HackAtari: Atari Learning Environments for Robust and Continual Reinforcement Learning

Quentin Delfosse[*,1] **Jannis Blüml**[*,1,2] **Bjarne Gregori**[1] **Kristian Kersting**[1,2,3,4]

[1]AI and ML Group, Technical University of Darmstadt, Germany
[2]Hessian Center for Artificial Intelligence (hessian.AI)
[3]Centre for Cognitive Science of Darmstadt
[4]German Research Center for Artificial Intelligence (DFKI)
quentin.delfosse@cs.tu-darmstadt.de, blueml@cs.tu-darmstadt.de

Abstract

Artificial agents' adaptability to novelty and alignment with intended behavior is crucial for their effective deployment. Reinforcement learning (RL) leverages novelty as a means of exploration, yet agents often struggle to handle novel situations, hindering generalization. To address these issues, we propose HackAtari, a framework introducing controlled novelty to the most common RL benchmark, the Atari Learning Environment. HackAtari allows us to create novel game scenarios (including simplification for curriculum learning), to swap the game elements' colors, as well as to introduce different reward signals for the agent. We demonstrate that current agents trained on the original environments include robustness failures, and evaluate HackAtari's efficacy in enhancing RL agents' robustness and aligning behavior through experiments using C51 and PPO. Overall, HackAtari can be used to improve the robustness of current and future RL algorithms, allowing Neuro-Symbolic RL, curriculum RL, causal RL, as well as LLM-driven RL. Our work underscores the significance of developing interpretable in RL agents. [1]

1 Introduction

Deep reinforcement learning (RL) agents struggle to adapt to environments with slightly perturbed goal, while neurosymbolic (NS) agents learn isolated explicit skills, that can easily be combined or adjusted to adapt to environments' modifications. Furthermore, learning agents are prone to shortcut learning, occurring when a model exploits superficial correlations in the training data, resulting in poor generalization to novel situations. To identify if agents base their decision on the wrong input parts, eXplainable AI (XAI) methods, such as importance maps have been used (Schramowski et al., 2020; Ras et al., 2022; Roy et al., 2022; Saeed and Omlin, 2023). In deep reinforcement learning (RL), such shortcut learning behavior is refereed to as goal misgeneralization (Koch et al., 2021; Shah et al., 2022; Tien et al., 2022), *i.e.* to solving a sub-goal aligned with the *true* objective. Such misalignments can be difficult to identify, as exemplified by di Langosco et al. (2022), who showed that agents trained on Coinrun learn to run to the end of the level (sub-goal) instead of learning to reach the coin (true objective) (*cf.* Fig. 1, Left) and by Delfosse et al. (2024b), with agents trained on Pong that learn to follow the enemy's paddle position instead of the ball's one (*cf.* Fig. 1, Right). They show that hiding the enemy or changing its policy (*i.e.* preventing it to move after it returns the ball) lead to agents incapable of catching the ball. Importance maps highlight the player's and enemy's paddles, and the ball, luring external observers into thinking that the agents "understood" that it needs to return the ball behind the enemy's paddle. Map-based explanations indicate the importance of an input element for a decision without indicating *why* this element is important, or how it is used within the decision process (Kambhampati et al., 2021; Stammer et al., 2021a; Teso et al., 2023).

[1]Code available at https://github.com/k4ntz/HackAtari, * Equal Contribution

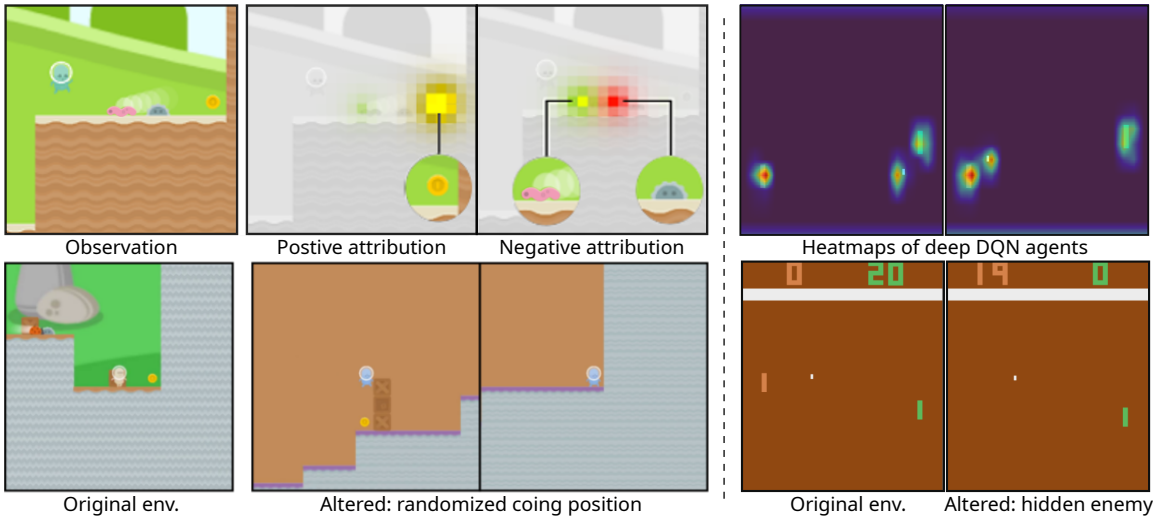


Figure 1: **Examples of misaligned agents.** In Coinrun (left), agents learn to reach the end of the level, instead of the coin. In Pong (right), agents learn to follow the enemy instead of the ball. Importance maps (top) are not enough for detecting such misalignments, environment variations are necessary.

The lack of generality of deep RL agents has previously been identified by Cobbe et al. (2020), who introduced Procgen, a set of procedurally generated environments, with varying assets (for the backgrounds and depicted objects) and level designs. The previously mentioned Coinrun game, with misaligned agents is part of Procgen, showcasing that varying the assets and procedural generation might not be enough. The authors also underline that the Atari Learning Environments (Bellemare et al., 2013) is the by far most used RL benchmark. This test bench completely dominate the test of deep RL agents (*cf.* Fig. 8 in the Appendix), as it incorporates a diverse set of challenges, does not require extensive computations, nor suffer from any experimenter bias. However, Atari games do not provide any variations, making it impossible to test for generality or misalignment.

In this work, we propose ***HackAtari***, a framework that introduces novelty to the Atari Learning Environments. HackAtari contains a set of in total 50 variations on 16 Atari Learning Environments. We show that these variations can be used to test identify RL agents' potential misgeneralizations, as well as to help in curriculum learning settings, by introducing simpler and more complex versions.

To learn robust policies, recent interpretable algorithms explicitly separate the extraction of neurosymbolic states from raw, pixel-based inputs (Lin et al., 2020; Delfosse et al., 2023c; Zhao et al., 2023) from the action selection process. This allows for transparent action selection process, based on *e.g.* first order logic (Delfosse et al., 2023b), on concept bottlenecks (Delfosse et al., 2024b), or on polynomial equations, later explained in natural language by an LLM (Luo et al., 2024). To efficiently train these methods, HackAtari incorporates the Object-Centric Atari framework (OCAtari) (Delfosse et al., 2023a), that provides object-centric representations of the games, to allow for training and comparing both deep and interpretable NS agents. HackAtari also allows for the development of Continual RL method, as many environments provide different level of difficulty, based on the mastering of different skills, each trainable in a curriculum learning fashion. Specifically, our contributions are:

(i) We introduce HackAtari, a set of modifications applied to different Atari environments, that allow for testing variations of the games. It allow to test generalization capabilities of RL agents, as well as training agents in curriculum learning settings.
(ii) We demonstrate that our environments can be used to test already trained agents, as well as to train other agents on our variations.
(iii) We use HackAtari to show the misgeneralisation of existing RL agents.
(iv) We use show that HackAtari allows for learning LLM-defined reward functions.

We start off by introducing the HackAtari framework. We experimentally evaluate the generalization and learning capabilities of RL agents. Before concluding, we touch upon related work.

2 HackAtari: Altered Atari Environments

In this section, we first explain how to create variations of the Atari Learning environments, then describe the 4 alteration categories, that can be used to train and test different RL agents' capabilities.

2.1 HackAtari: step, reset and reward modifications

While being -by far-, the most used benchmark for training and testing RL agents, the Atari Learning environments do not openly provide any source code. We thus cannot directly modify the source code of these environments. However, Anand et al. (2019) have identified that the many useful information about the depicted objects can be observed in the RAM, and proposed *AtariARI*, a wrapper that augments the info dictionary with pointers to the parts of the RAM responsible for *e.g.* the position of the agent. Inspired by this work, Delfosse et al. (2023a) created *OCAtari*, a set of augmented Atari environments that directly provide neurosymbolic (object-centric) states of the games. They identify which part of the RAM are responsible for the object's properties (*e.g.* visibility, position, color, orientation). They are thus able to provide both RGB and neurosymbolic states.

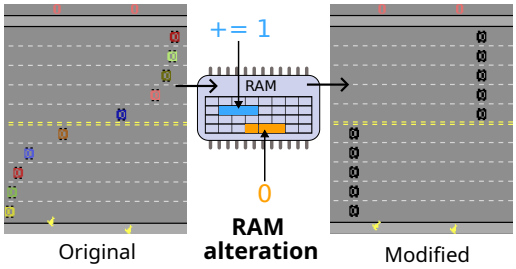


Figure 2: **RAM alteration allows for modified environments**. Exemplified on Freeway. Altering the some RAM cells leads to color and speed changes.

In our work, we use the mappings of RAM values to object states to understand the inner working of the game. We then can alter these RAM values to modify the behaviors of the objects. This is depicted in Fig. 2 on *Freeway*. For this game, the cars colors are encoded at the RAM cells 77 to 86. By setting each of their values to 0, we change the colors of each car to black. Furthermore, we identified the RAM position responsible for the movement of each cars (from 33 to 43). By setting all of them to the same number, we can modify the speed of all the cars, such that they *e.g.* now drive in columns. For some game, some object's properties can be encoded at multiple RAM cells. This is the case for the enemy's paddle position in *Pong*, encoded both at RAM cells 21 and 50. If we overwrite the RAM cell 50 (to modify the behavior of the enemy's paddle, as shown in Fig. 3), the value will not be used by the program and will be overwritten at the next time step by the game. Altering the RAM cell 21 allows for modifying the enemy's policy.

Modifying *Pong* to obtain a *Lazy Enemy Pong* version is more complicated than the previously illustrated changes. Our goal is to have the enemy static after it touches the ball to return it (and a moving enemy after the agent returned the ball). For this, we need to keep track of the position of the ball and of the enemy. By comparing the ball's positions at the previous and current timestep, we can decide whether to enforce the enemy to remain static, or not. If the ball is going to the agent's side, we overwrite the enemy's position to the one it had when it touched the ball.

Our framework incorporates several modification types, that can be combined (*e.g.* altering the cars' color and the cars' speed on *Freeway*). To make these modifications, 3 types of functions can be used:

(i) *state modification at each step*, *e.g.* when the position of an object is being altered, or when objects are being disabled (*i.e.* the corresponding RAM cells are constantly being overwritten).

(ii) *state modification at reset time*, *e.g.* when the level layout is being swapped, or when the agent is spawning at a randomized initial position.

(iii) *reward modification* (at each step), when reward is changed to modify the goal in the environment, such as encouraging peaceful policies in shooting games (or discouraging the use of AMO).

We have explained how to alter the RAM to obtain a resource efficient way of altering the learning environments. For implementation details on other environments, please look at our open-source repository, that we release together with this paper. In the following, we explain how we categorize the environments' modifications based on the type of training and/or testing that they allow for.

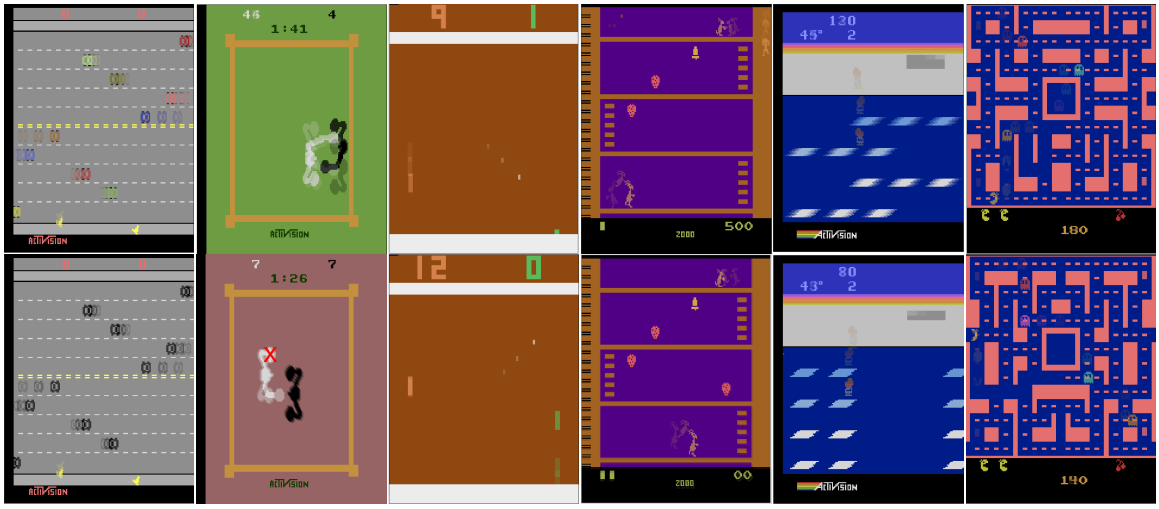


Figure 3: **HackAtari provides variations of Atari environments.** These include color changes (Freeway and Boxing), gameplay shifts (Boxing, MsPacman), continual learning settings (Kangaroo and Frostbite). The original games (top) are compared to HackAtari's modified versions (bottom). Superposed frames show the game dynamics.

2.2 Testing visual and dynamics robustness, curriculum RL and adaptability.

HackAtari environments alterations aim at testing different capabilities of RL agents. We thus categorize these modifications into the 4 following paradigms.

Table 1: **HackAtari variations.** ✓ mark at least one existing variation of the environment within HackAtari.

Modification	Bankheist	BattleZ.	Boxing	Breakout	Carnival	ChopperC.	DonkeyK.	FishingD.	Freeway	Frostbite	Kangaroo	Montezuma.	MsPacman	Pong	Riverraid	Seaquest	Skiing	SpaceInv.	Tennis
Color			✓						✓	✓									
Gameplay	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Skill	✓	✓			✓	✓	✓	✓	✓	✓	✓		✓	✓	✓				✓
Goal							✓	✓					✓		✓	✓		✓	

1. Visual Domain Adaptation. Altering the RAM allows changing the colors of the depicted objects in many environments. This is showcased in Fig. 2 on *Freeway*. These shifts can be used to test simple shortcut learning. For this type of shortcut learning, neural networks learn to associate specific pixel values to actions, as identified by Stammer et al. (2021b) on classification tasks. We also provide a general way of modifying any pixel color for any ALE game, exemplified in Fig. 3 on *Boxing*, using simple pixel values mapping and replacement. HackAtari incorporates a testing script, that automatically randomly swap the present pixel values. We however advise RL practitioners to use these settings to test their agents only, or to build offline RL datasets, as the color processing of the frame is resource intensive. For faster execution, we recommend using the color swaps done by our RAM alterations techniques for training agents. We mark them in the *Color* line of Tab. 1.

2. Dynamics Adaptation. For this changes, we alter the gameplay, and agents have to adapt to small perturbations. These games can be used to test the robustness and adaptability of RL agents. Examples of gameplay shifts are: small drifts in ball games (*e.g. Breakout* and *Tennis*), gravity slightly pulling the agent downwards (in *e.g.* Boxing, Pong, Seaquest), level layout change *e.g.* in *MsPacman* (*cf.* Fig. 3), velocity modifications (*e.g.* on cars in *Freeway*, on the projectiles in *Carnival*). For this category of games, there is no distribution shift in term of pixel values, but the agents have to adapt to novel situations. They also allow to detect another shortcut learning type, such as the one described in introduction on Pong, where the agent learns to follow the enemy instead of the ball (Delfosse et al., 2024b; Kohler et al., 2024). HackAtari incorporates a version of *Pong*, where the enemy is static after returning the ball (*cf.* Fig. 3). Some games, such as *MsPacman*, *Kangaroo*, *Montezuma's Revenge* also include multiple levels or level parts, that are usually accessible if the first level is completed. For these, we provide one type of modification that randomly select one level/part

after each death, or each reset of the environment. This allows the agent to learn more generalizable policies (or to be tested), for abilities such as navigate another maze layout, *cf. MsPacman* in Fig. 3).

3. Curriculum Reinforcement Learning (CRL). In this learning paradigm, the task complexity is incrementally increased. CRL environments are used to assess the agents' ability to learn different skills in a structured manner with incremental complexity. For example, we created versions of *Kangaroo* with disabled monkeys and disabled thrown coconuts, of *Seaquest* without enemy and with unlimited oxygen, or Freeway with stopped cars. These can be used to separately learn navigations from other skills (Mirowski et al., 2016; Sharifi et al., 2023), or to learn different options (*i.e.* high level action) using small networks (Stolle and Precup, 2002; Bacon et al., 2017). Novel research is also looking into neural merging (Kirkpatrick et al., 2016; Brahma et al., 2021), lately brought to continual RL (Gracla et al., 2023). Finally, neurosymbolic methods often explicitely learn separated skills (Kimura et al., 2021; Delfosse et al., 2023b) on their logic-based RL agents.

4. Reward Signal Adaptation. We also incorporate an easy reward modification implementation, to test the agent's ability to rapidly adapt to new objectives, potentially using large language models (LLMs) to generate the reward signals (Ma et al., 2023; Xie et al., 2023), or to evaluate how well RL agents align with human societal values (Pan et al., 2021). HackAtari enables users to specify rewards defined by large language models (LLMs) and to incorporate alternative reward structures that change the game's objectives. For example, in the game Kangaroo, the agent can be rewarded more for punching monkeys than for saving the joey. By reducing the reward for aggressive actions and increasing it for reaching the joey, a different policy is created that is more aligned with the original game's goals (Delfosse et al., 2024b). More broadly, this approach can be used to decrease incentives for aggressive behaviors, such as in shooting games (Pan et al., 2021), or to leverage LLMs for generating various reward types in skill acquisition tasks. A comprehensive example of modifying and adjusting the reward function is provided in App. D.

3 Experimental Evaluation

In our evaluation, we investigate several benefits of using HackAtari over the original Atari Learning Environments. Specifically, we aim at answering the following research questions:

- **(Q1)** Are our HackAtari variations usable for training RL agents?
- **(Q2)** Can HackAtari variation reveal misalignment and be used to correct it?
- **(Q3)** Can alternative reward functions allow for adjusting learned behaviors?
- **(Q4)** Can some HackAtari environments be used for curriculum learning?

Experimental Setup. We compare PPO (Schulman et al., 2017) and C51 (Bellemare et al., 2017) agents' on our modified environments to ones trained on the original ones (all averaged over 3 seeds). We also let human agents train on the original environments, and test them on both the original and modified versions (*cf.* App. B.4 for details on this user study). We average human scores across at least 3 human users. All agents use the classical DQN input representation (Mnih et al., 2015), *i.e.* process $4 \times 84 \times 84$ frame stack and are trained on 10M frames, on a 40 GB DGX A100 server. We used PPO implementation of Huang et al. (2022) with its default hyperparameters (*cf.* App. C.1). We focus the evaluations on robustness of the used policy, rather than achieving the best possible scores.

Game variations details. We here concisely present the games' variations that we refer to in this section. A more detailed list can be found in App. B.1 and a list of all modifications in App. E.
One Armed (*Boxing*): The agent can only punch its opponent with its right arm (instead of both).
No Barrel (*DonkeyKong*): We cancel falling barrels that the agent must avoid.
Aligned Cars (*Freeway*): All cars have identical speed, and thus travel aligned.
Mono-Colored (*Freeway*): All the cars have identical colors (*e.g.* black, *cf.* Fig. 3).
Stopped Cars (*Freeway*): All the cars are visible but static, allowing any agent to safely cross.
No Danger (*FrostBite*): The horizontally traveling enemies (in between the ice blocks) are canceled.
Static Ice (*FrostBite*): The ice blocks are static instead of moving sideways (*cf.* Fig. 3).
No Danger (*Kangaroo*): We cancel both monkey enemies and deadly falling coconuts.
Safe and Close (*Kangaroo*): No Danger + The agent randomly spawns on 1^{st}, 2^{nd} or 3^{rd} floor.
Swap Level (*MsPacman*): The agent spans in a level with a different maze layout (*cf.* Fig. 3).

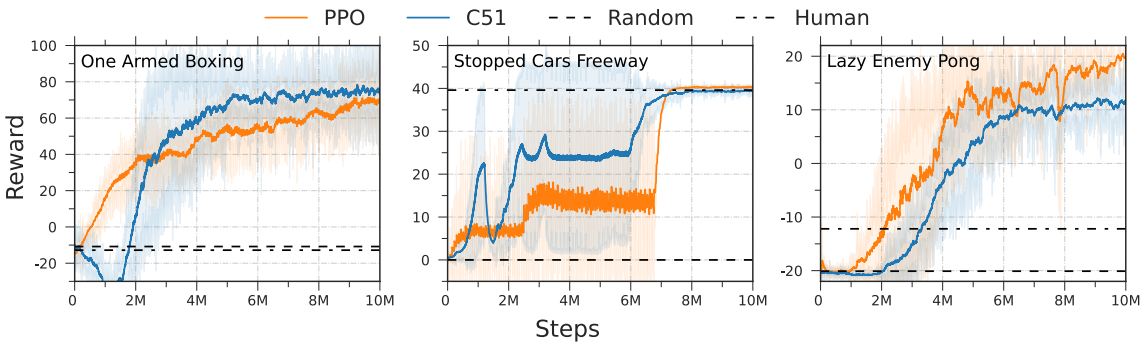


Figure 4: **RL agents can learn on altered environments**, exemplified on *One armed Boxing*, *Mono-Colored Freeway* and *Lazy Enemy Pong*, by PPO and C51 agents. These agents are able to progressively improve from random to (or beyond) the human level. *Freeway*'s high variance is due to the number of frames needed before each seeded agent reaches the top.

Lazy Enemy (*Pong*): The enemy stands still after returning the ball instead of following it (*cf.* Fig. 3).
Infinite Oxygen (*Seaquest*): The level of oxygen does not decrease (stays at 100%).
No Shield (*SpaceInvaders*): Shield that cancel missiles from the agents and the enemies are removed.

HackAtari modified environments can be used for learning (Q1). Before other investigations, we need to verify that our environments are usable to train RL agents, *i.e.* that they do not lead to games impossible to complete. After training on our modified versions, PPO and C51 agents consistently demonstrate the ability to master the introduced changes (*cf.* learning curves in Fig. 4, PPO vs. Random in Tab. 2 and the additional curves in App. B.2), *i.e.* are able to increase their scores in our variants. Our findings confirm that these game variants can be used to train policies and evaluate the generalization capabilities of agents trained on the original environments (to *e.g.* compare them to humans or agents directly trained on the variations).

HackAtari can help uncover flaws of trained agents (Q2). Let us now test the generalization capabilities of artificial agents using HackAtari's variations. PPO's performances remarkably drop on all of our tested variations, whereas humans' ones increase in all but *Boxing* (*cf.* Tab. 2). For this game, the PPO agents trained on this variation are able to adapt their policies and achieve comparable performances, even with the absence of the second arm. For *Kangaroo* and *DonkeyKong*, the lack of enemies/barrels simplifies the game, allowing humans to safely reach the end of the level, a behavior that PPO agents have not learned, as they were not able to observe reward for this behavior during this training. These results show that autonomous agents could benefit from incorporating other learning or planning modules, *e.g.* (vision) language models, to guide the agent in constantly changing environments. We can also confirm the findings of Delfosse et al. (2024b): RL agents trained on *Pong* learned misaligned behavior, heavily relying on the enemy's position (instead of the ball's one). Rearranging the walls also lead to performance drops in *MsPacman*, showing that these agents follow a remembered pattern instead of learning navigation (Sukhbaatar et al., 2018; Burda et al., 2018).

HackAtari allows to learn alternative behaviors (Q3). We make use of the integration of alternative reward function of HackAtari to test if agents can learn based on *e.g.* LLM provided guidance (detailed in App. D) instead of the original score, included in the ALE games. The original Seaquest game does not reward agents for saving divers, but only for shooting enemies. In this experiment, we encourage this more pacifist behavior by rewarding for rescuing them. The LLM provided us with a reward signal aligned with this goal (*cf.* App. D), that we used to train PPO agents (blue line in Fig. 5). The LLM was able to produce a useful reward function, allowing the agents to learn. Similarly, expert defined functions can be used, as done by Delfosse et al. (2024b), who redefine reward signals in Pong (adding a penalty on distance between the ball and the player), Kangaroo (rewarding the progress of the mother kangaroo towards its joey), and on Skiing (fixing the reward ill-defined reward, by directly rewarding the agents when they pass in between the poles). We were able to reproduce their results by integrating their provided reward function to the HackAtari environments.

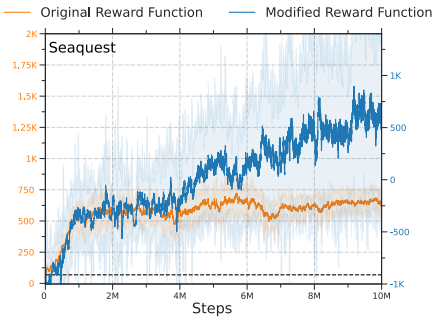


Figure 5: **LLM can guide RL agents.** Performances of PPO agents trained using an LLM-provided reward function (blue) and the original reward (orange).

Simplifications enable skill learning (Q4). As shown in Tab. 2, agents are able to learn the game of *Freeway* within $10M$ steps. However, in our experiment, we observed that C51 agents fail to do it if we do not use the frame skipping optimization (slightly increasing the sparsity). To observe reward, the randomly playing agent needs to cross the 10 lines, knowing that it is pushed back down when hit by a car. Using our *Stopped Cars* variation of *Freeway*, we decrease this sparsity, allowing the agents to observe reward in less than a million steps. We were thus able to first train the agent for $7M$ steps on this simpler variation, allowing it to learn that reaching the top provides reward, before continuing learning for the last $3M$ steps, that allow it to adjust to avoid cars (*cf.* Fig. 6). These experiments showcase that HackAtari gameplay simplifications can be used to now evaluate curriculum RL techniques on ALE.

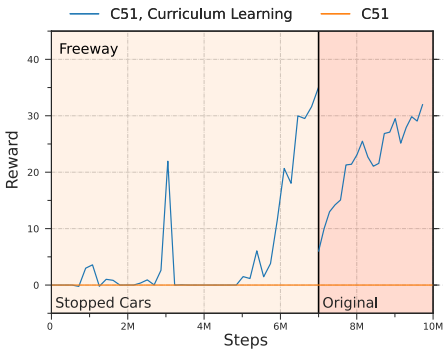


Figure 6: **Curriculum learning, using a simplified version.** C51 agent learn to reach the top (for $7M$ steps on easier variant), before learning to avoid cars.

Our next experiments demonstrate the usability of HackAtari for parallel skills' acquisition.

In *Frostbite*, the agent has to collect ice blocks to build its igloo (*cf.* Fig. 3), by jumping on moving ice

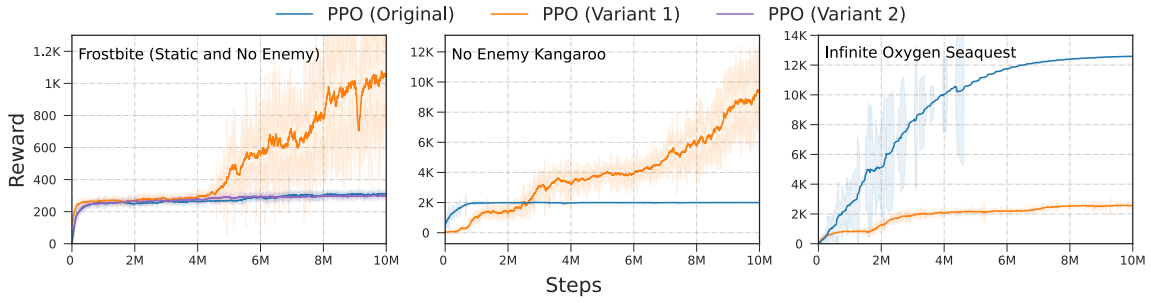


Figure 7: **Alternate versions allow to learn different skills:** (i) In *Frostbite*, RL agents can learn to safely collect ice blocks and fishes on the *No Danger* variant, then to avoid the enemies on a simpler *Static Ice* second variant (left), (ii) with *Safe and Close* agents learn to reach the joey, while they learn to punch Monkey in the original game (center), (iii) in *Infinite Oxygen Seaquest*, agents can also learn to kill enemies (as in the original), but also to collect divers' group (right).

platforms, while avoiding enemies. It can also collect fishes, which provides additional reward. Collecting blocks without falling in the water can be more easily learned, using our first *No Danger* variant, while the enemy-aware navigation skill can be acquired on the second *Static Ice* variant (*cf.* Fig. 7, left). Merging these overlapping skills could be achieved using neural merging techniques.
In *Kangaroo*, Delfosse et al. (2024b) identified that agents learn to punch monkeys instead of saving the joey, as this originally intended goal is more difficult to reach, and use an alternate reward signal to further incentivize for saving the joey. We here identify that fighting monkeys leads to 4.5 times higher scores than the goal described in the game manual[2] (*cf.* Fig. 7, center). Indeed, the agents learn to reach the joey on our *Safe and Close* variation, that cancels the monkey enemies. Interpretable skill-based agents could learn these two behaviors in a curriculum learning manner, as well as when to select each, and alternate reward signals could also boost the learning speed of the saving behavior.
Finally, in *Seaquest*, the agents usually learn to shoot enemies until their oxygen bar is depleted. Our *Infinite Oxygen* version allows it to learn to kill more enemies, as well as surfacing when enough divers have been saved (*cf.* Fig. 7, right). We also provide a way to cancel the enemies (even if not used here). Again, this demonstrates that HackAtari's gameplay simplications can allow for controlled separate skill learning (or options (Bacon et al., 2017)). Other environments, such as *MsPacman* with *Caged Ghosts*, could allow for first learning to navigate the different mazes, before releasing the ghosts and have the agent learning to adapt its behavior to avoid getting killed by them (or to collect the superpower pills that allow to chase them). We leave this for future work.

Overall, our experimental evaluations have demonstrated that HackAtari can be used to train and evaluate agents on alternate tasks within the same (or similar) environment, for which humans require no retraining. We also demonstrated that they can be used to emphasize the acquisition of alternative skills, or of curriculum learning (on simplified environments *e.g. Stopped Cars Freeway*).

4 Related Work

Novelty in RL. In general, novelty can be attributed to the characteristics of an object, to an event involving the object, or to an act of manipulation with an object. Although there is no strict definition of novelty in RL or ML, it often relies on either novelty detection (Markou and Singh, 2003a;b; Schmidhuber, 2008; Pimentel et al., 2014) or intrinsic-motivation-based exploration Oudeyer and Kaplan (2007); Singh et al. (2010); Barto et al. (2013); Siddique et al. (2017). The latter, as described by Harlow (1950), describes intrinsic motivation as the drive to manipulate and explore features, *i.e.* explore uncertain or novel elements of the environment. While HackAtari works as a benchmark or test suite for novelty detection, in this work, we focus on the aspect of HackAtari as a valuable tool for identifying and addressing misalignment within reinforcement learning environments.

Evaluating Generalization and Robustness. Assessing generalization and robustness gets increasing attention Tec et al. (2023); Linial et al. (2023); Busch et al. (2024), particularly in RL, as it ensures that AI agents can apply their learned behaviors effectively. Recently, some started to develop special benchmarks to grapple with this in multiple ways (Nichol et al., 2018; Justesen et al., 2018; Juliani et al., 2019; Cobbe et al., 2019). Procgen (Cobbe et al., 2020), provides a standardized platform to evaluate generalization and robustness across diverse, procedural generated environments. This innovation addresses critical challenges, *e.g.* , overfitting or studying transfer learning capabilities effectively and plays a pivotal role in advancing RL algorithms (Cobbe et al., 2020; Mohanty et al., 2021). Atari (Bellemare et al., 2013) on the other hand, as stated by Cobbe et al. (2020), is the gold standard for benchmarking in RL but lacks variety in the state spaces to be used to evaluate generalization and robustness. While the first steps have been done in this direction (Farebrother et al., 2018; Tomilin et al., 2024), with HackAtari, we start adding more of the needed variety to the ALE by generating new versions of already known games. Other work have looked into plasticity to help agents to adapt to change. Thus, Delfosse et al. (2024a) categorize Atari games based on their amount of change during the learning phase, but do not evaluate agents on unseen data.

[2] www.retrogames.cz/play_195-Atari2600.php

Continual reinforcement learning benchmarks. Atari games serve as crucial testing grounds for increasingly complex RL methods (Hessel et al., 2018; Hafner et al., 2020; Badia et al., 2020; Fan et al., 2021; Farebrother et al., 2022; Xuan et al., 2024). Despite achieving superhuman performance, challenges such as efficient exploration, algorithmic efficiency, planning with sparse rewards, sample inefficiency, and generalization failures persist. The need for expanded Atari benchmarks is recognized by researchers such as Toromanoff et al. (2019) and Fan (2021), who propose additional metrics for accurate performance assessment. Further, efforts like the Atari $100k$ benchmark by Kaiser et al. (2020), curated subsets of Atari Learning Environment (ALE) environments by Aitchison et al. (2023), and the Mask Atari benchmark for POMDPs by Shao et al. (2022) contribute to advancing Atari as a benchmarking tool in RL research even further. HackAtari follow on from here and offers a wide range of applications, such as benchmarking for generalization, to help with explainable RL tasks or create new tasks for continuous RL approaches. We believe that comparison methods between different continual methods (Mundt et al., 2021) should be brought to RL.

5 Limitations

While HackAtari provides a robust framework for evaluating RL agents, it is currently limited to the set of Atari Learning Environments. This restricts the ability to fully assess transfer learning and adaptability across a broader spectrum of tasks. The findings may not always generalize to more complex or varied environments, highlighting the need to extend HackAtari to incorporate a wider variety of scenarios, including 3D environments and more complicated real-time strategy games.

Additionally, we believe that further research is needed to integrate human learning principles more effectively into RL frameworks. Moreover, while large language models and vision models show promise for enhancing RL agents, they introduce complexities and require significant computational resources, posing challenges for seamless integration. Addressing these limitations will be crucial for realizing the full potential of agents able to adapt to HackAtari variations in driving autonomous agents' innovation. To better evaluate the gap between such agents and human ones, we are conducting a broader user study investigation, that involve more subjects on more game variations.

6 Conclusion

In this study, we introduce HackAtari, a framework designed to test the generalization, robustness, and curriculum learning capabilities of RL agents on the most commonly used set of Atari Learning Environments. By offering a wide range of modifications to existing Atari games, HackAtari will enable researchers to create more human-like and adaptive agents, addressing key challenges. It also serves as a valuable tool for studying the behavior and decision-making processes of RL agents, offering insights into how agents adapt to novel environments and tasks, and helping to uncover various shortcut learning behaviors, such as RL agents following the enemy on *Pong*, or learning a navigation path on *MsPacman*. Through a series of experiments, we evaluate the efficacy of HackAtari in uncovering such misalignments, testing RL agents' adaptability to novel environments, and enabling curriculum learning in the Atari games.

7 Ethical Consideration and Broader Impact

The development and use of HackAtari raise ethical considerations. Researchers must be vigilant about the potential misuse of adaptive agents. Furthermore, while our variations are lightweight, the computational resources required for advanced models may have environmental impacts, necessitating efforts to develop more energy-efficient algorithms. HackAtari also has potential for broader impacts. By facilitating the creation of more human-like and adaptable autonomous agents, it can drive advancements in various fields, from autonomous systems to interactive entertainment. However, it is essential to balance innovation with ethical responsibility, ensuring that these technologies benefit society as a whole and do not exacerbate existing inequalities or create new ethical dilemmas.

Acknowledgment

We would like to express our sincere gratitude to our students Felicia Benkert, Lars Wassmann, Mohammad Azimpour, Svenja Droll, Annabell Schmidt, Julian Mehler, Markus Lang and Tim Blaschzyk for dedication and feedback. Their enthusiasm and perseverance have been instrumental in achieving the results presented in this paper. We also thank our user study participants for their time and insights. Thank you for your continuous efforts and commitment.

This research work has been funded by the German Federal Ministry of Education and Research, the Hessian Ministry of Higher Education, Research, Science and the Arts (HMWK) within their joint support of the National Research Center for Applied Cybersecurity ATHENE, via the "SenPai: XReLeaS" project as well as their cluster project within the Hessian Center for AI (hessian.AI) "The Third Wave of Artificial Intelligence - 3AI".

References

Matthew Aitchison, Penny Sweetser, and Marcus Hutter. Atari-5: Distilling the arcade learning environment down to five games. In *International Conference on Machine Learning, ICML*, 2023.

Ankesh Anand, Evan Racah, Sherjil Ozair, Yoshua Bengio, Marc-Alexandre Côté, and R. Devon Hjelm. Unsupervised state representation learning in atari. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems*, 2019.

Pierre-Luc Bacon, Jean Harb, and Doina Precup. The option-critic architecture. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

Adrià Puigdomènech Badia, Bilal Piot, Steven Kapturowski, Pablo Sprechmann, Alex Vitvitskyi, Zhaohan Daniel Guo, and Charles Blundell. Agent57: Outperforming the atari human benchmark. In *Proceedings of the 37th International Conference on Machine Learning, ICML*, 2020.

Andrew Barto, Marco Mirolli, and Gianluca Baldassarre. Novelty or surprise? *Frontiers in psychology*, 2013.

Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 2013.

Marc G. Bellemare, Will Dabney, and Rémi Munos. A distributional perspective on reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning*, 2017.

Dhanajit Brahma, Vinay Kumar Verma, and Piyush Rai. Hypernetworks for continual semi-supervised learning. *ArXiv*, 2021.

Yuri Burda, Harrison Edwards, Amos Storkey, and Oleg Klimov. Exploration by random network distillation. In *International Conference on Learning Representations*, 2018.

Florian Peter Busch, Roshni Kamath, Rupert Mitchell, Wolfgang Stammer, Kristian Kersting, and Martin Mundt. Where is the truth? the risk of getting confounded in a continual world. *arXiv*, 2024.

Karl Cobbe, Oleg Klimov, Christopher Hesse, Taehoon Kim, and John Schulman. Quantifying generalization in reinforcement learning. In *Proceedings of the 36th International Conference on Machine Learning*, 2019.

Karl Cobbe, Chris Hesse, Jacob Hilton, and John Schulman. Leveraging procedural generation to benchmark reinforcement learning. In *International conference on machine learning*, 2020.

Quentin Delfosse, Jannis Blüml, Bjarne Gregori, Sebastian Sztwiertnia, and Kristian Kersting. Ocatari: Object-centric atari 2600 reinforcement learning environments. 2023a.

Quentin Delfosse, Hikaru Shindo, Devendra Singh Dhami, and Kristian Kersting. Interpretable and explainable logical policies via neurally guided symbolic abstraction. *Advances in Neural Information Processing (NeurIPS)*, 2023b.

Quentin Delfosse, Wolfgang Stammer, Thomas Rothenbacher, Dwarak Vittal, and Kristian Kersting. Boosting object representation learning via motion and object continuity. In *European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML)*, 2023c.

Quentin Delfosse, Patrick Schramowski, Martin Mundt, Alejandro Molina, and Kristian Kersting. Adaptive rational activations to boost deep reinforcement learning. In *International Conference on Learning Representations (ICLR)*, 2024a.

Quentin Delfosse, Sebastian Sztwiertnia, Wolfgang Stammer, Mark Rothermel, and Kristian Kersting. Interpretable concept bottlenecks to align reinforcement learning agents. 2024b.

Lauro Langosco di Langosco, Jack Koch, Lee D. Sharkey, Jacob Pfau, and David Krueger. Goal misgeneralization in deep reinforcement learning. In *International Conference on Machine Learning ICML*, 2022.

Jiajun Fan. A review for deep reinforcement learning in atari: Benchmarks, challenges, and solutions. 2021.

Jiajun Fan, Changnan Xiao, and Yue Huang. GDI: rethinking what makes reinforcement learning different from supervised learning. 2021.

Jesse Farebrother, Marlos C. Machado, and Michael Bowling. Generalization and regularization in DQN. 2018.

Jesse Farebrother, Joshua Greaves, Rishabh Agarwal, Charline Le Lan, Ross Goroshin, Pablo Samuel Castro, and Marc G Bellemare. Proto-value networks: Scaling representation learning with auxiliary tasks. In *The Eleventh International Conference on Learning Representations*, 2022.

Steffen Gracla, Carsten Bockelmann, and Armin Dekorsy. A multi-task approach to robust deep reinforcement learning for resource allocation. In *WSA & SCC 2023; 26th International ITG Workshop on Smart Antennas and 13th Conference on Systems*, 2023.

Danijar Hafner, Timothy P. Lillicrap, Jimmy Ba, and Mohammad Norouzi. Dream to control: Learning behaviors by latent imagination. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*, 2020.

Harry F Harlow. Learning and satiation of response in intrinsically motivated complex puzzle performance by monkeys. *Journal of comparative and physiological psychology*, 1950.

Matteo Hessel, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. In *Proceedings of the AAAI conference on artificial intelligence*, 2018.

Shengyi Huang, Rousslan Fernand Julien Dossa, Chang Ye, Jeff Braga, Dipam Chakraborty, Kinal Mehta, and João G.M. Araújo. Cleanrl: High-quality single-file implementations of deep reinforcement learning algorithms. *Journal of Machine Learning Research*, 2022.

Arthur Juliani, Ahmed Khalifa, Vincent-Pierre Berges, Jonathan Harper, Ervin Teng, Hunter Henry, Adam Crespi, Julian Togelius, and Danny Lange. Obstacle tower: A generalization challenge in vision, control, and planning. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI*, 2019.

Niels Justesen, Ruben Rodriguez Torrado, Philip Bontrager, Ahmed Khalifa, Julian Togelius, and Sebastian Risi. Illuminating generalization in deep reinforcement learning through procedural level generation. 2018.

Lukasz Kaiser, Mohammad Babaeizadeh, Piotr Milos, Blazej Osinski, Roy H. Campbell, Konrad Czechowski, Dumitru Erhan, Chelsea Finn, Piotr Kozakowski, Sergey Levine, Afroz Mohiuddin, Ryan Sepassi, George Tucker, and Henryk Michalewski. Model based reinforcement learning for atari. In *8th International Conference on Learning Representations, ICLR, 2020*, 2020.

Subbarao Kambhampati, Sarath Sreedharan, Mudit Verma, Yantian Zha, and L. Guan. Symbols as a lingua franca for bridging human-ai chasm for explainable and advisable ai systems. In *AAAI Conference on Artificial Intelligence*, 2021.

Daiki Kimura, Masaki Ono, Subhajit Chaudhury, Ryosuke Kohita, Akifumi Wachi, Don Joven Agravante, Michiaki Tatsubori, Asim Munawar, and Alexander Gray. Neuro-symbolic reinforcement learning with first-order logic. In *Conference on Empirical Methods in Natural Language Processing*, 2021.

James Kirkpatrick, Razvan Pascanu, Neil C. Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A. Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, Demis Hassabis, Claudia Clopath, Dharshan Kumaran, and Raia Hadsell. Overcoming catastrophic forgetting in neural networks. *Proceedings of the National Academy of Sciences*, 2016.

Jack Koch, Lauro Langosco, Jacob Pfau, James Le, and Lee Sharkey. Objective robustness in deep reinforcement learning, 2021.

Hector Kohler, Quentin Delfosse, Riad Akrour, Kristian Kersting, and Philippe Preux. Interpretable and editable programmatic tree policies for reinforcement learning. *arXiv*, 2024.

Zhixuan Lin, Yi-Fu Wu, Skand Vishwanath Peri, Weihao Sun, Gautam Singh, Fei Deng, Jindong Jiang, and Sungjin Ahn. SPACE: unsupervised object-oriented scene representation via spatial attention and decomposition. In *International Conference on Learning Representations*, 2020.

Ori Linial, Guy Tennenholtz, and Uri Shalit. Benchmarks for reinforcement learning with biased offline data and imperfect simulators. 2023.

Lirui Luo, Guoxi Zhang, Hongming Xu, Yaodong Yang, Cong Fang, and Qing Li. Insight: End-to-end neuro-symbolic visual reinforcement learning with language explanations. *arXiv*, 2024.

Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Eureka: Human-level reward design via coding large language models. In *The Twelfth International Conference on Learning Representations*, 2023.

Markos Markou and Sameer Singh. Novelty detection: a review - part 1: statistical approaches. *Signal Process.*, 2003a.

Markos Markou and Sameer Singh. Novelty detection: a review - part 2: : neural network based approaches. *Signal Process.*, 2003b.

Piotr Wojciech Mirowski, Razvan Pascanu, Fabio Viola, Hubert Soyer, Andy Ballard, Andrea Banino, Misha Denil, Ross Goroshin, L. Sifre, Koray Kavukcuoglu, Dharshan Kumaran, and Raia Hadsell. Learning to navigate in complex environments. *ArXiv*, 2016.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin A. Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 2015.

Sharada Mohanty, Jyotish Poonganam, Adrien Gaidon, Andrey Kolobov, Blake Wulfe, Dipam Chakraborty, Gražvydas Šemetulskis, João Schapke, Jonas Kubilius, Jurgis Pašukonis, et al. Measuring sample efficiency and generalization in reinforcement learning benchmarks: Neurips 2020 procgen benchmark. 2021.

Martin Mundt, Steven Lang, Quentin Delfosse, and Kristian Kersting. Cleva-compass: A continual learning evaluation assessment compass to promote research transparency and comparability. In *International Conference on Learning Representations*, 2021.

Alex Nichol, Vicki Pfau, Christopher Hesse, Oleg Klimov, and John Schulman. Gotta learn fast: A new benchmark for generalization in RL. 2018.

Pierre-Yves Oudeyer and Frédéric Kaplan. What is intrinsic motivation? A typology of computational approaches. *Frontiers Neurorobotics*, 2007.

Alexander Pan, Kush Bhatia, and Jacob Steinhardt. The effects of reward misspecification: Mapping and mitigating misaligned models. In *International Conference on Learning Representations*, 2021.

Marco A.F. Pimentel, David A. Clifton, Lei Clifton, and Lionel Tarassenko. A review of novelty detection. *Signal Processing*, 2014.

Gabrielle Ras, Ning Xie, Marcel van Gerven, and Derek Doran. Explainable deep learning: A field guide for the uninitiated. *Journal of Artificial Intelligence Research*, 2022.

Nicholas A. Roy, Junkyung Kim, and Neil C. Rabinowitz. Explainability via causal self-talk. 2022.

Waddah Saeed and Christian W. Omlin. Explainable AI (XAI): A systematic meta-survey of current challenges and future opportunities. *Knowledge-Based Systems*, 2023.

Jürgen Schmidhuber. Driven by compression progress: A simple principle explains essential aspects of subjective beauty, novelty, surprise, interestingness, attention, curiosity, creativity, art, science, music, jokes. In *Anticipatory Behavior in Adaptive Learning Systems, From Psychological Theories to Artificial Cognitive Systems*, 2008.

Patrick Schramowski, Wolfgang Stammer, Stefano Teso, Anna Brugger, Franziska Herbert, Xiaoting Shao, Hans-Georg Luigs, Anne-Katrin Mahlein, and Kristian Kersting. Making deep neural networks right for the right scientific reasons by interacting with their explanations. *Nature Machine Intelligence*, 2020.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. 2017.

Rohin Shah, Vikrant Varma, Ramana Kumar, Mary Phuong, Victoria Krakovna, Jonathan Uesato, and Zachary Kenton. Goal misgeneralization: Why arxivect specifications aren't enough for arxivect goals. *ArXiv*, 2022.

Yang Shao, Quan Kong, Tadayuki Matsumura, Taiki Fuji, Kiyoto Ito, and Hiroyuki Mizuno. Mask atari for deep reinforcement learning as POMDP benchmarks. 2022.

Iman Sharifi, Mustafa Yusuf Yildirim, and Saber Fallah. Towards safe autonomous driving policies using a neuro-symbolic deep reinforcement learning approach. *ArXiv*, 2023.

Nazmul H. Siddique, Paresh Dhakan, Iñaki Rañó, and Kathryn E. Merrick. A review of the relationship between novelty, intrinsic motivation and reinforcement learning. *Paladyn Journal Behavior Robotics*, 2017.

Satinder Singh, Richard L. Lewis, Andrew G. Barto, and Jonathan Sorg. Intrinsically motivated reinforcement learning: An evolutionary perspective. *IEEE Trans. Auton. Ment. Dev.*, 2010.

Wolfgang Stammer, Patrick Schramowski, and Kristian Kersting. Right for the right concept: Revising neuro-symbolic concepts by interacting with their explanations. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR*, 2021a.

Wolfgang Stammer, Patrick Schramowski, and Kristian Kersting. Right for the right concept: Revising neuro-symbolic concepts by interacting with their explanations. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2021b.

Martin Stolle and Doina Precup. Learning options in reinforcement learning. In *Abstraction, Reformulation, and Approximation: 5th International Symposium, SARA*, 2002.

Sainbayar Sukhbaatar, Zeming Lin, Ilya Kostrikov, Gabriel Synnaeve, Arthur Szlam, and Rob Fergus. Intrinsic motivation and automatic curricula via asymmetric self-play. In *International Conference on Learning Representations*, 2018.

Mauricio Tec, Ana Trisovic, Michelle Audirac, and Francesca Dominici. Space: The spatial confounding (benchmarking) environment. 2023.

Stefano Teso, Öznur Alkan, Wolfgang Stammer, and Elizabeth Daly. Leveraging explanations in interactive machine learning: An overview. *Frontiers in Artificial Intelligence*, 2023.

Jer Yi Tien, Jerry Zhi-Yang He, Zackory M. Erickson, Anca D. Dragan, and Daniel S. Brown. Causal confusion and reward misidentification in preference-based reward learning. In *International Conference on Learning Representations*, 2022.

Tristan Tomilin, Meng Fang, Yudi Zhang, and Mykola Pechenizkiy. Coom: A game benchmark for continual reinforcement learning. *Advances in Neural Information Processing Systems*, 2024.

Marin Toromanoff, Émilie Wirbel, and Fabien Moutarde. Is deep reinforcement learning really superhuman on atari? 2019.

Tianbao Xie, Siheng Zhao, Chen Henry Wu, Yitao Liu, Qian Luo, Victor Zhong, Yanchao Yang, and Tao Yu. Text2reward: Reward shaping with language models for reinforcement learning. 2023.

Chunyu Xuan, Yazhe Niu, Yuan Pu, Shuai Hu, and Jing Yang. Rezero: Boosting mcts-based algorithms by just-in-time and speedy reanalyze. *arXiv preprint*, 2024.

Xu Zhao, Wenchao Ding, Yongqi An, Yinglong Du, Tao Yu, Min Li, Ming Tang, and Jinqiao Wang. Fast segment anything, 2023.

A On the impact of Atari games in RL.

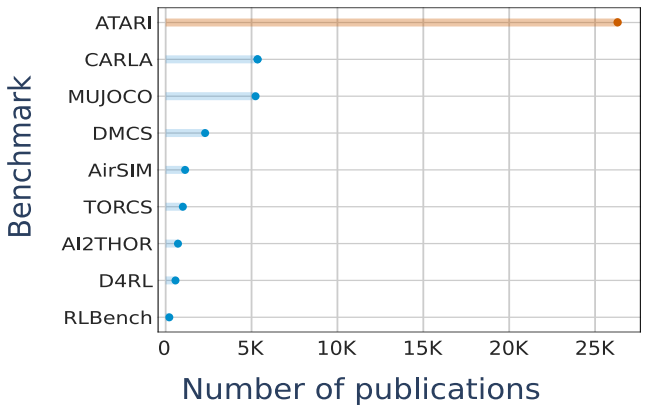


Figure 8: **RL research needs diverse Atari environments.** The Atari Learning Environments is, by far, the most used RL benchmark among the ones listed on paperswithcode.com.

B Details on and Additional Results

This section provides supplementary information to support the findings presented in the paper. We include descriptions of the game variations and additional results to further validate our conclusions. This information ensures transparency and reproducibility, offering deeper insights into our research methodologies and outcomes.

B.1 Game Variants/Modifications used in our Evaluation (Section 3

These are the environments, used in our evaluation. We shortly state the changes we made as well as the modifications (*cf.* Section E) we used to create these environments. A shorter version can be found in Section 3 and additional environments follow in the next section.

(i) **One Armed *Boxing*.** In *Boxing*, the agent controls the white boxer that has to punch an automated opponent (while avoiding getting punched). In the original game, both boxers can hit their opponents with both arms. We introduce *One armed*, where the agent can only use its right arm, forcing the agent to adapt its behavior. This variates the gameplay instead of modifying the color shift, for which human players also might face performance changes.

(ii) **No Barrel *DonkeyKong*.** This version of *DonkeyKong*, we removed the barrels, making it easier to get to Peach. We use the *no_barrel* modification here.

(iii) **Aligned Cars *Freeway*** This variant aligned the cars so that each car has the same speed. The first 5 cars are driving from left to right in one line. The last 5 from right to left. We used the modification *stop2* for this game variant.

(iv) **Mono-colored *Freeway*.** In *Freeway*, the agent controls a chicken targeted with crossing a road and is rewarded only after reaching the other side of the road. At every contact with an incoming car, the agent is pushed back down, making the probability of crossing (while playing randomly) quite low. *Freeway*'s rewarding system can thus be considered sparse. To test for generalization and robustness, we provide multiple color variations, where all cars are colored with the same color using the *color* modification. In Table 2 we used *color1*, resulting in black.

(v) **Stopped Cars Freeway.** While *Aligned Cars Freeway* aligns the cars to move with the same speed, this version stops the cars to make crossing the street even easier. The modification for this variant is *stop3* and can be seen in Figure 4.

(vi) **No Danger *Frostbite*.** In *Frostbite* there are birds pushing you into the water. To remove this danger and enable skill learning, we used the modification *enemies1*.

(vii) **Static Ice *Frostbite*.** Another way to loose the game is to jump into the water when missing a floating ice shelf. To make the game a little easier, we use the modification *static60* to fixate the ice shelves.

(viii) **No Danger *Kangaroo*.** In Kangaroo the agent has to learn to deal with monkeys and coconuts that are thrown by the monkeys or are falling from the top floor. In this variant, we removed these elements by using the modifications *disable_monkeys* and *disable_coconut*.

(ix) **Swap Level *MsPacman*** MsPacman has multiple levels to play, however, most agents only train on the first level. The difference between the levels primarily changes the layout of the maze. This game variant does now really change the gameplay but lets the agents train or test their ability on another level. We used the *change_level1* modification for this.

(x) **Lazy Enemy *Pong*.** Also explained already in Section 3, this variant let the enemy stop after hitting the ball and only allows movement again after the player hits the ball with its paddle. The used modification is *lazy_enemy*.

(xi) **Infinite Oxygen Seaquest** This version has a similar goal as No Enemy Seaquest by removing the necessity to get to the surface to refill oxygen. The player can concentrate on the other tasks. We used the modification *unlimited_oxygen*.

(xii) **No Shields *SpaceInvaders*.** In SpaceInvaders the user has 3 shields to hide behind. These shields can be removed, using the modification *disable_shields*.

B.2 Additional Game Variants

In addition to the variants presented in the paper, we also conducted experiments on the following game variations to show the variety of possible environments, that can be created using HackAtari. This is by no means an exhaustive list and more variants can be created using the modifications, described in Appendix E.

(xiii) **Drunken Boxing.** An alternative to One-Armed Boxing is our Drunken Boxing version. In this variant the player is moved into a random position at each timestep, making the move actions more challenging. For this, we used the *drunken_boxing* modification.

(xiv) **Gravity Breakout** In Breakout you have to hit the ball before it reaches the bottom of the screen. To make the game more challenging, we added gravity, pulling the ball downwards using an artificial strength. We used the modification *strength1* and *gravity*.

(xv) **Easy FishingDerby** In Fishing Derby the player has to dodge the sharks while catching the fish. In this variant we stopped the sharks from moving and enhanced the amount of fish on the players side. We used the modifications *fish_mode0* and *shark_mode0*.

(xvi) **No Ghosts MsPacman.** MsPacman has four ghosts trying to catch the player. When catched the player looses one of its lives. In this version of the game, we cage the ghosts to their square in the middle of the maze using *caged_ghosts*. This version enables the agent to ignore the ghosts completely.

(xvii) **No Fuel Riverraid** This versions removes the necessity to collect fuel within the game. The player always have a full tank. The modification is *no_fuel*.

(xviii) **No Enemy Seaquest** In Seaquest the player has multiple tasks, like saving divers, always have oxygen in the tank and shooting as well as dodging the enemies. In this version we removed the enemies with *disable_enemies*.

(xix) **Relocated Shields SpaceInvaders** Instead of removing the shields completely, HackAtari also enables us to relocate them. In this version the shields behave like in the original game but are moves aside slightly. We used *relocate40*.

Similar to Figure 4, Figure 9 displays the training process of a PPO agent in these additional environments as well as environments from App.B.1. It can be seen that PPO agents are able to learn in all of them, except *No Enemy Seaquest*. A reason for the latter can be seen in their sparsity regaring rewards. Without enemies to shoot, the only way to gain rewards is to save 6 divers. This is similar Kangaroo in Table 2.

B.3 Extended Experimental Setup

In Table 2, we compared agents trained in the original environment against their performance in some of HackAtari's new variants. The performance of a similar agent, trained in the game variation instead of the original environment, is also added to show that higher scores are reachable. The performances were measured over a span of 30 episodes. Next to ending an episode naturally by winning or failing the game, we also add a maximum number of 100000 frames after which the episode is truncated as well as a maximum reward of 50000 after which the game is also truncated. As error we displayed the standard deviation over all episodes.

Fig. 4 to 7 display training runs over 10M frames/steps. The exact hyperparameters can be found in App. C.1. As rewards we used the internal ones given by the Atari games and for Fig. 5 added our own reward function as additional axis. To mitigate noise and fluctuations, we use exponential moving average (EMA) smoothing over the mean of all seeds. We use an effective window size of 50, resulting in a smoothing factor $\alpha = 2/(1+50) \approx 0.039$ used in the following formula:

$$EMA_t = (1-\alpha) \cdot EMA_{t-1} + \alpha \cdot y_t \,. \tag{1}$$

To manage irregular training intervals due to rewards are not always being reported in the same timestep, we ignore missing values when computing the average, relying on the EMA smoothing to provide a continuous curve. For the error bands we used the standard deviation of your data within a rolling window over all seeds.

B.4 Evaluating Human Performance on HackAtari Environments

All experiments conducted in this study that involve evaluating human performance strictly adhere to ethical guidelines. We place the utmost importance on the anonymity and confidentiality of all participants. Data collected during the study is anonymized, ensuring that no personal information can be linked back to the participants.

The goal of this small study was to investigate, how humans react to unknown changes in familiar Atari games and the impact of these changes on game outcomes. The data collected, including game interaction data and

Figure 9: **Addtional environments RL agents can learn in.** PPO agents are able to progressively improve in these new game variants over 10M training steps.

results, will be used solely for scientific purposes. We are committed to conducting this research with the highest ethical standards, respecting the privacy and rights of all participants.

Participants were asked to make themselves familiar with a selected set of games by playing them until they felt comfortable in them. Games were chosen randomly per participant. Then we evaluated their performance in each of the original environments for 5 minutes before switching to the HackAtari variant of that specific game. We also set a cap in reward to 50000 in games like Kangaroo or DonkeyKong where our game variant removed the natural cause of death and truncated the game if a player reaches this reward. Note that participants have not seen or played the variant before and had no prior training time in it, compared to the original environment.

The results in Table 2 show the mean over all participants. All participants were informed about the study before taking part in it. This includes potential risks, usage of information recorded (*i.e.* the performance), the tasks within the study. The were also informed that withdrawal was possible at all times, resulting in the

removal of all personal information (*i.e.* the performance scores recorded until withdrawal). All participants had to give their consent before taking part. There was no compensation for participation.

The study was taken under the guidelines of the institutional ethics committee.

C Reproducing Training and Train new Models

Hackatari provides a robust environment for training reinforcement learning agents on a wide range of game modifications. By keeping it modular and near the original gymnasium environment, we emphasize leveraging frameworks like Stable Baselines and CleanRL, to develop and fine-tune agents to excel in customized game scenarios. This section will walk you through the steps to set up training processes and adapt to various game dynamics using Hackatari. We emphasize the importance of reproducibility, ensuring that experiments can be reliably replicated and validated. For this, we will also provide any needed information of our specific training, starting with the hyperparameters.

C.1 Hyperparameter Configuration for Training

In this section, we provide a detailed overview of the hyperparameters employed during the training and optimization of our models. Hyperparameters play a pivotal role in determining the performance and generalization ability of machine learning models. In our experiments, we followed the parameter set by Huang et al. (2022) for both our C51 and PPO agents. We do not provide any grid search or hyperparameter optimization.

C.2 Hardware Specification and Computational Costs

The experiments were run on a setup, described in Table 5, using the NVIDIA GPU Cloud (NGC) docker container for pytorch[3]. As stated before, all needed data is openly available.

For the PPO agents, the main load of training these networks was done on the CPU. While training the process did 400 to 600 steps per second (SPS), resulting in a overall runtime of 5-7h per agent per seed. Since we were running the experiments on a shared server, this of course was highly depending on the general load of the machine. For this paper we trained round about 70-100 agents (including failed runs). C51 agents were trained using the GPUs to accelerate. When training an agent on one of the V100 GPUs, training took about 10-12h. Since C51 were only used as an alternative to PPO for some experiments, we only trained about 15 agents. Overall this results in about 750 hours of training time.

C.3 Training and Evaluating Agents using HackAtari

Node that HackAtari alone is more a wrapper for gymnasium which can then be included in training environments like Stable-baselines or CleanRL.

To reproduce our training, one needs to replace the gymnasium in CleanRL with HackAtari. We provide[4] a fork already doing this for C51 and PPO. (See *ppo_atari.py* and *c51_atari.py*). The training and evaluation process follows CleanRL, using its codebase for tracking and logging results and models.

Examples on how our agents are trained and evaluated, can be found in the scripts folder, together with a ReadMe about how to train your own agents and a small installation guide in form of a bash script. For this it is neccessary to first download the HackAtari repository[5] and add it to the subfolder.

The codebase for training will be released together with the HackAtari repository.

D Replacing the Reward Function

In this section, we demonstrate the process of replacing the reward function in HackAtari.

[3] https://catalog.ngc.nvidia.com/orgs/nvidia/containers/pytorch, accessed 2024-05-22
[4] https://github.com/BluemlJ/oc_cleanrl
[5] https://github.com/k4ntz/HackAtari

By customizing the reward function, you can tailor the reinforcement learning environment to better suit your research objectives and explore new dimensions of agent behavior, e.g., like being a pacifistic agent. We'll cover the steps needed to implement your own reward structure, ensuring you can effectively modify the learning incentives within the game.

Example: Skiing. In the Atari game "Skiing," the reward function is based on the player's score, which is determined by the time taken to complete the course and the penalties incurred for missing gates. Here is a more detailed breakdown of how the reward is typically calculated:

- **Time-Based Scoring** The primary goal in Skiing is to navigate through a series of gates as quickly as possible. The faster the player completes the course, the higher the score. The game keeps track of the time taken to finish the course. A lower time translates into a better score.
- **Gate Penalties** Players must pass through gates correctly to avoid penalties. Each missed gate results in a penalty, which usually adds extra time to the player's overall time. These penalties decrease the score since the total time increases.

However, the result is giving to the agent as a combined score after each episode making it hard to learn that game. The result a non-functional agent, not able to play the game at all. To enable agents to learn the game however, we propose to change the reward function over the training process, starting with simple skills like skiing downwards before adding poles to it.

To use an own reward function, one can provide the path to a valid python file including a function called *reward_function*.

```python
LAST_SCORE=32 # save the RAM value from the last turn

def reward_function(self) -> float:
    global LAST_SCORE
    score = self.get_ram()[107] #nr. of gates successfully passed
    if score != LAST_SCORE:
        reward = 100
    else:
        reward = 0
    LAST_SCORE = score
    return reward
```

The example above shows a simple reward function only rewarding passing successfully through gates and sets the reward to 0 for every other turn. The number of successfully passed gates is saved as a decreasing number from 32 to 0 in the RAM at coordinate 107. The complexity of the new reward function depends only on the user and can also be created using LLMs or game objects.

Generating a reward function with a LLM in Seaquest. The following example of a reward function, for the game of Seaquest, was created using ChatGPT 3.5 and the results can be seen in Figure 5. For the input to the LLM, we used game objects, object properties and a short description of the game. The object-centric environment context is given by the classes provided by the OCAtari framework (Delfosse et al., 2023a), i.e., the parent game object class[6] and the game-specific objects [7]. The game objects related to the score were omitted. The game description is taking from the Atari description[8], the gymnasium website[9] and some shortening from us. For the game of Seaquest the description is the following

```
You a sub (Player) able to move in all directions and fire torpedoes.
The goal is to retrieve as many divers as you can, while dodging and blasting enemy subs
and killer sharks.
```

[6] https://github.com/k4ntz/OC_Atari/blob/master/ocatari/ram/game_objects.py
[7] e.g., https://github.com/k4ntz/OC_Atari/blob/master/ocatari/ram/pong.py
[8] https://www.retrogames.cz/play_221-Atari2600.php
[9] https://gymnasium.farama.org/environments/atari/seaquest/

```
The game begins with one sub and three waiting on the horizon. Each time you increase your score by 10,000 points, an extra sub will be delivered to your base.
Your sub will explode if it collides with anything except your divers.The sub has a limited amount of oxygen that decreases at a constant rate during the game. When the oxygen tank is almost empty, you need to surface and if you don't do it in time, your sub will blow up and you'll lose one diver.
Each time you're forced to surface, with less than six divers, you lose one diver as well.
```

The object-centric reward function created by the LLM for Seaquest:

```python
from ocatari.ram.seaquest import *

def check_collision(obj1, obj2):
    """
    Check if two GameObjects collide based on their bounding boxes.
    """
    return (obj1.x < obj2.x + obj2.w and
            obj1.x + obj1.w > obj2.x and
            obj1.y < obj2.y + obj2.h and
            obj1.y + obj1.h > obj2.y)

def update_game_state(objects):
    """
    Update positions of all game objects and check for collisions.
    """
    collisions = []
    for obj in objects:
        # Update position based on velocity
        obj.xy = (obj.x + obj.dx, obj.y + obj.dy)

        # Check for collisions with other objects
        for other in objects:
            if obj != other and check_collision(obj, other):
                collisions.append((obj, other))
    return collisions

def manage_oxygen_and_lives(player, oxygen_bar, lives):
    """
    Decrease oxygen levels and manage lives based on oxygen and collisions.
    """
    # Decrease oxygen
    oxygen_bar.value -= 1
    if oxygen_bar.value <= 0:
        player.lives -= 1
        oxygen_bar.value = 100  # Reset oxygen after surfacing or losing a life

    # Check if lives are depleted
    if lives.value <= 0:
        print("Game Over")

def update_score_and_divers(player, divers_collected, score):
    """
    Update score based on collected divers and manage divers.
    """
    for diver in divers_collected:
        if check_collision(player, diver):
            score.value += 1000  # Increment score for each diver collected
            divers_collected.remove(diver)  # Remove diver from the game

def fire_torpedo(player, torpedoes):
    """
    Create a new torpedo at the player's location and add it to the torpedoes list.
    """
    new_torpedo = PlayerMissile()
    new_torpedo.xy = player.xy
```

```
    torpedoes.append(new_torpedo)


def reward_function(self) -> float:
    game_objects = self.game_objects()
    reward = 0.0

    # Define categories for easy identification
    player = None
    divers = []
    enemies = []
    player_missiles = []
    enemy_missiles = []
    oxygen_bar = None

    # Classify objects
    for obj in game_objects:
        if isinstance(obj, Player):
            player = obj
        elif isinstance(obj, Diver):
            divers.append(obj)
        elif isinstance(obj, Shark) or isinstance(obj, Submarine):
            enemies.append(obj)
        elif isinstance(obj, PlayerMissile):
            player_missiles.append(obj)
        elif isinstance(obj, EnemyMissile):
            enemy_missiles.append(obj)
        elif isinstance(obj, OxygenBar):
            oxygen_bar = obj

    # Check for collisions and manage interactions
    if player:
        for diver in divers:
            if check_collision(player, diver):
                reward += 0.1  # Scaled down reward for collecting a diver
                divers.remove(diver)  # Assume diver is collected and removed from the
                ↪  game

        for enemy in enemies:
            if check_collision(player, enemy):
                reward -= 0.1  # Scaled down penalty for colliding with an enemy

        for missile in enemy_missiles:
            if check_collision(player, missile):
                reward -= 0.05  # Scaled down penalty for getting hit by an enemy missile

        # Reward for hitting enemies with missiles
        for missile in player_missiles:
            for enemy in enemies:
                if check_collision(missile, enemy):
                    reward += 0.05  # Scaled down reward for destroying an enemy
                    enemies.remove(enemy)  # Assume enemy is destroyed and removed from
                    ↪  the game
                    player_missiles.remove(missile)  # Remove missile after hitting

    # Manage oxygen levels
    if oxygen_bar and oxygen_bar.value <= 20:
        reward -= 0.05  # Scaled down penalty for low oxygen levels

    # Encourage surfacing if oxygen is too low
    if oxygen_bar and oxygen_bar.value <= 10:
        reward -= 0.1  # Scaled down higher penalty for critically low oxygen

    return reward
```

Table 2: **Deep RL agents' performances drop when confronted with slightly novel situations.** PPO agents scores, trained and/or evaluated the original and variation of different Atari games. Gameplay or color changes mostly lead to performance drop, whereas most human scores increase.

Game	**PPO**			**Human**		**Random**
Training **Testing**	original original	original variation	variation variation	original original	original variation	- variation
Boxing (OA)	90.9±1.5	1.9±10.2	82.2±9.3	0.6±2.7	-12.8±18.8	-10.8±0.9
DonkeyKong (NB)	3480±1032	0±0	0±0	7320±3961	50000±0	0±0
Freeway (AC)	31.4±1.5	20.4±0.7	29.1±1.8	21.7±4.8	22.4±1.6	0±0
Freeway (MC)	31.4±1.5	24.6±2.7	32.7±0.8	21.7±4.8	29.3±1.5	0±0
Frostbite (SI)	313±13.1	265±25.4	991±390	4916±3278	29360±19120	59.4±43
Kangaroo (ND)	1838±650	0±0	0±0	2344±1434	12200±1555	0±0
MsPacman (SL)	2312±465	456±260	2228±428	4592±3725	6149±5097	135±65
Pong (LE)	16.0±3.4	-12.6±2.4	18.1±4.4	-13.7±2.3	-12.2±6.4	-20.1±0.4
SpaceInv. (NS)	724±123	496±78	1181±292	640±368	726±616	109±32

Table 3: **Hyperparameter Configuration for Experimental Settings (PPO).** This table provides a comprehensive overview of the essential hyperparameters utilised in our experimental section.

Hyperparameter	**Value**	**Hyperparameter**	**Value**
batch size	1024	Clipping Coef.	0.1
γ	0.99	KL target	None
minibatch size	256	GAE lambda	0.95
seeds	42,73,91	input representation	4x84x84
total timesteps	10M	gym version	0.28.1
learning rate	0.00025	pytorch version	1.12.1
more information	https://docs.cleanrl.dev/rl-algorithms/ppo/		

Table 4: **Hyperparameter Configuration for Experimental Settings (C51).** This table provides a comprehensive overview of the essential hyperparameters utilized in our experimental section.

Hyperparameter	**Value**	**Hyperparameter**	**Value**
batch size	32	optimizer	Adam
buffer size	100k	loss	cross-entropy loss
γ	0.99	input representation	4x84x84
seeds	42,73,91	gym version	0.28.1
total time steps	10M	pytorch version	1.12.1
learning rate	0.00025		
more information	https://docs.cleanrl.dev/rl-algorithms/c51/		

Table 5: Hardware configuration for our experimental section. (NVIDIA DGX-2 Working station)

Hardware	**Description**
CPU	Intel(R) Xeon(R) Platinum 8174 CPU @ 3.10GHz
GPU	16 × NVIDIA® Tesla V100
Memory	1.5 TB 2,133 MHz DDR4 RDIMM
Operating System	Ubuntu 20.04 LTS

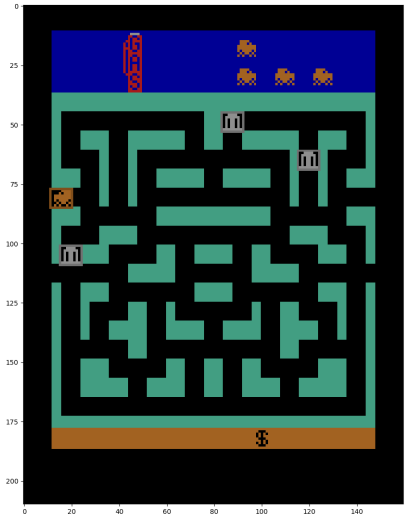
E Modifications

This chapter delves into the extensive range of modifications HackAtari offers, providing users with a versatile toolkit to customize and enhance their reinforcement learning experiments. From changing the game dynamics and creating new challenges to integrating novel algorithms, we explore the myriad ways HackAtari can be adapted to meet diverse research needs.

E.1 Game-specific Modifications

E.1.1 BankHeist details

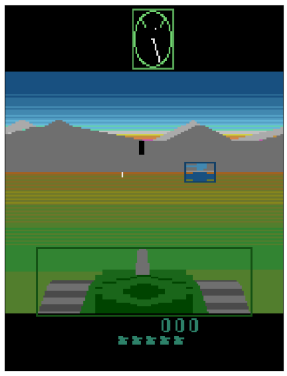
You are a bank robber and (naturally) want to rob as many banks as possible. You control your getaway car and must navigate maze-like cities. The police chases you and will appear whenever you rob a bank. You may destroy police cars by dropping sticks of dynamite. You can fill up your gas tank by entering a new city.At the beginning of the game you have four lives. Lives are lost if you run out of gas, are caught by the police,or run over the dynamite you have previously dropped.



Modification	Effect
unlimited_gas	Unlimited gas all the enemies.
no_police	Removes police from the game.
only_police	No banks only police.
random_city	Randomizes which city is entered next.
revisit_city	Allows player to go back one city.

E.1.2 BattleZone details

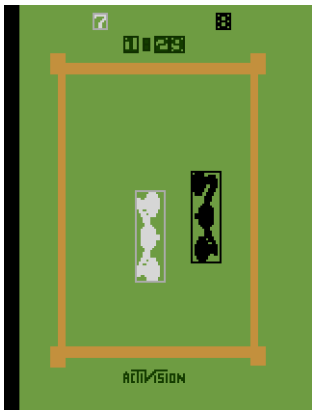
You control a tank and must destroy enemy vehicles. This game is played in a first-person perspective and creates a 3D illusion. A radar screen shows enemies around you. You start with 5 lives and gain up to 2 extra lives if you reach a sufficient score.



Modification	Effect
no_radar	Removes the radar content.

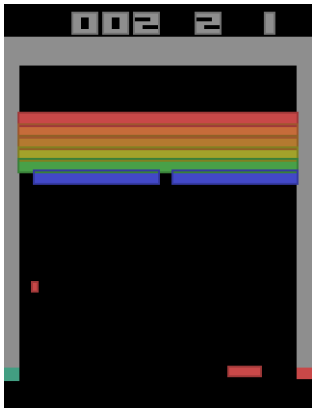
E.1.3 Boxing details

You fight an opponent in a boxing ring. You score points for hitting the opponent. If you score 100 points, your opponent is knocked out.



Modification	Parameter	Effect
gravity$X$	$X \in (1, 5)$	Add a permanent downwards movement with strength X.
one_armed		Disables the "hitting motion" with the right arm permanently
drunken_boxing		Applies random movements to the players input.
color_pX	$X \in (0, 4)$	Changes the color of the player to [Black, White, Red, Blue, Green] by choosing a value 0-4
color_eX	$X \in (0, 4)$	Changes the color of the enemy to [Black, White, Red, Blue, Green] by choosing a value 0-4
switch_p		Switches the position of player and enemy

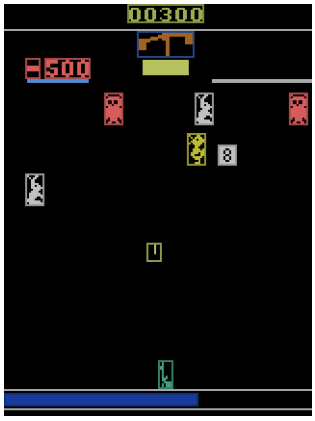
E.1.4 Breakout details



Another famous Atari game. The dynamics are similar to pong: You move a paddle and hit the ball in a brick wall at the top of the screen. Your goal is to destroy the brick wall. You can try to break through the wall and let the ball wreak havoc on the other side, all on its own! You have five lives.

Modification	**Parameter**	**Effect**
strength$X$	X	Set strength of drift to X.
drift$X$	$X \in \{r, l\}$	Set drift direction to left or right.
gravity		Pull the ball down by changing the corresponding ram positions
inverse_gravity		Pushes the ball up by changing the corresponding ram positions
color_pX	$X \in (0, 4)$	Changes the color of the player to [Black, White, Red, Blue, Green] by choosing a value 0-4
color_bX	$X \in (0, 4)$	Changes the color of all blocks to [Black, White, Red, Blue, Green] by choosing a value 0-4
color_rXY	$X, Y \in (0, 4)$	Changes the color of row X to color Y [Black, White, Red, Blue, Green] by choosing a value 0-4 for both row and color

E.1.5 Carnival details

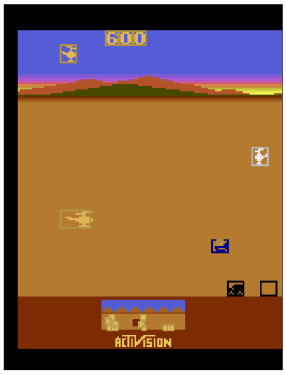


This is a “shoot ‘em up” game. Targets move horizontally across the screen and you must shoot them. You are in control of a gun that can be moved horizontally. The supply of ammunition is limited and chickens may steal some bullets from you if you don’t hit them in time.

Modification	Parameter	Effect
no_flying_ducks		Ducks in the last row disappear instead of turning into flying ducks.
unlimited_ammo		Ammunition doesn't decrease.
fast_missiles$X$	$X \in (1, 3)$	The projectiles fired from the players are faster. Uses the values 1-3 to determine how by how much to speed the missile up.

E.1.6 ChopperCommand details

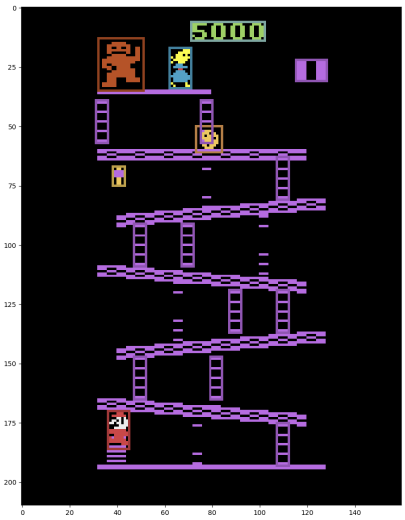
You control a helicopter and must protect truck convoys. To that end, you need to shoot down enemy aircraft.A mini-map is displayed at the bottom of the screen.



Modification	Parameter	Effect
delay_shots		Puts time delay between shots
no_enemies		Removes all Enemies from the game
no_radar		Removes the radar content
invis_player		Makes the player invisible
color$X$	$X \in (0, 4)$	Changes the color of background to [Black, White, Red, Blue, Green] by choosing a value 0-4. This also affects the enemies colors

E.1.7 DonkeyKong details

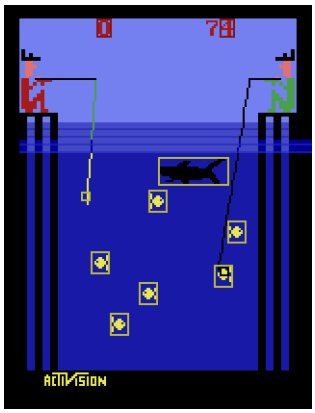
You play as Mario trying to save your girlfriend who has been kidnapped by Donkey Kong. Remove rivets and jump over fireballs, with a score that starts high and counts down throughout the game.



Modification	Effect
no_barrel	Remove barrels from the game
random_start	Set the start position to a random pre-defined start position

E.1.8 FishingDerby details

Your objective is to catch more sunfish than your opponent.



Modification	Parameter	Effect
fish_mode$X$	$X \in \{0, 1, 3\}$	Allows for alterations in the behavior of the shark as specified
shark_mode$X$	$X \in (0, 3)$	Allows for alterations in the behavior of the fish as specified

E.1.9 Freeway details

Your objective is to guide your chicken across lane after lane of busy rush hour traffic. You receive a point for every chicken that makes it to the top of the screen after crossing all the lanes of traffic.

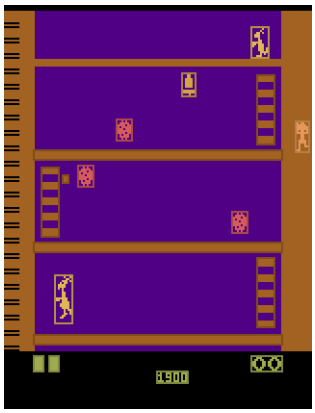
Modification	Parameter	Effect
stop$X$	$X \in (1, 3)$	Manipulate the move pattern of the car or let them stop.
color$X$	$X \in (0, 8)$	Set color of cars to one of 9 pre-defined colors

E.1.10 Frostbite details

In Frostbite, the player controls "Frostbite Bailey" who hops back and forth across across an Arctic river, changing the color of the ice blocks from white to blue. Each time he does so, a block is added to his igloo.

Modification	Parameter	Effect
color$X$	$X \in (0, 3)$	Adjusts the colors of the ice floes bases on the specified values.
line$X$	$X \in (1, 5)$	Specify row/line for color change
ui_color$X$	$X \in (0, 3)$	Adjust color of the UI
enemies$X$	$X \in (0, 3)$	Adjusts the memory based on the specified number of enemies selected by the user.
floesXXX	$X \in (0, 160)$	Adjusts the memory based on the specified new position of the ice floes.
staticXXX	$X \in (0, 160)$	Adjusts the memory based on the specified new position of the ice floes and hold it in place for the game.

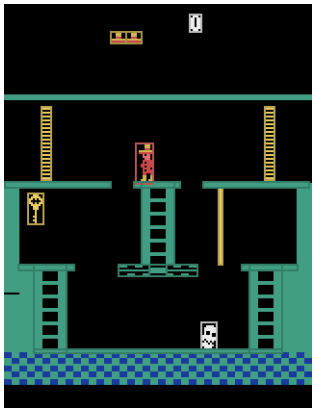
E.1.11 Kangaroo details



The object of the game is to score as many points as you can while controlling Mother Kangaroo to rescue her precious baby. You start the game with three lives.During this rescue mission, Mother Kangaroo encounters many obstacles. You need to help her climb ladders, pick bonus fruit, and throw punches at monkeys.

Modification	Parameter	Effect
disable_monkeys		Disables the monkeys in the game
disable_coconut		Disables the coconuts in the game
random_init		Randomize the floor on which the player starts.
set_floor$X$	$X \in (0, 2)$	Set the floor on which the player starts.
change_level$X$	$X \in (0, 2)$	Changes the level according to the argument number 0-2. *change_level*, selects random level.

E.1.12 MontezumaRevenge details

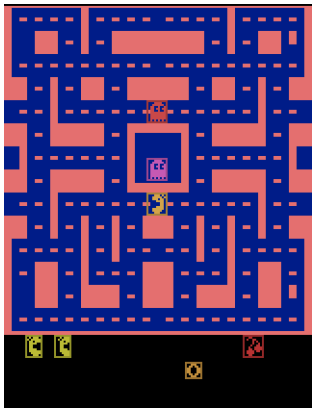


Your goal is to acquire Montezuma's treasure by making your way through a maze of chambers within the emperor's fortress. You must avoid deadly creatures while collecting valuables and tools which can help you escape with the treasure.

Modification	Parameter	Effect
random_position_start		Randomize the start position within the room
level$X$	$X \in (0, 9)$	Changes the level to a more difficult version. Level 0, 1, 2 are different versions, afterwards level determines map layout.
randomize_items		Randomize which item is found in which room.
full_inventory		Adds all items to inventory.

E.1.13 MsPacman details

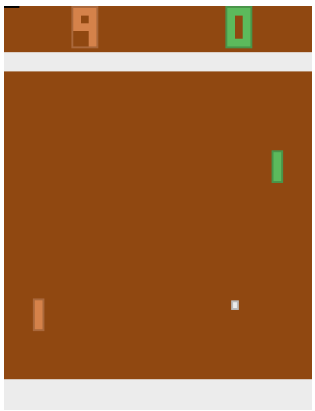
Your goal is to collect all of the pellets on the screen while avoiding the ghosts.



Modification	Parameter	Effect
caged_ghosts		Fix the position of the ghost inside the square in the middle of the screen.
disable_orange		Fix the position of the orange ghost only.
disable_red		Fix the position of the red ghost only.
disable_cyan		Fix the position of the cyan ghost only.
disable_pink		Fix the position of the pink ghost only.
power$X$	$X \in (0, 4)$	Switching the specified number of power pills with normal edible tokens.
edible_ghosts		All ghost will be made edible the entire game
inverted		All ghost will be edible the entire game until the player eats a power pill. After eating a power pill the ghost will return to "normal" for a certain amount of time
change_level$X$	$X \in (0, 3)$	Changes the level according to the argument number 0-3.

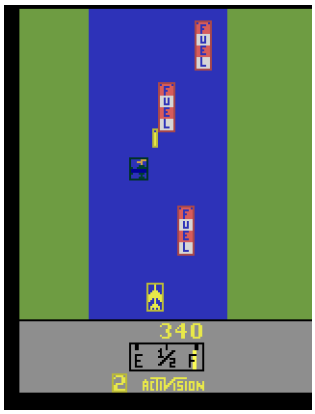
E.1.14 Pong details

You control the right paddle, you compete against the left paddle controlled by the computer. You each try to keep deflecting the ball away from your goal and into your opponent's goal.



Modification	Parameter	Effect
lazy_enemy		Enemy does not move after returning the shot until player hits ball.
up_drift$X$	$X \in (1, 5)$	Makes the ball drift upwards by changing the corresponding ram positions
down_drift$X$	$X \in (1, 5)$	Makes the ball drift down by changing the corresponding ram positions
left_drift$X$	$X \in (1, 5)$	Makes the ball drift to the left by changing the corresponding ram positions
right_drift$X$	$X \in (1, 5)$	Makes the ball drift to the right by changing the corresponding ram positions

E.1.15 RiverRaid details



You control a jet that flies over a river: you can move it sideways and fire missiles to destroy enemy objects. Each time an enemy object is destroyed you score points (i.e. rewards). You lose a jet when you run out of fuel: fly over a fuel depot when you begin to run low.You lose a jet even when it collides with the river bank or one of the enemy objects (except fuel depots). The game begins with a squadron of three jets in reserve and you're given an additional jet (up to 9) for each 10,000 points you score.

Modification	Effect
no_fuel	Removes the fuel deposits from the game.

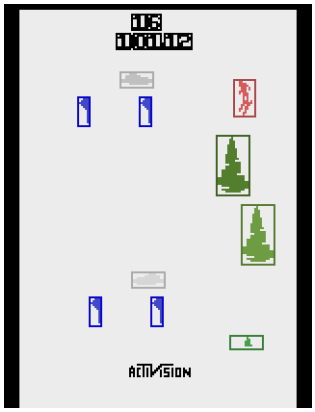
E.1.16 Seaquest details

You control a sub able to move in all directions and fire torpedoes. The goal is to retrieve as many divers as you can, while dodging and blasting enemy subs and killer sharks; points will be awarded accordingly. The game begins with one sub and three waiting on the horizon. Each time you increase your score by 10,000 points, an extra sub will be delivered to your base. You can only have six reserve subs on the screen at one time.Your sub will explode if it collides with anything except your own divers.The sub has a limited amount of oxygen that decreases at a constant rate during the game. When the oxygen tank is almost empty, you need to surface and if you don't do it in time, your sub will blow up and you'll lose one diver. Each time you're forced to surface, with less than six divers, you lose one diver as well.

Modification	Effect
unlimited_oxygen	Changes the behavior of the oxygen bar to remain filled
gravity	Enables gravity for the player.
disable_enemies	Disables all the enemies.
random_color_enemies	The enemies have new random colors each time they go across the screen.

E.1.17 Skiing details

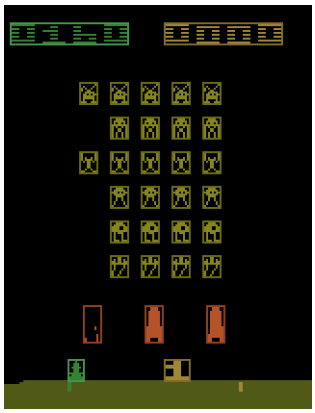
You control a skier who can move sideways. The goal is to run through all gates (between the poles) in the fastest time. You are penalized five seconds for each gate you miss. If you hit a gate or a tree, your skier will jump back up and keep going.



Modification	Effect
invert_flags	Switches the flag color from blue to red

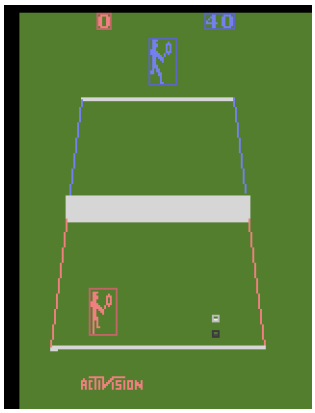
E.1.18 SpaceInvaders details

Your objective is to destroy the space invaders by shooting your laser cannon at them before they reach the Earth. The game ends when all your lives are lost after taking enemy fire, or when they reach the earth.



Modification	Parameter	Effect
disable_shield_left		Disables the left shield.
disable_shield_middle		Disables the middle shield.
disable_shield_right		Disables the right shield.
disable_shields		Disables all shields.
curved		Makes the shots travel on a curved path.
relocateXX	$X \in (35, 53)$	Allows for the relocation of the shields via an offset.

E.1.19 Tennis details



You control the orange player playing against a computer-controlled blue player. The game follows the rules of tennis. The first player to win at least 6 games with a margin of at least two games wins the match. If the score is tied at 6-6, the first player to go 2 games up wins the match.

Modification	Effect
wind	Sets the ball in the up and right direction by 3 pixles every single ram step to simulate the effect of wind
upper_pitches	Changes the ram so that it is always the upper persons turn to pitch.
lower_pitches	Changes the ram so that it is always the lower persons turn to pitch.
upper_player	Changes the ram so that the player is always in the upper field
lower_player	Changes the ram so that the player is always in the lower field