppose that the first environment chosen by the exponential search is $E_1 = (4.0, 1.6)$ and that P_A of the agent on E_1 is 1.0, i.e. the agent always adapts in the 3 training runs, which means that after training the agent on E_1, the agent is evaluated on E_1 and it gets at least 380 points of reward in all the 3 training runs. Since the adaptation probability is > 0.5, the regression probability is also computed at Lines 12–19. The for loop at Line 14 is executed $n_{tr} = |As_{env}.E| = 3$ times in which we may get $P_R = 0.0$: in all such runs the RC function returns false, which means all successful agents, when evaluated on E_0 over a certain number of episodes n_{er}, gets an average reward of at least 450.

Then, the do-while loop continues and the next environment chosen at Line 3 could be $E_2 = (2.0, 0.3)$. E_2 is dominated by E_1 in which the agent adapts. Therefore, the adaptation probability of the agent on E_2 is inferred to be 1.0 and the regression probability to be 0.0, without having actually executed the training process on E_2 (Lines 7–8). Finally, let us suppose that the next environment being sampled is $E_3 = (10.0, 4.0)$ and that in this environment the agent is not able to adapt. Specifically, its adaptation probability on E_3 is $P_A = 0.0$. For this environment the regression probability is not computed since the condition at Line 12 is false. We do not compute the regression probabilities of agents whose adaptation is not successful.

3.3.2 Binary Search. Algorithm 4 shows the next step of the search phase, i.e. the binary search phase, called by the main program Algorithm 1 at Line 9. At Line 2 a new search point is constructed where the environment is the initial environment that by definition has $P_A = 1.0$ and $P_R = 0.0$ since the agent A was trained on E_0. Then, the main loop at

Algorithm 4: Binary search function

Input : $\mathcal{E}_F$: search point $\langle E, P_A \leq 0.5, P_R \rangle$
E_0: initial environment $\{p_1, \ldots, p_n\}$, $p_i = \langle v_{0,i}, v_i = v_{0,i}, L_i = \langle l_i, h_i \rangle, m_i = -1 \rangle$
A: agent trained on E_0 until desired performance is reached
Ac $(T_r) \rightarrow T|F$: adaptation condition function for the environment, T_r = training or evaluation trace
Rc $(T_r) \rightarrow T|F$: regression condition function for the environment, T_r = training or evaluation trace
D_S: dataset of already computed search points
ϵ: constant that determines the stopping condition for binary search
n_{tr}: number of training runs
Output: f: frontier pair according to Definition 1
D_{S_b}: dataset of search points for binary search

```
1  D_{S_b} ← ∅
2  ℰ_T ← ⟨E_0, P_A = 1.0, P_R = 0.0⟩
3  do
4      ℰ ← CHOOSE(ℰ_T, ℰ_F)
5      do
6          i ← CHOOSEPARAMINDEX(ℰ.E)
7          ℰ.E[i].v ← (ℰ_T.E[i].v + ℰ_F.E[i].v) / 2
8      while ℰ ∈ D_S
9      {...}                      ▷ Same as exponential search Algorithm 3 lines 4–20
10     if ℰ.P_A > 0.5 then
11         ℰ_T ← ℰ
12     else
13         ℰ_F ← ℰ
14     end
15     D_{S_b} ← D_{S_b} ∪ ℰ
16     q ← dist(ℰ_F.E, ℰ_T.E)
17 while q > ϵ
18 f ← ⟨ℰ_F, ℰ_T⟩
19 return f, D_{S_b}
```

Lines 3–17 executes until the condition for a frontier pair (see Definition 1) is satisfied. For brevity we did not include in the algorithm a timeout that is necessary for the while loop to terminate when ϵ is too small for the given environment and agent. In such situation the algorithm would return the frontier pair that is the closest to ϵ when the timeout expires.

At Line 4, either $\mathcal{E}_T$ or $\mathcal{E}_F$ is randomly chosen as starting point for the construction of a new environment (initially the latter is the output of exponential search). Then, the loop at Lines 5–8 performs the binary search operation by randomly choosing a parameter index (Line 6) and by assigning (Line 7) the value of the parameter corresponding to that index to be the average of the values of the corresponding parameters of the two environments where in one of them the agent is able to adapt (i.e. $\mathcal{E}.T$) and in the other the agent is not able to adapt (i.e. $\mathcal{E}.F$). The loop ends when a new search point is found that does not belong to the set of already computed/evaluated search points. For simplicity we omitted the further stopping condition that makes sure that the loop terminates when all the possible combinations of environment configurations have been tried; in that case we return $f = null$ which will not be included in the dataset of frontier pairs.

At Line 9 the new search point $\mathcal{E}$ is evaluated in the same way as in the exponential search (see Algorithm 3 at Lines 4-20). Then, given the P_A of $\mathcal{E}$ (Line 10) either $\mathcal{E}_T$ (Line 11) or $\mathcal{E}_F$ (Line 13) is reassigned. At Line 16 the distance between the two candidate frontier environments $\mathcal{E}_T$, $\mathcal{E}_F$ is computed and if the condition for a frontier pair is satisfied at Line 17, the main loop of the binary search function stops and such frontier pair is returned, together with the dataset of search points produced in the given iteration.

Considering our running example, the environment of the search point returned by the exponential search in the previous step is $E_3 = (10.0, 4.0)$ and the original environment is $E_0 = (1.0, 0.1)$. Let us suppose that at Line 4 we choose $\mathcal{E}_F$ and that at Line 6 the index is 0. Then, the new environment will be $E_4 = \left(\frac{1.0+10.0}{2} = 5.5, 4.0\right)$ and let us suppose that the adaptation probability of the agent on E_4 is $P_A = 0.33$, which means that it adapts once out of three times, since $n_{tr} = 3$. Since $P_A \leq 0.5$, we say that the agent is not able to adapt on E_4 and that $\mathcal{E}_F = \mathcal{E} = \langle E_4, P_A = 0.3, P_R \rangle$. Assuming that we chose $\epsilon = 0.5$ the current search points $\mathcal{E}_T$ and $\mathcal{E}_F$ do not form a frontier pair since the distance between them, computed at Line 16, is $1.64 > \epsilon = 0.5$ (for reference, the two environments to consider are $E_0 = (1.0, 0.1)$ and $E_4 = (5.5, 4.0)$). Therefore, the main loop continues and let us say that we still select the search point $\mathcal{E}_F$ at Line 4, that now contains E_4 as environment. At Line 6 this time we select the second parameter, i.e. index 1, in order to obtain the new environment $E_5 = \left(5.5, \frac{4.0+0.1}{2} = 2.05\right)$. Let us suppose that the agent A is able to adapt to E_5, i.e. $P_A = 0.67 > 0.5$ (the agent is able to adapt twice out of three times) and therefore $\mathcal{E}_T = \mathcal{E} = \langle E_5, P_A = 0.67, P_R \rangle$. This time the two environments to consider for the calculation of the frontier pair are $E_4 = (5.5, 4.0)$ and $E_5 = (5.5, 2.05)$, and the distance between them, computed at Line 16, is $0.32 \leq \epsilon = 0.5$. Hence, the current search points $\mathcal{E}_F = \langle E_4, P_A = 0.3, P_R \rangle$ and $\mathcal{E}_T = \langle E_5, P_A = 0.67, P_R \rangle$ form a frontier pair f that the binary search function returns.

3.4 Volume Computation Phase

The next phase of our approach is the volume computation phase which starts at Line 14 in Algorithm 1. It consists of the interpolation of the missing adaptation/regression probabilities from the existing ones (parameter space approximation sub-phase) for the entire grid and then the count of the grid points with $P_A > 0.5$ and $P_{R|P_A>0.5} \leq 0.5$ respectively (grid counting sub-phase).

For parameter space approximation, the main algorithm calls the NEARESTNEIGHBOR function, which takes as input the dataset of search points D_S produced by the search and returns two fully filled grids, namely A_{gr}, the adaptation grid, and R_{gr}, the regression grid. First, the NEARESTNEIGHBOR function creates an n-dimensional grid, where n is the number of parameters in the initial environment E_0. The grid will be filled with an adaptation probability value for each combination of parameters (for brevity we only refer to A_{gr} but the same procedure applies to R_{gr}). In particular, for each parameter p_i, we compute the step size of the array a_i forming the i-th dimension of the grid using the following formula:

$$step_i = \frac{p_i.L.h - p_i.L.l}{g \times 100} \tag{2}$$

where g is the *granularity* parameter that determines how granular the resolution of the values of p_i (i.e. $p_i.v$) in the i-th dimension of the grid is (with $g = 1$ we get a grid containing 100 discrete slots along each dimension). The higher the parameter g, the higher the resolution. The number of values $nv = len(a_i)$ that p_i can assume in the i-th dimension is $\lceil g \times 100 \rceil$, therefore the grid will be a hyper-square matrix with dimensions $nv \times \cdots \times nv = nv^n$. It is initialized with *null* values.

Once the grid A_{gr} is built the next step is to *map* the search points in D_S into the grid. For each search point $\mathcal{E}_j \in D_S$, considering its associated environment $\mathcal{E}_j.E$, and for each parameter p_i of the environment (i.e. $\mathcal{E}_j.E[i]$) the index in the i-th dimension of the grid of the search point $\mathcal{E}_j$ is computed as follows:

$$index_j[i] = \left\lfloor (nv - 1) \times \frac{\mathcal{E}_j.E[i].v - \min a_i}{\max a_i - \min a_i} \right\rceil \tag{3}$$

where nv is the length of each array a_i (we subtract 1 because the index starts from 0), $\min a_i$ is the minimum value of the parameter p_i in the grid (correspondingly $\max a_i$ is the maximum) and $\lfloor \cdot \rceil$ indicates that the product is approximated to the nearest integer. Once we have the index in the grid corresponding to the environment $\mathcal{E}_j.E$, we insert the adaptation probability value into the associated grid cell: $A_{gr}[index_j] = \mathcal{E}_j.P_A$. However, if the granularity g is too small, there could be *collisions*, i.e. multiple search points, and consequently multiple environment configurations, have the same index in the grid. In that case for the value in $A_{gr}[index_j]$ we take the mean of the adaptation probabilities of the colliding search points.

At this point A_{gr} has the adaptation probabilities of all the search points in D_S, together with *null* values where the search procedure did not sample any point. In order to compute the volume to quantify the capability of adaptation of the agent on the possible environment configurations, we need to approximate the missing values of the adaptation probabilities. We apply the nearest neighbor technique: we replace each *null* value in A_{gr} with the mean value of the *neighboring* points when there exists at least one neighboring value different from *null*. At first, the grid is scanned for indices with *null* values. Then, the values of these *null* cells are updated in batch and the procedure is repeated until there are no more indexes with *null* values. By performing a batch update of all *null* entries at the same time, we ensure that the nearest neighbor approximation does not depend on the order in which the indexes are scanned, as would happen if the updates were in-place. The *neighborhood* to consider for each update is composed of $3^n - 1$ points, where n is the number of parameters of the environment E, i.e. the number of dimensions of the grid A_{gr}. In fact, along each dimension a point has two neighbors, obtained by incrementing or decrementing its index. This means that a neighborhood consists of 3 points per dimension, or 3^n points overall, except for the initial, unchanged point (hence, $3^n - 1$). Once the grid A_{gr} is fully filled, we can compute the adaptation volume (Line 15 in Algorithm 1) by counting the number of points in the grid that have an adaptation probability value of > 0.5, normalized by the total number of points in the grid.

If the environment has two parameters (Line 16 in Algorithm 1) then we also build a two-dimensional adaptation probability heatmap from A_{gr}, for visualization by users. The color in each two-dimensional heatmap cell represents the adaptation probability ranging from red ($P_A = 0.0$) to green ($P_A = 1.0$). However, instead of plotting the grid directly, we apply a further interpolation function that makes the transitions (in yellow) between the adaptation part (in green) and the non-adaptation part (in red) *smoother* than they are on the grid, so as to make the adaptation frontier more visible in the heatmap.

On the other hand, if the environment has more than two parameters, besides the adaptation volume, we provide the users with a two-dimensional visualization of the frontier based on the t-SNE (t-distributed Stochastic Neighbor Embedding) [89] dimensionality reduction technique. t-SNE preserves the local structure, such that the frontier points that are close in the n-dimensional space remain close to each other in the two-dimensional space. Despite the information on the actual shape of the adaptation frontier is necessarily lost in the two-dimensional space, the clusters resulting from the application of the dimensionality reduction technique give the users an idea of the regions of the parameter space in which the frontier points represent similar environments. In order to better show the separation between the

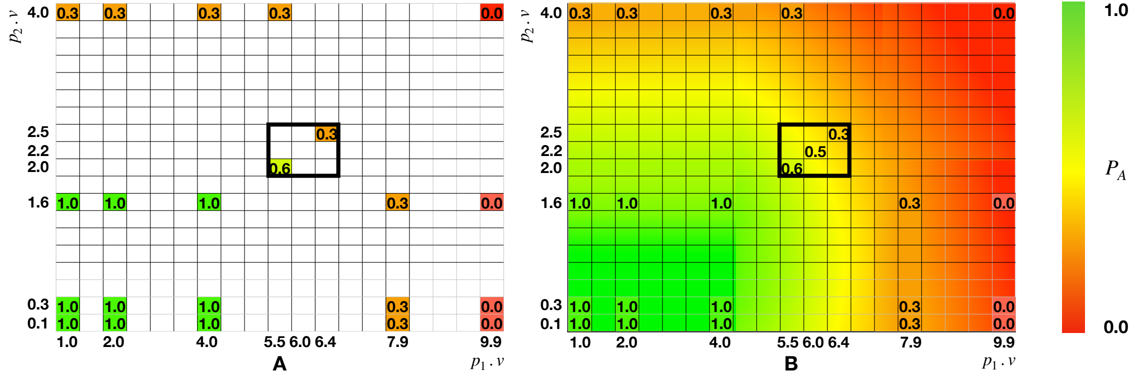

Fig. 4. Adaptation probability heatmaps

different frontier pairs we apply *clustering* to the output of the dimensionality reduction method. In particular, we use the *K-means* clustering algorithm and we determine the optimal number of clusters k^* by performing *silhouette analysis* on a range of candidates k. The silhouette score falls within the range $[-1, 1]$ and we choose as k^* the k that gives the highest silhouette score, resulting in dense and well separated clusters. Then, once each frontier pair has been assigned a cluster label we train a *Decision Tree* that determines the critical environment parameters for which a certain frontier pair belongs to a cluster rather than another. Finally, we plot the frontier pairs as given by the dimensionality reduction technique, within colored regions indicating the respective clusters they belong to, together with the decision tree plot.

Figure 4 shows two maps for the CartPole running example. The two parameters are $p_1 = masscart$ and $p_2 = masspole$. On the left hand side Figure 4.A shows the grid filled only with the search points, whereas Figure 4.B shows the result of applying nearest neighbor approximation and smoothing to the initial grid.

Specifically Figure 4.A contains the search points explored during exponential and binary search sections (including the ones discussed above: $\mathcal{E}_{1_T} = \langle E_1 = (4.0, 1.6), P_A = 1.0\rangle, \mathcal{E}_{2_T} = \langle E_2 = (2.0, 0.3), P_A = 1.0\rangle, \mathcal{E}_{3_F} = \langle E_3 = (10.0, 4.0), P_A = 0.0\rangle, \mathcal{E}_{4_F} = \langle E_4 = (5.5, 4.0), P_A = 0.3\rangle, \mathcal{E}_{5_T} = \langle E_5 = (5.5, 2.05), P_A = 0.67\rangle$) together with the initial search point $\mathcal{E}_{0_T} = \langle E_0 = (1.0, 0.1), P_A = 1.0\rangle$ and other search points that might have been sampled during the search. First, let us construct the grid by computing the step size for the two dimensions. By setting $g = 0.192$ we get a step size $step_1 = 0.47$ for p_1 ($p_1.L = [1.0, 10.0]$) and a step size $step_2 = 0.20$ for p_2 ($p_2.L = [0.1, 4.0]$). The two arrays a_1 and a_2, containing the remapped values for p_1 and p_2, have 20 elements. Therefore the grid A_{gr} is a two dimensional matrix with a shape of 20×20. In Figure 4 the x axis of each grid shows the values of p_1 and the y axis the values of p_2. Let us now map the search point $\mathcal{E}_{3_F}$ into the grid. The environment E_3 has coordinates $(10.0, 4.0)$ and according to Equation 3 its indexes on the grid are:

$$index_3[0] = \left\lfloor 19 \times \frac{10.0 - 1.0}{9.9 - 1.0} \right\rceil = \lfloor 19.2 \rceil = 19 \tag{4}$$

$$index_3[1] = \left\lfloor 19 \times \frac{4.0 - 0.1}{4.0 - 0.1} \right\rceil = \lfloor 19 \rceil = 19 \tag{5}$$

hence we can set the value at $index_3 = (19, 19)$ in A_{gr} to be $\mathcal{E}_{3_F}.P_A = 0.0$, i.e. $A_{gr}[index_3] = 0.0$.

After repeating the same mapping procedure for all the other search points, we get the heatmap in Figure 4.A. The next step is to approximate the missing adaptation probability values (blank cells in the figure) using the nearest neighbor algorithm. For example, let us take the missing value with coordinate $(6.0, 2.2)$, i.e. the central point of the black square in the middle of the heatmaps. The neighborhood of that point is composed of $3^2 - 1 = 8$ points, since $n = 2$. These are the points delimited by the black square, excluding the central point, i.e., the point under analysis. There are two other values in the neighborhood of $(6.0, 2.2)$, therefore the adaptation probability value for such point is inferred to be $\frac{0.6+0.3}{2} = 0.5$. In Figure 4.B we can see that the point is on the frontier delimited by the yellow curved region.

3.5 Implementation

We implemented our approach in an open-source tool called AlphaTest written in Python [47]. The environments we considered in our empirical evaluation are available from the *Gym* library [10] (v. 0.16.0) and the DRL algorithms we trained on those environments are implemented in a library called *stable-baselines* [30] (v. 2.10.1) a fork of the popular *baselines* [15] library from OpenAI. For plotting the heatmaps we used *seaborn* and *matplotlib* whereas for dimensionality reduction we used the *t-SNE* method [89] implemented in the *scikit-learn* Python library [58]. We used the scikit-learn also for the *k-means* and the *decision trees* implementations. For what regards the smoothing technique, we used the *radial basis function* method of the *scipy* Python library [13].

4 EMPIRICAL EVALUATION

We consider the following research questions:

RQ_1 (effectiveness): How effective is AlphaTest in finding frontier pairs that characterize the adaptation behavior of a deep reinforcement learning algorithm in a given environment w.r.t. a random exploration of the parameter space?

RQ_1 aims at empirically comparing AlphaTest with the random approach on the characterization of the adaptation frontier. We deem a given approach as *effective* when it is able to characterize the adaptation frontier *accurately* (i.e., with high resolution across the entire parameter space), which is important to understand the regions of the parameter space where the agent is able to adapt and the regions where it is not able to adapt.

RQ_2 (discrimination): How does AlphaTest discriminate between different deep reinforcement learning algorithms, that exhibit different degrees of plasticity?

The goal of **RQ_2** is to determine whether AlphaTest is able to discriminate a deep reinforcement algorithm that has high adaptation capabilities from another deep reinforcement learning with low adaptation capabilities. The discrimination capability is crucial in practical scenarios to guide the decision of the most appropriate algorithm that adapts better overall or in a desired region of the parameter space.

RQ_3 (hyperparameters): What are the critical hyperparameters of AlphaTest and what is the best way to fine tune them?

In **RQ_3** we want to study what is the impact of the hyperparameters of AlphaTest on its performance and on the discrimination metrics. Such empirical assessment will guide the users of AlphaTest to choose a proper trade-off, depending on the computational resources available and the algorithm under test.

4.1 Subject systems

Table 1 shows the environments we considered in our evaluation together with the associated parameters. We kept the same parameter names that can be found in the source code of the respective environments except in the case of

CartPole, where we added a *cartfriction* parameter, taken from the implementation of Patanjali et al. [57], that was not present in the original implementation. Moreover, we modified the constructors of the environment classes to receive the values of such parameters, which were previously hardcoded. Table 1 can be read in the following way: considering a specific RL algorithm we tested its behaviors in the parameter space of an environment determined by either two, three or four parameters. The parameters indicated in the *3 parameters* and in the *4 parameters* Columns are added to the parameters of the previous columns. For example, when we write *CartPole 2*, we are considering the environment CartPole where during the search phase we vary only two parameters, namely *length* and *cartfriction*, while the other two parameters, namely *masspole* and *masscart*, are fixed at their respective original values. When we write *CartPole 3* we are considering the environment CartPole where we vary three parameters during the search phase, namely *length*, *cartfriction* and *masspole*, while the fourth parameter *masscart* remains fixed. The *dash* sign in a table entry means that for that environment there are no further parameters to consider. For example, for the Pendulum and the MountainCar environments, we can consider at most three parameters.

We already described the CartPole environment in the previous section (see Section 3.1) as our motivating example, together with the parameters that we vary during the search process. The *Pendulum* environment [54] can be described an inverted pendulum, like the CartPole environment, but the problem is framed in a different way. In the Pendulum environment the pendulum starts in a random position, and the goal is to swing it up so that it stays upright. The reward function depends on the angle of the pendulum and it gives the agent maximum reward if the pendulum is upright. The action space is continuous and it consists of a single action corresponding to the torque applied on the joint of the pendulum. The observation space is also continuous and it is a vector of two components, namely the pendulum angle w.r.t. the rest position θ and the angular velocity $\dot{\theta}$. The Pendulum environment is episodic and the condition for the end of an episode is based on a fixed maximum number of time-steps (i.e. 200). Pendulum is an *unsolved* environment, as opposed to the CartPole environment, meaning that it does not have a specified reward threshold at which it is considered solved. The parameters that can be changed in such environment are *dt*, *length* and *mass*. The latter two parameters are relative to the length and the mass of the pendulum pole respectively, while the former determines the rate at which the angle θ of the pendulum can change.

In the *MountainCar* environment [52], described for the first time by Moore et al. [51], a car starts positioned between two mountains in a one-dimensional track. The objective is to reach the goal position up the right mountain. The car engine is not powerful enough to reach the goal in a single pass, therefore it needs to drive back and forth between the two mountains to build up momentum. The agent receives a negative reward at every time-step so that it is encouraged to reach the goal as soon as possible. The action space is discrete and the agent can either choose to apply a force to push the car in the right direction, to apply the same force to push the car in the left direction or, alternatively, not to apply any force. The observation space, instead, is continuous and it is a one dimensional vector containing the position of the car and its linear velocity. Also the MountainCar environment is episodic and an episode ends when 200

Table 1. Environments and their parameters

	2 parameters	3 parameters	4 parameters
Cartpole [5, 6]	*length*, *cartfriction*	*masspole*	*masscart*
Pendulum [54]	*dt*, *length*	*mass*	-
MountainCar [51, 52]	*force*, *gravity*	*goalvelocity*	-
Acrobot [22, 78, 79]	$linklength_1$, $linkcompos_1$	$linkmass_2$	$linkmass_1$

time-steps have passed or the agent reaches the goal position. Like the CartPole environment, also the MountainCar environment is *solvable* and it is considered solved when the agent gets an average reward of at least −110.0 over 100 consecutive evaluation episodes. The parameters that can be changed in such environment are *force*, *gravity* and *goalvelocity*. The *force* parameter represents the magnitude of the action applied to the car to move it either to the left or to the right. The *gravity* parameter also affects the difficulty of the task and it can be considered equivalent to changing the mass of the car. Lastly, the *goalvelocity* parameter imposes a constraint on the velocity that the car needs to have when reaching the goal position.

The *Acrobot* environment [78], first described by Sutton et al. [79] and later refined by Geramifard et al. [22], is a system composed of two joints and two links where the joint between the two links is actuated. The initial position of the system is with the two links hanging downwards and the goal of the agent is to bring the end of the lower link to a given height. The agent gets a reward of −1 for each time-step and a reward of 0 when it manages to reach the goal. The action space of the agent is discrete so that it can either apply a positive torque on the actuated joint between the two links, a negative torque of the same magnitude or, alternatively, it can choose to apply no torque. The observation space, instead, is continuous and it consists of $\sin\cdot$ and $\cos\cdot$ of the two rotational joint angles θ_1, θ_2 and the joint angular velocities $\dot{\theta}_1$, $\dot{\theta}_2$. The Acrobot environment is an episodic environment and an episode ends when either the goal is reached or 500 time-steps have passed. The Acrobot environment has a reward threshold for which it is considered solved, which is −100 over 100 evaluation episodes. The parameters that can be changed in such environment are *$linklength_1$* (length of the first link, changing the length of the second link has no effect), *$linkcompos_1$* (position of the center of mass of the first link), *$linkmass_2$* (mass of the second link) and *$linkmass_1$* (mass of the first link).

4.2 Subject algorithms

The deep reinforcement learning (DRL) algorithms we selected for our empirical evaluation are *Proximal Policy Optimization* (PPO) [71], *Soft Actor Critic* (SAC) [26] and *Deep Q Network* (DQN) [50]. They belong to the categories presented in Section 2, respectively policy gradients (PPO), hybrid (SAC) and value-based (DQN) methods. Moreover these DRL algorithms are mature and widely used and as such many model-free reinforcement learning libraries provide a stable implementation of them.

The implementation of SAC we used [30] only supports continuous action spaces, while DQN only supports discrete action spaces. The PPO implementation supports both continuous and discrete action spaces, but we always chose to use PPO in its discrete version. Therefore, we modified the Pendulum environment to also support a discrete action space for the two possible actions. This way a DQN agent, as well as a PPO agent with discrete actions, can either swing the pendulum left with maximum torque or right with the same torque magnitude (we did not include a *do nothing* action). Moreover, we also modified the CartPole, MountainCar and Acrobot environments to support continuous action spaces, so that a SAC agent would be able to control them. In particular, we defined such continuous action spaces as a linear interpolation between the minimum and the maximum *force* that can be applied to the system. In all environments with discrete action spaces the *left* and *right* actions correspond to the extremes of the respective continuous action spaces and the *do nothing* action (in the MountainCar and Acrobot environments) is the middle point in the continuous space.

4.3 Procedure and Metrics

4.3.1 Procedure. We trained all DRL algorithms under test on the environments with the default parameters. The hyperparameters of the DRL algorithms were, in part, taken from the open-source repository *rl-baselines-zoo*[2], which contains both hyperparameters and trained agents for the DRL algorithms implemented in *stable-baselines* [30]. We changed the given hyperparameters only when it was possible to achieve a better cumulative reward and/or to decrease the training time. The final set of hyperparameters for each DRL algorithm was chosen as the one that gave the highest average reward over 10 distinct training runs (each one with a different random seed). Such set of hyperparameters was then used to train the model that we used in our evaluation, i.e. the *model under test*. In particular, when training the model under test, we evaluated it for 100 episodes every N time-steps, where N is a fraction (i.e., 10%) of the total training time measured in number of time-steps, which changes for every algorithm and every environment. The best model was chosen as the one with highest average reward considering all the evaluation runs. Table 2 shows the performance, both in terms of average reward and in terms of training time, of the best models produced by each DRL algorithm on the respective environments. In particular, Column *Avg reward* shows the average reward a trained agent gets over 100 evaluation episodes. Column *Time to train* indicates the time taken, in minutes, to train a certain DRL algorithm on the given environment. The table also shows that for all the environments that have a reward threshold (i.e. they are *solvable*, see column *Reward threshold*) the average reward over 100 episodes for all the DRL algorithms is above the threshold. For the Pendulum environment, which does not have a reward threshold, we took the average reward obtained when training the SAC algorithm with the hyperparameters we found in the *rl-baselines-zoo* repository and trained both PPO and DQN algorithms to achieve the same average performance.

In order to enable continual learning for the selected DRL algorithms we had to change some of their specific hyperparameters. Since no guidelines exist on how to best set the hyperparameters of DRL algorithms for continual learning, we acted mainly on those hyperparameters that control the amount of exploration of the agent during training. Indeed, when continual learning is enabled, the objective is to preserve the performance achieved in the previous training phase while also learning the new behaviors that might be necessary in the new, changed environments. In order to preserve the previous performance, we saved the replay memory resulting form the training phase on the original environment for DQN and SAC since, being off-policy algorithms (see Section 2), they learn from transitions stored in such memories. Then, when continual learning is enabled for such algorithms, the memory is restored and, as

[2]https://github.com/araffin/rl-baselines-zoo

Table 2. RL algorithms performance after training on the original environments.

		PPO		SAC		DQN	
	Reward threshold	Avg reward	Time to train	Avg reward	Time to train	Avg reward	Time to train (min)
Cartpole	475	500.0	1.81	500.0	2.33	500.0	2.08
Pendulum	-	-145.87	2.12	-140.79	1.92	-153.32	2.55
MountainCar	-110	-108.93	2.38	-108.47	2.80	-104.86	3.55
Acrobot	-100	-83.20	0.95	-89.62	3.32	-86.93	3.88

the subsequent training phase proceeds, the older knowledge is replaced by the newer one, coming from the interactions of the agent with the new environment. Regarding the specific hyperparameters of the DQN algorithm (described in Section 2.2.2) when continual learning is enabled, we set the initial ϵ of the ϵ-greedy policy to be equal to the final value of ϵ resulting from the previous training phase (usually a small value, such as 0.01). For the SAC algorithm (described in Section 2.2.3) the entropy regularization coefficient α that controls the exploration-exploitation trade-off is automatically adjusted in the implementation we used. Therefore, we did not act on this parameter when enabling continual learning. Finally, for the PPO algorithm (described in Section 2.2.1), before enabling continual learning, we modified the parameter ϵ that controls how much the new policy can be *different* from the old policy. This parameter is usually in the interval $[0.1, 0.3]$ but we noticed that in continual learning the policy changes in a way that the previous performance cannot be restored if the value of such parameter is set within this interval. Therefore, we set $\epsilon = 0.08$ before starting the training phase on the new environment. Moreover, we checked that the hyperparameters for all the DRL algorithms were reasonably set by running 10 continual learning runs for half of the training time on the same environment where a certain DRL algorithm was originally trained. Then, we made sure that in all the runs the performance of the agent (average reward over 100 evaluation episodes) after each continual learning phase matched the initial performance (i.e., the one reached at the end of the initial training phase). In other words, we made sure that we could restore the performance of an agent on the same environment it was trained on and maintain it for a certain number of time-steps.

In order to choose the environment parameters to consider for evaluation we instantiated each environment with all the possible parameters (i.e. *CartPole 4*, *Pendulum 3*, *MountainCar 3* and *Acrobot 4*), we computed the multipliers for each parameter according to Algorithm 2 and stopped AlphaTest (see line 2 in Algorithm 1). The adaptation condition is the same for all environments, i.e. we deem the adaptation successful if the average reward of the agent trained and evaluated in an environment does not fall below 20% of the average reward the agent gets in the original environment. Likewise, the regression condition is also the same for all environments, with a regression threshold of 5%. Since each discovered multiplier is such that the adaptation condition returns false, we could compute the precise *drop in performance* of the agent w.r.t. the average reward the agent had achieved after the initial training phase. Then, for each environment, we ranked the parameters that are potentially more *critical* in terms of adaptation based on such performance drops. The results of such analysis are shown in Table 1 where the parameters for each environment are ranked by their *performance drops*. For example, the parameters *length* and *cartfriction* are the most critical in the CartPole environment. The parameter *masspole* is the next most critical among the four and *masscart* is the less critical. We have chosen to prioritize the parameters based on *criticality* because evaluating all the possible combinations of parameters for all DRL algorithms would have been too computationally expensive.

Once we have set the multipliers for each parameter, we determined the respective limits, i.e. $L = \langle l, h \rangle$. In particular, we set the limits of each parameter to be $L = \langle l = p.v_0, h = l \times p.m \times 4 \rangle$ if $p.m > 1.0$ otherwise, if $0.0 < p.m < 1.0$, $L = \langle l = \frac{h \times p.m}{4}, h = p.v_0 \rangle$. We multiplied and divided the discovered limits by 4 in order to have a reasonably large range of values for each parameter. Given a certain parameter, the multipliers can be different for each DRL algorithm and, therefore, also the respective limits are different. Hence, to be able to compare the output of different DRL algorithms, when we construct the adaptation grids and compute the volumes, we consider the limits associated with the smallest range of values across all DRL algorithms. For example, if for the CartPole environment, considering the parameter *length*, the limits for SAC are $L_{SAC} = \langle l = 0.5, h = 10.0 \rangle$ and the limits for PPO are $L_{PPO} = \langle l = 0.5, h = 5.0 \rangle$, we construct the adaptation grids considering L_{PPO}. In this way, we are sure that the comparison is fair and the adaptation volumes are comparable. Regarding the anti-regression volume, we computed it as the number of points in the grid

within the adaptation frontier of each algorithm (i.e. with $P_A > 0.5$) that have $P_R \leq 0.5$. The anti-regression volume is normalized over the total number of grid points within the adaptation frontier. For this reason the normalization factor can be different for every algorithm, even considering the same environment configuration. Therefore, anti-regression volumes of different algorithms within the same environment configuration are not comparable in an absolute way, but only relatively to the respective adaptation volumes.

To form a *baseline* for comparison, we devised a *random* exploration procedure of the parameter space. In particular, at each iteration, we randomly choose a possible value in the defined range for each parameter and then train the agent on the resulting environment. The resulting search point $\mathcal{E}$ may have an adaptation probability either below or above threshold (we do not compute the regression probability in this case). Then, to understand if the randomly selected search point $\mathcal{E}$ belongs to the adaptation frontier, we determine the *neighborhood* of $\mathcal{E}$ by modifying the environment parameters in a way that the resulting search points are at a distance ϵ from the already executed search point $\mathcal{E}$, according to the distance function in Equation 1. In particular, for each parameter, we considered two values, respectively greater and smaller than the current parameter value, that respect the distance function, therefore the neighborhood of an executed search point is composed of $n \times 2$ search points, where n is the dimension of the search space (number of parameters being searched). The resulting neighboring search points $\mathcal{E}_i$ are executed and their adaptation probabilities are computed. If a pair $(\mathcal{E}, \mathcal{E}_i)$ satisfies Definition 1, i.e. their adaptation probabilities are one below and one above threshold, a frontier pair is found.

In order to establish a fair comparison between the random baseline and AlphaTest we determined the number of iterations of the random baseline in the following way. For each environment and for each DRL algorithm we first executed AlphaTest for five repetitions and computed the maximum number of search points M resulting from the experiments. We then set the number of iterations for the random method to be $\lfloor \frac{M}{n \times 2} \rceil$, where $\lfloor \cdot \rceil$ means the approximation to the nearest integer. In fact, for each iteration, the random method executes $n \times 2 + 1$ search points, i.e. one search point determined at random plus the $n \times 2$ search points in the neighborhood.

For both the random method and AlphaTest we set the number of repetitions to be equal to five for each experiment (in order to compare the two methods statistically), where each experiment is a pair (*environment configuration*, *DRL algorithm*), the number of training runs for probability estimation (*rpe*) to be equal to three (in order to account for the randomness of DRL algorithms the adaptation probability is estimated with multiple continual learning runs on each environment configuration) and the continual learning time (*clt*) to be equal to half of the initial training time (measured in number of time-steps). For each training run the model is evaluated every N time-steps, where N is a fraction (i.e. 20%) of the total continual learning time, for 20 episodes in order to evaluate the adaptation condition. If the adaptation condition returns true at some point during training, then training is stopped. Otherwise, training goes on until the continual learning time expires and the adaptation condition is also evaluated at the end.

4.3.2 Metrics. In order to characterize accurately the adaptation frontier it is important to obtain the highest number of frontier pairs and maximize their sparseness across the parameter space. The more frontier pairs are found in the parameter space and the more scattered they are, the better the adaptation frontier will be characterized. Therefore, to assess *effectiveness* (RQ_1) we ran both AlphaTest and the random exploration on all the environments by varying only two parameters (see first column of Table 1) for all DRL algorithms. We measured the number of frontier pairs found by AlphaTest and by random exploration, and compute both the *p-value* with the non-parametric *Wilcoxon* test and the *Vargha-Delaney* $\hat{A}_{12}$ effect size to compare the two methods statistically [4]. We also measured the *sparseness* of the frontier pairs. In particular, for each frontier pair we considered a search point with parameter values corresponding

to the average of the parameter values of the two search points in the frontier and measured the pairwise distances between each resulting point and all the others. Each pairwise distance is normalized by dividing it by the maximum distance between two points in the parameter space. We then considered the maximum pairwise distance for each search point in the frontier, averaged them and compared them statistically.

We evaluated *discrimination* (RQ_2) by computing both the adaptation volume and the anti-regression volume for all the DRL algorithms and all the combinations of environment parameters (see Table 1) that we considered in our evaluation (the granularity hyperparameter for volume computation was set to 1, i.e. $g = 1$). We also computed the adaptation and anti-regression heatmaps for all the environment configurations with two parameters, and built frontier visualization plots by applying the t-SNE dimensionality reduction technique together with clustering and decision trees, for all the environment configurations with more than two parameters.

Concerning the *hyperparameters* (RQ_3), we measured the impact on the number of runs skipped and, hence on the time saved during search, of the *dominance* option of AlphaTest. Moreover, we measured the impact on the volume of increasing the number of runs for probability estimation from three to five. In particular, we measured the standard error of the mean (SEM) of the adaptation volume, i.e. the standard deviation of the adaptation volume divided by the square root of the runs used for probability estimation. We also varied the granularity of the adaptation grid. Specifically we doubled the original value considered in RQ_2 and we halved it, to measure the impact both on the adaptation volume metric and on the percentage of search points that have a collision in the grid. Finally, for each environment, we took the worst performing algorithm in terms of adaptation volume and carried out other five repetitions of AlphaTest by increasing the continual learning time to be equal to the initial training time in order to measure the impact on adaptation and anti-regression volumes of the continual learning time hyperparameter.

Table 3. RQ_1: effectiveness results

	# search points		# frontier pairs		sparseness	
	alphatest	random	alphatest	random	alphatest	random
CartPole 2						
PPO	46	55	**5.6**	0.8	**0.58**	0.12
SAC	54	68	**6.0**	0.6	**0.42**	0.00
DQN	49	56	5.8	6.6	0.50	0.68
Pendulum 2						
PPO	112	136	**14.0**	3.4	0.06	0.06
SAC	83	103	**8.0**	0.4	0.01	0.00
DQN	157	197	**17.0**	4.0	0.03	0.08
MountainCar 2						
PPO	108	126	**15.0**	4.0	0.06	0.03
SAC	312	350	**43.6**	7.2	0.04	0.03
DQN	361	396	**50.4**	8.0	0.05	0.05
Acrobot 2						
PPO	148	179	**19.0**	1.2	**0.11**	0.05
SAC	85	110	**11.0**	0.8	**0.09**	0.01
DQN	119	154	**15.0**	0.8	**0.09**	0.01

4.4 Results

Effectiveness (RQ$_1$). Table 3 shows the comparison between AlphaTest and random for all DRL algorithms in the four environments in terms of number of search points (Columns 1–2), number of frontier pairs (Columns 3–4) and sparseness of the pairs in the frontier (Columns 5–6).

We can notice that AlphaTest and random both have roughly the same *number of search points* executed, on average, although there is a slight advantage for random which explores more search points in all environments. This data confirms that the comparison between AlphaTest and random is fair.

In terms of *number of frontier pairs* Table 3 shows that AlphaTest finds more frontier pairs than random. Bold values indicate that the difference between the two means is statistically significant (i.e. p-value is below 0.05) and the underline indicates when the magnitude of the effect size $\hat{A}_{12}$ is large. In all but one case (DQN in *CartPole 2*), the number of frontier pairs found by AlphaTest is larger than random and the difference between the two means is both statistically significant and the magnitude of the effect size is large.

Regarding *sparseness* there is a significant difference only in two out of four environments, namely *CartPole 2* (considering PPO and SAC) and *Acrobot 2*. In the remaining environments, i.e. *Pendulum 2* and *MountainCar 2*, the effect size is large in favor of AlphaTest when considering the algorithm SAC. The small values of sparseness in *Pendulum 2*, *MountainCar 2* and *Acrobot 2* are due to the frontier pairs being concentrated near the origin of the two-dimensional parameter space.

RQ$_1$: Considering the same number of search points, AlphaTest finds *significantly* more frontier points than random. Moreover, in two out of four environments and for most of the DRL algorithms, the frontier pairs found by AlphaTest are *significantly* more sparse than those found by random.

Discrimination (RQ$_2$). The first part of Table 4 (i.e. Columns 1–4) shows how AlphaTest discriminates between different DRL algorithms in different environments. In particular Column 1 shows the average adaptation volume considering five repetitions of AlphaTest and Column 2 shows the standard deviation of the adaptation volume in percentage. Similarly, Columns 3–4 show the average anti-regression volume and its standard deviation in percentage respectively. The adaptation and anti-regression volumes were computed by considering the granularity $g = 1.0$ except for the environments *CartPole 4* and *Acrobot 4* where the volumes were computed with $g = 0.5$, in order to save computation time. The adaptation volumes of different DRL algorithms can be compared within the same environment configuration, since the adaptation grids are constructed with the same parameter ranges. Regarding the anti-regression volumes, each normalization factor depends on the number of points within the adaptation frontier of the specific DRL algorithm. Therefore, each anti-regression volume indicates how much a certain DRL algorithm is able to remember how to behave in the original environment (i.e. it does not regress) relative to its adaptation volume. For example in *CartPole 2* the PPO algorithm, on average, has regressions only within 5% of the adaptation frontier (i.e., the anti-regression volume is 95%).

Considering the average adaptation volumes, we marked in green (with suffix (1)) the best adaptation volume in each environment configuration, in yellow (with suffix (2)) the second best adaptation volume and in red (with suffix (3)) the worst adaptation volume among the three DRL algorithms. We can see that PPO achieves the best adaptation volume in three environments out of four (i.e. except for *Pendulum*) and it is never the worst among the three. SAC is the best in *Pendulum* but it is the worst in two environments, namely *CartPole* and *MountainCar*. DQN, on the other hand, is never

Table 4. RQ_2 discrimination results and RQ_3 results for AlphaTest's dominance hyperparameter

	RQ_2				RQ_3 †				
	adapt. vol.	% adapt. vol.	anti-regr. vol.	% anti-regr. vol.	# s.p.	# s.p. skipped	% s.p. skipped	search time (h)	% time saving
CartPole 2									
PPO	$2.9e^{-1}$ (1)	30.06	0.95	6.00	46	28	60.87	0.32	76.02
SAC	$1.2e^{-1}$ (3)	16.94	0.86	10.28	54	33	61.85	0.99	145.41
DQN	$2.7e^{-1}$ (2)	11.37	0.76	17.27	49	21	43.21	1.83	70.90
CartPole 3									
PPO	$5.0e^{-2}$ (1)	43.06	0.99	0.62	198	119	59.88	1.43	82.20
SAC	$3.2e^{-2}$ (3)	26.05	0.95	3.12	224	148	66.07	4.31	159.19
DQN	$3.5e^{-2}$ (2)	12.23	0.95	3.20	236	130	54.95	7.00	108.49
CartPole 4*									
PPO	$1.4e^{-2}$ (1)	31.91	1.00	0.16	655	443	67.65	4.32	110.78
SAC	$5.0e^{-3}$ (2)	18.90	0.99	0.48	785	595	75.83	11.52	229.70
DQN	$4.0e^{-4}$ (3)	11.57	0.95	2.33	856	548	64.04	21.08	155.80
Pendulum 2									
PPO	$3.0e^{-2}$ (2)	8.92	0.80	9.57	112	85	75.62	1.08	229.87
SAC	$5.0e^{-2}$ (1)	52.18	0.85	4.84	83	65	78.99	0.71	186.98
DQN	$1.4e^{-2}$ (3)	9.64	0.88	8.26	157	132	83.82	1.10	230.97
Pendulum 3									
PPO	$5.0e^{-3}$ (2)	9.42	0.91	3.63	1062	958	90.24	3.70	353.55
SAC	$1.4e^{-2}$ (1)	41.48	0.90	4.23	778	718	92.36	2.35	380.76
DQN	$2.0e^{-3}$ (3)	14.61	0.82	8.55	1916	1822	95.09	3.50	455.60
MountainCar 2									
PPO	$1.0e^{-2}$ (1)	4.76	0.55	19.56	108	78	72.27	3.09	207.91
SAC	$2.0e^{-3}$ (3)	34.84	0.82	17.42	312	275	88.16	2.69	360.04
DQN	$4.0e^{-3}$ (2)	6.29	0.80	14.59	361	321	88.87	3.30	386.62
MountainCar 3									
PPO	$2.0e^{-3}$ (1)	5.91	0.86	7.97	371	281	75.78	7.74	226.79
SAC	$4.0e^{-4}$ (3)	4.57	0.95	4.20	1016	897	88.25	7.21	365.50
DQN	$7.0e^{-4}$ (2)	8.11	0.95	1.03	1199	1060	88.36	10.59	371.03
Acrobot 2									
PPO	$1.0e^{-2}$ (1)	8.18	1.00	0.00	148	110	74.22	1.43	188.99
SAC	$9.0e^{-3}$ (2)	6.17	0.93	9.55	85	53	62.59	1.88	155.01
DQN	$9.0e^{-3}$ (3)	4.80	1.00	0.00	119	86	71.81	3.57	207.91
Acrobot 3									
PPO	$6.0e^{-4}$ (1)	2.85	1.00	0.00	664	538	81.04	4.21	272.13
SAC	$3.0e^{-4}$ (3)	3.66	1.00	0.27	391	296	75.81	5.18	271.89
DQN	$4.0e^{-4}$ (2)	6.08	1.00	0.20	544	442	81.23	10.30	351.29
Acrobot 4*									
PPO	$4.0e^{-5}$ (1)	5.46	1.00	0.00	1970	1640	83.23	11.21	325.94
SAC	$2.0e^{-5}$ (3)	7.70	1.00	0.56	1276	1054	82.59	11.67	372.27
DQN	$2.0e^{-5}$ (2)	3.46	0.99	0.34	1726	1455	84.26	27.13	383.58

* *$g = 0.5$ to save time; $g = 1.0$ in all other cases* † *hyperparameter dominance analysis*

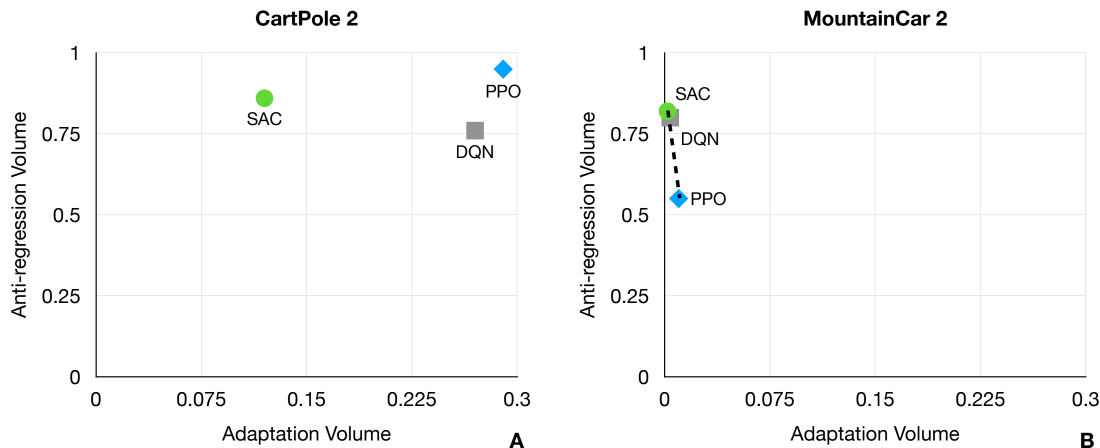

Fig. 5. Trade-off between adaptation and anti-regression volumes

the best and it is the worst in *Pendulum*. Interestingly the ranking is maintained in *Pendulum* and *MountainCar* when passing from two parameters, i.e. *Pendulum 2* and *MountainCar 2*, to three parameters, i.e. *Pendulum 3* and *MountainCar 3*. Moreover, also in *CartPole* and *Acrobot* the algorithm that obtains the best adaptation volume, i.e. PPO, remains the best when considering more than two parameters. On the other hand, in *Acrobot* the algorithm SAC is moved from the second to the third position when considering three and four parameters, i.e. *Acrobot 3* and *Acrobot 4*, being surpassed in the ranking by DQN (in *Acrobot 2* the average adaptation volume of DQN is smaller than that of SAC only by a negligible amount). In *CartPole*, instead, the switch in the ranking between second and third position takes place when moving from three to four parameters, i.e. from *CartPole 3* to *CartPole 4*, and it involves SAC and DQN with the latter moving from the second to the third position in *CartPole 4*. In summary, although PPO is the DRL algorithm that adapts better in three out of four considered environments, our experiments are not sufficient to deem PPO superior than others in absolute terms, since the number of environments we trained PPO on is relatively small and we did not carry out any hyperparameter tuning, which DRL algorithms have been shown to be sensitive to [29].

The other interesting dimension along which we compare the selected DRL algorithms is the anti-regression volume. Figure 5 shows the trade-off between adaptation and anti-regression volumes. In particular, Figure 5.A shows such trade-off in *CartPole 2* whereas Figure 5.B shows it in *MountainCar 2*. We can see that in *CartPole 2* there is clearly a DRL algorithm, namely PPO, that *dominates* the others, i.e. it has the highest adaptation volume and, at the same time, it also has the least amount of regression (i.e. the highest anti-regression volume). On the other hand, Figure 5.B shows that there is no clear winner among the three algorithms applied to *MountainCar 2*. In fact, the *Pareto front*, indicated with a dashed black line, suggests that as the adaptation volume increases, there is also an increase in the amount of regressions that a certain algorithm has (i.e., the anti-regression volume decreases). In other words, in *MountainCar 2* and for all DRL algorithms, adapting to new environments means forgetting how to behave in the original environment. For DQN, whose average adaptation volume increases by $\approx 39\%$ w.r.t. the average adaptation volume of SAC, its average anti-regression volume decreases only by $\approx 2\%$. Instead, PPO has an average adaptation volume that is $\approx 80\%$ higher than the one of DQN but its average anti-regression volume is $\approx 38\%$ smaller. This could be due to the fact that DQN and SAC are both off-policy algorithms (see Section 2). The replay memory helps these

algorithms to replay the previous experience while learning new behaviors, hence forgetting less how to behave in the original environment than PPO which, being an on-policy algorithm, does not use a replay memory to learn. However, this trade-off seems to be significant only when considering two parameters in an environment. When the number of parameters increases, correspondingly the adaptation volume of a certain algorithm decreases and its anti-regression volume tends to increase (except for DQN in *Pendulum* and DQN from *Acrobot 3* to *Acrobot 4*). One possible explanation of this phenomenon might be that, since the adaptation volume is smaller in higher dimensions, the new environments are more similar to the original environment (i.e. the origin) and, as a consequence, the algorithm has less regressions w.r.t. new environments that are not so far away from the origin.

Besides the quantification of the adaptation capabilities of a certain algorithm with the computation of the adaptation volume, our approach also produces the *adaptation heatmap* for a certain algorithm when the considered environment has two parameters. Figure 6 shows the adaptation heatmaps for PPO, SAC and DQN (respectively Figure 6.A, Figure 6.B, and Figure 6.C) in the *CartPole 2* environment. The x axis shows the values of the parameter *length* that spans from 0.50 to 4.0, whereas the y axis shows the values of the parameter *cartfriction* that has the interval $[0.0, 51.20]$. The ranges of these two parameters are the same for each DRL algorithm and the colors in each heatmap indicate the adaptation probability, as shown by the color bar on the bottom right corner of the figure. The adaptation frontier is the yellow continuous line between the green (adaptation successful) and the red (adaptation failed) regions of the heatmap, while the black dots indicate the search points sampled by the search procedure of AlphaTest. From the maps we can see that the parameter *length* is more critical in terms of adaptation capabilities than the *cartfriction* parameter, for all the algorithms. In particular, no DRL algorithm adapted when *length* = 4.0, although DQN seems to be the best at tolerating the increase of such parameter, but all DRL algorithms adapted when *cartfriction* = 51.20. The adaptation heatmaps are easily interpretable by developers, as they show the regions of the parameter space where the adaptation frontier lies and where we can expect a certain algorithm to successfully adapt or not when the initial conditions of the environment change.

Figure 7 shows the adaptation and anti-regression heatmaps of the DQN algorithm in the *CartPole 2* environment (respectively Figure 7.A and Figure 7.B, with Figure 7.A the same as Figure 6.C). In the anti-regression heatmap (Figure 7.B) the color code is reversed w.r.t. the adaptation heatmap: the heatmap is red where the regression probability is 1.0 and it is green when the regression probability is 0.0. The gray color indicates the region of the parameter space where the anti-regression heatmap is not defined, i.e. outside the boundary delimited by the adaptation frontier indicated by the yellow continuous line in Figure 7.A. We can notice that the anti-regression frontier is not continuous and there can be islands of regressions inside regions of the parameter space where the algorithm does not regress (see the red region in Figure 7.B, where the *length* parameter is within $[1.20, 1.76]$ and the *cartfriction* parameter is within $[0.00, 2.05]$).

When the environment has more than two parameters, AlphaTest provides a visualization of the adaptation frontier by means of the t-SNE dimensionality reduction technique and k-means clustering applied to the t-SNE lower dimensional vectors. Figure 8.A shows the frontier pairs found by AlphaTest in the *CartPole 3* environment for PPO, i.e. by considering the three parameters *length*, *cartfriction* and *masspole*. After reducing the search space dimensionality from 3 to 2 by means of t-SNE, we perform silhouette analysis to find the optimal number k of clusters produced by k-means. In Figure 8.A, the six resulting clusters are represented as regions with the same background color. A magenta star with a label indicating the cluster ID (*class-i*) is positioned at each cluster centroid. In the figure the original environment is indicated with a blue circle and each frontier pair is displayed as two points, one green (where

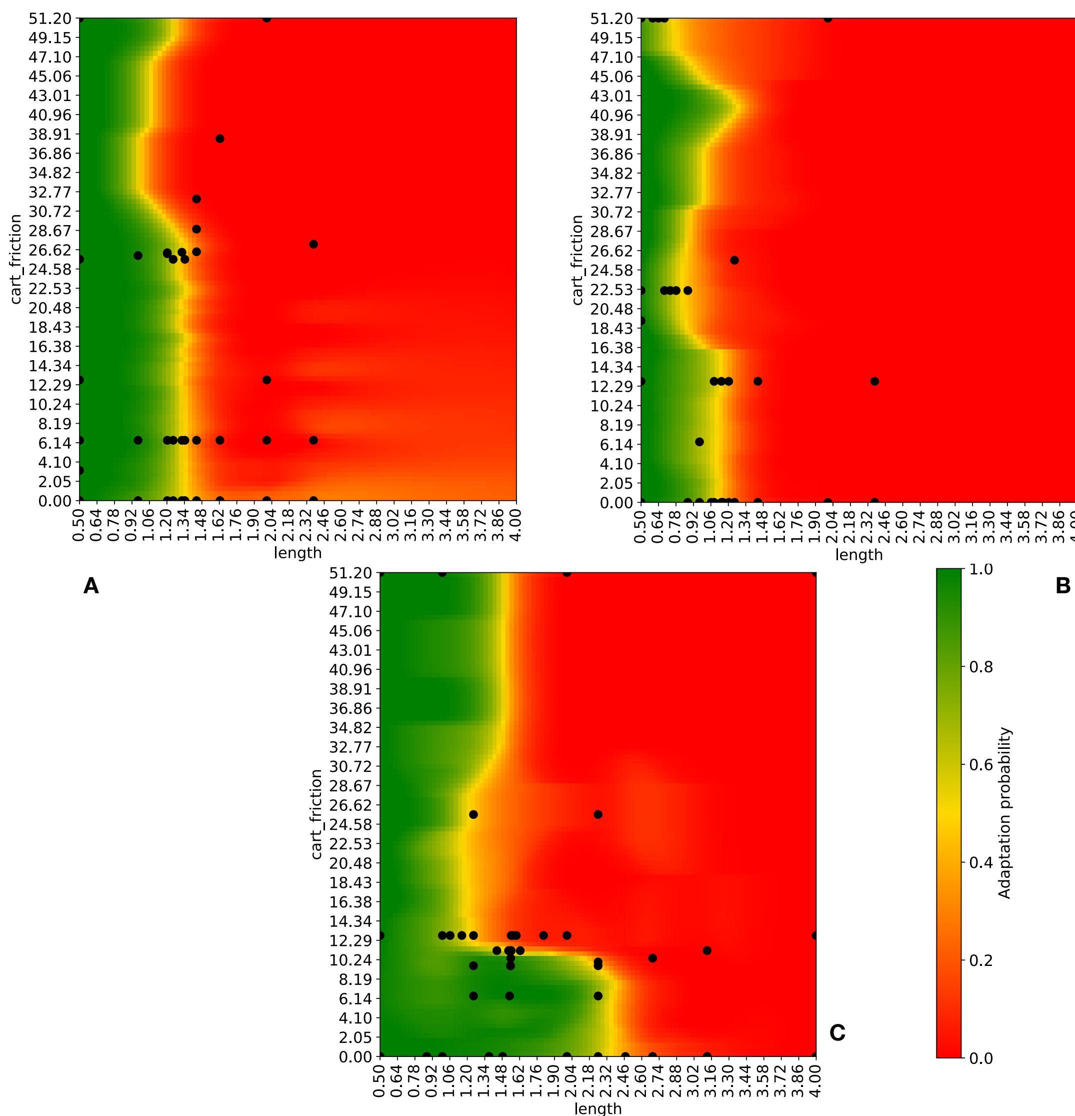

Fig. 6. Adaptation probability heatmaps for PPO, SAC and DQN in the CartPole 2 environment

the algorithm adapted) and one red (where the adaptation was not successful), connected by a dashed gray line (for visual clarity we omitted the labels indicating the values of the three parameters of each component of a frontier pair).

Then, we trained a decision tree to classify each point of a frontier pair by the cluster they belong to, based on its *features*, i.e. the values of the parameters for that particular environment. Figure 8.B shows the decision tree trained on the frontier pairs, using the cluster IDs as class labels, for the clusters shown in Figure 8.A. The leaf nodes are highlighted in yellow and each leaf node is *pure* (i.e. its *Gini* impurity metric is 0.0). The decision tree tells us that the *length* parameter is not crucial for the classification of the frontier pairs, since it is not present in any of the decision

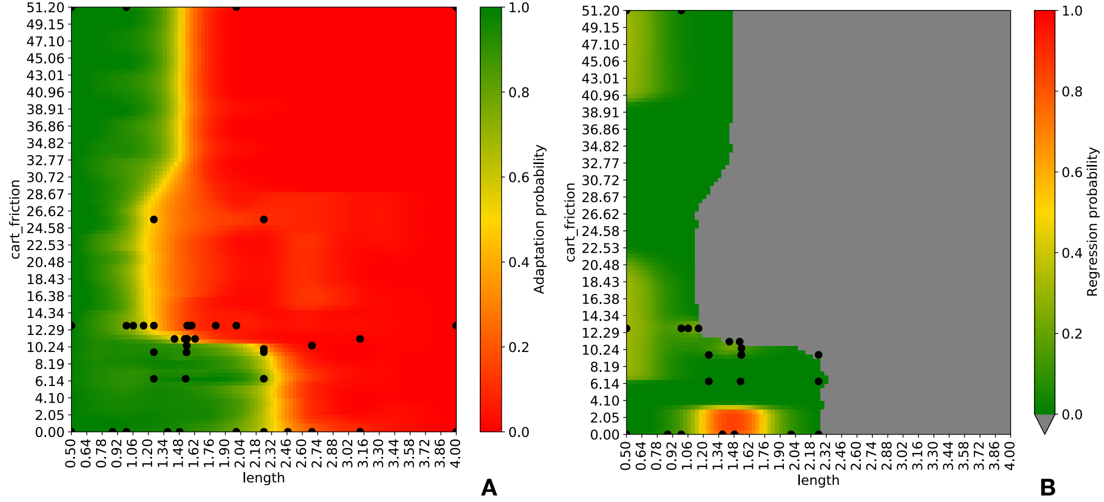

Fig. 7. Adaptation and anti-regression probability heatmaps for DQN algorithm in the CartPole 2 environment

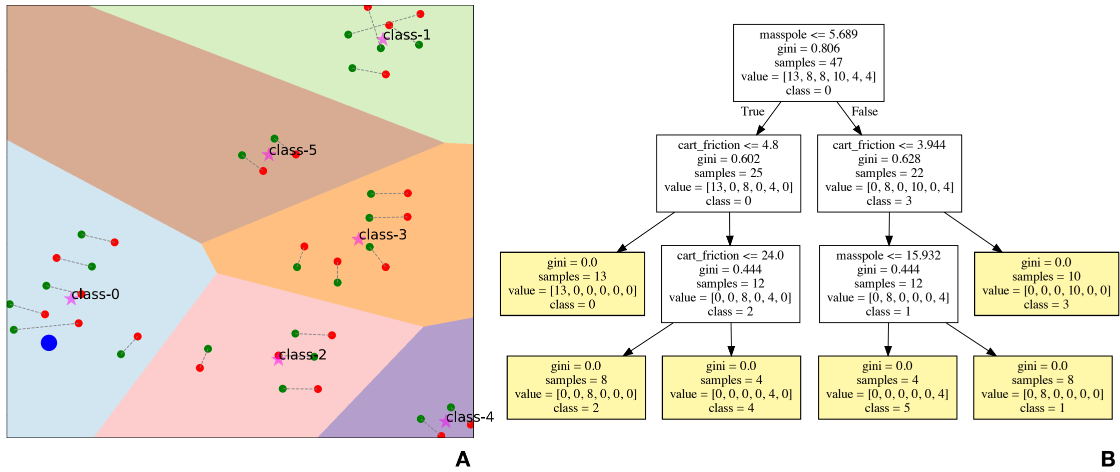

Fig. 8. Visualization of frontier pairs found by AlphaTest in the Cartpole 3 environment for PPO and their classification by means of a decision tree

nodes nor in the root node. Moreover, the decision tree tells us in which regions of the parameter space the frontier pairs are clustered. For example, from the decision tree in Figure 8.B, we can see that the six frontier pairs that have *masspole* ≤ 5.689 and *cartfriction* ≤ 4.8 are clustered together (with label *class-0* on the bottom left corner of Figure 8.A). When *cartfriction* $\in (4.8, 24.0]$, instead, we get the four frontier pairs that belong to the cluster with ID *class-2* (bottom center of Figure 8.A).[3].

[3]The visualizations of the frontier pairs in more than two dimensions and of the heatmaps for all the environment configurations we considered are available for download at [47]

RQ_2: The adaptation volume allows the users of AlphaTest to discriminate a DRL algorithm with high adaptation capabilities from another DRL algorithm with low adaptation capabilities. Sometimes such adaptation comes at the cost of more regressions on the original environment (i.e. higher anti-regression volume), therefore the user needs to decide which of the two properties is more important for the specific situation. AlphaTest also provides the users with a visualization of the adaptation frontier to better evaluate the behaviors of the DRL agent in the parameter space of the environment of interest, both when two (heatmap) and when more than two (clusters and decision tree) parameters are considered.

Hyperparameters (RQ_3). Table 4 shows, on the right hand side of the table, the results for the hyperparameter *dominance*. Column 5 shows the average number of search points, across five repetitions, sampled by AlphaTest during the search phase. Such number increases when moving from an environment with two parameters to an environment with three parameters, and from three parameters to four parameters, since the combinations of possible environment configurations increases. Column 6 shows the average number of search points skipped, i.e. not executed, by enabling dominance analysis both in the exponential and in the binary search sub-phases. Column 7 shows the percentage of search points skipped out of the total number of search points, whereas Column 8, shows the search time, measured in hours, of the search phase of AlphaTest for a single repetition. Column 9 shows the average percentage of time saved by enabling dominance analysis, measured as the ratio between the estimated execution time for the skipped points and the actual execution time of the search, with dominance analysis enabled. It gives the percentage of the actual execution time that would be added if search point skipping were disabled. In practice, we compute it by multiplying the number of skipped search points by the time to execute a single search point (estimated as the average across all the executed search points in all repetitions for a certain algorithm) and dividing the product by the actual search time.

Interestingly, for all environments except *MountainCar* and for all algorithms, the average number of skipped search points in percentage increases by ≈ 13% when moving from the environment configuration with the lowest number of parameters (e.g. *CartPole 2*) to the environment configuration with the highest number of parameters (e.g. *CartPole 4*). In *MountainCar* such increase is only 1%. Moreover, dominance analysis seems to be quite dependent on the specific environment. In particular, considering the environment configurations with two parameters, in *MountainCar 2* dominance analysis is able to skip the execution of the highest number of search points (i.e. ≈ 83%), while in *CartPole 2* it skips the lowest number of search points (i.e. ≈ 55%). The impact of the specific DRL algorithm on dominance is less significant but not negligible in some environments. For example, in *CartPole*, dominance analysis skips 54% of the total number of search points for DQN and 68% of the total number of search points for SAC. Similarly, in *MountainCar*, dominance analysis skips 74% of the total number of search points for PPO and 88% for DQN. The differences in the remaining environments are less pronounced. The corresponding percentage of time saved depends on the time to execute a single search point that can greatly vary across different algorithms. For example, in *Acrobot 2*, the percentage of skipped search points is higher for PPO than for DQN (respectively 74% vs 71%) but the time saving percentage is lower for PPO than for DQN (respectively 189% vs 207%). In fact, the time to execute a single search point for PPO is 1.5 minutes, whereas for DQN it is 5.2 minutes.

Table 5 shows the results for the remaining hyperparameters of AlphaTest. The first five columns report the metric values obtained when running AlphaTest with the default parameters, i.e. granularity $g = 1.0$, runs for adaptation probability estimation $rpe = 3$ and continual learning time $clt = half$ the initial training time. In particular, Column 1 shows the average percentage of search points colliding when constructing the adaptation grids with granularity

$g = 1.0$. Such adaptation grids are constructed considering the original limits for each algorithm (not the lowest across algorithms, as in Table 4 where different algorithms are being compared). Column 2 shows the average adaptation volumes when *clt* = *half* and Column 4 is the average search time, in hours, for a single repetition of AlphaTest. Column 3 reports the average standard error of the mean (i.e. SEM) of the adaptation volume when considering three runs for estimating the adaptation probability. Column 5 shows the average anti-regression volumes.

Columns 6–9 are related to the granularity hyperparameter. In particular, we analyze the percentage of collisions and the adaptation volume percentage variation when the granularity is half the original (i.e. $g = 0.5$) and when the granularity is twice as much (i.e. $g = 2.0$). As expected, the percentage of collisions approximately doubles on average when halving the granularity, i.e. $g = 0.5$, and it becomes half of the original value on average when $g = 2.0$. More interesting is what happens to the adaptation volume that, on average, varies by ≈ 10% when $g = 0.5$ (the maximum variation of the adaptation volume, i.e. 24%, happens in *MountainCar 2* for SAC and the minimum variation, i.e. 1%, happens in *CartPole 2* for DQN), whereas when $g = 2.0$ the average variation is ≈ 5% (the maximum variation, 11.49%, happens in *Pendulum 2* for DQN and the minimum variation, 0.55%, in *CartPole 2* also for DQN). We can also notice that the adaptation volume percentage difference is high when the percentage of collisions is also high. In particular, when considering $g = 2.0$, i.e. a grid where the adaptation volume is estimated better, with less collisions, than at $g = 1.0$, the

Table 5. RQ$_3$: hyperparameters results

	orig.†					granularity				prob. est.				time	
						g = 0.5		g = 2.0		rpe = 5				clt = full	
	% collisions	adapt. vol.	SEM $\left(\frac{\sigma}{\sqrt{3}}\right)$	search time (h)	anti-regr. vol.	% collisions	adapt. vol. % diff.	% collisions	adapt. vol. % diff.	SEM $\left(\frac{\sigma}{\sqrt{5}}\right)$	SEM % decr.	search time (h)	search time % incr.	adapt. vol. % incr.	anti-regr. vol. % decr.
CartPole 2															
PPO	5.7	$1.4e^{-1}$	$2.1e^{-2}$	0.3	0.97	15.3	2.2	2.9	0.7	$9.9e^{-3}$	54.1	0.6	104.3	-	-
SAC	12.9	$6.7e^{-2}$	$6.6e^{-3}$	1.0	0.94	24.9	6.4	3.8	1.4	$3.5e^{-3}$	47.1	2.0	106.8	47.4	2.6
DQN	2.5	$2.8e^{-1}$	$1.8e^{-2}$	1.8	0.76	7.1	1.0	1.7	0.5	$9.4e^{-3}$	48.4	2.9	56.8	-	-
Pendulum 2															
PPO	17.4	$2.5e^{-2}$	$1.1e^{-3}$	1.1	0.78	29.8	3.7	6.9	3.8	$1.0e^{-3}$	12.1	1.6	50.4	-	-
SAC	20.5	$5.8e^{-2}$	$1.1e^{-2}$	0.7	0.88	33.9	10.8	10.7	3.6	$5.4e^{-3}$	56.1	1.1	54.8	-	-
DQN	22.2	$1.3e^{-2}$	$3.0e^{-4}$	1.1	0.89	39.5	13.7	9.1	11.5	$1.0e^{-4}$	67.8	1.7	56.0	43.7	13.0
MountainCar 2															
PPO	10.6	$7.6e^{-3}$	$2.0e^{-4}$	3.1	0.57	25.5	12.4	4.6	8.0	$1.0e^{-4}$	51.4	4.1	31.9	-	-
SAC	31.5	$3.0e^{-3}$	$1.0e^{-4}$	2.7	0.83	50.2	24.1	17.5	9.3	$4.0e^{-5}$	34.9	4.4	62.4	23.4	20.5
DQN	32.6	$4.9e^{-3}$	$2.0e^{-4}$	3.3	0.81	50.1	17.5	18.0	5.7	$1.0e^{-4}$	49.6	6.3	90.4	-	-
Acrobot 2															
PPO	17.9	$1.1e^{-2}$	$5.0e^{-4}$	1.4	1.00	35.7	6.0	9.0	5.2	$4.0e^{-4}$	22.5	2.2	56.6	-	-
SAC	16.1	$6.6e^{-3}$	$2.0e^{-4}$	1.9	0.92	32.6	9.4	6.0	4.2	$1.0e^{-4}$	70.8	3.6	89.9	-	-
DQN	16.5	$7.3e^{-3}$	$2.0e^{-4}$	3.6	1.00	33.1	12.5	6.2	3.2	$1.0e^{-4}$	67.7	6.7	88.9	24.4	3.8

† *g = 1.0, rpe = 3, clt = half*

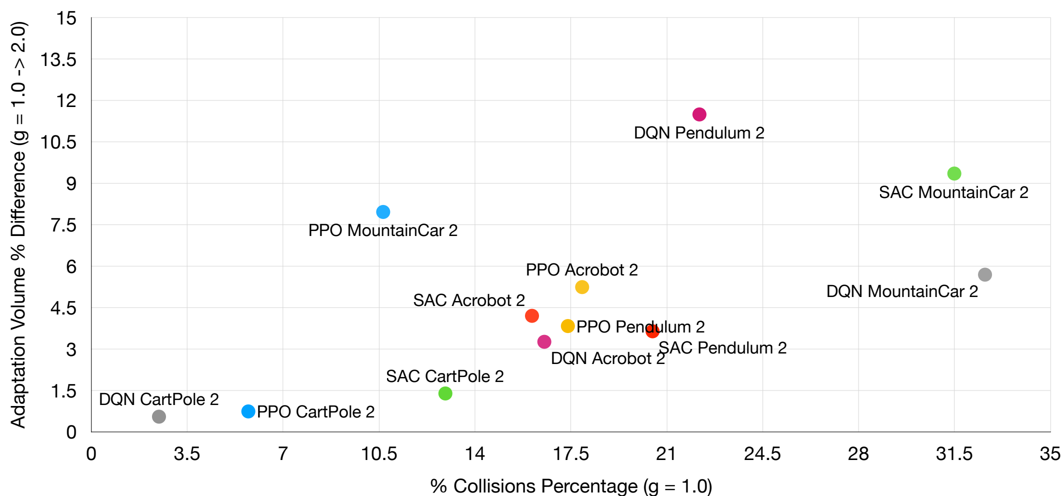

Fig. 9. Relation between collisions percentage and adaptation volume percentage difference

adaptation volume percentage difference is non negligible (> 1%) only when the initial percentage of collisions is high (> 10%). As Figure 9 shows, the adaptation volume percentage difference, when the granularity is doubled, increases linearly (with angular coefficient between zero and one) with the collisions percentage. In terms of execution time, halving the granularity decreases the time to approximate the adaptation grid with the nearest neighbor algorithm by 60% (the number of points in the grid decreases from $10k$ to $2.5k$), while doubling the granularity increases such time by 200% (the number of points in the grid increases from $10k$ to $40k$).

Columns 10–13 are about the runs of continual learning needed to estimate the adaptation probability of a certain algorithm given an environment configuration. Specifically, Column 10 shows the standard error of the mean (SEM) of the adaptation volume when $rpe = 5$, i.e. when we use five runs of continual learning to estimate the adaptation probability. Column 11 shows the percentage decrease of SEM when moving from $rpe = 3$ to $rpe = 5$. We can see that SEM always decreases (on average it decreases by 48%), suggesting that the adaptation volume across 5 repetitions is more stable, since the adaptation probability is better estimated. The maximum decrease, i.e. 71%, happens in the environment *Acrobot 2* for SAC, whereas the minimum decrease, i.e. 12%, happens in *Pendulum 2* for PPO. The search time percentage increase is reported in Column 13 and, on average, the time to carry out the search phase in AlphaTest increases by 71%. The maximum increase happens in *CartPole 2* for SAC, where the search time more than doubles (i.e. it increases by 107%), whereas the minimum increase, i.e. 32%, happens in *MountainCar 2* for PPO.

Finally, Columns 14–15 are related to the time hyperparameter, specifically the continual learning time that was originally set to *half* of the initial training time. We want to study the impact of increasing the continual learning time, making it equal to the *full* training time, on the adaptation volume and the anti-regression volume. We considered one algorithm in each environment for this study, in particular the algorithm that had the worst adaptation volume in the comparison done in Table 4. We can see in Column 14 that, on average the adaptation volume increases by 35%, with the maximum being 47% in *CartPole 2* for PPO and the minimum being 23% in *MountainCar* for SAC. Column 15 shows that the anti-regression volume always decreases, on average by 10%, with the maximum being 20.5% in *MountainCar* for SAC and the minimum being 2.6% in *CartPole* also for SAC.

RQ_3 (dominance): The most critical hyperparameter of AlphaTest is the *dominance* option which should always be enabled, since it is beneficial to save computation time (on average, in our experiments, we measured a percentage saving of $\approx$ 250%, i.e. 2.5× more computation time would be needed without dominance).

RQ_3 (granularity): The granularity hyperparameter g needs to be chosen as a trade-off between the number of collisions and the time to carry out the parameter space approximation phase. We recommend decreasing the granularity hyperparameter when the number of parameters in an environment increases. Indeed, the time to compute the volume increases by an order of magnitude when adding a new parameter and considering the same granularity.

RQ_3 (number of runs): The number of runs for adaptation probability estimation (hyperparameter *rpe*) is important for those algorithms that are more *unstable*, i.e. those that tend to produce different results when trained on the same environment multiple times. Our results show that, when increasing *rpe* from three to five, on average, the standard error of the adaptation volume mean decreases by 35% for PPO, by 52% for SAC and by 58% for DQN. As a consequence increasing *rpe* seems more beneficial for SAC and DQN. Correspondingly, the search time increases, on average, by 71% (considering all algorithms).

RQ_3 (continual learning time): As expected, by increasing the continual learning time hyperparameter we can increase the adaptation capabilities of the agent but, at the same time, we also decrease its capabilities to perform well in the original environment when adaptation is successful. In some environments such trade-off is negligible (e.g. in *CartPole 2* the ratio between the anti-regression volume decrease and the adaptation volume increase is only 0.05), whereas others it can be significant (e.g. in *MountainCar 2* such ratio is 0.88).

4.5 Threats to Validity and Limitations

In this section we discuss the threads to validity that could affect our results [91]. Threats to *internal validity* might come from how the empirical study was carried out. To be fair, when considering effectiveness, we compared AlphaTest and random under identical parameter settings (e.g., same number of search points), on the same environments and DRL algorithms.

Threats to *conclusion validity* are related to random variations and inappropriate use of statistical tests. To mitigate these threats, we ran each experiment (both with AlphaTest and random) five times and used the non-parametric Wilcoxon test and the Vargha-Delaney effect size for statistical testing. Moreover, to account for the random initialization of DRL algorithms, we ran them three times on the same environment configuration with different random seeds, so as to better estimate adaptation and anti-regression probabilities.

Using a limited number of environments poses an *external validity* threat. Although more subjects would be needed to fully assess the generalizability of our results, we have chosen all the four classic control environments that are highly used in the DRL community, as they are part of the popular *gym* library.

With respect to *reproducibility* of our results, the source code of AlphaTest, the experimental results and all the environments are available online [47], making the evaluation repeatable and our results reproducible.

One limitation of our approach is that the time to approximate the frontier in the parameter space, needed to compute the adaptation and anti-regression volumes, increases exponentially with the number of parameters of the environment if the granularity hyperparameter stays fixed. To address this limitation, developers can consider two parameters at a time and indeed in most of our experiments we also considered two parameters at a time, although we also conducted experiments involving three or four parameters. The latter experiments confirmed the results obtained on the reduced dimensionality space, showing that it is often possible to extrapolate the experimental outcomes. In our future work we plan to investigate techniques to scale parameter approximation to larger dimensionality.

Our approach makes the assumption that a total order relation exists between the parameter values, such that the complexity of the environment increases/decreases in the direction (or in the opposite direction) of the parameter change. If a parameter satisfies this requirement then it is possible to apply the search phase of AlphaTest, i.e. exponential search and binary search. Another assumption, made by the binary search, is that two environment configurations can be moved ϵ-close to each other in the parameter space. If a parameter assumes discrete values then this might not be possible. In our future work we plan to address these limitations by supporting environments with discrete parameters, including those that do not admit a total order relation.

5 DISCUSSION

Let us consider a context in which a software engineer needs to evaluate and test a system that has adaptation capabilities (e.g. a reinforcement learning based system). In the following we consider three questions that the software engineer may want to address in such context, and describe the analyses and visualizations that AlphaTest offers to answer those questions.

Will the system adapt to new unseen environments? The ability to adapt to new unseen environments is the primary objective of a continual learning system. Assuming that the environment can be parameterized AlphaTest can efficiently manipulate the parameters to find the adaptation frontier of the system. This is useful for the software engineer not only because it provides information on whether the system is able to adapt but also in which regions of the parameter space the adaptation is successful or not. For example in the *CartPole 2* environment the DQN algorithm (see the adaptation heatmap in Figure 6.C) is able to adapt for values of the *length* parameter (which determines the length of the pole) less than 2.32 when the friction on the cart (i.e. parameter *cartfriction*) is 0, i.e. its default value. The range of values for the *cartfriction* parameter in which the system can adapt is, instead, much bigger. The adaptation heatmap produced by AlphaTest tells the software engineer which parameters are more critical for adaptation and in which range of values the system is expected to work once deployed.

Does the system have regressions when adaptation is successful? Depending on the task at hand, it might be desirable to preserve an *acceptable* performance on the environment the system was initially trained on while adapting successfully to new environment conditions. To this aim, the anti-regression heatmap produced by AlphaTest helps the software engineer understand in which regions of the parameter space regressions arise. For example, the anti-regression heatmap in Figure 7.B shows that the DQN algorithm in the *CartPole 2* environment was able to adapt in the region where $length \in [1.2, 1.7]$ and $cartfriction \in [0.0, 3.0]$ but once adapted to those environment configurations it was not able to *perform well* on the environment it was initially trained on (i.e. *length = 0.5* and *cartfriction = 0.0*, the axes origin of the heatmap). The regression regions in the anti-regression heatmap provide an indication for the software engineer to start investigating the issues causing the regressions, in case they matter for the task at hand.

How to compare the performance of different algorithms? Ultimately, the software engineer needs to choose the system to deploy. The choice might be between different available algorithms, or even different versions of the same algorithm. AlphaTest offers the adaptation volume metric to allow the software engineer to discriminate a system with high adaptation capabilities from another with low adaptation capabilities. The anti-regression volume is also computed, such that the software engineer can decide to trade off one property for another depending on the context. Adaptation volumes provide an overall measure of performance of the adaptation capabilities of the system regardless of the number of parameters the environment has. However, a more in depth analysis of the adaptation capabilities on specific regions of the parameter space might be needed. For example, Figure 6 shows the adaptation heatmaps of three different DRL algorithms in the *CartPole 2* environment. Visually the figure shows that the DQN algorithm (Figure 6.C) has a bigger adaptation area (i.e. the green area) w.r.t. the other algorithms, i.e. PPO and SAC (respectively Figure 6.A and Figure 6.B). However, the DQN algorithm has its adaptation frontier for values of *length* above 1.20 and *cartfriction* $\in [12.3, 24.5]$ whereas the PPO algorithm is well inside the adaptation boundary for the same ranges of values. Hence, despite the DQN adaptation area being overall bigger than the one of PPO, the latter might be preferable in terms of adaptation in specific regions of the parameter space. When different algorithms are available, AlphaTest helps supporting the decision of the software engineer with both qualitative and quantitative adaptation measures.

6 RELATED WORK

In this section we first summarize the current approaches to test reinforcement learning (RL) based systems. Then, we discuss the techniques proposed to test deep learning (DL) based systems in general and, finally, we present how the major issue of catastrophic forgetting is addressed in the context of continual learning. The literature on transfer learning is not discussed since the objective of transfer learning techniques is to study how to transfer knowledge acquired in a source domain to a target domain so as to speed up learning in the latter [42, 56, 82, 96]. This goes beyond the scope of the present paper, which is studying the adaptation boundary of a learned policy when the environment changes and the policy needs to be adapted incrementally.

6.1 Testing of Reinforcement Learning Systems

The problem of testing RL based systems is not much explored w.r.t. testing supervised learning (SL) based systems [61, 93]. The first body of work in RL testing draws from the SL literature that studies *adversarial attacks*, i.e. techniques to craft inputs on which trained neural networks perform very poorly despite having very good average performance on the test set. For example, classifiers trained to classify images are vulnerable to perturbations to the input image added by an adversary, possibly causing misclassification [81]. In the same way also DRL algorithms can be vulnerable to adversarial attacks, since DRL algorithms can learn end-to-end behaviors as well (i.e. from raw inputs, e.g. images, to actions). Huang et al. [31] explored this hypothesis, finding that the policies trained to play Atari games [7] from raw pixels are also prone to adversarial attacks that can degrade their performance at test time. The authors analyzed the robustness to adversarial attacks of different DRL algorithms considering both white-box and black-box adversarial techniques (i.e whether the adversary has access to the policy network or not). The results show that even in black-box scenarios, i.e., when the adversary has only access to the training environment (i.e., the simulator), it is possible to confuse DRL policies in a computationally efficient way. More recently Lin et al. [44] proposed adversarial attacks that exploit the sequential nature of RL systems. In fact, they designed a technique to perturb the observations received by the agent only when the perturbations are likely to be effective instead of performing attacks at every time-step (i.e., uniform attacks). Such strategically-timed attacks, applied at selective time-steps, can lower the reward of DRL

agents while being less likely to be detected w.r.t. uniform attacks. Moreover, the authors propose another attack, called *enchanting attack*, which uses a generative model in combination with a planning algorithm to lure the agent to a certain, possibly dangerous, state. The definition of frontier pair (see Definition 1) presents some similarities with the concept of adversarial example. Indeed, a frontier pair is made by two environment configurations that are close to each other and that trigger a different adaptation behavior of the agent. Similarly, an adversarial search technique seeks to find the minimal perturbation of the input (e.g. a pixel, if the input space of the model is an image) that triggers a misbehavior of the model (e.g. a misclassification). However, there are some important differences that distinguish an adversarial example from a frontier pair. First of all, the frontier pair is defined on environment configurations determining the whole environment the agent will be trained on, whereas an adversarial example is defined on a single input the agent receives (e.g. through a camera) and processes for each prediction. Hence, gradient-based techniques to generate adversarial input examples are not applicable in the context of this paper.

Adversarial evaluation is proposed by Uesato et al. [88] to find failures in trained DRL agents without generating out-of-distribution inputs, unlike previous work on adversarial examples in DRL [31, 81]. The objective of the authors is to efficiently find inputs, i.e. initial conditions (e.g. the shape of the track in a driving scenario, as it is generated by the environment), that cause a catastrophic failure (e.g. the car hits a wall in the driving scenario). Towards such objective a failure probability predictor (AVF for short) is learnt which takes as input an initial condition and outputs a binary signal, indicating a catastrophic failure. To do that efficiently the authors propose to use data from intermediate agents taken at different stages during the training process, since such agents are less robust and therefore more prone to failure. The underlying assumption of the approach is that agents fail early on in the training phase in similar ways as the final agent does. Their results confirm the hypothesis in two tasks, namely the Humanoid task on the MuJoCo [84] simulator and a self-driving scenario on the TORCS [92] simulator, where their approach was able to find failures in the final agents significantly faster than a Monte Carlo method (e.g. repeated random trials).

Another active area of research somehow related to RL testing is the study of generalization and overfitting. *Procedurally generated environments* (PGEs) have been proposed to help alleviate both concerns [12, 18]. In fact, PGEs can provide significant variation during training so that the agents are encouraged to learn general strategies to solve the problem rather than overfitting a specific instantiation of the environment. Of particular interest is the work by Ruderman et al. [18, 65], which explores the question of whether specific failures can emerge when training DRL agents in such environments. Specifically, when navigating in procedurally generated mazes, agents can suffer from catastrophic failures despite having a high average-case performance at evaluation time. The search for environment settings which cause catastrophic failures is a local search process. Initially, a set of mazes is sampled from the training distribution and the trained agent is evaluated on each environment. Afterwards, the maze where the agent has the lowest score is selected, new candidate mazes are generated by randomly changing the wall positions and the process repeats. The authors found that this search procedure can effectively discover mazes that the trained agent fails to solve in a two minute episode (i.e. a catastrophic failure, according to their definition). Moreover, the failure-causing mazes transfer among different DRL agents and different architectures.

Drawing from the DL testing literature, which we discuss in the next section, Trujillo et al. [85] applies the concept of *neuron coverage* [59] to DRL systems. Given a test set of inputs, neuron coverage is defined as the proportion of activated neurons over all neurons when all available test inputs are supplied to a neural network. According to such metric, a test set is adequate if it is able to activate a high proportion of the neural network neurons, hence, thoroughly exercising its “logic”. Trujillo et al. measured neuron coverage during training and testing of a DQN [50] agent on the Mountain Car problem [80] and investigated the correlation between neuron coverage and cumulative reward. The

preliminary results show that neuron coverage tends to be higher when the agent explores, i.e. in the early stages of training, and does not correlate, or it correlates negatively, with the cumulative reward. Therefore, neuron coverage is not sufficient to reach substantial conclusions about the quality of neural networks for DRL agents, even though more extensive studies would be needed to confirm such result.

Rupprecht et al. [66] instead, focused on the visual aspect of DRL agents that learn from images, in order to understand the relationship between the actions made by an agent and its visual input, with the objective of identifying potential problems in the learnt behavior. In particular, the authors learn a generative model of the environment aiming at evaluating the agent's behavior in particular classes of states created by the optimization. The rationale is that, often, a trained DRL agent is evaluated on a set of scenarios which rarely include potential failure cases. Instead, by training a generative model, the authors were able to sample out-of-distribution states where a certain target condition is satisfied. For example, when training a DQN agent, it may be of interest to see what happens in states where the action-values are high or low. Alternatively, states where one action yields a high expected return and another one is not beneficial at all are also potentially very interesting.

Such works relate to ours because they evaluate and test RL algorithms. In particular, our work is built on the idea of *changing the environment* in which the agent is originally trained, similar to what Ruderman et al. [18, 65] accomplishes with PGEs. One difference w.r.t. such works is that the environments we consider are not procedurally generated. We instead modify critical parameters of the environment at runtime. The other fundamental difference is that we test the plasticity of the learning algorithm when exposed to the new environment: we do not just evaluate a trained agent in a new environment to find its sensitivity to the changed conditions; we rather let the agent learn in the changed environment to study its behaviors in the new conditions. None of the existing work tested the continual learning capabilities of RL agents.

6.2 Testing of Deep Learning Systems

Testing DL systems is a very active area of research. Research work in such context comprises all aspects of software testing, from test input generation [8, 21, 24, 45, 59, 62, 83, 95] to test oracles [17, 34, 46, 59, 72, 77], including test adequacy criteria [38, 45, 59]. Although none of those works specifically address RL based systems such approaches could still be applied to DRL, i.e., to RL when a neural network is used as function approximator. The works most related to ours are those on test input generation. For a more in-depth and thorough discussion of the topic, the interested reader can refer to the systematic mapping by Riccio et al. [61] and the survey by Zhang et al. [94].

According to Riccio et al [61] the most widely applied test input generation technique is *input mutation* where existing inputs to a DL based system are mutated with the constraint of preserving their semantics. Most of the works are focused on changing the input in a way that is imperceptible for humans [16, 17], but others focus also on mutating the inputs, especially images, with the objective of simulating real environmental changes [64, 83], e.g., changes in weather conditions, occlusions, lens distortions and object movements. Besides input mutation another popular way to generate test inputs is the *search based* approach [1, 2, 8, 21, 62]. The objective of such works is to generate challenging scenarios, i.e., environment configurations, for the system under test to detect as many system failures as possible. Moreover, search based approaches are also used to carry out *boundary input analysis* [60]. In fact, instead of finding single failures, these approaches aim at finding similar inputs that trigger different behaviors of a DL based system, being at the *boundary* (or *frontier*) of the behaviors of such system. This analysis is, for example, performed by Mullins et al. [53], Tuncali et al. [87] and Riccio et al. [62] for autonomous systems using different search techniques. Indeed, Mullins et al. [53] use adaptive search to discover inputs at the frontier of behaviors of a system using a minimal number

of samples. On the contrary, Tuncali et al. [87] utilize rapidly-exploring random trees to find pairs of environment configurations at the collision boundary of an autonomous car, i.e., one environment configuration in which the collision is unavoidable and the other, *close* to it, in which the collision is avoidable. Likewise, Riccio et al. [62] use a model-based multi-objective search technique to characterize the frontier of behaviors of both an autonomous car and a handwritten digit classifier. The results show that the points found by this technique are spread across the frontier and that the scenarios produced by the approach are realistic.

Our work builds on the idea of characterizing the frontier of behaviors of a DL based system [53, 62, 87]. We differ from such existing techniques in two ways. First, we search for the frontier of behaviors of a DRL system not just by evaluating the trained agent on different environments, but also by letting the agent learn in the new environments, in continual training mode. In this way, we characterize both how the agent learns new behaviors (i.e., the adaptivity heatmap, see Section 3), as well as how the agent behaves in the environment it was initially trained on, once it has adapted to the new environments (i.e., the anti-regression heatmap, see Section 3). Second, we use a combination of systematic search algorithms to sample the parameter space, while enabling continual learning, and we use the nearest neighbor algorithm to approximate the behaviors of the agent in the remaining parts of the parameter space. In fact, the high cost associated with a single, complete (continual) training run is not compatible with the use of population based, evolutionary algorithms, such as those used in previous works for frontier exploration [62].

6.3 Continual Learning

The continual learning problem was first studied by McCloskey et al. [48], who investigated whether neural networks can acquire new knowledge incrementally (or sequentially). Such question was explored in a supervised learning setting by the authors, who trained a neural network to perform single-digit additions. They showed that when a new task is learnt incrementally, the new knowledge interferes with the existing one, replacing it completely. McCloskey et al. referred to this failure mode of neural networks in the continual learning setting as *catastrophic forgetting* or *catastrophic interference*. When learning a new task, the neural network's inability to learn sequentially is a major problem in several scenarios. For instance, if training from scratch takes a long time, it might be impossible to reorder and replay all training data, to ensure high performance on all the tasks, including the old ones.

Catastrophic interference is a manifestation of a more general problem of neural networks, the so-called *stability-plasticity* problem [11, 20, 23]. The fundamental question is how to design a system that is *plastic* (or sensitive) with respect to new inputs in order to incorporate new knowledge, but that, at the same time, does not forget the already acquired knowledge (i.e. the system is *stable* w.r.t. old inputs). Catastrophic forgetting is a failure of stability, in which new experience overwrites previous experience. The algorithms that address the continual learning problem, which we present next, are designed to find a trade-off between stability and plasticity.

Approaches that address and mitigate catastrophic forgetting are divided into three main categories [55], namely *regularization approaches*, *dynamic networks* and *complementary learning systems. Regularization* approaches retrain the whole network while regularizing to prevent forgetting of the previously learnt tasks. One such approach is represented by the work of Zhizhong et al. [43], who proposed the *learning without forgetting* (LwF) algorithm. The LwF algorithm considers a network with shared parameters across tasks and some task specific parameters. When a new task needs to be learnt, the approach optimizes the parameters of the new task together with the shared parameters with the constraint that the predictions of the network on the old tasks do not change significantly. Another regularization approach is the *elastic weight consolidation* (EWC) algorithm proposed by Kirkpatrick et al. [40] in supervised and reinforcement learning scenarios. In EWC, the objective is to identify the weights that are important for past tasks and,

while learning a new task, penalize their updates w.r.t. the updates on weights that have less significance for past tasks. Differently from the previous two approaches, Kaplanis et al. [37] propose a policy consolidation model for continual RL that does not require the knowledge of task boundaries. Such approach can also be viewed as an extension of the PPO algorithm [71] (see Section 2), which constrains the new policy to be close to the old one, thus preventing catastrophic forgetting at a very short timescale. In the policy consolidation model [37], instead, the constraint for the policy is applied to multiple gradient steps in order to maintain the knowledge acquired at several stages in the training history.

Dynamic networks approaches selectively train the network and expand it if necessary to learn new tasks. For instance, Rusu et al. [67] propose to freeze changes to the networks trained on previous tasks and, when a new task is presented, add novel sub-networks with fixed capacity to be trained for such next task.

Finally, *complementary learning systems* use mechanisms to replay old experience while learning new tasks in order to consolidate the acquired knowledge. Shin et al. [73] train a generative model to output synthetic data that follows the same distribution as the original training data. In such a way, when learning a new task, the training data for previous tasks can easily be sampled and interleaved with those for a new task even if the training data the network was trained on is no longer available. In the same way, Rolnick et al. [63] uses experience replay as a way to reduce catastrophic forgetting in multi-task RL. Such approach, called *CLEAR*, mixes on-policy learning from novel experiences (for plasticity) and off-policy learning from replay experiences (for stability).

Continual learning in RL is mostly studied in multi-task settings [40, 63, 76], e.g. an agent trained to play a certain game is then challenged to sequentially learn to play another game without forgetting how to play the first one. In contrast, Fedus et al. [19] explores the question of whether catastrophic forgetting may arise within a single game environment. They show that in Atari games [7] catastrophic interference causes the agent performance to plateau, i.e. learning one segment of the game degrades the performance of the policy on previously learnt segments of the game.

In our work, we devise a novel methodology that helps developers understand the frontier of the adaptive behaviors of a given RL algorithm, when continual learning is enabled, i.e., when a trained agent learns incrementally to adapt to an environment which is different from the one it was originally trained on. Since continual learning is a key property of RL based systems, which can continue to learn as new, unlabeled data are acquired, finding the limits where such property holds is fundamental for any practical application that involves runtime adaptation via continual learning.

7 CONCLUSION AND FUTURE WORK

In this paper we proposed the first approach to test the adaptation and anti-regression capabilities of a DRL based system. We characterize the adaptation frontier of a DRL algorithm along the parameters that define the environment in which the agent operates. We provide a visualization of such frontier and we propose a volume metric to quantify both the adaptation capabilities of an agent and its ability to remember how to perform in the original environment when the adaptation is successful. We implemented the approach in a tool called AlphaTest [47] and we carried out an extensive evaluation on three DRL algorithms and four continuous control environments, considering several parameter combinations.

AlphaTest has been successfully applied to four subjects taken from the popular *gym* library. Experimental results indicate that AlphaTest is effective at finding the adaptation frontier points and that it can be very useful in characterizing and discriminating the adaptation and anti-regression capabilities of alternative DRL algorithms. AlphaTest provides developers with an interpretable visual output, which consists of the adaptation/anti-regression heatmaps (when two parameters are considered) or a clustered two-dimensional projection, accompanied by a decision tree (when dealing with more than two parameters).

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).

9 EXTENDED BACKGROUND

9.1 Formalization

The RL problem can be formalized as a *Markov Decision Process* (or MDP) [3, 80]. MDPs model decision making problems, where actions determine not just the immediate reward but also the next states and future rewards.

At each time-step t, the environment sends the agent information that includes the reward $r_t \in \mathcal{R} \subset \mathbb{R}$ and the environment state $s_t \in \mathcal{S}$ (or an observation $o_t \in \mathcal{O}$, since the environment may not be fully observable). The agent decides how to act in the environment by choosing action $a_t \in \mathcal{A}$. Then, at the next time-step $t+1$, the environment transitions to the next state s_{t+1} and gives the agent the reward r_{t+1}. The action a_t is determined by the agent's *policy*: a function mapping states to actions, indicated as $\pi : \mathcal{S} \rightarrow \mathbb{P}(\mathcal{A})$ when stochastic, while it is indicated as $\mu : \mathcal{S} \rightarrow \mathcal{A}$ when deterministic. The agent uses its policy to interact with the MDP producing a *trajectory* $\tau = (s_1, a_1, r_1, s_2, a_2, r_2, \ldots)$ of states, actions and rewards over $\mathcal{S} \times \mathcal{A} \times \mathcal{R}$.

In a *finite* MDP, the sets of states, actions, and rewards all have a finite number of elements. Assuming that the environment is stochastic, the *one-step dynamics* can be defined as $p : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$ with $p(s_{t+1}|s_t, a_t)$ being the probability that the environment transitions to state s_{t+1} when it starts from state s_t and action a_t is taken[4]. Moreover, if $p(s_{t+1}|s_1, a_1, \ldots, s_t, a_t) = p(s_{t+1}|s_t, a_t)$ then the MDP obeys the *Markov property*, saying that the future s_{t+1} is conditionally independent from the past $(s_{t-1}, s_{t-2}, \ldots, s_1)$, being only dependent on the present s_t. In other words the present state summarizes all aspects of the past agent–environment interactions that make a difference for the future.

Due to the Markov property, an MDP can be defined by a *reward function* $R : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow \mathbb{R}$. The goal of the agent is to maximize the cumulative reward obtained over the future trajectory τ, indicated as $R^*(\tau)$ (also called *return*). One kind of return is the finite-horizon undiscounted return $R^*(\tau) = \sum_{t=1}^{T} r_t$ which is the sum of rewards obtained in a window T of time-steps. Another kind of return is the infinite-horizon discounted return, which is the sum of all rewards obtained by the agent, but discounted depending on when in the future they are obtained. The infinite-horizon discounted return includes a *discount factor* $\gamma \in [0, 1)$ and it is defined as: $R^*(\tau) = \sum_{t=1}^{\infty} \gamma^t r_t$. The discount factor mathematically guarantees that the reward is finite even in the infinite-horizon case (if the reward is always the constant +1 then $R^*(\tau) = \frac{1}{1-\gamma}$) and practically determines the relevant time horizon of the agent, giving more weight to the rewards in the near future with respect to those far off.

To summarize, an MDP is a 5-tuple, $\langle \mathcal{S}, \mathcal{A}, R, p, p_1 \rangle$, where:

- $\mathcal{S}$ is the set of all states;
- $\mathcal{A}$ is the set of all actions (to simplify the notation, we assume all actions to be *valid* in all states);
- $R : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow \mathbb{R}$ is the reward function, with $r_t = R(s_t, a_t, s_{t+1})$ which is interpreted as the reward the agent gets at time-step t after transitioning from s_t to s_{t+1} by taking action a_t. It is often simplified to just a dependence on the current state, $r_t = R(s_t)$, or to a state-action pair $r_t = R(s_t, a_t)$;
- $p : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$ is the one-step dynamics or transition probability function of the environment.

[4]The reward r_{t+1} was dropped to simplify the notation.

- $p_1\colon \mathcal{S} \longrightarrow [0, 1]$ indicates the initial state (s_1) distribution. In other words it is the probability that the environment starts in a specific state $s_1 \in \mathcal{S}$.

9.2 The Reinforcement Learning Problem

The probability of a trajectory τ that is T time-steps long given a policy π is $\rho(\tau|\pi) = p_1(s_1) \prod_{t=1}^{T} \pi(a_t|s_t) p(s_{t+1}|s_t, a_t)$. The RL objective $J(\pi)$ can be expressed as:

$$J(\pi) = \int_{\tau} \rho(\tau|\pi) R^*(\tau) = \mathbb{E}_{\tau\sim\pi}[R^*(\tau)] \tag{6}$$

assuming that the state space $\mathcal{S}$ and the action space $\mathcal{A}$ are continuous (the integral is replaced by a sum when such spaces are discrete). Equation 6 computes the expected return for a policy π as the return $R^*(\tau)$ averaged over all possible trajectories τ each having a probability $\rho(\tau|\pi)$. The $\sim$ symbol indicates that the trajectory τ is *sampled* from the probability function $\rho(\cdot|\pi)$. Hence, the central optimization problem in RL can be expressed as:

$$\pi^* = \arg\max_{\pi} J(\pi) \tag{7}$$

i.e., finding the optimal policy π^* that maximizes $J(\pi)$.

9.3 Value Functions and Optimality

Value Functions. The *value* of a state or state-action pair is defined as the expected return the agent will get if it starts on that state or state-action pair and then follows a given policy π. The value function is defined as:

$$v_\pi(s_t) = \mathbb{E}_{\tau\sim\pi}[R^*(\tau)|s_t] \tag{8}$$

It tells us how good is a state s_t given the policy π. Note that if we consider $v_\pi(s_1)$ we obtain the RL objective in Equation 6. In a similar way we can define the action-value function as:

$$q_\pi(s_t, a_t) = \mathbb{E}_{\tau\sim\pi}[R^*(\tau)|s_t, a_t] \tag{9}$$

It tells us how good it is being in state s_t and taking action a_t, given the policy π. Sometimes it is useful to know how much an action is better than the others on average in a certain state. This is quantified by the *advantage function* $\hat{a}_\pi(s_t, a_t) = q_\pi(s_t, a_t) - v_\pi(s_t)$.

Optimality. Equation 8 and Equation 9 can be applied to the optimal case. Equation 8 gives us the optimal value function as $v_*(s_t) = \max_\pi v_\pi(s_t), \forall s_t \in \mathcal{S}$, and Equation 9 the optimal action-value function as $q_*(s_t, a_t) = \max_\pi q_\pi(s_t, a_t)$, $\forall s_t \in \mathcal{S}, a_t \in \mathcal{A}$. These equations give the expected return the agent will get if it starts on state s_t and, in case of q^*, it chooses action a_t, and then it follows the *optimal* policy π^* forever after. The optimal policy π^* can be obtained from the optimal action-value function by choosing at each time-step the action that maximizes the optimal action-value function: at time-step t the optimal action is given by $a_t^* = \arg\max_a q^*(s_t, a)$.

Bellman Equations. A fundamental property of value functions is that they must satisfy the recursive consistency relationships expressed by the Bellman equations. For any policy π and any state s_t, the following consistency condition holds between the value of s_t and the value of its possible successor states:

$$v_\pi(s_t) = \mathbb{E}_{a_t\sim\pi, s_{t+1}\sim p}[R(s_t, a_t) + \gamma v_\pi(s_{t+1})] = \int_{\mathcal{A}} \pi(a_t|s_t) \int_{\mathcal{S}} p(s_{t+1}|s_t, a_t)[R(s_t, a_t) + \gamma v_\pi(s_{t+1})] \tag{10}$$

Equation 10 expresses a relationship between the value of a state s_t and the values of all its possible successor states s_{t+1}, weighting each possibility by its probability of occurring (given a policy π and the one step dynamics $p(s_{t+1}|s_t, a_t)$) and averaging over all possibilities.

A similar relationship holds for the action-value function as well:

$$q_\pi(s_t, a_t) = \mathbb{E}_{s_{t+1}\sim p}\Big[R(s_t, a_t) + \gamma \mathbb{E}_{a_{t+1}\sim\pi}[q_\pi(s_{t+1}, a_{t+1})]\Big] = \int_{\mathcal{S}} p(s_{t+1}|s_t, a_t) \int_{\mathcal{A}} \pi(a_{t+1}|s_{t+1})[R(s_t, a_t) + \gamma q_\pi(s_{t+1}, a_{t+1})] \tag{11}$$

Since v^* and q^* are respectively the value function and action-value function for an optimal policy π^*, they must also satisfy the self-consistency condition given by the respective Bellman equations (Equation 10 and Equation 11). The only difference with respect to Equation 10 and Equation 11 is that the expectation over actions is replaced by the max over actions, because to act optimally the agent must select the action which leads to the highest value of the value function or of the action-value function.

9.4 Reinforcement Learning Algorithms

9.4.1 Model-Free Algorithms. In the following we describe the three main categories model-free algorithms belong to, namely *policy gradients*, *value-based* and *hybrid*.

Policy Gradients. In this category of algorithms the policy is represented explicitly as $\pi(a_t|s_t;\theta)$. The parameters θ are the parameters of the function approximator (e.g., the weights of a neural network) which estimates the policy. Those parameters θ are updated by directly computing the gradient of the RL objective $J(\pi(\theta))$ (see Equation 6) and then taking a step in the direction of steepest ascent. This optimization is performed *on-policy*, which means that each update is carried out with data coming only from the current version of the policy. In policy gradients an approximator $V_\pi(s_t;\phi)$ (or $Q_\pi(s_t, a_t;\phi)$) [5] is usually learnt, that approximates the true value function $v_\pi(s_t)$ (resp. $q_\pi(s_t, a_t)$), and such approximator is used to update the policy. The parameters ϕ of the value function are usually optimized by minimizing the mean-squared error between the actual estimate of the value function and the return computed by interacting with the environment using the policy. The state-of-the-art policy gradient algorithm is *Proximal Policy Optimization* (PPO) [71].

Value-Based. In value-based methods the approximator $Q(s_t, a_t;\phi)$ for the optimal action-value function $q^*(s_t, a_t)$ is learnt. The objective function does not coincide with the RL objective (see Equation 6), rather it is based on the Bellman equation (see Equation 11). This optimization is performed *off-policy*, which means that the optimization can be done with data coming from any policy. At the end of training the corresponding policy is derived from the trained action-value function, by greedily choosing the action that maximizes the function for each specific state ($a_t = \arg\max_a Q(s_t, a;\phi)$, $\forall t$). The most popular example of value-based method is *Deep Q Network* (or DQN) [50], which actually started the field of DRL and was improved over the years with various optimizations [28, 69, 90].

Hybridization between Policy Gradients and Value-Based Methods. The main advantage of policy gradient methods is that they determine directly the policy (i.e., RL's ultimate goal) by computing the gradient of the RL objective. Value-based methods, instead, try to solve the Bellman equation via fixed point iterations that cannot be proven to converge under non-linear function approximation [80]. Therefore, value-based methods are less reliable than policy

[5] When a state dependent function (value function v, action-value function q or advantage function $\hat{a}$) is approximated we indicate it with a capital letter (respectively V, Q and $\hat{A}$)

gradients methods but, when they work, they can be substantially more sample efficient. In fact, being off-policy methods, they can reuse data more extensively and effectively. In order to get the best of both worlds, many algorithms that use both policy gradients and value-based methods have been proposed. One notable example from this category is *Soft Actor Critic* (or SAC) [26].

9.4.2 Model-Based Algorithms. Broadly speaking model-based algorithms either assume to have a model of the environment to use or such model is learnt. The most popular example of model-based algorithm when the model is given is *AlphaZero* [75] which mastered the games of Go, Chess and Shogi (i.e. Japanese chess). Afterwards, the approach was made more general by removing the assumption that the rules of the game were known (i.e. the model of the environment was known). In fact *MuZero* [70] learns the rules of the game by playing it and then apply the same machinery of AlphaZero once the model is available. The effort of building the model of the environment when it is not available plays a crucial part for the success of a model-based algorithm but actually what differentiates model-based algorithms is how they choose to use the model once it is available. A few of those options are discussed below.

An algorithm may choose to not represent the policy explicitly but only use planning techniques (like *model-predictive control*) that compute a plan which is optimal with respect to the model and then use the plan to select actions in the environment. Alternatively, planning techniques can be used in combination with an explicit representation of the policy. The current policy is used in a planning algorithm (like *Monte Carlo Tree Search* [14] if the action space is discrete) to approximate the search when the state space is large. By planning ahead it is possible to find better actions than what the policy alone would have produced and, therefore, those actions can be used update the current policy to become more like the planning algorithm. Another strategy is to use a model-free approach but use the model to either augment the real experiences obtained by running the policy on the real environment or use only the fictitious experience produced by the model to train a model-free algorithm [25].

9.5 Deep Q Network

The basic idea behind the DQN algorithm is to estimate the action-value function by solving the Bellman optimality equation iteratively. When a non-linear function approximator is used to represent the action-value function, the iterative algorithm is not guaranteed to converge [50, 86]. One cause of instability is the correlation present in the sequence of states the agent experiences: small updates to the Q function may significantly change the policy derived from it and therefore the state sequence observed over time. The other cause of instability is the correlation between the left-hand side of the Bellman equation ($Q(s_t, a_t)$, the action-values) and the right-hand side (often indicated as $y = r_t + \gamma \max_a Q(s_{t+1}, a)$), representing the target values.

The first cause of instability is addressed by the authors of DQN with a technique known as *experience replay*. The agent's experiences, often called transitions, are stored at each time-step, (s_t, a_t, r_t, s_{t+1}), in a data set $\mathcal{D}$ called *replay memory* (see Line 8 in Algorithm 5). Then, samples of experiences are drawn uniformly at random from $\mathcal{D}$ to perform minibatch updates of the Q function weights ϕ. With sample randomization there is no longer correlation between experiences and therefore the variance of the updates is reduced. Moreover, each experience is potentially used in many weight updates, which increases sample efficiency. The use of experience replay makes the DQN algorithm *off-policy*, which means that the policy used to generate the experiences and the policy that is being learnt (the greedy strategy $a_t = \max_a Q(s_t, a; \phi)$) are different. In practice the experiences are collected using an ϵ-greedy policy (see Line 6 in Algorithm 5).

Algorithm 5: DQN algorithm pseudocode, adapted from Mnih et al. [50]

```
1  Initialize replay memory D to capacity N
2  Initialize action-value function Q with random weights φ
3  Initialize target action-value function Q⁻ with weights φ⁻ ← φ
4  foreach episode = 1, M do
5      foreach t = 0, 1, 2, ... do
6          Observe state s_t and, following ε-greedy policy, select:
               a_t = { a random action,            with probability ε.
                     { argmax_a Q(s_t, a; φ),      otherwise.
7          Execute action a_t in the environment and observe reward r_t and state s_{t+1}
8          Store transition (s_t, a_t, r_t, s_{t+1}) in D
9          Sample random minibatch of transitions (s_j, a_j, r_j, s_{j+1}) from D
10         Compute targets for the Q function
               y_j = { r_j,                                   if episode terminates at step j + 1.
                     { r_j + γ max_{a'} Q⁻(s_{j+1}, a'; φ⁻),  otherwise.
11         Perform a gradient descent step on (y_j − Q(s_j, a_j; φ))² wrt φ
12         Every C steps update φ⁻ ← φ
13     end
14 end
```

The second cause of instability is addressed by using a separate Q network (namely Q^-) for computing the targets y_j in the Q-learning update (see Lines 10–11: the equation at Line 11 is also called Bellman error loss). Every C updates the weights of the target network ϕ^- are updated to reflect the online Q network (with weights ϕ, see Line 12). This way the update of Q does not immediately affect the targets y_j making the overall process more similar to supervised learning and more stable.

Improvements to the DQN algorithm. Since the DQN algorithm we use in our experiments [30] incorporates recent improvements to the original algorithm discussed above, we briefly describe them following the temporal order in which they were proposed.

The first improvement is *Double Q-Learning* [28]. The authors show that the original DQN algorithm tends to numerically overestimate the action values under certain conditions. Overestimation is due to both the use of an approximated Q function and the use of the max operator in the target computation (see Line 10 in Algorithm 5). The max operator, in fact, selects the action with the biggest value, which is often also the action associated with the biggest estimation error. The idea to overcome this problem is to select the action according to the online Q network but to compute its value using the target Q network ($r_t + \gamma Q^-(s_{t+1}, \arg\max_{a'} Q(s_{t+1}, a'; \phi); \phi^-)$), so that the error affecting the selection of the action does not affect its evaluation.

The second improvement regards the replay memory [69]. In the original DQN algorithm, transitions from the replay memory are uniformly sampled, giving equal importance to all transitions. To make experience replay more efficient and effective, the authors propose to prioritize (or replay more frequently) transitions with high expected learning progress, as measured by the magnitude of their temporal-difference (TD) error (i.e. the difference between the right-hand side of the Bellman equation for the action-value function and the left-hand side).

The last improvement regards the architecture of the networks used in the algorithm [90]. The authors represent the Q function using two separate estimators: one for the state value function V and one for the advantage function $\hat{A}$. The

Algorithm 6: PPO algorithm pseudocode (Clip version), adapted from Schulman et al. [71] and Jushua et al. [3]

```
1  Initialize policy π with random weights θ
2  Initialize value function V with random weights ϕ
3  Initialize old weights of the policy θ_old ← θ
4  repeat
5    Collect set of trajectories τ_1, ..., τ_N by running policy with weights θ_old in the environment
6    Compute returns for each time-step for each trajectory: (G_11, ..., G_T1), ..., (G_1N, ..., G_TN)
7    Compute advantage estimates for each trajectory based on V(s;ϕ): (Â_11, ..., Â_T1), ..., (Â_1N, ..., Â_TN)
8    Perform gradient ascent steps on the following objective wrt θ with minibatch size ≤ NT:
9    Perform gradient descent steps on (V(s;ϕ) − G)^2 wrt ϕ
10   θ_old ← θ
11 until convergence
```

$$L(s, a, \theta_{old}, \theta) = \min\left(\frac{\pi(a|s;\theta)}{\pi(a|s;\theta_{old})}\hat{A}(s,a), g(\epsilon, \hat{A})\right),\ g(\epsilon, \hat{A}) = \begin{cases} (1+\epsilon)\hat{A}, & \text{if } \hat{A} \geq 0. \\ (1-\epsilon)\hat{A}, & \text{if } \hat{A} < 0. \end{cases}$$

key insight of this architecture is that for many states it is unnecessary to estimate precisely the value of each action choice. It is however important the accurate estimation of how valuable each state is (i.e the value function), which is where this architecture puts more resources. In fact, in the dueling architecture, for each Q value update the value function V is updated whereas in the single-network architecture only the state value for the selected action is updated.

9.6 Proximal Policy Optimization

Algorithm 6 shows the pseudocode of a particular version of the PPO algorithm (Clip version) implemented in the library used in our empirical study [30]. First (Line 5), a set of trajectories of length T is collected by running the current policy (with weights θ_{old}). Such trajectories can also be collected in parallel by running the same agent in different independent environments. Then (Line 6), the returns $G_i = \sum_{t=i}^{T} \gamma^t r_t$ are computed for each time-step t_i and each trajectory τ_j. Based on the returns the advantage function $\hat{A}$ is computed at each time-step. One common choice for such computation is $\hat{A}_i(s_i, a_i) = G_i - V(s_i;\phi)$ which tells us how much an action a_i is better or worse than the average, as obtained from the current value function $V(s_i;\phi)$.

Line 8 contains the objective function used by PPO. The main idea of this objective is to increase the probabilities of the actions that are better than average (i.e., their advantage is positive) and decrease the probabilities of the actions that are worse than average (i.e., their advantage is negative). The constraint we need to respect during each gradient step that updates the policy is that the new policy (weights θ) being updated is not *too far away* from the old one (weights θ_{old}). If the constraint is not respected the advantage estimates computed using the old policy are going to be very inaccurate and cannot be trusted to get a policy which is better than the previous one. To figure out the intuition behind the formula at Line 8, let's look at a single state-action pair (s, a), and consider the two cases. When the advantage is positive ($g(\epsilon, \hat{A}) = 1 + \epsilon$) the optimization makes $\pi(a|s;\theta)$ larger. However, the min operator constraints the policy with an upper bound on its increase. In particular the upper bound on L when the advantage is positive is $(1+\epsilon)\hat{A}$. Conversely when the advantage is negative ($g(\epsilon, \hat{A}) = 1 - \epsilon$) the optimization makes $\pi(a|s;\theta)$ smaller. However, the max operator (since $\hat{A}$ is negative we can turn the min operator to a max operator) constraints the policy with a lower bound on its decrease. In particular the lower bound on L when the advantage is negative is $(1-\epsilon)\hat{A}$. *Clipping*, obtained through the min operator in the objective formula, prevents the policy from changing too much and the hyperparameter

Algorithm 7: SAC algorithm pseudocode, adapted from Haarnoja et al. [26] and Jushua et al. [3]

```
1  Initialize policy π with random weights θ
2  Initialize action-value functions Q_1, Q_2 with random weights φ_1, φ_2
3  Initialize replay memory D to capacity N
4  Initialize target action-value functions Q_1^-, Q_2^- with weights φ_1^- ← φ_1, φ_2^- ← φ_2
5  repeat
6      Observe state s_t and select action a_t ~ π(·|s; θ)
7      Execute action a_t in the environment and observe reward r_t and state s_{t+1}
8      Store transition (s_t, a_t, r_t, s_{t+1}) in D
9      foreach gradient step do
10         Sample random minibatch of transitions (s_j, a_j, r_j, s_{j+1}) from D
11         Set targets for the Q functions
               y_j = { r_j,                                                                   if episode terminates at step j + 1.
                     { r_j + γ( min_{i=1,2} Q_i^-(s_{j+1}, ã_j; φ_i^-) − α log π(ã_j|s_{j+1}; θ) ),   otherwise, with ã_j ~ π(·|s_{j+1}).
12         Perform a gradient descent step on (y_j − Q_i(s_j, a_j; φ_i))^2 wrt φ_i for i = 1, 2
13         Perform a gradient ascent step on ( min_{i=1,2} Q_i(s_j, f(s_j, η); φ_i) − α log π(f(s_j, η)|s_j; θ) ) wrt θ
14         Update target action-value functions with φ_i^- ← ρφ_i^- + (1 − ρ)φ_i, for i = 1, 2
15     end
16 until convergence
```

ϵ determines how much the new policy can change w.r.t. the old one, while, at the same time, increasing the value of the objective. Usually this parameter takes a small value (in the range 0.1 to 0.3) and the larger it is, the larger the allowed distance between the new policy and the old one.

After the policy is updated, the value function weights ϕ are updated by minimizing the mean squared error between the actual estimate of the value function and the return obtained by running the policy in the environment (Line 9).

9.7 Soft Actor Critic

SAC concurrently learns a policy $\pi(a_t|s_t;\theta)$ and two Q functions instead of one ($Q_1(s_t, a_t; \phi_1)$ and $Q_2(s_t, a_t; \phi_2)$). It uses two Q functions to overcome the overestimation of the Q values that we discussed for DQN. In particular, it uses the smaller of the two Q values to form the target in the Bellman error loss function. Lines 6–10 in Algorithm 7 are similar to the DQN algorithm. Line 11 computes the targets for the Bellman error loss. Apart from the two Q functions mentioned above, another difference w.r.t. DQN is the presence of the term $\alpha \log \pi(\tilde{a}_j|s_j;\theta)$, associated with the policy entropy ($\log \pi(\tilde{a}_j|s_j;\theta)$). The coefficient α explicitly controls the exploration-exploitation trade-off, with higher α leading to more exploration. Depending on the implementation, the parameter α is either a hyperparameter of the algorithm that may require careful tuning or a variable automatically adjusted during training (the implementation of SAC we used [30] supports both). Line 12 performs a gradient descent step on the mean squared Bellman error as in DQN. Line 13 updates the policy weights by gradient ascent. In order to optimize the policy the authors made use of a technique called reparametrization trick [39], which parameterizes the policy as $f(s, \eta)$, where η is a noise vector (e.g., Gaussian noise) used to determine the expected value. Line 14 updates the weights of the two target networks. The amount of update is determined by a parameter, ρ, between $[0, 1]$, usually chosen quite close to 1. The idea is to update the target network weights slowly but at every gradient step.

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] Achiam, J. Spinning Up in Deep Reinforcement Learning. *Website* (2018).

[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] Barto, A. G., Sutton, R. S., and Anderson, C. W. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE transactions on systems, man, and cybernetics*, 5 (1983), 834–846.

[6] Barto, S., and Anderson, C. Cartpole-v1 openai gym. https://gym.openai.com/envs/CartPole-v1/, 2020.

[7] Bellemare, M. G., Naddaf, Y., Veness, J., and Bowling, M. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research 47* (2013), 253–279.

[8] 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.

[9] Berner, C., Brockman, G., Chan, B., Cheung, V., Dębiak, P., Dennison, C., Farhi, D., Fischer, Q., Hashme, S., Hesse, C., et al. Dota 2 with large scale deep reinforcement learning. *arXiv preprint arXiv:1912.06680* (2019).

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

[11] Carpenter, G. Self organization of stable category recognition codes for analog input patterns. *Applied Optics 3* (1987), 4919–4930.

[12] Cobbe, K., Hesse, C., Hilton, J., and Schulman, J. Leveraging procedural generation to benchmark reinforcement learning. In *International conference on machine learning* (2020), PMLR, pp. 2048–2056.

[13] community, T. S. Scipy is an open-source software for mathematics, science, and engineering. https://docs.scipy.org/doc/scipy/reference/index.html, 2020.

[14] Coulom, R. Efficient selectivity and backup operators in monte-carlo tree search. In *International conference on computers and games* (2006), Springer, pp. 72–83.

[15] Dhariwal, P., Hesse, C., Klimov, O., Nichol, A., Plappert, M., Radford, A., Schulman, J., Sidor, S., Wu, Y., and Zhokhov, P. Openai baselines. https://github.com/openai/baselines, 2017.

[16] Ding, J., Kang, X., and Hu, X.-H. Validating a deep learning framework by metamorphic testing. In *2017 IEEE/ACM 2nd International Workshop on Metamorphic Testing (MET)* (2017), IEEE, pp. 28–34.

[17] Du, X., Xie, X., Li, Y., Ma, L., Liu, Y., and Zhao, J. Deepstellar: Model-based quantitative analysis of stateful deep learning systems. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2019), pp. 477–487.

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

[19] Fedus, W., Ghosh, D., Martin, J. D., Bellemare, M. G., Bengio, Y., and Larochelle, H. On catastrophic interference in atari 2600 games. *arXiv preprint arXiv:2002.12499* (2020).

[20] French, R. M. Catastrophic forgetting in connectionist networks. *Trends in cognitive sciences 3*, 4 (1999), 128–135.

[21] 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.

[22] Geramifard, A., Dann, C., Klein, R. H., Dabney, W., and How, J. P. Rlpy: a value-function-based reinforcement learning framework for education and research. *J. Mach. Learn. Res. 16*, 1 (2015), 1573–1578.

[23] Grossberg, S. T. *Studies of mind and brain: Neural principles of learning, perception, development, cognition, and motor control*, vol. 70. Springer Science & Business Media, 2012.

[24] Guo, J., Jiang, Y., Zhao, Y., Chen, Q., and Sun, J. Dlfuzz: differential fuzzing testing of deep learning systems. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE* (2018), pp. 739–743.

[25] Ha, D., and Schmidhuber, J. World models. *arXiv preprint arXiv:1803.10122* (2018).

[26] 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).

[27] 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.

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

[29] Henderson, P., Islam, R., Bachman, P., Pineau, J., Precup, D., and Meger, D. Deep reinforcement learning that matters. In *Proceedings of the AAAI Conference on Artificial Intelligence* (2018), vol. 32.

[30] Hill, A., Raffin, A., Ernestus, M., Gleave, A., Kanervisto, A., Traore, R., Dhariwal, P., Hesse, C., Klimov, O., Nichol, A., Plappert, M.,

Radford, A., Schulman, J., Sidor, S., and Wu, Y. Stable baselines. https://github.com/hill-a/stable-baselines, 2018.

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

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

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

[34] Jahangirova, G., and Tonella, P. An empirical evaluation of mutation operators for deep learning systems. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)* (2020), IEEE, pp. 74–84.

[35] 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.

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

[37] Kaplanis, C., Shanahan, M., and Clopath, C. Policy consolidation for continual reinforcement learning. *arXiv preprint arXiv:1902.00255* (2019).

[38] Kim, J., Feldt, R., and Yoo, S. Guiding deep learning system testing using surprise adequacy. In *Proceedings of the 41st International Conference on Software Engineering* (Piscataway, NJ, USA, 2019), ICSE '19, IEEE Press, pp. 1039–1049.

[39] Kingma, D. P., and Welling, M. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114* (2013).

[40] Kirkpatrick, J., Pascanu, R., Rabinowitz, N., Veness, J., Desjardins, G., Rusu, A. A., Milan, K., Quan, J., Ramalho, T., Grabska-Barwinska, A., et al. Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences 114*, 13 (2017), 3521–3526.

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

[42] Lazaric, A. Transfer in reinforcement learning: a framework and a survey. In *Reinforcement Learning*. Springer, 2012, pp. 143–173.

[43] Li, Z., and Hoiem, D. Learning without forgetting. *IEEE transactions on pattern analysis and machine intelligence 40*, 12 (2017), 2935–2947.

[44] 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).

[45] Ma, L., Juefei-Xu, F., Zhang, F., Sun, J., Xue, M., Li, B., Chen, C., Su, T., Li, L., Liu, Y., et al. Deepgauge: Multi-granularity testing criteria for deep learning systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (2018), pp. 120–131.

[46] Ma, L., Zhang, F., Sun, J., Xue, M., Li, B., Juefei-Xu, F., Xie, C., Li, L., Liu, Y., Zhao, J., et al. Deepmutation: Mutation testing of deep learning systems. In *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)* (2018), IEEE, pp. 100–111.

[47] Matteo, B., and Paolo, T. AlphaTest: A tool for testing the plasticity of reinforcement learning based systems. https://github.com/testingautomated-usi/rl-plasticity-experiments, 2021.

[48] McCloskey, M., and Cohen, N. J. Catastrophic interference in connectionist networks: The sequential learning problem. In *Psychology of learning and motivation*, vol. 24. Elsevier, 1989, pp. 109–165.

[49] Minsky, M. Steps toward artificial intelligence. *Proceedings of the IRE 49*, 1 (1961), 8–30.

[50] 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.

[51] Moore, A. W. Efficient memory-based learning for robot control.

[52] Moore, B., and Sutton. Mountaincar-v0 openai gym. https://gym.openai.com/envs/MountainCar-v0, 2020.

[53] Mullins, G. E., Stankiewicz, P. G., Hawthorne, R. C., and Gupta, S. K. Adaptive generation of challenging scenarios for testing and evaluation of autonomous vehicles. *Journal of Systems and Software 137* (2018), 197–215.

[54] OpenAI. Pendulum-v0 openai gym. https://gym.openai.com/envs/Pendulum-v0/, 2020.

[55] Parisi, G. I., Kemker, R., Part, J. L., Kanan, C., and Wermter, S. Continual lifelong learning with neural networks: A review. *Neural Networks 113* (2019), 54–71.

[56] Parisotto, E., Ba, J. L., and Salakhutdinov, R. Actor-mimic: Deep multitask and transfer reinforcement learning. *arXiv preprint arXiv:1511.06342* (2015).

[57] Patanjali, A. Modified version of the cartpole-v0 openai environment. https://github.com/AadityaPatanjali/gym-cartpolemod, 2017.

[58] 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.

[59] Pei, K., Cao, Y., Yang, J., and Jana, S. Deepxplore: Automated whitebox testing of deep learning systems. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP '17, ACM, pp. 1–18.

[60] Pezzè, M., and Young, M. *Software testing and analysis: process, principles, and techniques*. John Wiley & Sons, 2008.

[61] 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).

[62] 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.

[63] Rolnick, D., Ahuja, A., Schwarz, J., Lillicrap, T., and Wayne, G. Experience replay for continual learning. In *Advances in Neural Information*

Processing Systems (2019), pp. 350–360.

[64] Rubaiyat, A. H. M., Qin, Y., and Alemzadeh, H. Experimental resilience assessment of an open-source driving agent. In *2018 IEEE 23rd Pacific Rim International Symposium on Dependable Computing (PRDC)* (2018), IEEE, pp. 54–63.

[65] Ruderman, A., Everett, R., Sikder, B., Soyer, H., Uesato, J., Kumar, A., Beattie, C., and Kohli, P. Uncovering surprising behaviors in reinforcement learning via worst-case analysis. *openreview.net* (2018).

[66] Rupprecht, C., Ibrahim, C., and Pal, C. J. Finding and visualizing weaknesses of deep reinforcement learning agents. *arXiv preprint arXiv:1904.01318* (2019).

[67] Rusu, A. A., Rabinowitz, N. C., Desjardins, G., Soyer, H., Kirkpatrick, J., Kavukcuoglu, K., Pascanu, R., and Hadsell, R. Progressive neural networks. *arXiv preprint arXiv:1606.04671* (2016).

[68] Satariano, A., and Metz, C. A warehouse robot learns to sort out the tricky stuff. https://www.nytimes.com/2020/01/29/technology/warehouse-robot.html, 2020.

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

[70] Schrittwieser, J., Antonoglou, I., Hubert, T., Simonyan, K., Sifre, L., Schmitt, S., Guez, A., Lockhart, E., Hassabis, D., Graepel, T., et al. Mastering atari, go, chess and shogi by planning with a learned model. *arXiv preprint arXiv:1911.08265* (2019).

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

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

[73] Shin, H., Lee, J. K., Kim, J., and Kim, J. Continual learning with deep generative replay. In *Advances in neural information processing systems* (2017), pp. 2990–2999.

[74] Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. Mastering the game of go with deep neural networks and tree search. *nature 529*, 7587 (2016), 484–489.

[75] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815* (2017).

[76] Silver, D. L., Yang, Q., and Li, L. Lifelong machine learning systems: Beyond learning algorithms. In *2013 AAAI spring symposium series* (2013).

[77] Stocco, A., Weiss, M., Calzana, M., and Tonella, P. Misbehaviour prediction for autonomous driving systems. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (2020), pp. 359–371.

[78] Sutton, Geramifard, D. Acrobot-v1 openai gym. https://gym.openai.com/envs/Acrobot-v1/, 2020.

[79] Sutton, R. S. Generalization in reinforcement learning: Successful examples using sparse coarse coding. *Advances in neural information processing systems* (1996), 1038–1044.

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

[81] 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).

[82] Taylor, M. E., and Stone, P. Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research 10*, 7 (2009).

[83] Tian, Y., Pei, K., Jana, S., and Ray, B. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th International Conference on Software Engineering* (New York, NY, USA, 2018), ICSE '18, ACM, pp. 303–314.

[84] 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.

[85] Trujillo, M., Linares-Vásquez, M., Escobar-Velásquez, C., Dusparic, I., and Cardozo, N. Does neuron coverage matter for deep reinforcement learning? a preliminary study. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops* (2020), pp. 215–220.

[86] Tsitsiklis, J. N., and Van Roy, B. Analysis of temporal-diffference learning with function approximation. In *Advances in neural information processing systems* (1997), pp. 1075–1081.

[87] Tuncali, C. E., and Fainekos, G. Rapidly-exploring random trees-based test generation for autonomous vehicles. *arXiv preprint arXiv:1903.10629* (2019).

[88] 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).

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

[90] 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.

[91] Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. *Experimentation in software engineering.* Springer Science & Business Media, 2012.

[92] Wymann, B., Espié, E., Guionneau, C., Dimitrakakis, C., Coulom, R., and Sumner, A. Torcs, the open racing car simulator. *Software available at http://torcs. sourceforge. net 4*, 6 (2000), 2.

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

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

(2020).

[95] Zhang, M., Zhang, Y., Zhang, L., Liu, C., and Khurshid, S. Deeproad: Gan-based metamorphic testing and input validation framework for autonomous driving systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (New York, NY, USA, 2018), ASE 2018, ACM, pp. 132–142.

[96] Zhu, Z., Lin, K., and Zhou, J. Transfer learning in deep reinforcement learning: A survey. *arXiv preprint arXiv:2009.07888* (2020).