`PyDSMC`: Statistical Model Checking for Neural Agents Using the Gymnasium Interface

Timo P. Gros[1,2], Arnd Hartmanns[3], Ivo Hoese[2], Joshua Meyer[1,2], Nicola J. Müller[1,2], and Verena Wolf[1,2]

[1] German Research Center for Artificial Intelligence (DFKI), Saarbrücken, Germany
`{timo.gros, ivo.hoese, joshua.meyer, nicola.mueller, verena.wolf}@dfki.de`
[2] Saarland University, Saarland Informatics Campus, Saarbrücken, Germany
[3] University of Twente, Enschede, The Netherlands
`a.hartmanns@utwente.nl`

Abstract. Artificial intelligence (AI) has achieved remarkable success in sequential decision-making. However, evaluating its neural agents remains challenging, as current methods often rely on interpreting training curves only, overlooking key statistical factors. Existing tools that allow a formal evaluation also require white-box formal models, making them impractical for most AI benchmarks based on the black-box Gymnasium interface. We introduce `PyDSMC`, a lightweight and easy-to-use Python tool for statistical model checking of neural agents on arbitrary Gymnasium environments. `PyDSMC` automates the selection of statistical methods to compute confidence intervals, supporting both convergence-based and resource-limited evaluation settings. We empirically demonstrate the importance of rigorous agent evaluation and showcase `PyDSMC`'s capabilities to more reliably judge and report an AI agent's performance.

1 Introduction

Artificial intelligence (AI) exerts a significant impact on both everyday life and contemporary research. Deep learning, in particular, is increasingly utilized for *sequential decision-making* (SDM) where it achieved considerable success in areas such as the training of large language models [1,17], improving the prediction of protein foldings [61], or mastering complex computer games such as StarCraft [68] or Dota 2 [12]. We call such AI-based decision-makers *neural agents.*

Despite these great achievements, a critical gap remains in accurately measuring the performance of neural agents. Even landmark papers introducing state-of-the-art algorithms like DQN [52], PPO [60], SAC [41], RND [22], and DreamerV2 [42] typically assess agent performance only through training curves, i.e., the accumulated sum of rewards/scores over training time. These assessments typically fail to account for important influences, such as the system's variance, the limited number of samples used for each score in the training curve, or the effects of exploration strategies [35,36]. This calls for a more formal approach.

In recent years, the gap between the AI and verification communities was bridged by tools that adapt and extend verification methods to neural agents. Notably, COOL-MC [40] and MoGym [34] provide ways to apply probabilistic [6, 7] and statistical model checking (SMC) [3,49,69], respectively, to evaluate the behavior of neural agents. These tools leverage the capabilities of established model checkers—Storm [28], and modes [19] of the Modest Toolset [43], respectively—to enhance the rigor of evaluations.

However, they crucially require a formal model of the environment, encoded in modeling languages such as JANI [20] or PRISM [48], while the benchmarks most commonly used for SDM in the AI context follow the Farama Foundation's Gymnasium [65] interface (the successor to OpenAI Gym [15]). COOL-MC and MoGym implement this interface for formal models, giving Gymnasium-based tools access to formal models, but not the reverse. Thus, formal evaluation methods remain inaccessible to the majority of AI community benchmarks—e.g., MuJoCo [63], Procgen [26], or Atari 2600 [10]—that are based on arbitrary (black-box) simulations.

With this paper, we aim to bridge this AI-verification gap from the other side: We introduce `PyDSMC` (publicly available at github.com/neuro-mechanistic-modeling/PyDSMC), a lightweight, easy-to-install, and easy-to-use tool that facilitates SMC of neural agents (in prior work called *Deep Statistical Model Checking* (DSMC) [35]) across any environment conforming to the Gymnasium interface, irrespective of the underlying implementation. `PyDSMC` provides predefined trajectory-based properties to evaluate, such as accumulated rewards, the number of steps until termination, or the goal-reaching probability. In addition, a simple interface allows users to define custom properties. `PyDSMC` computes confidence intervals to either (1) achieve a predefined error margin and level of confidence, or (2) report the error margin given a confidence level once a specified resource limit (runtime or number of samples) is reached. For every property, `PyDSMC` automatically selects the appropriate statistical method. Thus, it provides accurate and adaptable statistical verification capabilities.

`PyDSMC` is designed for ease of use and compatibility with a broad range of environments using the Gymnasium interface. Notably, using SMC as the underlying analysis technique allows `PyDSMC` to remain agnostic w.r.t. the environment's underlying implementation. We believe that these features will facilitate greater adoption of SMC within the AI community and improve the state of the art in how the performance of neural agents is measured and reported.

Related Work. COOL-MC and MoGym make AI algorithms available for use with formal models—whereas `PyDSMC` makes formal evaluation available for use with AI agents. A similar goal is achieved by MultiVeStA for economic agent-based models [66,67]. The area of neural network verification (see, e.g., [4,9,27, 54,59]) is a wide field that includes constraint- and abstraction-based symbolic and explicit verification methods and tools, typically aimed at ensuring the correctness or safety [5,39,47] of a neural agent w.r.t. a specification. In contrast, `PyDSMC`'s purpose is specifically to provide a toolbox for the AI practitioner to easily evaluate the performance of their agent in a sound, formally justified way.

Outline. We elaborate the motivation to use SMC for neural agents in Section 2. Then Sections 3 and 4 introduce Gymnasium and the statistical methods used in our work, respectively. In Section 5 we present the PyDSMC Python package and demonstrate some exemplary results in Section 6.

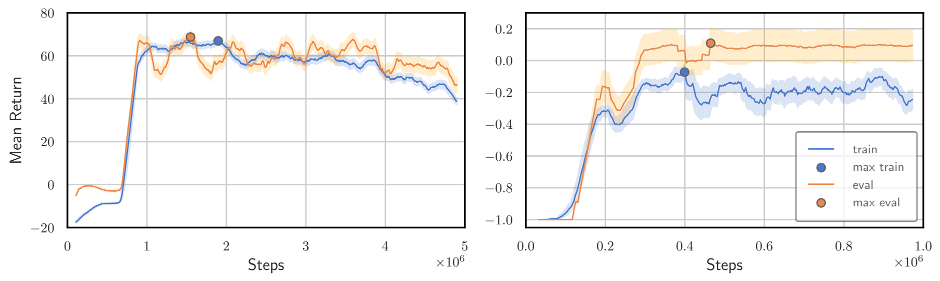
2 The Importance of Verifying Neural Agents

This section highlights the critical role of verification in assessing the performance of neural agents effectively.

2.1 Current State of the Art of Reporting Agents' Performances

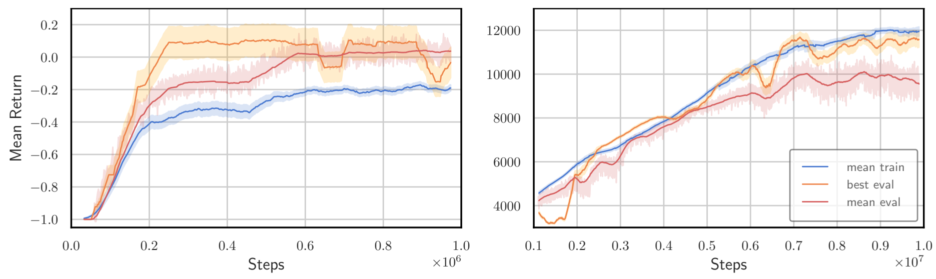
Several papers evaluate the performance of their neural agent using the training curve. The training curve plots the number of training steps on the x-axis against the estimated expected return, i.e., the cumulative sum of (discounted) rewards, on the y-axis. Broadly, there are three common ways how the learning curve's data is used for agent evaluation:

Single Random Seed. It is common for papers—including those introducing influential algorithms such as DQN [52], DreamerV2 [42], PPO [60], or Rainbow [44]—to present the training curve of a single training run/random seed. Typically, this curve is smoothed using a sliding mean over the most recent steps. Regularly, this is done without providing any confidence interval or any additional information beyond the training curve itself. Consider the blue curve



(a) Racetrack (b) MinigridSafe

Fig. 1: Mean return on two common benchmarks during training for a single seed. The blue curve depicts the values observed during training, whereas the orange curve was additionally computed after every training step through PyDSMC by using additional evaluation runs. The shaded area represents the 95% confidence interval over the sliding window (training) or over the evaluation samples (evaluation), respectively.



(a) MinigridSafe (b) Half Cheetah (MuJoCo)

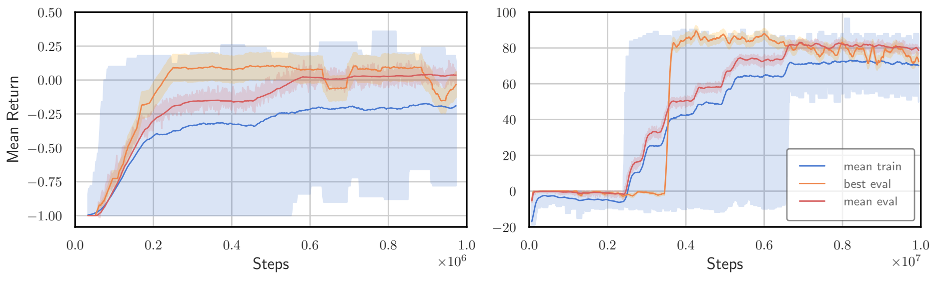
Fig. 2: Mean return on two common benchmarks during training over 10 seeds. The blue curve depicts the values observed during training, whereas the red and orange curves were additionally computed after every training step using `PyDSMC` by performing additional evaluation runs. The orange curve was computed by using the best random seed, while the red curve averages over all used random seeds. The shaded area represents the 95% confidence interval over the 10 different random seeds.

of Fig. 1, which exemplarily provides the training curves for two different benchmarks (left Racetrack [8,31,38], right MinigridSafe [57]) trained with PPO. The shaded blue area here additionally provides the 95% confidence interval computed with the sequential Student's-t method (details in Section 4) using the samples from the sliding mean.

Mean of Multiple Random Seeds with Additional Confidence Information. Another common approach is to report the mean over several training runs, for example, done by Burda et al. [22], Duan et al. [29], or Agarwal et al. [2]. In addition to the mean, these papers typically report either the standard deviation [2], or the 95% confidence interval across these multiple runs [29].

Consider Figure Fig. 2, which provides an example for such a training curve in blue for two different benchmarks (MinigridSafe trained with PPO and Half Cheetah (MuJoCo) [63] trained with SAC [41]). The shaded blue area provides the 95% confidence interval over the 10 different random seeds used.

Mean of Multiple Random Seeds with Additional Min/Max Information. Alternatively, some papers report the minimum and maximum values observed across multiple runs, e.g., SAC [41]. Consider Figure Fig. 3, providing such an example for Racetrack and MinigridSafe. While the blue curve still represents the mean, the shaded blue area now represents the range of observed values across the different training runs.



(a) MinigridSafe (b) Racetrack

Fig. 3: Mean return on two common benchmarks during training over 10 seeds. The blue curve depicts the values observed during training, whereas the red and orange curves were additionally computed after every training step through `PyDSMC` by using additional evaluation runs. The orange curve was computed by using the best random seed, while the red curve averages over all used random seeds. The shaded area represents the 95% confidence interval (red and orange) or the range of minimal and maximal observed values (blue), respectively, over the 10 different random seeds.

2.2 Failure to Assess the Agents' Real Performances

Although all three approaches described to measure a neural agent's performance are common, we will now show that they fail to capture their real performance. To measure their actual performance, we use `PyDSMC` throughout the training to perform additional evaluation runs, i.e., runs without additional exploration influences that are exclusively used to measure the current performance and not for training. In Fig. 1, the orange curve depicts the results of these additional evaluation runs, with the shaded area indicating the 95% confidence interval. In Fig. 2 and Fig. 3, we provide two additional curves computed with `PyDSMC`: the orange curve depicts the performance of the best-performing seed, while the red curve shows the average performance across all seeds. For both curves, the shaded area indicates the 95% confidence interval computed over the used evaluation samples. We measure both the best and the average performance to enable a more comprehensive comparison: while most papers report the average over several training runs, in practice, one naturally selects the best agent for (real-world) deployment *after* training. Thus, we compare against both the actual best and the actual average performance.

Two major discrepancies become apparent when comparing the performance captured by `PyDSMC` (orange, red) with the training curves (blue):

Discrepancy in Performance Measurement. The substantial gap between the training and the evaluation curves underscores the inadequacy of training curves as reliable indicators of the expected return. Mostly, the training curves do not even lie within the confidence interval of the `PyDSMC`'s calculated value, and vice versa. This is evident from the difference between the orange and blue curves for a single seed in Fig. 1, as well as when comparing in the multiple seed setting: both the orange curve (best seed) and the red curve (mean over seeds) differ significantly from the training curve (blue), mostly falling outside each other's confidence interval.

Comparing the confidence interval of the best (orange) and average (red) performance to the min/max shaded area (blue) in Fig. 3 area reveals that the min/max information does not provide any insights about agents' performances.

Therefore, the blue training curve must be considered insufficient for accurately measuring an agent's performance.

Variability in Optimal Performance Timing. There is considerable variation in the timing of peak returns between training and evaluation. While this occurs in both single-seed and multiple-seed settings, it can most clearly be observed when considering a single seed (Fig. 1).

For both Racetrack and MinigridSafe, we compare (i) the time x_t where the training curve (blue) reaches its maximum—corresponding to the point where the agent would typically be selected for deployment after training—with (ii) the time x_e at which the evaluation curve (orange) reaches its maximum, indicating the agent's true peak performance. These two points in time, x_t and x_e, differ significantly. Moreover, the evaluation curve (orange) reveals that the performance at agent selection time x_t is substantially lower than at its actual peak x_e. This implies that the selected agent is far from optimal at the point it would be chosen based on the training curve alone.

2.3 Consequence for Training Neural Agents

These experiments[1] reveal a significant discrepancy in performance measurement. The training curve is affected by additional factors such as exploration, limited samples (especially in the multiple-seed setting), and policy changes during sampling due to ongoing learning steps. Therefore, it does not reflect the true performance, which can only be determined through rigorous verification.[2]

Additionally, the observed variability in optimal performance timing indicates a substantial risk of selecting suboptimal agents when relying solely on

[1] It is important to note that the experiments presented in this section—involving evaluation after every learning step—are computationally expensive and might (depending on the domain) therefore be impractical. Nevertheless, they clearly demonstrate the importance of incorporating verification *at regular intervals* during training.

[2] It is worth noting that, in rare cases that we were especially looking for, the training curve was expressive enough for performance measures. We observed either only negligible differences between the training and evaluation curves, i.e., the training curve resembled the actual performance, or that the extraction point x_t yielded performance comparable to that at x_e. However, the fact that such discrepancies can occur—even if not always—demonstrates the necessity of rigorous verification, as otherwise the actual performance will remain unknown.

training performance. Therefore, selecting the best-performing agent and accurately assessing its performance requires repeated verification throughout and after the training process.

These observations underscore the necessity of employing robust verification tools like `PyDSMC` to ensure that the reported performance of neural agents accurately reflects their true capabilities.

3 The Gymnasium Interface

The Farama Foundation's Gymnasium [64], the successor to OpenAI Gym [16], is an open-source library designed to standardize the interface between neural agents and their environments. To this end, it provides a feature-rich and extensible API to abstract the agent-environment interactions, enabling easy interoperability between libraries and tools. Its widespread adoption across various libraries, tools, and environments serves as evidence of its success.

Available Tools. Stable Baselines3 [58], a popular reinforcement learning framework, allows users to train agents based on a broad variety of algorithms, such as PPO [60] and DQN [52], given that they follow the Gymnasium interface. Other training libraries supporting the Gymnasium interface include CleanRL [46], which provides clean and minimalistic implementations of learning algorithm, and Dopamine [23], a research framework developed by Google that focuses on reproducibility and simplicity.

Having access to numerous training algorithms naturally requires compatible environments. While Gymnasium already bundles example environments like MountainCar, and MuJoCo [63], there is no shortage of custom environments implementing the Gymnasium interface either. Examples include the Atari Learning Environment [11], running original Atari 2600 games on an emulator, and Minigrid [24], providing easy access to implement arbitrary grid-based environments.

Interface. An environment striving to be Gymnasium API-compatible has to implement two methods: `step` and `reset`. Both return a tuple consisting of an *observation*, a *reward*, a *termination* flag, a *truncated* flag, and an *info* dictionary, defining additional (possibly environment-specific) information.

On top of that, keeping efficiency in mind, Gymnasium provides easy vectorization functionality by grouping environments together, enabling batch inference. By default, each contained environment can either be managed by its proper subprocess, or by the main thread itself. The better choice depends on many factors, including the environment's complexity.

4 Statistical Methods

The core of SMC is Monte Carlo simulation: Generate k *simulation runs* (i.e. random executions of the neural agent acting within its Gymnasium environment), which give rise to samples $X_1, \ldots, X_k$ of the random variable X of the property of interest, and return the sample mean $\hat{X} = \frac{1}{k}\sum_{i=1}^{k} X_i$ as an estimate for the value of the property. For example, if the property concerns the probability p to reach a goal, then X is 1 for every run that reaches a goal and 0 for all others (i.e. we estimate a binomial proportion), so $p = \mathbb{P}(X = 1) = \mathbb{E}(X)$; if the property queries for the expected accumulated discounted reward r, then X for a run is the discounted sum of the rewards along that run, and $r = \mathbb{E}(X)$. In the latter case, if we know that rewards are non-negative and the maximum reward of any step is $\leq r_{max}$, we can derive that the support of the distribution of X lies within $[0, \frac{r_{max}}{1-\gamma}]$ for discount factor $\gamma < 1$, and we say that X is *bounded*. Without knowledge of r_{max}, or for $\gamma = 1$, we have to assume unbounded X.

SMC additionally provides a formal statistical guarantee on the correctness of its results. `PyDSMC` uses *confidence intervals* (CIs) as an easy-to-understand way to express its guarantee: In addition to $\hat{X}$, it returns an interval $I = [\ell, u]$ such that, in $(1 - \kappa) \cdot 100\,\%$ of the times such an interval is returned by `PyDSMC`, the (unknown) true value $x = \mathbb{E}(X)$ lies within I. Additionally, we ensure that $u - \ell \leq 2\varepsilon$ (absolute width) or $u - \ell \leq \varepsilon \cdot (\ell + u)$ (relative width). The significance level κ must always be specified. The absolute or relative error bound ε *can* be specified by the user. If it is, then `PyDSMC` generates runs until it can deliver an ε-interval with confidence $1 - \kappa$ (the *sequential* setting). If ε is not given, k must be specified—either directly, or indirectly via a bound on the runtime after which to stop generating runs. Then, once k runs have been collected, `PyDSMC` returns an interval with confidence $1 - \kappa$ (the *fixed-runs* setting), and ε is implicitly given by the interval's half-width (and thus the distinction between absolute and relative ε does not apply). `PyDSMC` uses the sequential setting with relative $\varepsilon = 0.05$ and $\kappa = 0.05$ (i.e. a $\pm 5\,\%$ error with $95\,\%$ confidence) by default.

The statistics literature provides many different statistical methods (SMs) to obtain CIs. `PyDSMC` implements the most widely-used "standard" SMs as well as a set of state-of-the-art "sound" methods recommended in recent surveys [21,51]. An important aspect of an SM is its *coverage probability* p_{cov}: the fraction of intervals that in the limit, if we perform SMC again and again to generate a sequence of independent CIs, contain the true value. A sound SM guarantees $p_{cov} \geq 1 - \kappa$, no matter what parameters we use and what distribution (from those supported by the SM) we sample from.

Standard Methods. The most widely-used SMs, in fields ranging from psychology over medical sciences to economics as well as in many SMC tools [21, Table 1], rely on the central limit theorem (CLT) and assume that they are used with a "large enough" number of samples. These CLT-based **standard SMs are not sound** [21,51]: They only attain $p_{cov} \approx 1 - \kappa$ *on average* over the supported distributions—e.g. on average when ranging over $p \in (0, 1)$ for binomial proportion intervals. We nevertheless include them in `PyDSMC` as they are the de-facto standard in statistical evaluation of results, flawed as they may be, and require few runs. Notably, for unbounded distributions, they are the only methods available [21, Section 4]. In `PyDSMC`, we offer the following standard methods:

- **Normal intervals** approximate the distribution of error by a normal distribution: $I = [\hat{X} - \frac{zs}{\sqrt{k}}, \hat{X} + \frac{zs}{\sqrt{k}}]$ where z is a $1 - \frac{1}{2}\kappa$ quantile from the standard normal distribution and s is the sample standard deviation. As they require k, normal intervals apply to the fixed-runs setting.

– **Student's-t intervals** use a quantile from the Student's-t distribution with $k-1$ degrees of freedom instead, which works a little better for small k.
– **Chow-Robbins' method** keeps generating runs until the normal interval for the current set of runs has half-width at most ε (absolute) or $\varepsilon \cdot \hat{X}$. Chow and Robbins showed that this method attains coverage $1-\kappa$ *in the limit* as $\varepsilon \to 0$ [25]. For any concrete $\varepsilon > 0$, p_{cov} may be much lower than $1-\kappa$.
– **Sequential Student's-t intervals** work the same way as the Chow-Robbins method but use Student's-t instead of normal intervals.

Sound Methods. If we know we estimate a binomial proportion, or the distribution's support is bounded to $[a,b]$, then **sound SMs** are available for all settings. In general, these require more runs than standard methods—but are arguably the methods of choice to evaluate any (safety-)critical application of AI. Finding efficient sound SMs is an area of active research [55,56]; in PyDSMC, we provide the methods recommended by [21] for the fixed-runs setting and for the sequential setting with absolute error, and EBStop [53] for relative error:

– **Clopper-Pearson** intervals [18] apply to binomial proportions only, where the method guarantees coverage $\geq 1-\kappa$. It requires around $2\times$ as many runs as normal intervals in our experiments using PyDSMC's defaults. For the sequential setting, we precompute k via exponential and binary search assuming the worst-case of $p = 0.5$ [51] before any runs are performed.
– **Hoeffding's inequality** [45] gives the relation $k \geq (b-a)^2 \cdot (\ln 2/\delta)/2\varepsilon^2$. We can thus precompute k to solve the sequential setting with absolute error, or solve for ε instead for the fixed-runs setting. Due to the quadratic influence of the range of the distribution, k may become very large.
– **DKW** uses the Dvoretzky-Kiefer-Wolfowitz(-Massart) inequality [30,50] to obtain CIs on the mean in the fixed-runs setting for bounded distributions as described in [21, Section 4.1]. This method delivers smaller CIs than Hoeffding's inequality that are usually asymmetric: the worst case of DKW coincides with Hoeffding's inequality; in the best case, intervals are half as wide.
– **EBStop** [53] is a truly sequential SM (i.e., it does not precompute k but determines whether to stop after every run) for the relative-error case based on Bernstein's inequality [13,14] used with an estimation of X's variance.

Choice of SM. Fig. 4 provides an overview of the SMs available in PyDSMC and the decision tree that it employs to select the method to use. The choice depends on the setting (fixed-runs or sequential), the kind of distribution underlying the property (binomial, bounded, or unbounded), how the interval width ε is specified (absolute or relative), and whether the user requests a sound method to be used. By default, PyDSMC uses the standard methods.

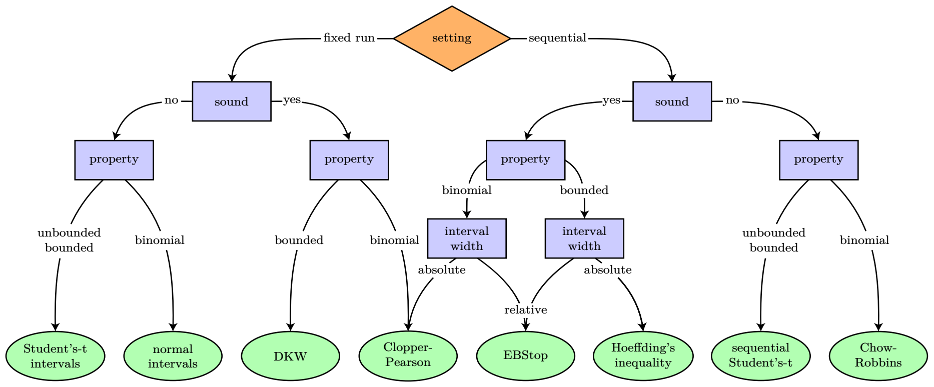


Fig. 4: Automated statistical method selection.

5 The `PyDSMC` Python Package

`PyDSMC` is implemented as a Python package that offers numerous SMC techniques and, thus, can easily be integrated into existing training pipelines. To access these functions, users only need to interact with two classes: First, the `Property` class, which allows specifying arbitrary trajectory-based properties, and second, the `Evaluator` class, which handles the sampling and checking of the properties.

Properties. `PyDSMC` offers various predefined properties, e.g., the return, the episode length, or the goal-reaching probability. These properties may be parameterized, for instance, by the discount factor used in the return calculation. Further, users can easily define custom properties by providing functions that check the property. The properties are thereby assumed to be trajectory-based, i.e., they can use the Gymnasium-provided information for all steps of the trajectory. If the user specifies ε to be `None`, `PyDSMC` uses the fixed run setting, and the sequential setting otherwise. Since the statistical methods depend on property-specific attributes (e.g., bounded, binomial), these have to be set during property initialization. For each property, the user can additionally set: (i) a name, (ii) a flag whether sound methods should be used, and (iii) a flag to toggle between absolute and relative error ε.

Evaluator. Having defined the properties, they have to be registered in an `Evaluator` instance, which manages the environment and the logging directory. The evaluation can then be started by calling `eval` on the `Evaluator` and providing the agent to verify. Additionally, `eval` takes the following arguments: (i) an optional resource limit (time, number of samples, or both), (ii) the number of samples taken between convergence checks, (iii) a flag whether to stop on convergence of all properties or to run until the specified resources are exhausted, (iv) a logging interval, (v) a flag whether to store every sample of every property, and (vi) the number of threads used for parallelization.

```python
from pydsmc import Evaluator, property as prop

# Create a predefined property
return_property = prop.create_predefined_property(
↪  property_id='return', epsilon=0.025, kappa=0.05,
↪  relative_error=True, bounds=(-1, 1), sound=True, gamma=0.99)
# Create a custom property
collision_property = prop.create_custom_property(
↪  name='obstacle_collision_prob',
↪  check_fn=lambda self, t: float(t[-1][2] == -1), epsilon=0.05,
↪  kappa=0.05, relative_error=False, bounds=(0, 1), binomial=True)

# Create the evaluator and register the properties
evaluator = Evaluator(env=env, log_dir="./example_logs")
evaluator.register_properties([return_property, collision_property])

# Evaluate the agent with respect to the registered properties
results = evaluator.eval(agent=agent, save_every_n_episodes=1000,
↪  time_limit=150, stop_on_convergence=True, num_threads=2)
```

Fig. 5: Example usage of `PyDSMC`.

Storing Results. Within the logging directory, `PyDSMC` creates a subdirectory for each property, where all files containing its evaluation results and its parameters are stored. `PyDSMC` additionally saves files storing the evaluation parameters and the utilized resources corresponding to the number of episodes and runtime.

Parallelization. To decrease runtime, `PyDSMC` supports vectorized environments and multithreading, where each thread manages a separate vector environment. We observed that vectorization significantly accelerates `PyDSMC`'s execution.

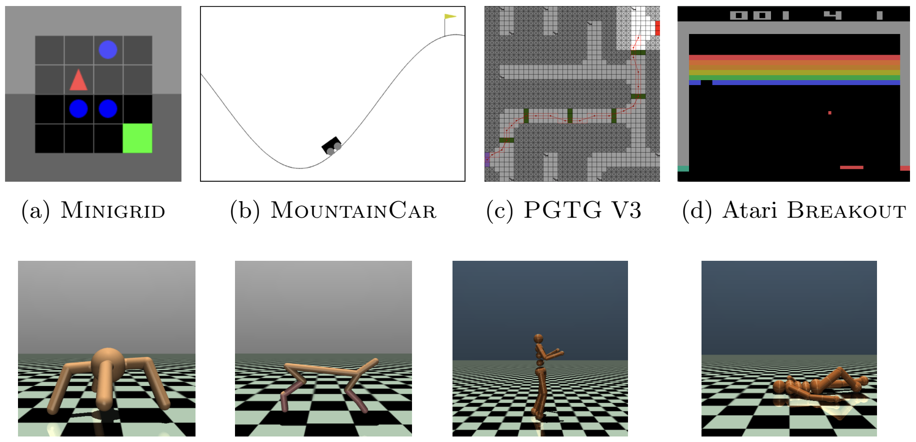
Example. Consider the example in Fig. 5 which shows how `PyDSMC` can be used to evaluate an agent on a given MiniGrid environment(see Fig. 6a). We first create a predefined return property with parameters `epsilon=0.025`, `kappa=0.05`, and the property-specific discount factor `gamma=0.99`, where ε describes the maximum tolerated relative error. Since the rewards in the sample environment lie within the interval $[-1, 1]$, we can use a bounded SM by setting `bounds=(-1, 1)`.

Second, we define a custom, environment-specific property that corresponds to the agent's probability of colliding with an obstacle, which, in this domain, can be identified by a negative reward in the trajectory's last step. The remaining arguments specify that we want to evaluate the absolute error of this binomial property. As the sound flag is not specified, `PyDSMC` defaults to an unsound SM.

Afterward, the evaluator is initialized and the properties are registered. We limit the evaluation to 2.5 hours, but stop early if all properties have converged while using two threads.

6 Analyzing Neural Agents Using `PyDSMC`

We demonstrate `PyDSMC` by evaluating neural agents trained using state-of-the-art deep reinforcement learning algorithms on eight Gymnasium benchmarks, including four from MuJuCo [63].



(a) Minigrid (b) MountainCar (c) PGTG V3 (d) Atari Breakout

(e) Ant (f) Half Cheetah (g) Humanoid (h) Humanoid Standup

Fig. 6: The eight Gymnasium benchmark environments used in this section. The second row depicts the four MuJoCo Benchmarks, all in version V5.

6.1 Benchmarks

We present exemplary results on eight benchmarks commonly used in the AI community: (a) Minigrid [24], (b) MountainCar [65], (c) ProcGrid Traffic Gym [33], (d) Breakout [10] , and four MuJoCo benchmarks (e) Ant, (f) Half Cheetah, (g) Humanoid, and (h) Humanoid Standup. Fig. 6 shows the eight benchmarks.

Minigrid. A Minigrid environment corresponds to a 2D navigation task, where the agent has to traverse a grid to reach a goal cell. The available actions are moving forward by one cell, rotating by 90 degrees to the left or right, picking up objects, and interacting with objects (e.g., opening doors). Based on its current direction, the agent can only observe a limited section of the grid.

We focus on the Minigrid Dynamic Obstacles environment, where, starting from a random cell, the agent has to reach the goal cell while avoiding three randomly moving obstacles. A reward of $1 - 0.9 \cdot \frac{steps}{144}$ is given when reaching the goal, -1 when colliding with an obstacle, and 0 otherwise.

MountainCar. MountainCar is a classic control benchmark, where a car is randomly placed in a valley. The goal is to reach the top of the right hill by accelerating either to the left or the right. The actions represent the car's directional force, which is within the range $[-1;1]$. The agent observes its current location and velocity. At each time step, a negative reward of $-0.1 \cdot action^2$ is given, with an additional positive reward of 100 if the agent reaches the goal. The (undiscounted) return bounds are $[-999;100]$, and the episode length bounds are $[1;999]$.

ProcGrid Traffic Gym. ProcGrid Traffic Gym (PGTG) is an extension of the popular Racetrack benchmark. The task is a rough simplification of autonomous driving, where the agent needs to drive from a starting line to a goal line across a randomly generated racetrack. Additionally, environments can be customized with different features such as ice, sand, or traffic. The observation is limited to the agent's surroundings, where a green lines function as a guidance toward the goal line. A reward of $\frac{100}{\text{number of subgoals}}$ is given when reaching a subgoal for the first time, a reward of 150 is given when reaching the goal line, a reward of -100 is given for crashing, and a reward of 0 otherwise. The (undiscounted) return bounds are $[-100;150]$, and the episode length bounds are $[1;100]$.

Breakout. Breakout is a classic Atari 2600 game in which the agent moves a paddle horizontally to bounce a ball such that it destroys the blocks at the top. Whenever the ball touches the bottom, the agent loses a life and a new ball spawns. The game is over when the agent has either lost all of its 5 lives or has destroyed all blocks. A reward is given when a block is destroyed, with the values ranging from 1 for blue blocks and 7 for red blocks. The (undiscounted) return bounds are $[0;864]$, but the episode length is unbounded as no time step limit is given.

Ant. In this MuJoCo environment, a four-legged 3D robot is tasked to move forward from a random initial state. The actions correspond to applying torque to the ant's joints, and an episode ends when the height of the ant's torso is outside a predefined range. Ant features a dense reward function, where the agent is rewarded for moving forward, and keeping its torso at a certain height, and is penalized for applying too much torque to the joints or when the external contact forces on the ant's body parts are too high. We do not assume any bounds on the return for all MuJoCo benchmarks.

Half Cheetah. This environment features a 2D robot resembling a cheetah, that ought to run as fast as possible from a random initial state. The action space represents the torque applied to the cheetah's joints. All episodes end after a fixed number of timesteps. The agent's rewards are based on how fast it moves forward, and it is penalized for applying too much torque.

Humanoid. A 3D robot resembling a human needs to quickly walk forward without falling from a random initial state. The actions are the torques applied to the robot's legs, arms, and torso, and an episode ends when the torso's height is outside a predefined range. The agent is rewarded for moving forward and keeping its torso at a certain height, and it is penalized for applying too much torque or when the external contact forces are too high.

Humanoid Standup. This environment is similar to Humanoid, with the difference that the robot does not have to move forward but starts lying on the ground and the task is to stand up. Unlike in humanoid, the episodes are not terminated early.

6.2 Properties

We evaluate the agent by using some standard properties. To further highlight `PyDSMC`'s flexibility, we also define a custom property for each environment.

Standard Properties. We evaluate the return with a discount factor of $\gamma = 0.99$, the undiscounted return (i.e., $\gamma = 1.0$), the average episode length, and the goal-reaching probability. For all properties, we use a significance level $\kappa = 0.05$, and an error bound $\varepsilon = 0.025$, which is relative for all properties but the goal-reaching probability, where we use the absolute setting.

Custom Properties. We define a custom property for each benchmark. In Minigrid, we define the custom property *Collision Prob.* as the probability of the agent moving onto a cell that is already occupied. In MountainCar we consider the average acceleration *Avg. Acceleration.* For PGTG, we examine the probability of crashing into a wall *(Crash Prob).* In Breakout, we analyze the number of steps until the agent loses its first life *First Life Lost.* In Ant, we consider *Sum. Control Cost* as the sum of all obtained penalties for applying too much torque. For Half Cheetah, we analyze the probability of ending an episode with a negative return (*Neg. Return Prob*). In Humanoid and Humanoid Standup, we customize *Sum. Contact Cost* as the sum of penalties that was given because of too much contact force.

6.3 Exemplary Evaluation

Setup. All experiments were performed on a single machine with an AMD Ryzen Threadripper PRO 5965WX 24 Core CPU, an NVIDIA RTX A6000 GPU, and 512 GB of memory.[3] We provide details about the used hyperparameters in Appendix A.

[3] Note that memory was never an issue and the machine used does not need this large amount of memory.

Property	**Stat. Method**	**Mean**	**St.D.**	**C.I.**	**Conv.**
MINIGRID (◷ $\approx$ 45m)					
Return† (γ=0.99)	EBStop	0.14	0.21	[0.14;0.15]	249000
Return† (γ=1.0)	EBStop	0.25	0.3	[0.25;0.26]	153000
Episode Length	Student's-t	112.2	41.73	[112.04;112.37]	1000
Goal-reaching Prob.	Normal Interval	0.49	0.5	[0.49;0.49]	2000
Collision Prob.*	Normal Interval	0.0	0.0	[0.0;0.0]	1000
MOUNTAINCAR (◷ $\approx$ 3m)					
Return† (γ=0.99)	EBStop	41.43	2.05	[40.39;42.46]	80000
Return† (γ=1.0)	EBStop	95.12	0.49	[92.78;97.53]	31000
Episode Length	Student's-t	81.21	4.48	[81.18;81.24]	1000
Goal-reaching Prob.	Normal Interval	1.0	0.0	[1.0;1.0]	1000
Avg. Acceleration*	Student's-t	0.08	0.08	[0.08;0.08]	1000
PGTG (◷ $\approx$ 2m)					
Return† (γ=0.99)	EBStop	170.49	77.3	[166.22;174.74]	23000
Return† (γ=1.0)	EBStop	208.79	97.48	[203.56;214.0]	23000
Episode Length	Student's-t	31.69	19.77	[31.43;31.95]	3000
Goal-reaching Prob.	Normal Interval	0.84	0.36	[0.84;0.85]	1000
Crash Prob.*	Normal Interval	0.08	0.28	[0.08;0.09]	1000
BREAKOUT (◷ $\approx$ 10h 38m)					
Return† (γ=0.99)	DKW	1.55	0.43	[1.52;13.28]	—
Return† (γ=1.0)	DKW	25.81	15.41	[24.84;37.54]	—
Episode Length	Student's-t	26484.0	15207.3	[26186;26782]	—
First Life Lost*	Student's-t	259.26	353.02	[252.34;266.18]	—
ANT (◷ $\approx$ 33m)					
Return (γ=0.99)	Student's-t	137.75	31.60	[137.27;138.22]	1000
Return (γ=1.0)	Student's-t	1017.36	497.49	[1009.88;1024.84]	2000
Episode Length†	EBStop	779.53	325.81	[760.08;799.03]	17000
Sum. Control Cost*	Student's-t	-502.42	208.53	[-505.6;-499.3]	2000
HALF CHEETAH (◷ $\approx$ 2m)					
Return (γ=0.99)	Student's-t	247.67	101.01	[243.24;252.10]	2000
Return (γ=1.0)	Student's-t	3247.40	1146.31	[3197.13;3297.67]	2000
Neg. Return Prob.*	Normal Interval	0.009	0.09444	[0.0049;0.0131]	1000
HUMANOID (◷ $\approx$ 12m)					
Return (γ=0.99)	Student's-t	485.97	9.64	[485.72;486.22]	1000
Return (γ=1.0)	Student's-t	4782.39	886.35	[4759.96;4804.82]	1000
Episode Length†	EBStop	937.04	172.30	[913.59;960.41]	6000
Sum. Contact Cost*	Student's-t	-119.48	22.34	[-120.0;-118.9]	1000
HUMANOID STANDUP (◷ $\approx$ 2m)					
Return (γ=0.99)	Student's-t	13940.7	285.2	[13928.2;13953.2]	1000
Return (γ=1.0)	Student's-t	146413.7	6684.6	[146120;146707]	1000
Sum. Contact Cost*	Student's-t	-51.97	3.96	[-52.14;-51.79]	1000

Table 1: Evaluation results. Properties marked with * are custom properties, whereas † denotes the usage of a sound statistical method. For BREAKOUT, the **Conv.** column is empty since the fixed runs setting was used. The reported time corresponds to the total runtime of all listed properties analyzed simultaneously, rounded to the nearest full minute.

Results. Table 1 provides all results obtained with `PyDSMC`. For all benchmarks, we provide the automatically selected SM, the approximated mean with its standard deviation, and the confidence interval. The **Conv.** column reports the number of samples that were needed to achieve the target confidence interval. Since we used the fixed runs setting with 10,000 episodes for BREAKOUT, the **Conv.** column is empty. We marked those properties where we enforced a sound SM.

Further Insights. Additionally analyzing other properties and not only the standard objective, i.e., the accumulated discounted return, can provide deeper insights about the agent.

For example, consider the MINIGRID results. Despite the low return ($\gamma = 0.99$) of 0.14, the goal-reaching probability is 49%, indicating that the agent reaches the goal almost every other try. By also taking into account the episode length of 112 and the obstacle collision probability of 0, we can conclude that the poor performance is due to the agent frequently standing still until the step limit is reached.

As another example, consider MOUNTAINCAR. We observe a goal-reaching probability of 100%, indicating that the agent has learned to always reach the top of the hill. The undiscounted return of 95.12 suggests that it does so with little acceleration, which is further confirmed by the small average acceleration of 0.08.

In ANT, the high standard deviation of the episode lengths suggests that sometimes the agent quickly fails to keep its torso at the required height. This can be explained by the large value of the summed control penalties, showing that the agent tends to apply large torques, which can lead to situations where it is impossible to prevent early termination.

Lastly, consider the results of HALF CHEETAH. The discounted return already indicates that the agent is performing well. While the additional information of the undiscounted reward already strengthens this finding, the custom property *Neg. Return Prob.* additionally shows that the agent rarely uses too much torque, which indicates that the agent has learned to precisely control the joints.

7 Conclusion and Future Work

In this paper, we presented `PyDSMC`, a Python tool for applying statistical model checking to arbitrary neural agents in any Gymnasium environment, independent of the underlying implementation. We highlighted the importance of statistical model checking for neural agents, as standard evaluation methods like training curves fail to capture key influences, potentially leading to suboptimal agent selection and misleading performance assessments. We demonstrated `PyDSMC`'s usage and illustrated how it can provide critical insights into agents' behaviors.

For the future, we have planned several extensions of `PyDSMC`. These include expanding the set of predefined properties and extending compatibility to *PettingZoo* [62], the standard interface for multi-agent reinforcement learning. Additionally, we aim to adapt `PyDSMC` for symbolic AI approaches such as planning. The algorithms *DSMC Evaluation Stages* [32,37] and *RARE* [39] have already integrated DSMC results into the training procedure to improve the performance of neural agents. In the future, we plan to integrate PyDSMC into these algorithms.

With an increasing range of applications, we also plan to integrate additional statistical methods to enhance both evaluation accuracy and efficiency.

Funding. This work was partially supported by the German Federal Ministry of Education and Research (BMBF) as part of project MAC-MERLin (grant agreement no. 01IW24007), by the German Research Foundation (DFG) under grant no. 389792660, as part of TRR 248, see https://perspicuous-computing.science and as GRK 2853/1 “Neuroexplicit Models of Language, Vision, and Action” (project no. 471607914), by the European Regional Development Fund (ERDF) and Saarland within the scope of (To)CERTAIN, by the European Union’s Horizon 2020 research and innovation programme under Marie Skłodowska-Curie grant agreement no. 101008233 (MISSION), by the Interreg North Sea project STORM_SAFE, and by NWO VIDI grant VI.Vidi.223.110 (TruSTy).

Data Availability. The models, scripts, and tools to reproduce our experimental evaluation are archived and publicly available at DOI 10.5281/zenodo.15267298.

References

1. Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F.L., Almeida, D., Altenschmidt, J., Altman, S., Anadkat, S., et al.: GPT-4 Technical Report. arXiv preprint arXiv:2303.08774 (2024). https://doi.org/10.48550/arXiv.2303.08774
2. Agarwal, R., Schuurmans, D., Norouzi, M.: An optimistic perspective on offline reinforcement learning. In: Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event. Proceedings of Machine Learning Research, vol. 119, pp. 104–114. PMLR (2020), http://proceedings.mlr.press/v119/agarwal20c.html
3. Agha, G., Palmskog, K.: A survey of statistical model checking. ACM Trans. Model. Comput. Simul. **28**(1), 6:1–6:39 (2018). https://doi.org/10.1145/3158668
4. Albarghouthi, A.: Introduction to neural network verification. Found. Trends Program. Lang. **7**(1-2), 1–157 (2021). https://doi.org/10.1561/2500000051
5. Alshiekh, M., Bloem, R., Ehlers, R., Könighofer, B., Niekum, S., Topcu, U.: Safe reinforcement learning via shielding. In: Proceedings of the AAAI conference on artificial intelligence. vol. 32 (2018)
6. Baier, C.: Probabilistic model checking. In: Esparza, J., Grumberg, O., Sickert, S. (eds.) Dependable Software Systems Engineering, NATO Science for Peace and Security Series – D: Information and Communication Security, vol. 45, pp. 1–23. IOS Press (2016). https://doi.org/10.3233/978-1-61499-627-9-1

7. Baier, C., de Alfaro, L., Forejt, V., Kwiatkowska, M.: Model checking probabilistic systems. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking, pp. 963–999. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_28
8. Baier, C., Christakis, M., Gros, T.P., Groß, D., Gumhold, S., Hermanns, H., Hoffmann, J., Klauck, M.: Lab conditions for research on explainable automated decisions. In: Trustworthy AI-Integrating Learning, Optimization and Reasoning: First International Workshop, TAILOR 2020, Virtual Event, September 4–5, 2020, Revised Selected Papers 1. pp. 83–90. Springer (2021). https://doi.org/10.1007/978-3-030-73959-1_8
9. Banerjee, D., Xu, C., Singh, G.: Input-relational verification of deep neural networks. Proc. ACM Program. Lang. **8**(PLDI), 1–27 (2024). https://doi.org/10.1145/3656377
10. Bellemare, M.G., Naddaf, Y., Veness, J., Bowling, M.: The arcade learning environment: An evaluation platform for general agents. Journal of Artificial Intelligence Research **47**, 253–279 (jun 2013)
11. Bellemare, M.G., Naddaf, Y., Veness, J., Bowling, M.: The Arcade Learning Environment: An Evaluation Platform for General Agents. Journal of Artificial Intelligence Research **47**, 253–279 (2013). https://doi.org/10.1613/jair.3912
12. Berner, C., Brockman, G., Chan, B., Cheung, V., Dębiak, P., Dennison, C., Farhi, D., Fischer, Q., Hashme, S., Hesse, C., et al.: Dota 2 with Large Scale Deep Reinforcement Learning. arXiv preprint arXiv:1912.06680 (2019). https://doi.org/10.48550/arXiv.1912.06680
13. Bernstein, S.: On a modification of Chebyshev's inequality and of the error formula of Laplace. Ann. Sci. Inst. Sav. Ukraine, Sect. Math **1**(4), 38–49 (1924)
14. Bernstein, S.: Theory of Probability. 2 edn. (1934)
15. Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., Zaremba, W.: Openai gym (2016)
16. Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., Zaremba, W.: OpenAI Gym. arXiv preprint arXiv.1606.01540 (2016). https://doi.org/10.48550/arXiv.1606.01540
17. Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J.D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al.: Language Models are Few-Shot Learners. In: Advances in Neural Information Processing Systems. vol. 33, pp. 1877–1901. Curran Associates, Inc. (2020), https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html
18. Bu, H., Sun, M.: Clopper-pearson algorithms for efficient statistical model checking estimation. IEEE Transactions on Software Engineering (01), 1–20 (2024). https://doi.org/10.1109/TSE.2024.3392720
19. Budde, C.E., D'Argenio, P.R., Hartmanns, A., Sedwards, S.: An efficient statistical model checker for nondeterminism and rare events. Int. J. Softw. Tools Technol. Transf. **22**(6), 759–780 (2020). https://doi.org/10.1007/S10009-020-00563-2
20. Budde, C.E., Dehnert, C., Hahn, E.M., Hartmanns, A., Junges, S., Turrini, A.: Jani: quantitative model and tool interaction. In: Tools and Algorithms for the Construction and Analysis of Systems: 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part II 23. pp. 151–168. Springer (2017)
21. Budde, C.E., Hartmanns, A., Meggendorfer, T., Weininger, M., Wienhöft, P.: Sound statistical model checking for probabilities and expected rewards. In: 31st

International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS). Lecture Notes in Computer Science, Springer (2025), to appear, preprint available at https://doi.org/10.48550/arXiv.2411.00559
22. Burda, Y., Edwards, H., Storkey, A., Klimov, O.: Exploration by random network distillation. arXiv preprint arXiv:1810.12894 (2018)
23. Castro, P.S., Moitra, S., Gelada, C., Kumar, S., Bellemare, M.G.: Dopamine: A Research Framework for Deep Reinforcement Learning. arXiv preprint arXiv.1812.06110 (2018). https://doi.org/10.48550/arXiv.1812.06110
24. Chevalier-Boisvert, M., Dai, B., Towers, M., de Lazcano, R., Willems, L., Lahlou, S., Pal, S., Castro, P.S., Terry, J.: Minigrid & miniworld: Modular & customizable reinforcement learning environments for goal-oriented tasks. CoRR **abs/2306.13831** (2023)
25. Chow, Y.S., Robbins, H.: On the Asymptotic Theory of Fixed-Width Sequential Confidence Intervals for the Mean. The Annals of Mathematical Statistics **36**(2), 457–462 (1965). https://doi.org/10.1214/aoms/1177700156
26. Cobbe, K., Hesse, C., Hilton, J., Schulman, J.: Leveraging procedural generation to benchmark reinforcement learning. In: International conference on machine learning. pp. 2048–2056. PMLR (2020)
27. Corsi, D., Marchesini, E., Farinelli, A.: Formal verification of neural networks for safety-critical tasks in deep reinforcement learning. In: de Campos, C.P., Maathuis, M.H., Quaeghebeur, E. (eds.) 37th Conference on Uncertainty in Artificial Intelligence (UAI). Proceedings of Machine Learning Research, vol. 161, pp. 333–343. AUAI Press (2021), https://proceedings.mlr.press/v161/corsi21a.html
28. Dehnert, C., Junges, S., Katoen, J.P., Volk, M.: A storm is coming: A modern probabilistic model checker. In: Computer Aided Verification: 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II 30. pp. 592–600. Springer (2017)
29. Duan, J., Guan, Y., Li, S.E., Ren, Y., Sun, Q., Cheng, B.: Distributional soft actor-critic: Off-policy reinforcement learning for addressing value estimation errors. IEEE Trans. Neural Networks Learn. Syst. **33**(11), 6584–6598 (2022). https://doi.org/10.1109/TNNLS.2021.3082568, https://doi.org/10.1109/TNNLS.2021.3082568
30. Dvoretzky, A., Kiefer, J., Wolfowitz, J.: Asymptotic Minimax Character of the Sample Distribution Function and of the Classical Multinomial Estimator. The Annals of Mathematical Statistics **27**(3), 642–669 (1956). https://doi.org/10.1214/aoms/1177728174
31. Gros, T.P.: Tracking the race: Analyzing racetrack agents trained with imitation learning and deep reinforcement learning. Master's thesis **5** (2021)
32. Gros, T.P., Groß, J., Höller, D., Hoffmann, J., Klauck, M., Meerkamp, H., Müller, N.J., Schaller, L., Wolf, V.: Dsmc evaluation stages: Fostering robust and safe behavior in deep reinforcement learning–extended version. ACM Transactions on Modeling and Computer Simulation **33**(4), 1–28 (2023)
33. Gros, T.P., Groß, D., Kamp, J., Gumhold, S., Hoffman, J.: Visual analysis of action policy behavior: A case study in grid-world driving. In: World Conference on Explainable Artificial Intelligence. Springer (2025)
34. Gros, T.P., Hermanns, H., Hoffmann, J., Klauck, M., Köhl, M.A., Wolf, V.: Mogym: Using formal models for training and verifying decision-making agents. In: International Conference on Computer Aided Verification. pp. 430–443. Springer (2022)

35. Gros, T.P., Hermanns, H., Hoffmann, J., Klauck, M., Steinmetz, M.: Deep statistical model checking. In: Formal Techniques for Distributed Objects, Components, and Systems: 40th IFIP WG 6.1 International Conference, FORTE 2020, Held as Part of the 15th International Federated Conference on Distributed Computing Techniques, DisCoTec 2020, Valletta, Malta, June 15–19, 2020, Proceedings 40. pp. 96–114. Springer (2020)
36. Gros, T.P., Hermanns, H., Hoffmann, J., Klauck, M., Steinmetz, M.: Analyzing neural network behavior through deep statistical model checking. International Journal on Software Tools for Technology Transfer **25**(3), 407–426 (2023)
37. Gros, T.P., Höller, D., Hoffmann, J., Klauck, M., Meerkamp, H., Wolf, V.: Dsmc evaluation stages: Fostering robust and safe behavior in deep reinforcement learning. In: Quantitative Evaluation of Systems: 18th International Conference, QEST 2021, Paris, France, August 23–27, 2021, Proceedings 18. pp. 197–216. Springer (2021)
38. Gros, T.P., Höller, D., Hoffmann, J., Wolf, V.: Tracking the race between deep reinforcement learning and imitation learning. In: Quantitative Evaluation of Systems: 17th International Conference, QEST 2020, Vienna, Austria, August 31–September 3, 2020, Proceedings 17. pp. 11–17. Springer (2020)
39. Gros, T.P., Müller, N., Höller, D., Hoffmann, J., Wolf, V.: Safe reinforcement learning through regret and state restorations in evaluation stages. Currently in publication (2024)
40. Gross, D., Jansen, N., Junges, S., Pérez, G.A.: Cool-mc: a comprehensive tool for reinforcement learning and model checking. In: International Symposium on Dependable Software Engineering: Theories, Tools, and Applications. pp. 41–49. Springer (2022)
41. Haarnoja, T., Zhou, A., Abbeel, P., Levine, S.: Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In: International conference on machine learning. pp. 1861–1870. PMLR (2018)
42. Hafner, D., Lillicrap, T., Norouzi, M., Ba, J.: Mastering atari with discrete world models. arXiv preprint arXiv:2010.02193 (2020)
43. Hartmanns, A., Hermanns, H.: The modest toolset: An integrated environment for quantitative modelling and verification. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 593–598. Springer (2014)
44. Hessel, M., Modayil, J., Van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M., Silver, D.: Rainbow: Combining improvements in deep reinforcement learning. In: Proceedings of the AAAI conference on artificial intelligence. vol. 32 (2018)
45. Hoeffding, W.: Probability inequalities for sums of bounded random variables. Journal of the American Statistical Association **58**(301), 13–30 (1963). https://doi.org/10.1080/01621459.1963.10500830
46. Huang, S., Dossa, R.F.J., Ye, C., Braga, J., Chakraborty, D., Mehta, K., Araújo, J.G.M.: CleanRL: High-quality Single-file Implementations of Deep Reinforcement Learning Algorithms. Journal of Machine Learning Research **23**(274), 1–18 (2022), http://jmlr.org/papers/v23/21-1342.html
47. Jansen, N., Könighofer, B., Junges, S., Serban, A., Bloem, R.: Safe reinforcement learning using probabilistic shields. In: 31st International Conference on Concurrency Theory (CONCUR 2020). Schloss-Dagstuhl-Leibniz Zentrum für Informatik (2020)

48. Kwiatkowska, M., Norman, G., Parker, D.: Prism 4.0: Verification of probabilistic real-time systems. In: Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23. pp. 585–591. Springer (2011)
49. Legay, A., Lukina, A., Traonouez, L.M., Yang, J., Smolka, S.A., Grosu, R.: Statistical model checking. In: Steffen, B., Woeginger, G.J. (eds.) Computing and Software Science – State of the Art and Perspectives, Lecture Notes in Computer Science, vol. 10000, pp. 478–504. Springer (2019). https://doi.org/10.1007/978-3-319-91908-9_23
50. Massart, P.: The tight constant in the Dvoretzky-Kiefer-Wolfowitz inequality. The Annals of Probability **18**(3), 1269–1283 (1990). https://doi.org/10.1214/aop/1176990746
51. Meggendorfer, T., Weininger, M., Wienhöft, P.: What are the odds? Improving the foundations of statistical model checking. CoRR **abs/2404.05424** (2024). https://doi.org/10.48550/ARXIV.2404.05424
52. Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G., et al.: Human-level control through deep reinforcement learning. nature **518**(7540), 529–533 (2015)
53. Mnih, V., Szepesvári, C., Audibert, J.Y.: Empirical Bernstein stopping. In: Cohen, W.W., McCallum, A., Roweis, S.T. (eds.) 25th International Conference on Machine Learning (ICML). ACM International Conference Proceeding Series, vol. 307, pp. 672–679. ACM (2008). https://doi.org/10.1145/1390156.1390241
54. Narodytska, N.: Formal verification of deep neural networks. In: Bjørner, N.S., Gurfinkel, A. (eds.) 18th Conference on Formal Methods in Computer Aided Design (FMCAD). IEEE (2018). https://doi.org/10.23919/FMCAD.2018.8603017
55. Parmentier, M., Legay, A.: Adaptive stopping algorithms based on concentration inequalities. In: Steffen, B. (ed.) 2nd International Conference on Bridging the Gap Between AI and Reality (AISoLA). Lecture Notes in Computer Science, vol. 15217, pp. 336–353. Springer (2024). https://doi.org/10.1007/978-3-031-75434-0_23
56. Phan, M., Thomas, P.S., Learned-Miller, E.G.: Towards practical mean bounds for small samples. In: Meila, M., Zhang, T. (eds.) 38th International Conference on Machine Learning (ICML). Proceedings of Machine Learning Research, vol. 139, pp. 8567–8576. PMLR (2021), http://proceedings.mlr.press/v139/phan21a.html
57. Pranger, S.: Minigridsafe: An extension of the minigrid library for safe reinforcement learning. https://github.com/PrangerStefan/MinigridSafe (2025)
58. Raffin, A., Hill, A., Gleave, A., Kanervisto, A., Ernestus, M., Dormann, N.: Stable-Baselines3: Reliable Reinforcement Learning Implementations. Journal of Machine Learning Research **22**(268), 1–8 (2021), http://jmlr.org/papers/v22/20-1364.html
59. Schlüter, M., Steffen, B.: Affinitree: A compositional framework for formal analysis and explanation of deep neural networks. In: Huisman, M., Howar, F. (eds.) 18th International Conference on Tests and Proofs (TAP). Lecture Notes in Computer Science, vol. 15153, pp. 148–167. Springer (2024). https://doi.org/10.1007/978-3-031-72044-4_8
60. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O.: Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347 (2017)
61. Senior, A.W., Evans, R., Jumper, J., Kirkpatrick, J., Sifre, L., Green, T., Qin, C., Žídek, A., Nelson, A.W., Bridgland, A., et al.: Improved protein structure prediction using potentials from deep learning. Nature **577**(7792), 706–710 (2020)

62. Terry, J., Black, B., Grammel, N., Jayakumar, M., Hari, A., Sullivan, R., Santos, L.S., Dieffendahl, C., Horsch, C., Perez-Vicente, R., et al.: Pettingzoo: Gym for multi-agent reinforcement learning. Advances in Neural Information Processing Systems **34**, 15032–15043 (2021)
63. Todorov, E., Erez, T., Tassa, Y.: MuJoCo: A physics engine for model-based control. In: 2012 IEEE/RSJ International Conference on Intelligent Robots and Systems. pp. 5026–5033 (2012). https://doi.org/10.1109/IROS.2012.6386109
64. Towers, M., Kwiatkowski, A., Terry, J., Balis, J.U., Cola, G.D., Deleu, T., Goulão, M., Kallinteris, A., Krimmel, M., KG, A., Perez-Vicente, R., Pierré, A., Schulhoff, S., Tai, J.J., Tan, H., Younis, O.G.: Gymnasium: A Standard Interface for Reinforcement Learning Environments. arXiv preprint arXiv.2407.17032 (2024). https://doi.org/10.48550/arXiv.2407.17032
65. Towers, M., Kwiatkowski, A., Terry, J., Balis, J.U., De Cola, G., Deleu, T., Goulão, M., Kallinteris, A., Krimmel, M., KG, A., et al.: Gymnasium: A standard interface for reinforcement learning environments. arXiv preprint arXiv:2407.17032 (2024)
66. Vandin, A.: Statistical model checking of python agent-based models: An integration of multivesta and mesa. In: Steffen, B. (ed.) 2nd International Conference on Bridging the Gap Between AI and Reality (AISoLA). Lecture Notes in Computer Science, vol. 15217, pp. 398–419. Springer (2024). https://doi.org/10.1007/978-3-031-75434-0_26
67. Vandin, A., Giachini, D., Lamperti, F., Chiaromonte, F.: Multivesta: Statistical analysis of economic agent-based models by statistical model checking. In: Bowles, J., Broccia, G., Pellungrini, R. (eds.) 10th International DataMod Symposium – From Data to Models and Back. Lecture Notes in Computer Science, vol. 13268, pp. 3–6. Springer (2021). https://doi.org/10.1007/978-3-031-16011-0_1
68. Vinyals, O., Babuschkin, I., Czarnecki, W.M., Mathieu, M., Dudzik, A., Chung, J., Choi, D.H., Powell, R., Ewalds, T., Georgiev, P., et al.: Grandmaster level in starcraft ii using multi-agent reinforcement learning. nature **575**(7782), 350–354 (2019)
69. Younes, H.L.S., Simmons, R.G.: Probabilistic verification of discrete event systems using acceptance sampling. In: Brinksma, E., Larsen, K.G. (eds.) 14th International Conference on Computer Aided Verification (CAV). Lecture Notes in Computer Science, vol. 2404, pp. 223–235. Springer (2002). https://doi.org/10.1007/3-540-45657-0_17

A Training Hyperparameters

All agents were trained using the algorithms provided by Stable Baselines 3. In Table 2, we list the training hyperparameters for each environment, where unspecified hyperparameters are set to their default values.

Parameter	**Value**
	MINIGRID
Algorithm	DQN
Learning Rate	0.0001
Total Time Steps	50,000
Initial Time Steps	1,000
Update Frequency	10
Wrappers	FlatObsWrapper
	MOUNTAINCAR
Algorithm	SAC
Learning Rate	0.0003
Total Time Steps	100,000
Update Frequency	32
Entropy Coefficient	0.1
Gamma	0.9999
Tau	0.01
Gradient Steps	32
Hidden Sizes	64, 64
	PGTG
Algorithm	DQN
Wrappers	TimeLimit (100 steps), FlattenObservation
	BREAKOUT
Algorithm	PPO
Learning Rate	0.00025
Total Time Steps	10,000,000
Update Frequency	128
Entropy Coefficient	0.01
Value Function Coefficient	0.5
Wrappers	AtariWrapper
	ANT
Algorithm	PPO
Total Time Steps	1,000,000
Wrappers	NormalizeObservation, TimeFeatureWrapper
Link to the Evaluated Agent	huggingface.co/sb3/ppo-ant-v3
	HALF CHEETAH
Algorithm	SAC
Total Time Steps	1,000,000
Initial Time Steps	10,000
Wrappers	NormalizeObservation
	HUMANOID
Algorithm	SAC
Total Time Steps	2,000,000
Initial Time Steps	10,000
Parallel Environments	16
	HUMANOID STANDUP
Algorithm	SAC
Total Time Steps	2,000,000
Initial Time Steps	10,000
Parallel Environments	16

Table 2: Training hyperparameters for all environments.