Universiteit Leiden
The Netherlands

Master Computer Science

How Stable are Baselines: analyzing the consistency of Reinforcement Learning Baseline Frameworks

Name: Lex Janssens
Student ID: 2989344

Date: 28/08/2025

Specialisation: Artificial Intelligence

1st supervisor: Álvaro Serra-Gómez
2nd supervisor: Aske Plaat

Master's Thesis in Computer Science

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

There is a wide availability of Reinforcement Learning baseline frameworks, all of which are motivated by specific design choices. They contain easy-to-use state-of-the-art implementations of Reinforcement Learning algorithms. They serve as a consistent and reliable base for algorithm use and development. It is expected that the results produced by these implementations across different baseline frameworks remain consistent under the same setting.

However, due to the rapid development of these algorithms and baseline frameworks, combined with the emphasis on the reproducibility crisis in Reinforcement Learning, we question the consistency of the performance of these implementations across different baseline frameworks.

We will be comparing three common algorithms among three baseline frameworks to see whether the results remain consistent in similar settings. In this work, we thoroughly analyze these implementations on the code level and expose their differences. Furthermore, we address whether implementations are able to match the results of the original algorithms. Additionally, we investigate the consistency of the results obtained from implementations across different baseline frameworks. Lastly, we will compare the relative ranking of implementations and evaluate their preservation across baseline frameworks.

Our results show that the implementations of algorithms by baseline frameworks can match the performance of the original algorithms. However, code-level details among implementations may lead to significantly different performance under the same hyperparameter configuration. The relative ranking of implementations remains mostly preserved across baseline frameworks. This does not hold when comparing implementations using the default hyperparameters provided by that framework.

This work sheds light on a relatively new problem in the topic of RL reproducibility, showing how code-level details may cause two algorithm implementations to significantly differ from one another. We suggest reporting all hyperparameters, implementation details, and the source of the implementation to ensure reliable and consistent reproduction.

Contents

1 Introduction

Setting Deep Reinforcement Learning (DRL) focuses on Reinforcement Learning using neural networks. Since the rise of DRL, there have been numerous algorithms (Mnih et al., 2013; Lillicrap et al., 2015; Fujimoto et al., 2018; Haarnoja et al., 2018, 2019; Schulman et al., 2015a, 2017) designed and published. These publications often show how their newly designed algorithm performs differently from other algorithms. Researchers have published repositories (Raffin et al., 2021; Bou et al., 2023; Huang et al., 2022b; Weng et al., 2022a; Hoffman et al., 2020; Castro et al., 2018) with a collection of these DRL algorithms as a way to establish accurate and consistent baselines. Furthermore, they provide a way for consistent reproduction and easy application of state-of-the-art algorithms.

As of today, there exist dozens of baseline frameworks (Raffin et al., 2021; Bou et al., 2023; Huang et al., 2022b; Weng et al., 2022a; Hoffman et al., 2020; Castro et al., 2018; Guadarrama et al., 2018; Fujita et al., 2021a; Liang et al., 2018; Plappert, 2016) with a subset of replicated DRL algorithms. The reason for this large collection of baseline frameworks stems from their motivation: from focusing on a specific subset of DRL algorithms, to specific environment support, research-friendly usability, educationally written implementations, accessible prototyping of new algorithms, support of a specific machine learning backend, or large-scale deployment. The wide availability of baseline frameworks ensures there is one that matches each individual's requirements.

Of course, the expectation is that all algorithms, regardless of the origin of the baseline framework, perform similarly in the same environment using the same hyperparameter configuration. This is important since it establishes a universal ground truth defined by the algorithm, hyperparameters, and environment. The choice of a baseline framework is based solely on its design principles rather than its performance.

Problem However, with the many baseline frameworks published recently, we question the consistency of the performance of algorithm implementations across different baseline frameworks. A comparison of a novel algorithm to algorithm implementations that differ in performance across one another produces only relative results. This makes it challenging for researchers to find out whether an improvement found in a novel algorithm is legitimate. Without a common ground truth, every comparison between algorithms is relative. This can become a major problem, since many papers that introduce novel algorithms (Schulman et al., 2017; Haarnoja et al., 2018, 2019; Fujimoto et al., 2018) typically rely on, re-implement, or fine-tune algorithm implementations from the baseline repositories as a way to compare the difference in performance.

The reproducibility of the published results has become increasingly difficult in the RL research community (Henderson et al., 2018; Baker, 2016). Earlier research emphasized the importance of reporting all implementation details, hyperparameters, and statistics used within works (Henderson et al., 2018; Islam et al., 2017). While some baseline frameworks are specifically praised for their complete documentation (Raffin et al., 2021; Bou et al., 2023), they do not adapt this. Given the different motivations behind these baseline frameworks, and the lack of documentation of implementation details that may critically affect the performance, we question the performance consistency of the algorithms implemented across baseline frameworks.

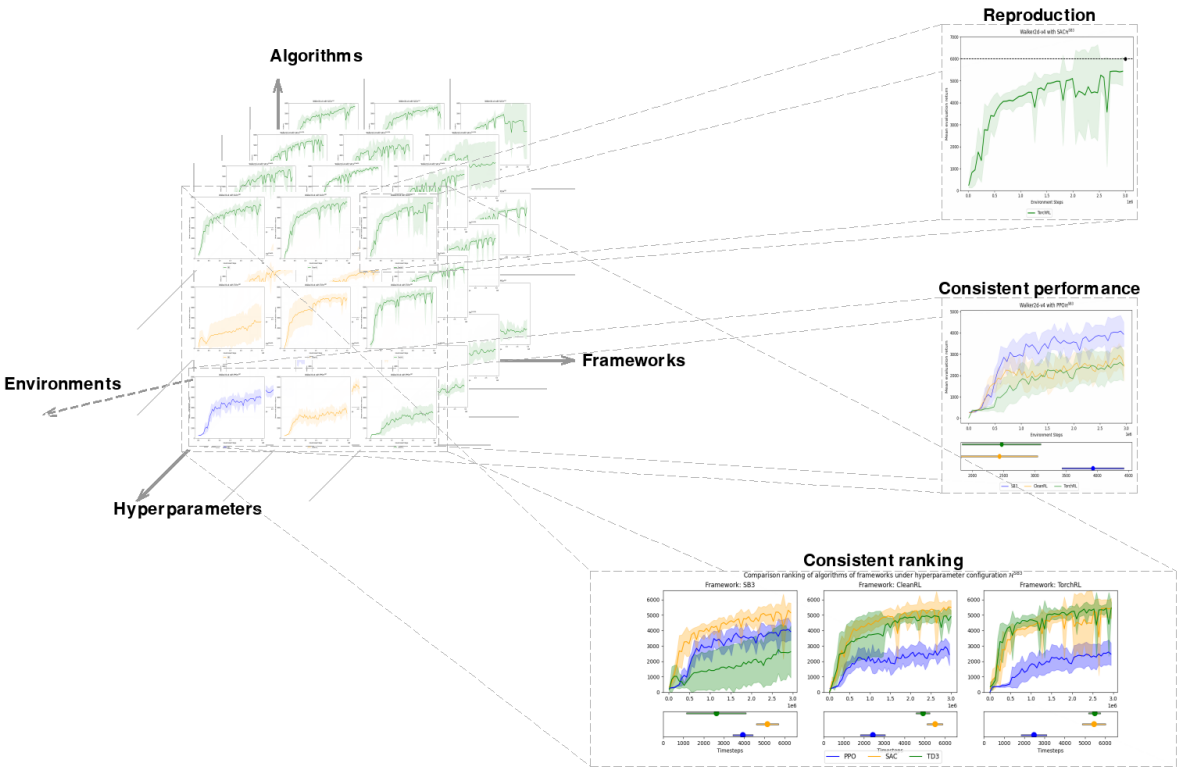


Figure 1: All variable dimensions that can be explored in this work that confirm consistency across frameworks. Each implementation should match the published algorithm results. Furthermore, implementations of the same algorithm under the same hyperparameter configuration should perform consistently. The relative ranking of algorithms across frameworks should similarly be consistent.

Earlier work by Hundal et al. (2025) has addressed the interchangeability of PPO implementations by different baseline frameworks. They applied PPO with the original hyperparameter configuration on 56 environments and found varying ratios of superhuman performance across implementations. They claim that code-level differences are responsible for inconsistencies in performance, but lack depth in their analysis. Our work covers three algorithms across three baseline frameworks using multiple hyperparameter configurations and a limited set of environments. This also allows us to expose the impact of the relative ranking of implementations' performances. Furthermore, we thoroughly analyze implementations and expose code-level differences.

Contribution In this work, we will address the comparability of implementations across multiple baseline frameworks. Figure 1 conceptually summarizes our work. We will look at code-level differences between implementations and how they may cause performance differences. Furthermore, we will be answering three research questions.

RQ1: *Are implementations of algorithms in baseline repositories able to match the results published by original papers?*

Research question **RQ1** focuses on the reproducibility of the original results of the implementations. This soundness check ensures that the implementations are replicas of the original algorithms. We will apply the hyperparameter configuration provided by the original paper to the implementations as closely as possible. We do not expect any significant differences between the results reported in the original algorithm paper and those of the implementations.

RQ2: *How consistent are the results obtained from implementations of algorithms across baseline repositories?*

Research question **RQ2** focuses on the comparability of implementations across different baseline frameworks. This reveals whether implementation details may lead to significantly different results. We will evaluate each algorithm implementation across the default hyperparameter configuration given by each baseline framework. By not focusing on a single hyperparameter configuration, we avoid the risk of using an implementation-specific, biased hyperparameter configuration. We expect that the implementation details of algorithms could result in distinctly different results.

RQ3: *How is the relative ranking of algorithms by a single baseline framework preserved against other baseline frameworks?*

Research question **RQ3** focuses on the preservation of the ranking of algorithm results across different baseline frameworks. Although individual algorithms might show divergent results compared to each other, it could still mean that the relative ranking of algorithms across baseline frameworks is preserved. If this is not the case, the impact of implementation details is so severe that different algorithms outperform each other. This may create a bias towards using a specific algorithm of a baseline framework in experimentation. We follow a similar methodology as **RQ2**, but instead compare the results of all implementations under the same default hyperparameter configurations for each algorithm. We expect that such high-level inconsistencies will not appear.

Results Our results highlight some of the code-level inconsistencies found among PPO (Schulman et al., 2017), TD3 (Fujimoto et al., 2018), and SAC (Haarnoja et al., 2018) implementations. These differences are often intended to stabilize training. Some of these differences reflect incorrect methodology or adaptations of the formulas introduced by the original papers. We can state that the implementations largely match the results published by the original papers. However, we observed inconsistent results across implementations. Implementation details and the way we aligned hyperparameters may help clarify some of these differences. This can result in bias and an inconsistent ground truth for the performance of an implementation under a specific hyperparameter configuration. Additionally, we saw that the relative ranking of the algorithms is mostly preserved. There are several experiments in which the magnitude of the differences varies, making some algorithms perform similarly in one experiment and differently in others. This does not apply to 'out-of-the-box' experimentation, where vastly different results emerge between framework-specific implementations of the same algorithms.

Structure This work is structured as follows. First, we review the related work in Section 2, discussing the reproducibility crisis and differential testing. Next, we describe the notation, engineering tricks, baseline frameworks, algorithms, and environments in Section 3. We then

describe our experiments in Section 4. We show a thorough code-level analysis in 5 and our results in Section 6. We conclude this work with a discussion in Section 7, where we highlight limitations, possible future extensions, and draw conclusions to this work.

2 Related Work

This work addresses the consistency among baseline frameworks. To get an overview of the current situation, we will discuss the reproducibility challenge in DRL. Furthermore, we will address works related to differential testing in DRL.

Reproducibility Reproducibility in Machine Learning is the property of being able to get similar results as presented in the research through a similar methodology. However, reproducibility has become a problem for both other researchers and the original authors (Baker, 2016; Henderson et al., 2018). Especially in Reinforcement Learning, issues such as non-determinism contribute to the difficulty of reproducibility (Nagarajan et al., 2019; Semmelrock et al., 2023). Additionally, due to work-in-progress code bases or to maintain a competitive advantage, code is not always published. This creates challenges for reproduction, as papers often use concise pseudocode and omit algorithmic details (Hutson, 2018). Furthermore, Pineau et al. (2021a) and Pineau et al. (2021b) categorize three problems that arise when addressing reproducibility: insufficient hyperparameter exploration, lack of documentation (code, data, methodology) for the published results (Gundersen and Kjensmo, 2018), and omission of, biased, or selective statistical analysis (Forde and Paganini, 2019; Henderson et al., 2018).

Regarding hyperparameter exploration, Melis et al. (2017) denotes that since the widespread use of neural networks, the deviations in the performance of old algorithms compared to new algorithms are getting smaller. Their work shows that well-established algorithms show performance comparable to more recent algorithms when hyperparameter optimization is applied. They want to emphasize that the methodology of newer algorithms must be explained thoroughly. Earlier work has shown that replication may yield diverse results due to omitted implementation details Tucker et al. (2018).

Similarly, Henderson et al. (2018) argues that the hyperparameter configuration used within the new implementation should match the configuration of the original performance. They emphasize that the results of the implementation should match the original algorithm. Islam et al. (2017) sees varying performance when altering key hyperparameters. They observe that baseline comparisons are highly dependent on the chosen hyperparameter configuration. They advise sticking with a well-working or close-to-optimal set of hyperparameters. *We stick to the hyperparameters provided by the baseline framework implementations, under the assumption that they generally perform well across tasks.*

It is important to sufficiently repeat experiments to draw conclusions on differences in performance between two algorithms, since experiments may exhibit high variance. For example, Colas et al. (2018) states that a minimum of 20 repetitions is needed, along with a tight statistical significance level, to achieve a rational false-positive rate when using the bootstrap test. Furthermore, both Henderson et al. (2018) and Islam et al. (2017) denote how two subsets of

results from the same experiment produce very different performance distributions. In addition to indicating the average return and standard deviation of all experiments, they also highlight the importance of reporting the number of repetitions. *We make use of 5 repetitions, which is considered low, but sufficient to summarize performance. This is due to computational constraints. Furthermore, we report both the evaluation averages and 95% confidence intervals at the end of training and during training in the form of learning curves, which indicate stability, in line with the key observations in Machado et al. (2018).*

Lastly, in line with the work of Melis et al. (2017), Henderson et al. (2018), and Islam et al. (2017), it is important not only to report all hyperparameters used but also all implementation details and experimental setup, together with the evaluation method used. The latter is of importance, as more researchers only publish the top-N repetitions, such as Wu et al. (2017). Additionally, Khetarpal et al. (2018) illustrates how an evaluation pipeline might look for a fair and consistent evaluation through shared metrics. *We apply a greedy evaluation on a fixed number of episodes and report the average return, 95% confidence interval, and the number of repetitions when reporting these metrics.*

Even the selection of environments used to evaluate the algorithms may include bias. Henderson et al. (2018) argues that stability and desired behavior are factors that could lead to favorable results. Similarly, Islam et al. (2017) highlights the performance sensitivity of algorithms with optimized hyperparameters when comparing HalfCheetah-v1 and Hopper-v1 environments. Some environments are generally easier and therefore yield good performance across many hyperparameter configurations. Other environments are notoriously more difficult, requiring an algorithm with optimized hyperparameters to yield good performance. *In this work, we consider multiple environments to overcome this bias.*

Differential testing Differential testing is a method originally used to detect potential bugs in software. It is a method of testing two or more implementations of the same function or algorithm against the same input. This can expose deviations in output, which can reveal if a bug or a different methodology is involved.

The work of Dai et al. (2022) performed a form of differential testing between the machine learning backends TensorFlow and PyTorch with respect to GPU computation time and performance. They note that the performance of similar implementations using TensorFlow or PyTorch is relatively similar. In contrast, the work of Novac et al. (2022) notes how differences between TensorFlow and PyTorch may occur, but they highlight that the study is limited in scope. A similar study was conducted when PyTorch was utilized in Stable Baselines 3, in contrast to TensorFlow in Stable Baselines 2. #48 shows that the performance of A2C (Mnih et al., 2016) differs between implementations by Stable Baselines 2 and Stable Baselines 3 across some environments.

Similar to this work, the work of Nouwou Mindom et al. (2023) compares baseline frameworks in the context of software engineering and applications. They show that implementations across baseline frameworks may already differ in performance. They argue that this is due to hyperparameter availability: the more hyperparameters a baseline framework algorithm offers, the more options for optimization can be utilized. In line with this, the work of Andrychowicz et al. (2021) shows how impactful implementation and hyperparameter-value differences are.

There are not many studies that compare baseline frameworks. The work of Hundal et al. (2025) is most related to this work. They observe that researchers often assume that differences in results are negligible and, as a result, assume that implementations of algorithms are interchangeable. However, as shown by Engstrom et al. (2019), small differences in the implementation of an algorithm can produce different results. Therefore, Hundal et al. (2025) questioned the interchangeability of PPO implementations by applying the original hyperparameter configuration to PPO implementations by five baseline frameworks. They used 56 environments of the ALE (Bellemare et al., 2013) suite. They found that there is a significant difference in the ratio of environments where superhuman performance is achieved. Specifically, SB3 (Raffin et al., 2021), CleanRL (Huang et al., 2022b), and OpenAI Baselines (Dhariwal et al., 2017) showed superhuman performance in 50% of the experiments, while RLLib (Liang et al., 2018) and Tianshou (Weng et al., 2022a) showed superhuman performance in less than 15% of the experiments. Furthermore, they highlight that the difference in the three well-performing implementations derives from implementation details. From their observations, they conclude that implementations across different baseline frameworks are not interchangeable. *Although similar, our work extends upon Hundal et al. (2025) by experimenting with multiple algorithms (on a smaller selection of baseline frameworks) and multiple sets of hyperparameters, the use of a widely used environment suite, a more in-depth analysis of code-level differences, and an overview of how these inconsistencies may affect the relative performance of algorithms.*

3 Preliminaries

In this section, we will first go over the notation used within this work, including engineering tricks applied in algorithm implementations. Furthermore, we will then address baseline frameworks, and highlight which frameworks we will use within this work. Additionally, we describe the algorithms used within this work. Finally, we will cover details of the environment suite used within this work.

3.1 Notation

Reinforcement Learning (RL) is a part of Machine Learning (ML) that deals with sequential decision-making tasks in the form of a Markov Decision Process (MDP) (Puterman, 1994). We denote an MDP as a 6-tuple $\{\mathcal{S}, \mathcal{A}, P, r, p_0, \gamma\}$, denoting the state space $s \in \mathcal{S}$, action space $a \in \mathcal{A}$, transition function $P : \mathcal{S} \times \mathcal{A} \to p(\mathcal{S})$, reward function $r : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$, initial state distribution $p_0 : \mathcal{S} \to \mathbb{R}$, and discount factor $\gamma \in (0, 1)$. We define a trace as $\tau := \{s_0, a_0, r_0, s_1, a_1, r_1, s_2, \ldots, s_T\}$, where T is the final timestep of the trace, satisfying a terminal or truncation condition. The cumulative discounted reward from timestep t is defined as $G_t(\tau) := \sum_{i=t}^{T} \gamma^{i-t} r_i$. We denote the return as $R_t := G_0$.

Reinforcement learning aims to maximize the objective $J(\theta) = \mathbb{E}_{\tau \sim p_{\pi_\theta}(\tau)}[R(\tau)]$, where $p_{\pi_\theta}(\tau) = p_0(s_0) \prod_{t=0}^{T-1} \pi_\theta(a_t|s_t) P(s_{t+1}|s_t, a_t)$ is the distribution of traces under policy π_θ. A parameterized policy $\pi_\theta : \mathcal{S} \to \mathcal{A}$ yields a conditional distribution over $\mathcal{A}$, which is categorical for discrete $\mathcal{A}$ and a Gaussian distribution $\mathcal{N}$ with mean $\mu_\theta(s)$ and standard deviation $\sigma^2_\theta(s)$ for continuous $\mathcal{A}$. Additionally, we denote the parameterized expected state-value function as

$V^{\phi}(s) := \mathbb{E}_{\tau_t \sim p(\tau_t)}[\sum_{i=0}^{T} \gamma^i r_{t+i} | s_t = s]$, the parameterized expected state-action value function (the Q-function) as $Q^{\phi}(s,a) := \mathbb{E}_{\tau_t \sim p(\tau_t)}[\sum_{i=0}^{T} \gamma^i r_{t+i} | s_t = s, a_t = a]$, the advantage function as $A_t^{\phi}(s_t, a_t) = Q^{\phi}(s_t, a_t) - V^{\phi}(s_t)$, and the entropy regularization (Mnih et al., 2016; Shannon, 1948) definition as $H(\pi_\theta(\cdot|s)) := \mathbb{E}_{a \sim \pi_\theta(a|s)}[-\log \pi_\theta(a|s)]$.

We are working with multiple algorithms and multiple hyperparameter configurations. To denote that we use, for instance, PPO from a specific framework, we use PPO^{FW}, where FW denotes the framework. Similarly, to denote a hyperparameter configuration from a specific framework, we use $\mathcal{H}^{FW}$. As an example, we denote SB3's PPO implementation using CleanRL's default hyperparameter configuration as $\text{PPO}^{\text{SB3}}(\mathcal{H}^{\text{CleanRL}})$, implicitly denoting that $\mathcal{H}^{\text{CleanRL}}$ refers to the PPO hyperparameters of said algorithm.

All notation used in this work can be found in Appendix A.

3.2 Engineering tricks

We will introduce some engineering tricks used within algorithm implementations beyond the pseudocode provided by the original papers.

Orthogonal initialization Orthogonal initialization initializes the weights of a neural network as orthogonal vectors. A scale factor adjusts the magnitude of the weights. A bias vector may be included to avoid dead neurons. Orthogonal initialization can be used to give the model a biased, positively influenced head-start on learning. Engstrom et al. (2019) has shown that orthogonal initialization has beneficial effects for training over other initialization methods.

Actionspace enforcement To ensure $a_i \in \mathcal{A}_i$, $i \in \{0, 1, \ldots, \dim \mathcal{A} - 1\}$, we define several functions for this purpose. Suppose actionspace bounds $[\mathcal{A}_{i,\text{low}}, \mathcal{A}_{i,\text{high}}]$ for each dimension i of $\mathcal{A}$. There are two approaches to enforce $a_i \in [\mathcal{A}_{i,\text{low}}, \mathcal{A}_{i,\text{high}}]$ for all dimensions i of $\mathcal{A}$. The first method is hard clipping via $\text{clip}(a_i, \mathcal{A}_{i,\text{low}}, \mathcal{A}_{i,\text{high}}) = \min(\max(a_i, \mathcal{A}_{i,\text{low}}), \mathcal{A}_{i,\text{high}})$. The second method is applying the hyperbolic tangent function $\tanh$ to enforce $a_i \in [-1, 1]$, and then proportionally unscale to meet the actionspace boundaries via $\text{unscale}(a_i) = \mathcal{A}_{i,\text{low}} + \frac{1}{2}(a_i + 1)(\mathcal{A}_{i,\text{high}} - \mathcal{A}_{i,\text{low}})$. Scaling via $\text{scale}(a_i) = 2\frac{a_i - \mathcal{A}_{i,\text{low}}}{\mathcal{A}_{i,\text{high}} - \mathcal{A}_{i,\text{low}}} - 1$ or unscaling are auxiliary functions that do not directly enforce action space boundaries. Both methods can work with an asymmetric $\mathcal{A}$. See Appendix C.3 for an experiment illustrating how these two methods may produce differences throughout training.

Minibatches Minibatches are a way to split the training data of a large batch into smaller batches. Off-policy methods use a replay buffer containing past experiences. When updating, uniform sampling is done over the replay buffer to make sure the batch is decorrelated. On-policy methods directly use gathered experiences. They sample replacement to create smaller minibatches. All samples of a batch are used by an on-policy method.

Generalized Advantage Estimation The bias-variance trade-off refers to how returns are estimated. REINFORCE (Williams, 1992) uses a full Monte Carlo rollout, which produces

low bias but high variance. Temporal Difference learning, such as TD(1), reduces variance by utilizing a value function, at the cost of introducing bias. To balance this trade-off, Schulman et al. (2015b) proposed Generalized Advantage Estimation (GAE).

Suppose performance measure $\Psi_t = \hat{A}_t(s_t, a_t) = Q(s_t, a_t) - V(s_t)$. Schulman et al. (2015b) proposes an estimation for $Q^\pi(s, a)$ to use variable-horizon exponentially-weighted Temporal Difference (TD) estimations. Advantage estimation for TD(1) is calculated as $\hat{A}_t^{(1)} := \delta_t = r_t + \gamma V^\phi(s_{t+1}) - V(s_t)$. This can be generalized for a TD(n) horizon: $\hat{A}_t^{(n)} = \delta_t + \gamma\delta_{t+1} + \gamma^2\delta_{t+2} + \cdots + \gamma^{n-1}\delta_{t+n-1} = \sum_{i=0}^{n-1} \gamma^i \delta_{t+i}$. GAE is the weighted summation of these advantages, formulated as

$$\hat{A}_t^{GAE} := (1-\lambda)(\hat{A}_t^{(1)} + \lambda\hat{A}_t^{(2)} + \lambda^2\hat{A}_t^{(3)} + \dots) = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l} \tag{1}$$

It introduces a new hyperparameter $\lambda \in (0, 1]$, which controls the bias-variance trade-off. The closer to 0, the higher the variance. Andrychowicz et al. (2021) saw improvement of performance using GAE over n-step returns.

To make sure that advantages are calculated sequentially but not across episodes, an identity condition is added, such that the definition of δ_t becomes

$$\delta_t = \begin{cases} r_t + \gamma \cdot V^\phi(s_{t+1}) - V^\phi(s_t) & \text{if } s_{t+1} \neq s_T \\ r_t - V^\phi(s_t) & \text{otherwise} \end{cases} \tag{2}$$

The correct calculation of δ_t should ensure that $s_t = S_T$ is only true for terminal s_T, such that proper bootstrapping is still done for states where a truncation condition is satisfied.

Truncation handling Truncation is a condition that ends an episode via non-terminal conditions, such as time limits or constraints. The last observed state s' is not terminal, and thus its state-value must be properly estimated.
The difference between bootstrapping and not bootstrapping yield $\delta_t = r_t + \gamma V^\phi(s_{T+1}) - V^\phi(s_T)$ and $\delta'_t = r_t - V^\phi(s_T)$, respectively. The non-stationarity of the expected state-value function will produce a bimodal action distribution. This results in sub-optimal policies, because the mean of a bimodal action distribution lies between two disjoint regions. It is said not to affect training much (Huang et al., 2022b). This is because the effects of improper bootstrapping often only occur when the agent is well-performing and fulfills a time-limit truncation condition. However, Pardo et al. (2017) illustrates how this leads to unstable learning and suboptimal policies.

Hyperparameter annealing Hyperparameter annealing is often done to stabilize training. Linear annealing is done by multiplying the initial hyperparameter value by the $1 - \frac{t}{T}$, the proportion of timesteps remaining given total training timesteps T. This is often applied to impactful hyperparameters, such as the learning rate. This may yield positive results (Engstrom et al., 2019), although the benefits are empirically shown to be small (Andrychowicz et al., 2021).

Advantage normalization Advantage normalization refers to rescaling the advantages for a batch of experiences such that they have a mean of 0 and unit variance. This stabilizes training and may improve learning speed. Given advantage estimates in a batch B as $\{\hat{A}\}_{\hat{A}\in B}$, normalization is done via

$$\hat{A}' \leftarrow \frac{\hat{A} - \mu(\{\hat{A}\}_{\hat{A}\in B})}{\sigma(\{\hat{A}\}_{\hat{A}\in B}) + 10^{-8}} \tag{3}$$

Network update ratios The network update ratio is the ratio of environment interactions to the number of gradient steps that follow. A gradient step may use a batch of samples per update. A ratio of 1:1 refers to one environment interaction to one gradient step. This causes stable learning at the cost of learning speed. A ratio of 1000:1000 is more data-efficient and faster, but may destabilize training. Skewed ratios (e.g., 2:1) delay network updates while performing fewer gradient steps overall.

Polyak averaging Polyak averaging (Polyak, 1964) is used to incorporate slow updates of the target networks. Polyak averaging is performed via $\theta' \leftarrow \tau_p\theta + (1-\tau_p)\theta'$, for network parameters θ, target network parameters θ'. A hyperparameter $\tau_p \in [0, 1]$, whose value is typically very small, controls the weight of the averaging between the two network parameters.

Standard deviation control To ensure stability and exploration, the value $\log \sigma^2_\theta(s)$ retrieved from a policy network may be bounded to a domain $[\log \sigma^2_{\theta,\min}, \log \sigma^2_{\theta,\max}]$. Preventing $\log \sigma^2_\theta(s)$ from becoming too small maintains exploration, while preventing it from becoming too large ensures stability.

3.3 Baseline Frameworks

A baseline framework is a collection of RL algorithm implementations, specifically designed for easy use, and often labeled reliable, well-documented, and consistent. Algorithms in this set can be utilized as baselines, allowing for easy reproduction, and providing a way to design new algorithms. It is thus important that the baseline used in these scenarios is clear and that any changes in expected behavior are properly traced back.

There are many (Hoffman et al., 2020; Kolesnikov, 2018; Huang et al., 2022b; Caspi et al., 2017; Niu et al., 2021; Fujita et al., 2021a; Dhariwal et al., 2017; D'Eramo et al., 2021; Plappert, 2016; Zheqing Zhu, 2023; Hill et al., 2018; Bou et al., 2023; Raffin et al., 2021; Weng et al., 2022a) baseline frameworks, all with their own motivation for development. Some baseline frameworks aim to educate the user through simplified and well-documented code bases (Huang et al., 2022b; Achiam, 2018). Other baseline frameworks aim for ease of use and adaptable algorithms via a wide variety of modular building blocks (Dhariwal et al., 2017; Hill et al., 2018; Raffin et al., 2021; Kuhnle et al., 2017; Zheqing Zhu, 2023; Liu et al., 2021a; Hoffman et al., 2020; Niu et al., 2021; garage contributors, 2019; Stooke and Abbeel, 2019; Guadarrama et al., 2018; Fujita et al., 2021b; D'Eramo et al., 2021; Bou et al., 2023; Weng et al., 2022a). Baseline frameworks may arise on the basis of offering large-scale experimentation via distribution methods (Liang et al., 2018; Zhi et al., 2020; Fan et al., 2018; Hoffman et al., 2020; Kolesnikov, 2018; Küttler et al., 2019; Nota, 2020), and some baseline frameworks support a wide variety of algorithms and/or environments

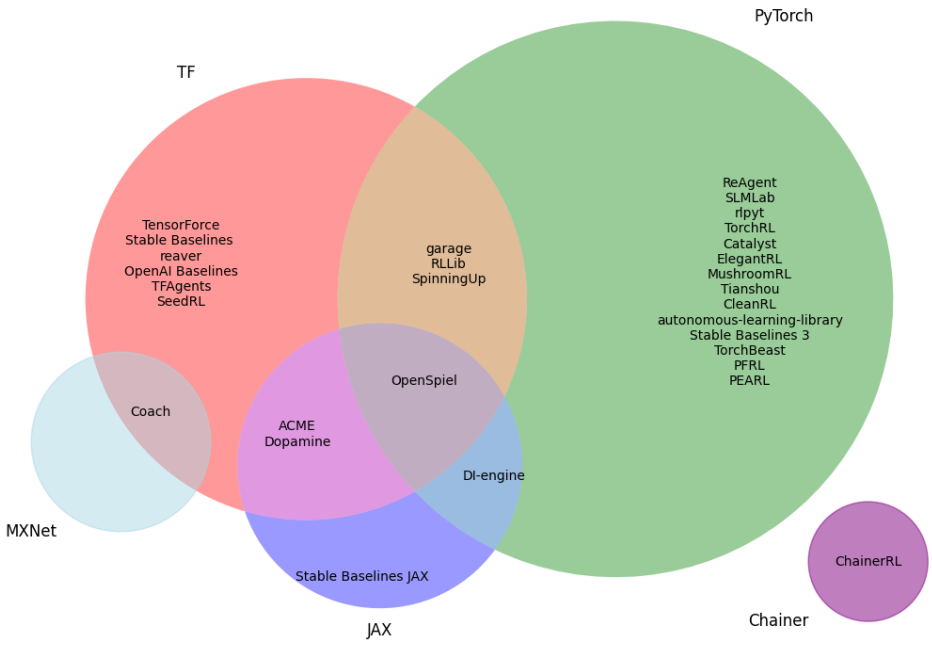


Figure 2: A Venn diagram showing the different machine learning backends used in a set of baseline frameworks. Some frameworks are built around a specific machine learning backend.

(Ring, 2018; Lanctot et al., 2020; Zheng et al., 2017; Niu et al., 2021). There are large differences in the support for machine learning backends in frameworks, as seen in Figure 2. Older frameworks used TensorFlow (TF) (Abadi et al., 2016) as a deep learning backend, but this has gradually shifted to PyTorch (Paszke et al., 2019). Currently, integration with the backend JAX (Bradbury et al., 2018) is gaining popularity due to its fast computation.

Educational implementations focus on readable codebases and often contain single-file standalone implementations. They are generally well-documented, and their main purpose is to expose the core mechanism of the algorithms to the user. This is done by highlighting what is necessary and omitting optimization or engineering tricks. Furthermore, these frameworks serve to extend existing algorithms, either by improving them or developing novel algorithms. This is easily accomplished because of their exposed, easily understandable, non-modular design. Examples of such baseline frameworks are SpinningUp (Achiam, 2018) and CleanRL (Huang et al., 2022b). Another benefit of compact implementations is comprehensibility. It allows for quick usage and enables rapid prototyping, as seen in Dopamine (Castro et al., 2018), or it may contribute to research-friendly building blocks, as seen in MushroomRL (D'Eramo et al., 2021).

Some baseline frameworks focus on the accessibility of adjustments that can be made to the algorithm. The architecture of such baseline frameworks often holds a modular design. Each part of the algorithm — the agent and the experimental loop — may be pre-built or constructed by the user via flexible class objects. Due to this flexibility, these baseline frameworks are often optimized and provide access to commonly used engineering tricks. Baseline frameworks such as OpenAI Baselines (Dhariwal et al., 2017), Stable Baselines (Hill et al., 2018), Stable

Baselines 3 (Raffin et al., 2021), TensorForce (Kuhnle et al., 2017), PEARL (Zheqing Zhu, 2023), ElegantRL (Liu et al., 2021a), and acme (Hoffman et al., 2020) have predefined agents and include an experimentation loop in the form of a complete class. Using these algorithms requires only a few lines of code. Their modular design makes it easy to read and adjust. Therefore, using an algorithm as a baseline does not require expert knowledge. Frameworks such as DI-engine (Niu et al., 2021), (garage contributors, 2019), rlpyt (Stooke and Abbeel, 2019), TF-Agents (Guadarrama et al., 2018), PFRL (Fujita et al., 2021b), MushroomRL (D'Eramo et al., 2021), and TorchRL (Bou et al., 2023) use a more modular approach, offering pre-made building blocks to design the agent and/or algorithm. Tianshou (Weng et al., 2022a) offers high- and low-level implementations of state-of-the-art baseline algorithms.

For large-scale experimentation and production-level workloads, several frameworks are aimed at supporting distributed RL. The frameworks' focus often relies on optimization of code and throughput via distribution methods, including multi-CPU, GPU, or support for multiple nodes. The use of these frameworks is often simple and accessible to the user via high-level abstractions. The user is discouraged from diving into the code base due to its complexity. Examples of these frameworks are RLLib (Liang et al., 2018), Fiber (Zhi et al., 2020), SURREAL (Fan et al., 2018), acme (Hoffman et al., 2020), Catalyst (Kolesnikov, 2018), and TorchBeast (Küttler et al., 2019). The term 'large-scale' can also apply to SLURM jobs, such supported by autonomous-learning-library (Nota, 2020).

Some frameworks were designed around specific niches. SLMLab (Keng and Graesser, 2017), as one of the first DRL baseline frameworks, focused on full accessibility, including parallelization and hyperparameter optimization. Furthermore, Arena (Wang et al., 2019) and MAgent (Zheng et al., 2017) are platforms that enable multi-agent RL algorithms. Additionally, the support for a smaller, more complex set of environments is provided in frameworks such as reaver (Ring, 2018), OpenSpiel (Lanctot et al., 2020), and ELF (Tian et al., 2017). Lastly, some frameworks are motivated by their support for a specific machine learning backend, such as coach (Caspi et al., 2017), ChainerRL (Fujita et al., 2021a), Stable Baselines JAX (Raffin et al., 2021), and KerasRL (Plappert, 2016).

Many frameworks are no longer supported. Furthermore, new baseline frameworks have been developed on top of older frameworks, which makes the predecessor irrelevant. Figure 3 shows the timeline of when these baselines were released until they were last updated. In this work, we will focus on three baseline frameworks. They are chosen based on their impact on the literature, as can be seen in Table 1. The frameworks are maintained and differ from each other in terms of motivation and design. Furthermore, each of the frameworks should support an overlapping set of algorithms, for which a subselection can be seen in Table 2. Based on these specifications and the frameworks' design structures, our experiments focused on Stable Baselines 3, CleanRL, and TorchRL.

3.3.1 Stable Baselines 3

Stable Baselines 3 (SB3) (Raffin et al., 2021) is a baseline framework that focuses on accessibility and reliability by finding a balance between modularity and educability. The predecessor of SB3 is Stable Baselines 2 (Hill et al., 2018), with the main difference being the complete rewrite to use PyTorch as opposed to TensorFlow, to improve understandability.

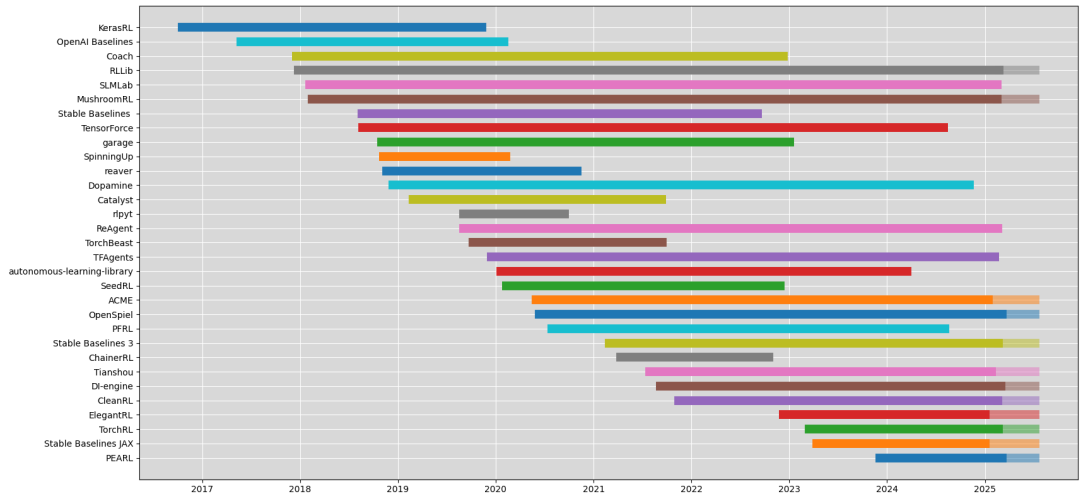


Figure 3: A timeline of baseline frameworks in terms of first release and last update. Shaded areas up to now are frameworks that are still regarded as maintained.

Framework	Citations	Stars	Contributors	Issues	PR	Forks
Stable Baselines 3	2884	10100	167	55/1461	18/544	1800
CleanRL	340	6500	47	52/141	25/279	719
TorchRL	42	2600	166	174/498	88/2056	347
RLLib	1150					
Tianshou	253	8300	86	145/614	5/447	1100
acme	275	3600	66	63/205	12/43	455
OpenSpiel	305	4400	182	22/548	14/709	970
DI-Engine		3300	53	6/210	13/589	395
PEARL	6	2800	17	13/48	0/48	179

Table 1: Impact of baseline frameworks. Citations refer to the citations of the paper published along with the framework. Stars, contributors, issues (open/closed), Pull requests (PRs) (open/closed), and forks refer to the GitHub of this framework. RLLib is part of the Ray repository, making GitHub statistics disproportionate. DI-Engine has no paper revolving around the framework specifically.

A comparative study is performed to see whether SB3 and SB2 match performance, in which they conclude that they are relatively equal. Additionally, there exists a proof-of-concept version of Stable Baselines 3 in JAX.

The library is high-level and concise, and only considers algorithms that are easy to use and maintain. This results in a small collection of state-of-the-art single-agent and model-free algorithms. However, acknowledging that this collection is compact, Stable Baselines 3 Contrib allows externally implemented algorithms that fall out of the design principles of Stable Baselines 3. The code base is fully documented and type-hinted, as the authors aim for readability and modifiability. Furthermore, it has a simple-to-use API that provides

Algorithm	SB3	CleanRL	TorchRL	Tianshou	acme	DI-engine	PEARL
DDPG	✓	✓	✓	✓	✗	✓	✓
DQN	✓	✓	✓	✓	✓	✓	✓
PPO	✓	✓	✓	✓	✓	✓	✓
SAC	✓	✓	✓	✓	✓	✓	✓
A2C	✓	✗	✓	✓	✗	✓	✗
TD3	✓	✓	✓	✓	✓	✓	✓

Table 2: State-of-the-art algorithms implemented by maintained impactful frameworks.

complete agents, along with network initialization (with options for feature extraction) and an experimentation loop. Plug-and-play usage is supported for all environments implemented in the Gymnasium environment framework. Additionally, Raffin (2020) provides pretrained agents trained in PyBullet environments, along with training, evaluation, hyperparameter tuning, and visualization options.

The algorithms provided by Stable Baselines 3 are often utilized within studies where the task is aligns with the structure of the Gymnasium framework, such as the works of Schäfer et al. (2024); Liu et al. (2021b); Lu (2023); Zhang et al. (2021). Other works have focused on comparing algorithms of Stable Baselines 3 on a single task, such as the works of De La Fuente and Guerra (2024); Silvestri et al. (2023); Babic (2024). The work of Wolk et al. (2022) focuses on a single task, but uses PPO methodologies from both SB3 and RLLib (Liang et al., 2018) interchangeably. Furthermore, the work of Benad et al. (2025) shows comparable performance between CleanRL and SB3 algorithms. Lastly, the paper by Young and Pugeault (2024) explores improving robustness by using SB3 methods. Overall, the fact that SB3 can be quickly deployed for experimentation makes it an appealing baseline framework to use.

3.3.2 CleanRL

Like SpinningUp (Achiam, 2018), CleanRL (Huang et al., 2022b) is designed to provide educational value. Instead of focusing on modular building blocks or single-class agents, it focuses on single-file implementations. Files are isolated for different algorithms, machine learning backends, action spaces, and environment suites. Overall, this helps the user understand the implementation at the cost of code duplication and maintenance. However, this design allows for quick debugging when designing novel algorithms. The educational value, along with easy access to algorithm development, is the core motivation of CleanRL.

Along with a wide variety of algorithm implementations, Huang et al. (2022b) also allows for extensive logging and visualizations by Weights and Biases (Biewald, 2020), and offers advanced modules for task-general hyperparameter optimization capabilities. Furthermore, it supports cloud integration via Amazon Web Services (AWS) for large-scale experimentation.

The motivation for the usage of CleanRL varies; it is driven by their high throughput, accessibility, and readability (Wan et al., 2023), integration with the Atari Learning Environment suite (ALE) (Bellemare et al., 2013), and the ability to easily adapt neural networks (Schiller, 2023). Furthermore, Suarez et al. (2023) shows that CleanRL has proven itself to be useful

in complex multi-agent environment settings by the overarching framework PufferLib (Suarez, 2024). Lastly, in the work by Huang et al. (2023), CleanRL is effective as the foundation of a framework that allows distributed methods. Although these works show high diversity in the use of CleanRL, its general use case is to be the foundation for new methodologies.

3.3.3 TorchRL

TorchRL (Bou et al., 2023) is a highly modular baseline framework, built on the motivation of full incorporation of the PyTorch machine learning framework and minimal other dependencies. It has been observed that there is a trade-off between offering functionality and complying with their restrictions. This ultimately affects the reusability of code. The idea that modular components can be reused, without too many usage restrictions, both enhances user experience and allows for easy experimentation. Through their concurrently introduced TensorDict data structure, they enable a new form of communication between components.

TorchRL offers a wide selection of modular components, such as network initialization, action selection, actors, critics, entropy regularization, loss functions, data collectors, environment setups, and replay buffers. The experimentation loop is not included within this modularity and must be defined externally. Although state-of-the-art example algorithms are provided, designing new algorithms or setting up experimentation would require expert knowledge in both RL and the PyTorch framework. Bou et al. (2023) recognizes Tianshou (Weng et al., 2022a) as a similar library, although they have different motivations: TorchRL is focused on algorithm development and stimulates users to explore this modularity within their experimentation, while Tianshou focuses on simplification of problem-solving.

TorchRL has been utilized in several research studies, such as generative chemical design (Bou et al.), a proposal on spacecraft navigation (Getzandanner and Martin, 2025), human-AI interaction (Zhang et al., 2024), a multi-agent autonomous vehicle framework (Akman et al., 2025), and LEGO robot training (Dittert et al., 2024). Reasons to use TorchRL often originate from its tight integration in the PyTorch framework and ease of use. Furthermore, (Tiapkin et al., 2024) motivates TorchRL's usage due to its high modularity. Lastly, the use of the TensorDict data structure is also cited as a motivation for the use of TorchRL (Berto et al., 2023; Flowers and Dey, 2024).

3.4 Algorithms

The algorithms we chose in this work are based on popularity and implementation availability. This selection includes PPO (Schulman et al., 2017), TD3 (Fujimoto et al., 2018), and SAC (Haarnoja et al., 2018).

3.4.1 PPO

Policy gradient methods are a class of methods that focus on optimizing a parameterized and differentiable stochastic policy π_θ directly through an objective $J(\theta)$, so that the policy parameters θ are updated proportionally to the gradient, i.e. $\nabla\theta \approx \alpha_\pi \frac{\delta J(\theta)}{\delta\theta}$ (Sutton et al., 1999), where α_π is the policy learning rate. Objective $J(\theta)$ is a measure of the performance of the agent, and as such, the objective is to maximize the expected return $J(\theta) = \mathbb{E}_{\tau\sim p_{\pi_\theta}}[R(\tau)]$. From the Policy Gradient Theorem, it follows that the gradient of $J(\theta)$ yields $\nabla_\theta J(\theta) =$

$\mathbb{E}_{\tau\sim p_{\pi_\theta}}\left[\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_t|s_t)\cdot G_t\right]$ [1], which is the basis of REINFORCE (Williams, 1992). By generalizing $\nabla_\theta J(\theta)$ to include a performance measure Ψ_t, we obtain

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim p_{\pi_\theta}}\left[\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_t|s_t)\cdot \Psi_t\right] \tag{4}$$

This versatile formulation allows for stability improvements through, among others, bootstrapped n-step target ($\Psi_t := \sum_{i=t}^{n-1}\gamma^i r_i + \gamma^n V^\phi(s_n)$) or baseline subtraction ($\Psi_t := \sum_{i=t}^{T}\gamma^i r_i - V^\phi(s_t)$).

Trust-Region Policy Optimization (TRPO) (Schulman et al., 2015a) is considered as the predecessor of PPO (Schulman et al., 2017). As the name suggests, TRPO uses trust regions in order to reduce the variance in policy methods. Trust regions contain policy updates to ensure stability. In the case of TRPO, this idea is adopted such that it tries to improve the policy as much as possible, while the difference between the old and new policy does not exceed a certain threshold. Given probability ratio $r(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$ as the divergence between π_θ and $\pi_{\theta_{\text{old}}}$ for some data, the objective of TRPO can be defined as

$$\max_\theta J(\theta) = \hat{\mathbb{E}}_t\left[r(\theta)\cdot \hat{A}_t\right] \quad \text{s.t.} \quad \hat{\mathbb{E}}_t\left[\mathsf{KL}\left[\pi_{\theta_{\text{old}}}(\cdot|s_t), \pi_\theta(\cdot|s_t)\right]\right] \leq c \tag{5}$$

where KL refers to the Kullback-Leibler Divergence (Kullback and Leibler, 1951) and c a threshold. This can also be formulated as an unconstrained quadratic problem,

$$\max_\theta J(\theta) = \hat{\mathbb{E}}_t\left[r(\theta)\cdot \hat{A}_t\right] - \beta\cdot \mathsf{KL}\left[\pi_{\theta_{\text{old}}}(\cdot|s_t), \pi_\theta(\cdot|s_t)\right] \tag{6}$$

where β is a Lagrange multiplier that alternatively weights this approximated constraint. Note that now, $\hat{A}_t$ is weighed and thus dependent on the probability ratio $r(\theta)$. Furthermore, Schulman et al. (2015a) calculated a fixed optimal value for β.

Schulman et al. (2017) addresses the problems occurring in TRPO, namely the difficulty of computing the constraint, and that a fixed β yields suboptimal behavior, even when tuned per problem. The main contribution of this paper is Proximal Policy Optimization with clipping (PPO-clip or PPO). Instead of using the KL divergence, PPO uses clipping on $r(\theta)$ such that it remains within bounds $[1-\epsilon, 1+\epsilon]$, i.e. $\mathsf{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)$, where ϵ is a hyperparameter controlling the maximum step size. Along with a pessimistic lower bound, the PPO objective combines the TRPO objective (without the constraint) with the surrogate clip objective to yield

$$J(\theta) = \hat{\mathbb{E}}_t\left[\min(r_t(\theta)\hat{A}_t, \mathsf{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)\right] \tag{7}$$

This objective is easier to calculate, has a less sensitive hyperparameter enforcing the update size, and strictly clips diverging policies.

The pseudocode described in Schulman et al. (2017) is concise but conceptually similar to their described formulation. Over time, improvements have been made in terms of ensuring stable training and scheduled exploration. In section 5.1, we will discuss these engineering tricks. A high-level pseudocode can be seen in Algorithm 1.

[1] We acknowledge that Sutton et al. (1999) uses R for every step t, but implies that later return estimates are influenced by rewards obtained prior to timestep t. For this, G_t resolves this issue

Algorithm 1 PPO (Schulman et al., 2017)

1: **while** total number of timesteps not exceeded **do**
2: Accumulate rollout of size n_{steps}
3: Compute advantage estimations $\hat{A}_1, \dots \hat{A}_{n_{steps}}$
4: Optimize surrogate formula 7 and update network parameters
5: **end while**

Loss calculation The PPO loss as introduced by Schulman et al. (2017) is cumulative, consisting of a policy loss, a value loss, and an entropy regularization term. The average probability ratio is calculated for a batch B of samples via

$$r_B(\theta) = |B|^{-1} \sum_{(s,a,\log \pi_{\theta_{old}}(a|s)) \in B} \exp(\log \pi_\theta(a|s) - \log \pi_{\theta_{old}}(a|s)) \tag{8}$$

The policy loss L^π is an average and is calculated according to the objective formula 7 via

$$L^\pi = -\min(r_B(\theta)\hat{A}, \mathsf{clip}(r_B(\theta), 1-\epsilon, 1+\epsilon)\hat{A}) \tag{9}$$

The value loss L^V is the Mean Squared Error (MSE) between $\{V^\phi(s)\}_{s \in B}$ and the $\{R\}_{R \in B}$:

$$L^V = (2|B|)^{-1} \sum_{R,s \in B} (R - V^\phi(s))^2 \tag{10}$$

Finally, the average entropy loss L^H is calculated as

$$L^H = |B|^{-1} \sum_{s,a \in B} -\pi_\theta(a|s) \log \pi_\theta(a|s) \tag{11}$$

This yields the cumulative loss $L = -L^\pi + c_1 L^V - c_2 L^H$, where c_1 and c_2 are hyperparameters controlling the weight of the value loss and entropy loss, respectively. Lastly, to prevent gradients from exploding during training, the gradients are clipped (Černỳ, 2008; Pascanu et al., 2013).

3.4.2 TD3

Another class of algorithms is the value-based methods. Instead of optimizing the policy directly, the policy is indirectly optimized through optimization of the value function. A fundamental algorithm within this class for DRL is the Deep Q-Network (DQN) algorithm (Mnih et al., 2013). This algorithm, used for environments with a discrete action space, adopts the fact that the optimal policy $\pi^*(s) = \arg\max_{a \in \mathcal{A}} Q^{\phi *}(s,a)$ is a product of the optimal Q-value function $Q^{\phi *}(s,a)$. The loss to be minimized is

$$L(\phi) = \mathbb{E}_{(s,a,r,s') \sim \mathcal{D}} \left[r + \gamma \max_{a'} Q^{\phi'}(s',a') - Q^\phi(s,a) \right] \tag{12}$$

where $\mathcal{D}$ is a replay buffer filled with past experiences, $Q^{\phi'}(s',a')$ a target Q-value network parameterized via ϕ' that is periodically updated, $Q^\phi(s,a)$ the Q-value network. The work of Mnih et al. (2013) addresses improvements, including the replay buffer and target network.

These address the issues of sample efficiency, correlation, and instability that arise from learning on a moving target.

Within DQN, the policy is implicit. However, moving towards a continuous action space, it is impossible to compute $\pi(s)$ via $\arg\max_{a \in \mathcal{A}} Q^{\phi}(s, a)$, since there are infinitely many a contained within $\mathcal{A}$. The policy $\pi(s)$ becomes part of the optimization process as a result. Note that $\pi_\theta(s)$ is still a deterministic policy. One fundamental formula is the Deterministic Policy Gradient (DPG) (Silver et al., 2014). This formula aims to maximize the objective

$$J(\theta) = \mathbb{E}_{s \sim \mathcal{D}} \left[Q^{\phi}(s, \pi_\theta(s)) \right] \tag{13}$$

where $\mathcal{D}$ is a replay buffer filled with past experiences. Given the fact that both Q^{ϕ} and π_θ are functions, one can use the chain rule to find the gradient to be

$$\nabla_\theta J(\theta) = \nabla_a Q^{\phi}(s, a)|_{a=\pi_\theta(s)} \cdot \nabla_\theta \pi_\theta(s) \tag{14}$$

Intuitively, $\nabla_a Q^{\phi}(s, a)|_{a=\pi_\theta(s)}$ indicates how much the return changes with respect to the action a, while $\nabla_\theta \pi_\theta(s)$ indicates how much action a changes with respect to parameters θ. In other words, the policy is improved by optimizing the value function (Moerland, 2021). This formula is implemented within the Deep Deterministic Policy Gradient (DDPG) algorithm (Lillicrap et al., 2015). Unlike DQN, which aims to fulfill the Bellman-Optimality equation, DDPG minimizes the Bellman error under the deterministic policy's Q-function.

DDPG is used as the foundation for the design of TD3 Fujimoto et al. (2018): Twin Delayed Deep Deterministic Policy Gradient. This work addresses the flaw of the overestimation bias that is present within Q-learning methods (Thrun and Schwartz, 2014; Sutton, 1988). Furthermore, it recognizes that Double Q-learning (Hasselt, 2010), a method to address overestimation bias using a pair of independently trained state-value networks, may still lead to high variance during training. Fujimoto et al. (2018) propose solutions for both of these problems. First, to address overestimation bias, they introduce Clipped Double Q-learning, formulated as

$$y = r + \gamma \min_{i=1,2} Q^{\phi'_i}(s', \pi_\theta(s')) \tag{15}$$

The intuition behind this approach is to avoid Q-value overestimation by taking the minimum Q-value estimation over two Q-value functions. More Q-value functions can be used, but a minimum of two are required. In this work, we will stick to two Q-value functions. The Q-value function $Q_{\phi'_1}$ is optimized directly, yielding an objective function

$$J(\theta) = \mathbb{E}_{s \sim \mathcal{D}} \left[Q^{\phi_1}(s, \pi_\theta(s)) \right] \tag{16}$$

with its gradient

$$\nabla_\theta J(\theta) = \nabla_a Q^{\phi_1}(s, a)|_{a=\pi_\theta(s)} \cdot \nabla_\theta \pi_\theta(s) \tag{17}$$

However, Q-function Q^{ϕ_1} may still be overestimating (i.e. when $Q^{\phi'_1}(s', \pi_\theta(s')) > Q^{\phi'_2}(s', \pi_\theta(s'))$). In that case, $Q^{\phi'_2}(s', \pi_\theta(s'))$ is being used as underestimation does not backpropagate unstable behavior. Similarly, when $Q^{\phi'_2}(s', \pi_\theta(s')) > Q^{\phi'_1}(s', \pi_\theta(s'))$, the update is no different from using a single Q-value function. Fujimoto et al. (2018) further proposes three methods to reduce high variance. First, instead of training the Q-value function on TD targets, future TD errors are subtracted from the value estimation. Second, similarly

to Mnih et al. (2013), target networks are used along with delayed policy updates to maintain stability. Third, the target policy $\pi_{\theta'}$ adds (clipped) Gaussian noise to ensure smoother value estimations, thus ensuring robustness.

The pseudocode as described by Fujimoto et al. (2018) can be found in Algorithm 2. In section 5.2, we will discuss in more detail the low-level workings of this algorithm along with differences in implementations by baseline frameworks.

Algorithm 2 TD3 (Fujimoto et al., 2018)

Initialize Q-value functions Q^{ϕ_1}, Q^{ϕ_2}, and actor π_θ randomly.
Initialize target networks $Q^{\phi'_1}$, $Q^{\phi'_2}$, $\pi_{\theta'}$ with $\phi'_1 \leftarrow \phi_1, \phi'_2 \leftarrow \phi_2$, and $\theta' \leftarrow \theta$
Initialize replay buffer $\mathcal{D}$
for $t = 1 \dots T$ **do**
    Select action, apply noise, and observe r, s'
    Store transition (s, a, r, s') in $\mathcal{D}$
    Sample minibatch of N transitions from $\mathcal{D}$
    Update all Q-value function parameters via the Mean Squared Error
    if t mod policy delay **then**
        Update the policy parameters via formula 16
        Update the target networks via Polyak averaging
    end if
end for

Action Selection Actions are selected through the deterministic policy π_θ. Gaussian exploration noise $\epsilon \sim \mathcal{N}(0, \sigma^2)$ is added to the actions to enhance exploration. To ensure the actions are within $[\mathcal{A}_{\text{low}}, \mathcal{A}_{\text{high}}]$, they are clipped. A 5-tuple (s, a, r, s', d) is stored in $\mathcal{D}$, where d denotes a finished episode flag.

Update Q-value function To update the Q-value function, a minibatch M of samples $\{(s_k, a_k, r_k, s'_k, d_k)\}_{(s,a,r,s',d)\in M}$ is used. Furthermore, the next action is sampled using a target network with clipped Gaussian noise via

$$a' \sim \text{clip}(\pi_{\theta'}(s'_k) + \text{clip}(\epsilon', -b, b), \mathcal{A}_{\text{low}}, \mathcal{A}_{\text{high}}), \quad \epsilon' \sim \mathcal{N}(0, \sigma^{2'}) \tag{18}$$

where b defines the target noise boundaries, and $\sigma^{2'}$ defines the target noise standard deviation. This ensures robustness during training. The cumulative Q-value loss is given by $L^V = \sum_i N^{-1} \sum_{k=1}^{N} (Q^{\phi_i}(s_k, a_k) - y)^2$, where

$$y = \begin{cases} r_k + \gamma \min_{i=1,2} Q^{\phi'_i}(s'_k, a') & \text{if not } d_k \\ r_k & \text{otherwise} \end{cases} \tag{19}$$

Update the policy and target networks The policy parameters and the target networks are periodically updated. The policy parameters θ are updated according to equation 16, using the policy loss formula

$$L^\pi = -N^{-1} \sum_{k=1}^{N} Q^{\phi_1}(s_k, \pi_\theta(s_k))$$

The target networks are updated via Polyak averaging between the target network parameters and the updated network parameters.

3.4.3 SAC

Soft Actor-Critic (SAC) (Haarnoja et al., 2018) utilizes entropy as a way to design an off-policy algorithm that is efficient and stable. Entropy is the measure of the diversity of the action distribution, given by $H(\pi_\theta(\cdot|s)) = \mathbb{E}_{a\sim\pi_\theta(a|s)}[-\log \pi_\theta(a|s)]$. Entropy is utilized through an additional term to the objective function, such as

$$J(\theta) = \mathbb{E}_{\tau\sim p_{\pi_\theta}}\left[R(\tau) + \alpha \sum_{t=0}^{T} H(\pi_\theta(\cdot|s_t))\right] \tag{20}$$

The temperature $\alpha > 0$ is a hyperparameter that weighs the stochasticity of the action distribution of the optimal policy. This ensures robustness by improving coverage of the state space through enhanced exploration (Haarnoja et al., 2018; Ziebart, 2010; Haarnoja et al., 2017).

The paper Haarnoja et al. (2019) builds on the work of Haarnoja et al. (2018). They have noticed that α used in Haarnoja et al. (2018) is nontrivial and is task-dependent. They argue that the influence of entropy should depend on the rewards obtained, which may differ between tasks and policy improvement. α should therefore scale with this uncertainty. Haarnoja et al. (2019) addressed this problem by parameterizing α. This allows α to adjust depending on the uncertainty of the policy for a given state s. This improved version of the original SAC algorithm is also adopted in the SAC implementations of the baseline frameworks discussed in this work.

In this newer version of SAC, the value function used in Haarnoja et al. (2018) is omitted. The original SAC used two Q-functions, a value function, and a policy function. The value function was used to approximate the soft value, defined by $\mathbb{E}_{a_t\sim\pi_\theta}[Q^\phi(s_t, a_t) - \alpha \log \pi_\theta(a_t|s_t)]$. This was found to be redundant (Haarnoja et al., 2019). The resulting objective of the policy becomes

$$J(\theta) = \mathbb{E}_{s\sim\mathcal{D}}\left[\mathbb{E}_{a\sim\pi_\theta(\cdot|s)}\left[-Q^\phi(s,a) + \alpha \log \pi_\theta(a|s)\right]\right] \tag{21}$$

The pseudocode as described in Haarnoja et al. (2019) can be found in Algorithm 3. Section 5.3 will elaborate on the algorithmic details, covering both similarities and differences in the implementations by baseline frameworks.

Action selection In contrast to TD3 (Fujimoto et al., 2018), the policy in SAC is stochastic. Therefore, the reparameterization trick must be applied, since directly sampled Gaussian noise is non-differentiable. An action can then be defined as a deterministic function $a := f_\theta(\epsilon, s) = \mu_\theta(s) + \sigma_\theta^2(s) \cdot \epsilon$, where noise ϵ is drawn from a Gaussian distribution $\epsilon \sim \mathcal{N}(0, 1)$. This way, the objective becomes

$$J(\theta) = \mathbb{E}_{s\in\mathcal{D},\epsilon\sim\mathcal{N}(0,1)}\left[-Q^\phi(s, f_\theta(\epsilon, s)) + \alpha \log \pi_\theta(f_\theta(\epsilon, s)|s)\right] \tag{22}$$

Algorithm 3 SAC with automated temperature parameter (Haarnoja et al., 2018)

Initialize Q-value functions Q^{ϕ_1}, Q^{ϕ_2}, and actor π_θ randomly.
Initialize target networks $Q^{\phi'_1}$, $Q^{\phi'_2}$, $\pi_{\theta'}$ with $\phi'_1 \leftarrow \phi_1, \phi'_2 \leftarrow \phi_2$, and $\theta' \leftarrow \theta$
Initialize replay buffer $\mathcal{D}$
while total number of timesteps not exceeded **do**
    for each environment step **do**
        Sample action and observe r, s'
        Store transition (s, a, r, s') in $\mathcal{D}$
    end for
    for each gradient step **do**
        Update the Q-value functions
        Update the policy parameters via objective function formula 21
        Update the temperature
        Update the target networks via Polyak averaging
    end for
end while

Q-value loss As mentioned, the value function in Haarnoja et al. (2018) gets replaced by a Q-value function in Haarnoja et al. (2019). The Q-value is calculated similarly, and together with this calculation, an entropy loss is applied

$$L^V := |B|^{-1} \sum_{i=1}^{|B|} \left(\frac{1}{2} \left(Q^\phi(s_i, a_i) - (r_i + \gamma Q^{\phi'}(s'_i, a') - \alpha \log \pi_\theta(a'|s'_i)) \right)^2 \right)$$

To address overestimation, multiple (target) Q-value networks are utilized and a pessimistic minimum is used, as in Fujimoto et al. (2018), following similar reasoning as Hasselt (2010). The target Q-value, i.e. the expected cumulative reward, can be defined as

$$y = \begin{cases} r_k + \gamma \min_{i=1,2} Q^{\phi'_i}(s'_k, a') - \alpha \log a' & \text{if not } d_k \\ r_k & \text{otherwise} \end{cases} \tag{23}$$

The loss can then be computed as the MSE between the Q-values of (s_t, a_t) and the target y.

$$L^V := (2|B|)^{-1} \sum_{i=1,2} \sum_{j=1}^{|B|} \left(Q^{\phi_i}(s_j, a_j) - y \right)^2 \tag{24}$$

Both summation followed by backpropagation and individual Q-loss backpropagation are similar, as $\forall i, \phi_i$ are disjoint.

Policy loss calculation The policy is computed according to the objective function as given in equation 21. This can be formulated as

$$L^\pi := |B|^{-1} \sum_{i=1}^{|B|} \left(\alpha \log a - \min_{j=1,2} Q^{\phi_j}(s_i, a) \right), \quad a \sim \pi_\theta(a|s_i) \tag{25}$$

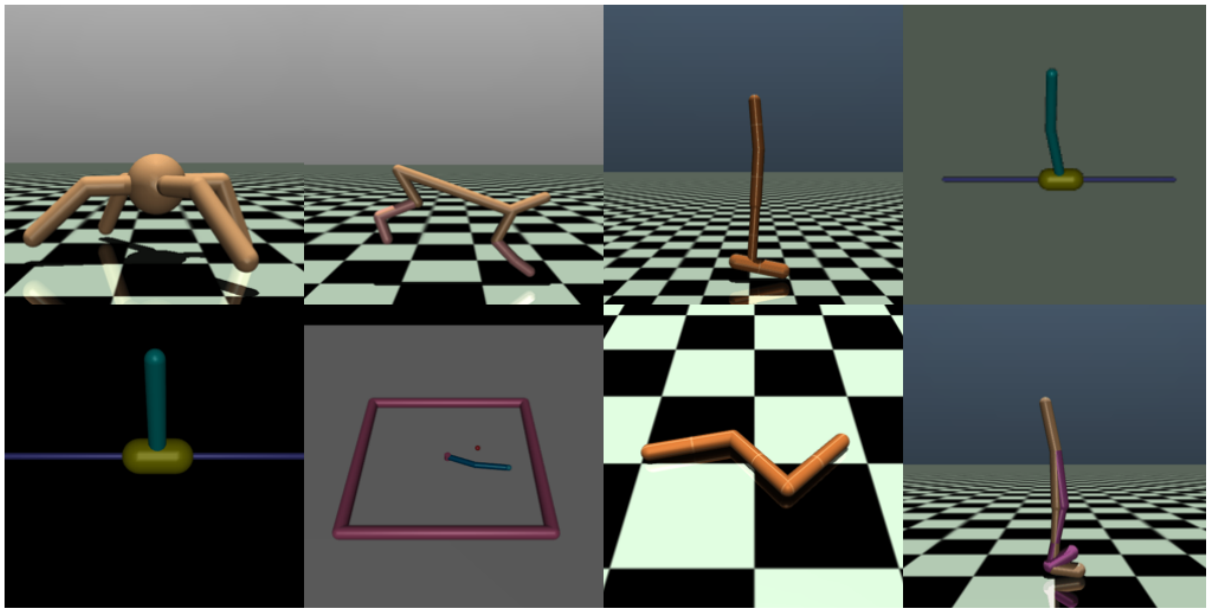


Figure 4: All MuJoCo environments considered in this work. From left to right, top to bottom: Ant-v4, HalfCheetah-v4, Hopper-v4, InvertedDoublePendulum-v4, InvertedPendulum-v4, Reacher-v4, Swimmer-v4, and Walker2d-v4.

Entropy loss calculation The entropy loss is given by

$$L^{\alpha} = |B|^{-1} \sum_{s \in B} \Big(-\alpha \log \pi_{\theta}(a|s) - \alpha \tilde{H}(s) \Big) \tag{26}$$

where $\tilde{H}(s)$ is the target entropy. In many cases, this is $\dim \mathcal{A}$.

Target Q-network update As in TD3, the target Q-networks are updated using a soft Polyak update.

3.5 Environments

In this work, we will focus on a fixed set of environments. All environments in this work have a continuous multidimensional state space and a continuous action space.

We will use the Farama Gymnasium library (Towers et al., 2024), a widely supported suite among popular frameworks (Raffin et al., 2021; Bou et al., 2023; Huang et al., 2022b; Weng et al., 2022a; Zheqing Zhu, 2023), as the interface of our environments. We use Multi-Joint dynamics with Contact (MuJoCo) environments (Todorov et al., 2012) to evaluate all algorithm implementations. By avoiding a mismatch of environment backends, we can factor out that the environment framework is the probable cause of any inconsistencies found. Furthermore, this environment suite is often used as a benchmarking suite (Schulman et al., 2017; Haarnoja et al., 2018; Fujimoto et al., 2018; Todorov et al., 2012; Schulman et al., 2015a; Haarnoja et al., 2019). Therefore, the results obtained using this suite are relevant to a larger community.

There are multiple versions of MuJoCo environments. Versions ≤ 1 are not supported by Gymnasium, and are left in Gym. Binary files have also changed when DeepMind acquired

MuJoCo, making the use of older versions more difficult. Furthermore, versions 2/3 and version 4 may differ slightly in performance, due to physics optimizations and emulator acceleration features (#2595). For consistency, all experiments are conducted using MuJoCo version 4. We are aware that reproducing results with a more recent MuJoCo version may differ.

The environments can be subdivided into several types of tasks: CartPole-like tasks (InvertedPendulum, InvertedDoublePendulum), coordination of a robotic arm (Reacher), 2D locomotive control (HalfCheetah, Hopper, Walker2D), and 3D locomotive control (Swimmer, Ant, Humanoid), most of which can be observed in Figure 4. The properties of the environments used in this work are summarized in Table 3. Not all environments have a termination condition. The environments that do have a termination condition calculate the 'healthiness' of the agent in the environment. Furthermore, all MuJoCo environments have symmetrical action space bounds, i.e. $\forall i \in [\dim \mathcal{A}], \mathcal{A}_{i,\text{low}} + \mathcal{A}_{i,\text{high}} = 0$.

Environment	Termination	Truncation	$\dim \mathcal{S}$	$\dim \mathcal{A}$
HalfCheetah-v4	✗	$T = 1000$	17	6
Hopper-v4	✓	$T = 1000$	11	3
InvertedDoublePendulum-v4	✓	$T = 1000$	9	1
InvertedPendulum-v4	✓	$T = 1000$	4	1
Reacher-v4	✗	$T = 50$	10	2
Swimmer-v4	✗	$T = 1000$	8	2
Walker2d-v4	✓	$T = 1000$	17	6
Ant-v4	✓	$T = 1000$	105	8
Humanoid-v4	✓	$T = 1000$	348	17

Table 3: Properties of MuJoCo environments. When a termination condition is present, it is denoted with a ✓. All environments have a truncation condition.

4 Experiments

In this section, we will go over the considerations when experimenting, the experimental design, and the experimental details.

4.1 Considerations

Each implementation of an algorithm may vary in the number of available hyperparameters and the available options for those hyperparameters. We define the set of experimental hyperparameters as those available in all implementations. A complete overview of these hyperparameters, along with their variable names and default values used, can be found in Appendix D. The rule of thumb to determine which hyperparameters we experiment with is:

If the hyperparameter in at least one baseline framework is not externally adaptable, it is considered a constant among all baseline frameworks.

With this, we avoid changing core implementation details integrated in each implementation, and only alter what is available. For example, the modular component offered by SB3 for

learning rate annealing is considered a valid hyperparameter, as it can be supplied to the algorithm without changing any core implementation details. We take into account that frameworks may have different objectives when replicating algorithms, such as providing 'state-of-the-art', 'matching the original paper', or 'task-independent, well-performing' implementations. Furthermore, we assume that the hyperparameters considered most important within an algorithm are available, whilst more complex or hyperparameters considered less influential by the framework authors are omitted.

Furthermore, to limit the inconsistencies to the algorithm implementations, we use similar environment setups. As a rule of thumb, we use

Environment feedback-altering wrappers are used in either no baseline framework or all baseline frameworks.

Environment wrappers that are feedback-altering are wrappers responsible for mapping the output of an environment. Examples of such wrappers are observation and reward normalization and/or transformation wrappers. Wrappers used for action space enforcement and logging statistics, however, do not alter feedback. An additional analysis of feedback-altering wrapped environments, along with an experiment performed in three environments, can be observed within Appendix C.1. There, we show how these wrappers can affect algorithmic performance.

4.2 Experimental design

To answer our research questions, we have to perform the following experiments. First of all, to answer whether the results produced by the implementations match the results reported in the original papers, we run the implementations with the original hyperparameter configuration on (a subset of) environments. We expect the implementations to match the performance of the original algorithm when using the same hyperparameters as in the original publication.

The second experiment compares the implementations under the same set of hyperparameters, including the original ones. We will be using the default hyperparameters given by each of the frameworks and applying them to all implementations of SB3, CleanRL, and TorchRL. Following the preliminaries, we evaluate the consistency in performance between different implementations via the use of a wide selection of MuJoCo environments. Ideally, all implementations perform similarly under the same hyperparameter configuration, regardless of the baseline framework.

Lastly, we will be comparing the relative performance ranking of all implementations from all baseline frameworks. We apply the default hyperparameter configurations provided by each baseline framework to all implementations. We then compare the relative performance ranking across different baseline frameworks, and observe to what extent this ranking is preserved.

We will trace back the found inconsistencies in the hyperparameter and code-level differences. Although we did a shallow analysis of the effects of some identified differences, it could be extended in further research. This also illustrates how these differences may affect training

and the resulting policies. This strengthens earlier work on the importance of reporting implementation details.

4.3 Experimental details

The implementations of each algorithm are listed in Table 4. We used a subset of the MuJoCo environments: Ant-v4, HalfCheetah-v4, Hopper-v4, InvertedDoublePendulum-v4, InvertedPendulum-v4, Reacher-v4, Swimmer-v4, and Walker2d-v4. The code can be found in this GitHub repository. A complete guide on how to use this code can be found in Appendix B.

Each experiment's mean and standard deviation are based on 5 random seeds, run for 3 million timesteps. Each point in the curve represents the average cumulative reward obtained using a deterministic (greedy) policy over 30 complete episodes. An evaluation interval of 10000 timesteps was used. Subsampling is applied to improve the readability of the results. This work was performed using the compute resources from the Academic Leiden Interdisciplinary Cluster Environment (ALICE) provided by Leiden University (ALICE - SHARK HPC team, 2025). Each experiment was run on a single node, using one core of an Intel Xeon Gold 6126 12-Core Processor.

ALG / FW	Stable Baselines 3	CleanRL	TorchRL
PPO	`PPO`	`ppo_continuous_action.py`	`ppo_mujoco.py`
TD3	`TD3`	`td3_continuous_action.py`	`td3.py`
SAC	`SAC`	`sac_continuous_action.py`	`sac.py`

Table 4: The references to the implementations of the different algorithms and respective baseline frameworks used in this work.

5 Analysis

In this analysis, we expose code-level inconsistencies between the implementations of PPO, TD3, and SAC. For clarity, we denote inconsistencies among implementations via the use of colors: Stable Baselines 3, CleanRL, and TorchRL. Furthermore, we refer to the line(s) of code (L) where the differences in these implementations originate by square [] brackets.

5.1 PPO analysis

Most engineering tricks used in PPO originate from Huang et al. (2022a)'s blog.

Model definition All PPO implementations use orthogonal initialization. However, as Box 1 displays, different scales are used when setting the weights of the policy and state-value networks in different implementations.

Box 1: PPO Orthogonal intialization

Stable Baselines 3 [L612-L631] and CleanRL [L106-L129]

- $\pi_\theta(a|s)$ last layer scaled by 1
- $V^\phi(s)$ last layer scaled by 0.01.
- Other layers in both $\pi_\theta(a|s)$ and $V^\phi(s)$ are scaled by $\sqrt{2}$ and bias 0.0.

TorchRL [L69-L73,L108-L111]

- All layers in both $\pi_\theta(a|s)$ and $V^\phi(s)$ are scaled by 1 and 0.01 respectively.

Hyperparameter annealing Two hyperparameters are annealed in different implementations. Learning rate annealing is used by default in PPO$^{\text{CleanRL}}$ and PPO$^{\text{TorchRL}}$. PPO$^{\text{SB3}}$ offers an additional learning rate schedule, but is not included by default. Similarly, clip range annealing is supported by PPO$^{\text{SB3}}$ and PPO$^{\text{TorchRL}}$, but not by PPO$^{\text{CleanRL}}$. Clip range annealing is not used by default in any implementation.

Action selection There are two different methods used to enforce the action space bounds, as observed in Box 2. PPO$^{\text{SB3}}$ and PPO$^{\text{CleanRL}}$ use hard clipping; however PPO$^{\text{CleanRL}}$ requires an environment wrapper to handle this, to enforce this. PPO$^{\text{TorchRL}}$ uses scaling to the range [-1,1] via a $\tanh$ function, and then applies a proportional unscaling to ensure $a \in \mathcal{A}$.

Box 2: PPO Action selection during rollout

Stable Baselines 3 [L213-L216] and CleanRL [L96][L106-L129]

$$a \sim \mathcal{N}(\mu_\theta(s), \sigma^2_\theta(s))$$

$$a \leftarrow \text{clip}(a, \mathcal{A}_{\text{low}}, \mathcal{A}_{\text{high}})$$

TorchRL [L401-L416]

$$a \sim \mathcal{N}(\mu_\theta(s), \sigma^2_\theta(s))$$

$$a \leftarrow \text{unscale}(\tanh(a))$$

Advantage estimation All PPO implementations make use of GAE to calculate advantage estimations. Furthermore, all PPO implementations apply advantage normalization by default. In PPO$^{\text{TorchRL}}$, this is a constant.

PPO$^{\text{SB3}}$ makes properly distinguish truncation and termination scenarioes. An additional bootstrapped expected cumulative discounted reward term $\gamma V^\phi(s_{T+1}))$ is added when a truncation condition is met[2]. PPO$^{\text{CleanRL}}$ does not properly distinguish termination and truncation scenarios, performing no bootstrapping either case [L221]. This open GitHub issue states this "does not seem to result in significant performance differences" (Huang et al., 2022b). PPO$^{\text{TorchRL}}$

[2]Denote that handling of truncation states is done via the reward adaptation, and not by masking truncation states and treating them differently [L236-L245].

makes a proper distinction between truncation and termination scenarios. By proper masking, they add an additional $\gamma V^{\phi}(s_{T+1}))$ term as the remaining expected cumulative discounted reward [L1502-L1503].

Minibatch selection All PPO implementations make use of minibatching to update the agent. However, the hyperparameter responsible differs in function. PPO[SB3] and PPO[TorchRL] use `batch_size`, which determines the size of the minibatch. PPO[CleanRL] uses `num_minibatches`, which determines the number of minibatches. All implementations use advantage estimates at the batch level and the probability ratios at the minibatch level.

Value loss clipping Value clipping, similar to how the policy loss is clipped, uses clipping such that updates of $V^{\phi}(s)$ remain stable. This is done by replacing the returns with state-values using the old state-value parameters ϕ_{old}, i.e.

$$V_{clipped}^{(\phi,\phi_{old})}(s) := V^{\phi_{old}}(s) + \text{clip}(V^{\phi}(s) - V^{\phi_{old}}(s), -\epsilon_V, \epsilon_V) \tag{27}$$

where ϵ_V is the value clipping epsilon hyperparameter. Box 3 shows how the PPO implementations differ when using value clipping. By default, PPO[SB3] and PPO[TorchRL] do not make use of value clipping. Though when applied, an independent variable ϵ_V can be set. PPO[CleanRL] uses value clipping by default, and $\epsilon_V = \epsilon$.. Furthermore, when using value clipping, PPO[SB3] forcibly uses these clipped values in the value loss calculation, whilst PPO[CleanRL] and PPO[TorchRL] use a pessimistic lower bound by taking the maximum over the clipped and unclipped state-values.

Box 3: PPO Clipped state-value loss formula

Stable Baselines 3: [L234-L244]

$$L^V = (2|M|)^{-1} \sum_{R,s \in M} \left(V_{clipped}^{(\phi,\phi_{old})} - V^{\phi}(s) \right)^2$$

CleanRL [L286-L297] and TorchRL [L622-L648]

$$L^V = (2|M|)^{-1} \max \left(\sum_{R,s \in M} \left(V_{clipped}^{(\phi,\phi_{old})} - V^{\phi}(s) \right)^2, \sum_{R,s \in M} \left(R - V^{\phi}(s) \right)^2 \right)$$

Gradient clipping PPO[TorchRL] does not seem to use gradient clipping. However, the work of Hundal et al. (2025) experimented with the use of gradient clipping and found no significant difference in the use of gradient clipping.[3]

[3] During experimenting, Walker2d-v4 and Hopper-v4 using $\text{PPO}^{\text{TorchRL}}(\mathcal{H}^{\text{TorchRL}})$ and $\text{PPO}^{\text{TorchRL}}(\mathcal{H}^{\text{CleanRL}})$, respectively, each failed 1/5 runs due to high/NaN actions returned. This can only occur when observations are NaN to begin with, which can happen when gradients have exploded. The absence of gradient clipping may contribute to this problem.

Environment setup The implementation PPO$^{\text{CleanRL}}$, specifically the one designed to support continuous action spaces, uses feedback-altering wrappers. These include observation normalization, observation transformation, reward normalization, and reward transformation wrappers. Implementations PPO$^{\text{SB3}}$ and PPO$^{\text{TorchRL}}$ do not share this same environment setup. Appendix C.1 explains the definitions of these wrappers and shows an additional experiment highlighting how it may affect training.

5.2 TD3 analysis

Model definition The default network hidden layer sizes and availability to adjust those differ among TD3 implementations. TD3$^{\text{SB3}}$ uses a hyperparameter controlling the hidden layer sizes, which are by default [400,300]. TD3$^{\text{CleanRL}}$ has no option to externally adjust the hidden layer sizes, and uses by default [256,256]. Similarly, TD3$^{\text{TorchRL}}$ uses a hyperparameter to control the hidden layer sizes, and is by default [256,256].

Action selection The TD3 implementations slightly differ in enforcing the action space and when exploration noise is applied, as is observed in Box 4. TD3$^{\text{CleanRL}}$ and TD3$^{\text{TorchRL}}$ use $\tanh$ to enforce $a \in [-1, 1]$, to then unscale this action to meet the action space boundaries proportionally. After applying Gaussian noise, the action is again hard-clipped to enforce action space boundaries. TD3$^{\text{SB3}}$ uses clipping to enforce $a \in [\mathcal{A}_{\text{low}}, \mathcal{A}_{\text{high}}]$. It then scales the action to $a \in [-1, 1]$, adds Gaussian noise, and unscales the action back to $a \in [\mathcal{A}_{\text{low}}, \mathcal{A}_{\text{high}}]$. Furthermore, TD3$^{\text{SB3}}$ saves the scaled action to the replay buffer and trains its networks with these scaled actions [L407-L409]. This issue along with this guide mentions its positive influence to normalize actions.

Box 4: TD3 Action selection during environment inference

Stable Baselines 3 [L400-L409]

$$a \leftarrow \pi_\theta(s)$$
$$a \leftarrow \text{clip}(a, \mathcal{A}_{\text{low}}, \mathcal{A}_{\text{high}})$$
$$a \leftarrow \text{clip}(\text{scale}(a) + \epsilon, -1, 1)$$
$$a \leftarrow \text{unscale}(a)$$

CleanRL [L130-L131,L209-L211] and TorchRL [L237-L247,L352-L366]

$$a \leftarrow \pi_\theta(s)$$
$$a \leftarrow \tanh(a)$$
$$a \leftarrow \text{unscale}(a)$$
$$a \leftarrow \text{clip}(a + \epsilon, \mathcal{A}_{\text{low}}, \mathcal{A}_{\text{high}})$$

Additionally, when retrieving next action a' via the target networks, TD3$^{\text{SB3}}$, by default, does not apply action noise. However, it offers a range of modules, extending beyond Gaussian noise, to sample noise. Equation 18 is adapted according to the normalized actions via

$$\text{clip}(\pi_\theta(s'_k) + \text{clip}(\epsilon', -b, b), -1, 1)$$

[L174-L176]. For TD3$^{\text{CleanRL}}$, $\sigma^{2'}$ within equation 18 is a hyperparameter, and is used by default. TD3$^{\text{TorchRL}}$ has this Gaussian noise with $\sigma^{2'} = 0.2$ as constant processed within the implementation.

Truncation TD3$^{\text{SB3}}$ does not make a distinction between terminal flags and truncation flags, meaning that truncated steps are not bootstrapped properly [L448-511][4]. TD3$^{\text{CleanRL}}$ [L230] and TD3$^{\text{TorchRL}}$ [L762-L770] make proper distinctions between termination and truncation, and thus perform proper bootstrapping when necessary.

Update delay TD3$^{\text{SB3}}$ and TD3$^{\text{TorchRL}}$ allow for different ratios of environment inferences to gradient steps performed by two hyperparameters. This enables ratios different from 1:1, allowing more environment inferences per gradient step and vice versa. By default, TD3$^{\text{SB3}}$ and TD3$^{\text{TorchRL}}$ use a ratio of 1:1 and 1000:1000, respectively. TD3$^{\text{CleanRL}}$ uses a single hyperparameter to affect both the policy- and target network parameter updates frequencies. Only ratios where environment inferences equal gradient steps are allowed.
Furthermore, cold-start update delays differ in default values. TD3$^{\text{CleanRL}}$ and TD3$^{\text{TorchRL}}$ uses a cold-start of 25000 environment inferences, whilst TD3$^{\text{SB3}}$ uses 100 environment inferences.

Target network updating TorchRL, given aforementioned formulas for Polyak updating, switched τ_p for $1-\tau_p$, meaning that the default hyperparameter TorchRL passes is large [L343]. This is taken into account when experimenting.

Vectorization Vectorization differs between TD3$^{\text{SB3}}$ and TD3$^{\text{CleanRL}}$. TD3$^{\text{SB3}}$ use proper accounting for vectorization. For n_{envs}, a total of T timesteps are performed. TD3$^{\text{CleanRL}}$ does not take into account that more vectorized environments produce more samples per timestep. For n_{envs}, a total of $T \cdot n_{envs}$ timesteps are performed. An additional experiment exposing this difference can be found in Appendix C.2.

5.3 SAC analysis

Learning rate hyperparameter There are three different networks that are being optimized in SAC: the policy network, Q-value networks, and the temperature network, all of which use an Adam optimizer. However, the availability to set learning rates differs. We denote the learning rates α_π, α_V, and α_H respectively. SAC$^{\text{SB3}}$ uses hyperparameters $\alpha_\pi = \alpha_V$, with a default value of 0.0003. α_H is a linearly annealed hyperparameter with an initial value of 1.0. SAC$^{\text{CleanRL}}$ uses two hyperparameters controlling α_π and $\alpha_V = \alpha_H$, with default values 0.0003 and 0.001 respectively. SAC$^{\text{TorchRL}}$ uses a single hyperparameter controlling $\alpha_\pi = \alpha_V$, with a default value of 0.0003. A constant value of 0.0003 is used as the value for α_H.

[4]The environment used during rollouts is of class `VecEnv`, which returns `dones` a boolean denoting a termination flag or truncation flag. Truncations can be filtered out through the use of provided `infos`, via 'TimeLimit.truncated'. The usage of this flag to filter truncations from terminations is not used within the aforementioned code snippet. Only the correct next observation is saved to the buffer, but it does not account for future rewards when truncated.

Action selection In terms of action sampling from the network, there are slight stability and normalization differences between SAC$^{\text{SB3}}$, SAC$^{\text{CleanRL}}$, and SAC$^{\text{TorchRL}}$. See Box 5 how action selection differs among SAC implementations. Similar to TD3$^{\text{SB3}}$, SAC$^{\text{SB3}}$ uses normalized actions when training the networks. The actions get unscaled to $[\mathcal{A}_{\text{low}}, \mathcal{A}_{\text{high}}]$ when inferring the environment.

All SAC implementations make sure the value $\sigma^2_\theta(s)$ retrieved from the policy remains within a range $[\log \sigma^2_{\theta,\min}, \log \sigma^2_{\theta,\max}]$. SAC$^{\text{SB3}}$ does so by clip$(\sigma^2_\theta(s), \log \sigma^2_{\theta,\min}, \log \sigma^2_{\theta,\max})$, with $[\log \sigma^2_{\theta,\min}, \log \sigma^2_{\theta,\max}]$ = [-20, 2]. SAC$^{\text{CleanRL}}$ uses the bounds $[\log \sigma^2_{\theta,\min}, \log \sigma^2_{\theta,\max}]$ = [-5, 2] instead. Furthermore, $\tanh$ and unscaling are applied to $\log \sigma^2$ to ensure to the bounds $[\log \sigma^2_{\theta,\min}, \log \sigma^2_{\theta,\max}]$. SAC$^{\text{TorchRL}}$ only enforces $\log \sigma^2_\theta$ to be at least some minimum value. All implementations discussed in this work apply $\tanh$ to enforce $\mathcal{N}(\mu_\theta, \sigma^2_\theta) \in [-1, 1]$ before unscaling to match the action space bounds. The log probability $\log a$ is corrected for the use of $\tanh$. In SAC$^{\text{CleanRL}}$ and SAC$^{\text{TorchRL}}$, it has also been corrected for unscaling to match the action bounds.

Q-value loss calculation Box 6 shows how SAC$^{\text{SB3}}$ applies a factor $\frac{1}{2}$ to the Mean Squared Error distance loss used, which is in line with Haarnoja et al. (2019). This is unrelated to the number of Q-networks used. SAC$^{\text{CleanRL}}$ and SAC$^{\text{TorchRL}}$ omit this halving factor, diverting from the original formula.

Updating delay Similar to the TD3 implementations, SAC$^{\text{SB3}}$ and SAC$^{\text{TorchRL}}$ allow for different ratios of environment inferences to gradient steps performed. By default, SAC$^{\text{SB3}}$ uses a ratio of 1:1, while SAC$^{\text{TorchRL}}$ uses 1000:1000. SAC$^{\text{CleanRL}}$ only allows for increasing the number of environment inferences per gradient step, at the cost of fewer gradient steps per total timesteps. By default, a ratio of 2:1 is used. This affects both the policy and temperature networks.
Furthermore, the default cold-start update delays differs: SAC$^{\text{SB3}}$ uses 100, SAC$^{\text{CleanRL}}$ uses 5000, and SAC$^{\text{TorchRL}}$ uses 25000 environment inferences as a cold-start delay.

Entropy loss calculation There are numerous discussions (Ray #15625, Stable Baselines #801, softlearning #37, Stable Baselines 3 #36, Tianshou #258) about the use of $\log \alpha$ instead of α during loss calculation. The benefit of using $\log \alpha$ is numerical stability, due to its positive nature. As shown in Box 7, SAC$^{\text{SB3}}$ and SAC$^{\text{TorchRL}}$ use $\log \alpha$ in its loss calculation, in line with Haarnoja et al. (2019). SAC$^{\text{CleanRL}}$ does not adopt this, and uses α instead.

Target network updating SAC$^{\text{TorchRL}}$, similarly to TD3$^{\text{TorchRL}}$, switched τ_p to $1 - \tau_p$ [L343]. This is taken into account when experimenting.

Vectorization The process of handling vectorization differs between SAC$^{\text{SB3}}$ and SAC$^{\text{CleanRL}}$ in a similar way vectorization differs between TD3$^{\text{SB3}}$ and TD3$^{\text{CleanRL}}$: SAC$^{\text{SB3}}$ accounts each environment step as a timestep towards the total number of timesteps, whilst SAC$^{\text{CleanRL}}$ accounts for a single timestep to the collective number of environment steps. See additional experiments in Appendix C.2.

Box 5: SAC Action selection

Stable Baselines 3: [L147-L165,L227-L240,L76-L79]

$$\mu_\theta, \log \sigma_\theta^2 \sim \pi_\theta$$

$$\log \sigma^2 \leftarrow \text{clip}(\sigma_t^2, \log \sigma_{\theta,\min}^2 - \log \sigma_{\theta,\max}^2)$$

$$x \sim \mathcal{N}(\mu_\theta, \exp \sigma^2)$$

$$a \leftarrow \tanh(x)$$

$$\log a \leftarrow \log \mathcal{N}(\mu_\theta, \sigma^2) - \sum \log(1 - \tanh(x)^2) + 10^{-6}$$

CleanRL [L128-L150,L76-L79]

$$\mu_\theta, \log \sigma_\theta^2 \sim \pi_\theta$$

$$\sigma^2 \leftarrow \log \sigma_{\theta,\min}^2 + \frac{1}{2}(\log \sigma_{\theta,\max}^2 - \log \sigma_{\theta,\min}^2)(\tanh(\sigma_\theta^2) + 1)$$

$$x \sim \mathcal{N}(\mu_\theta, \exp \sigma^2)$$

$$a \leftarrow \tanh(x)\frac{\mathcal{A}_{\text{high}} - \mathcal{A}_{\text{low}}}{2} + \frac{\mathcal{A}_{\text{high}} + \mathcal{A}_{\text{low}}}{2}$$

$$\log a \leftarrow \log \mathcal{N}(\mu_\theta, \sigma^2) - \sum \log \left(\frac{\mathcal{A}_{\text{high}} - \mathcal{A}_{\text{low}}}{2}(1 - \tanh(x)^2) + 10^{-6} \right)$$

TorchRL [L408-L417,L431-L453,L251-255,L83-L87,L76-L79]

$$\mu_\theta, \sigma_\theta^2 \sim \pi_\theta$$

$$\sigma \leftarrow \max(\log(1 + \exp \sigma_\theta^2), \sigma_{\theta,\min})$$

$$x \sim \mathcal{N}(\mu_\theta, \exp \sigma_\theta^2)$$

$$a \leftarrow \tanh(x)\frac{\mathcal{A}_{\text{high}} - \mathcal{A}_{\text{low}}}{2} + \frac{\mathcal{A}_{\text{high}} + \mathcal{A}_{\text{low}}}{2}$$

$$\log a \leftarrow \log \mathcal{N}(\mu_\theta, \sigma^2) - \sum \log \left(\frac{\mathcal{A}_{\text{high}} - \mathcal{A}_{\text{low}}}{2}(1 - \tanh(x)^2) + 10^{-6} \right)$$

Box 6: SAC Q-value loss formula

Stable Baselines 3: [L267]

$$L^V := (2|B|)^{-1} \sum_{i=1,2} \sum_{j=1}^{|B|} \left(Q^{\phi_i}(s_j, a_j) - y \right)^2$$

CleanRL [L273-L275] and TorchRL [L832-L836]

$$L^V := |B|^{-1} \sum_{i=1,2} \sum_{j=1}^{|B|} \left(Q^{\phi_i}(s_j, a_j) - y \right)^2$$

Box 7: SAC Entropy loss formula

Stable Baselines 3: [L237] and TorchRL [L879]

$$L^{\alpha} = |B|^{-1} \sum_{s \in B} \Big(-\log \alpha (\log \pi_{\theta}(a|s) + \tilde{H}(s)) \Big)$$

CleanRL [L299]

$$L^{\alpha} = |B|^{-1} \sum_{s \in B} \Big(-\alpha (\log \pi_{\theta}(a|s) + \tilde{H}(s)) \Big)$$

6 Results

In this section, we will discuss the results of the experiments as described in Section 4. We will highlight a few results that stand out. All of the results can be found in Appendix E.

6.1 Reproduction

We have applied the original hyperparameters of each algorithm to each implementation of Stable Baselines 3, CleanRL, and TorchRL. We have taken into account that some hyperparameters are unavailable in some implementations. Furthermore, the original results use a variable number of timesteps. We will account for this difference accordingly.

Frequency of inconsistencies Most of the experiments are in line with the original results, although with high standard deviations. The standard deviations reported in the original results were not taken into account. We use a 95% confidence interval to assess whether implementation results perform worse, similarly, or better. Across all implementations and baseline frameworks, 38.9% of runs align, 31.5% show worse performance, and 29.6% show improved performance over the reported results. It is expected to observe improved performance, as implementations use engineering tricks to improve stability. Figure 5 summarizes our findings across implementations and frameworks on multiple environments. SAC implementations often perform worse than the original results. Furthermore, we observe a similar degree of variability in the ability to reproduce the original results across frameworks.

Causalities First, we acknowledge that the difference in performance may originate from the different MuJoCo environment versions (for PPO and TD3, v1; for SAC, v2). Different MuJoCo versions may have altered reward functions, introduced additional loss penalties, or caused numerical differences within the physics engine. Furthermore, we acknowledge that the differences may originate from different Python versions and dependency versions.

Hyperparameter differences may result in different performance. For example, $\text{PPO}^{\text{TorchRL}}$ does not use gradient clipping, while this was used in the original algorithm. Furthermore, $\text{TD3}^{\text{TorchRL}}$ and $\text{SAC}^{\text{TorchRL}}$ use delayed network updates via an alternative update-to-data ratio: 1000:1000, in contrast to 1:1 in Fujimoto et al. (2018) and Haarnoja et al. (2019). Additionally, TD3^{SB3} uses a hidden layer sizes of [400,300] as opposed to [256,256]. In addition, unlike a fixed learning rate of 0.0003 across all networks in SAC (Haarnoja et al.,

Figure 5: This figure shows whether the implementations statistically perform worse/similar/better than the reported results of the original algorithm. In an ideal case, all implementations show no significant statistical difference. The frequency of statistical differences in each framework among all implementations is similar. Furthermore, we observe more statistical differences for implementations of SAC.

2019), SAC$^{\text{SB3}}$ also uses a linearly decaying temperature learning rate with an initial value of 1.0. SAC$^{\text{CleanRL}}$ instead uses a Q-value learning rate and temperature learning rate of 0.001, with delayed policy updates. Lastly, in contrast to the original algorithm, all SAC implementations do not make use of reward scaling.

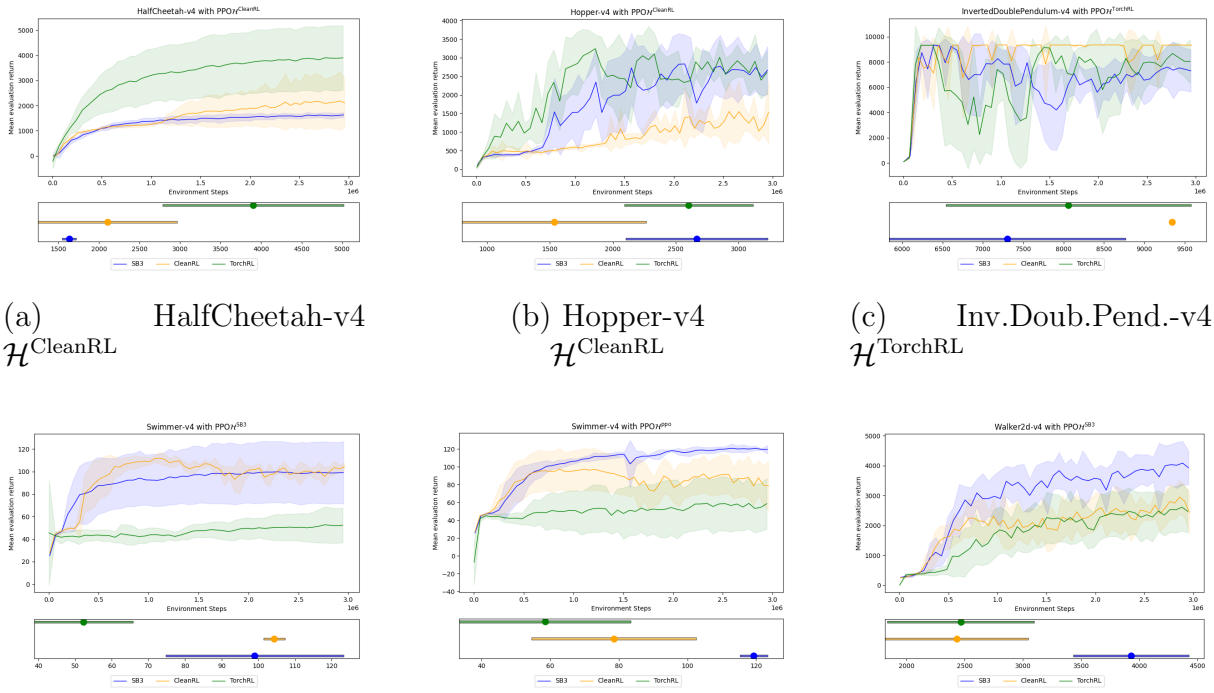
We have not focused on the original algorithm at the code level. Therefore, we cannot conclude whether methodological differences occur between implementations and the algorithm. However, between the provided pseudocode and the implementations, we can identify some additional differences. For example, TD3$^{\text{SB3}}$ and SAC$^{\text{SB3}}$ train the networks on scaled actions (i.e. $a \in [-1, 1]$), to increase stability. Additionally, we identified some differences between the original formulas and implementations, as observed in Boxes 6 and 7. Furthermore, some methodology is incorrectly implemented, such as distinguished bootstrapping.

6.2 Direct comparability

In this section, we will compare the performance of algorithm implementations using different baselines for each default hyperparameter configuration. Since we have not evaluated the impact of individual code-level differences, we must assume that the observed differences between implementations originate from a combination of these differences.

PPO All results involving the consistency of PPO implementations can be found in Appendix E: Figures 15 and 18, and Table 9.

We see that PPO$^{\text{CleanRL}}$ stably outperforms PPO$^{\text{SB3}}$ and PPO$^{\text{TorchRL}}$ in InvertedDoublePendulum-v4 (see figure 6c). This environment has both termination and truncation conditions. We therefore hypothesize that the performance gap stems from the difference in truncation handling. Furthermore, we note that $\mathcal{H}^{\text{CleanRL}}$ is the only hyperparameter configuration that uses value loss clipping, and is therefore the only setting where inconsistent performance between PPO$^{\text{SB3}}$ and others may become apparent due to



(a) HalfCheetah-v4 $\mathcal{H}^{\text{CleanRL}}$ (b) Hopper-v4 $\mathcal{H}^{\text{CleanRL}}$ (c) Inv.Doub.Pend.-v4 $\mathcal{H}^{\text{TorchRL}}$

(d) Swimmer-v4 $\mathcal{H}^{\text{SB3}}$ (e) Swimmer-v4 $\mathcal{H}^{\text{ppo}}$ (f) Walker2d-v4 $\mathcal{H}^{\text{SB3}}$

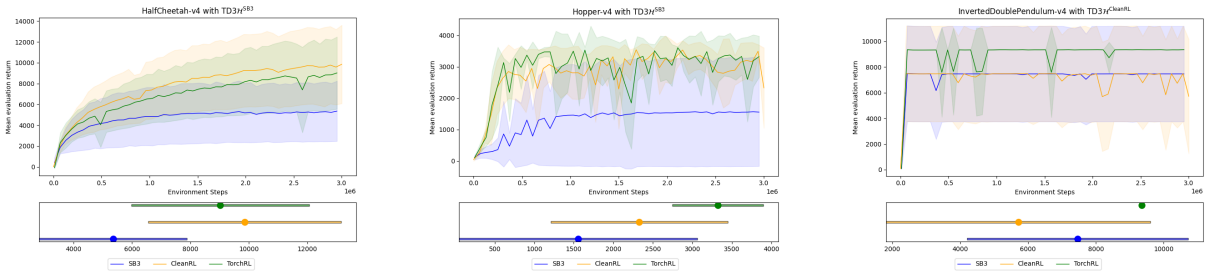
Figure 6: A highlight of inconsistent behaviour in PPO implementations by Stable Baselines 3, CleanRL, and TorchRL, using a similar hyperparameter configuration. The top plots show the mean and standard deviation return over 30 evaluation episodes. Bottom plots show the 95% confidence interval over the last reported evaluation mean and standard deviation.

differences in the formulas for value loss clipping. However, Figures 6a and 6b do not reveal performance differences due to this.

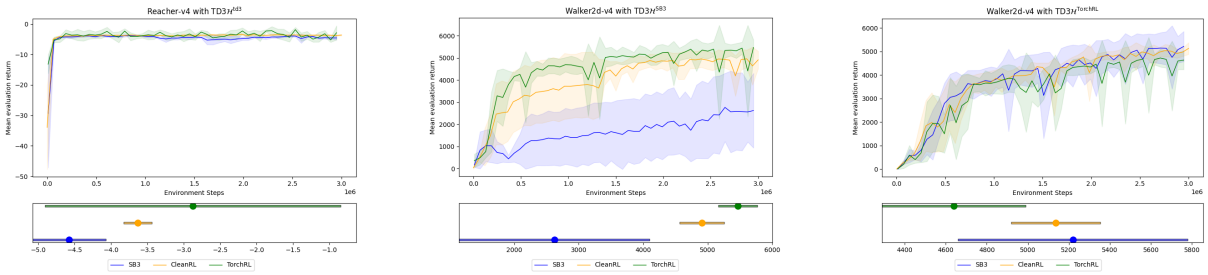
The impact of individual code-level differences is difficult to isolate. For example, PPO$^{\text{TorchRL}}$ may differ in performance due to its different orthogonal weight initialization, or show less stability due to the absence of gradient clipping. Similarly, exploration differences that may occur due to differences in action selection methodologies are difficult to expose. Figures 6d-f show examples of cases where extreme performance differences across implementations can be observed.

TD3 All results involving the consistency of TD3, as described in this section, can be found in Appendix E: Figures 16 and 19, and Table 10.

The implementation TD3$^{\text{SB3}}$ stands out the most. It uses a larger neural network than TD3$^{\text{CleanRL}}$ and TD3$^{\text{TorchRL}}$, which may both slow down training but also yield better performance. Furthermore, TD3$^{\text{SB3}}$ does not use exploration noise by default, which, in contrast to TD3$^{\text{CleanRL}}$ and TD3$^{\text{TorchRL}}$, may affect the policy robustness. Furthermore, TD3$^{\text{SB3}}$ trains on scaled actions, which may positively contribute to stability during training. Lastly, TD3$^{\text{SB3}}$ does not distinguish between truncation and termination. A combination of these differences may either positively (see Figure 7f) or negatively (see Figure 7a, b, and e)



(a) HalfCheetah-v4 $\mathcal{H}^{\text{SB3}}$ (b) Hopper-v4 $\mathcal{H}^{\text{SB3}}$ (c) Inv.Doub.Pend.-v4 $\mathcal{H}^{\text{CleanRL}}$



(d) Reacher-v4 $\mathcal{H}^{\text{td3}}$ (e) Walker2d-v4 $\mathcal{H}^{\text{SB3}}$ (f) Walker2d-v4 $\mathcal{H}^{\text{TorchRL}}$

Figure 7: A highlight of inconsistent behaviour in TD3 implementations by Stable Baselines 3, CleanRL, and TorchRL, using a similar hyperparameter configuration. The top plots show the mean and standard deviation return over 30 evaluation episodes. Bottom plots show the 95% confidence interval over the last reported evaluation mean and standard deviation.

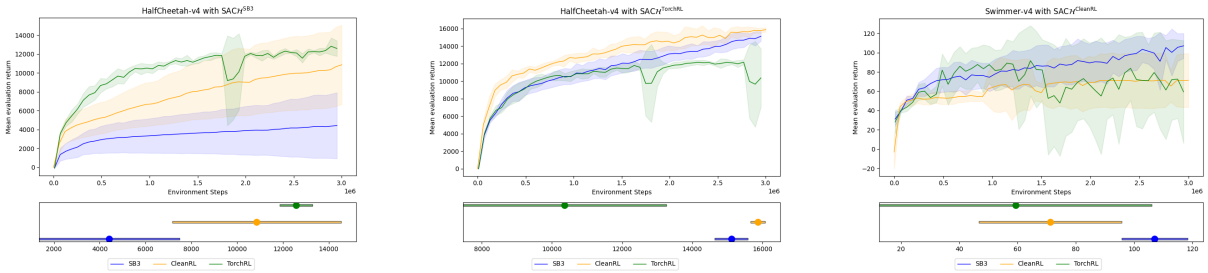
influence performance.

Similarly, we observe that TD3$^{\text{TorchRL}}$ and TD3$^{\text{CleanRL}}$ differ only between update ratios (1000:1000 opposed to 1:1). We can see TD3$^{\text{TorchRL}}$ displaying more oscillations during training in Figures 7b and f. Conversely, it could also show beneficial results, as seen in Figures 7d and e

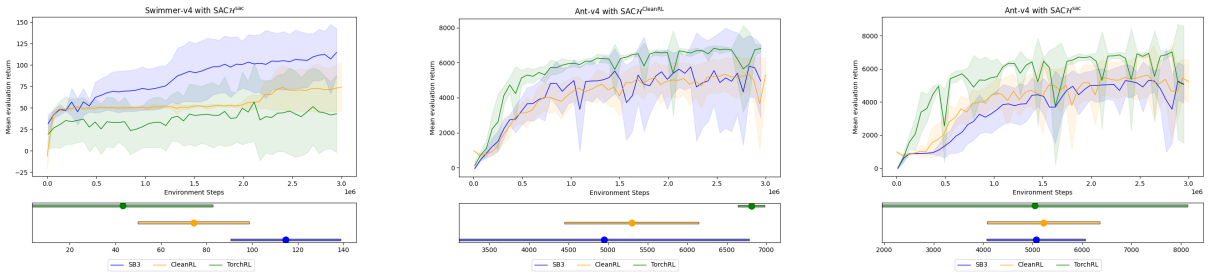
We hypothesize that the difference in performance across implementations in InvertedPendulum-v4 and InvertedDoublePendulum-v4 (see Figure 7c) is because of suboptimal starting seeds. The action space dimensionality of both environments is 1, and is therefore prone to converging to local optima.

SAC All the results involving the consistency of SAC described in this section can be found in Appendix E: Figures 17 and 20, and Table 11.

We observe differences in performance across SAC implementations. We assume that the hyperparameters controlling the policy, Q-value, and temperature networks are likely responsible for the observed differences. However, differences such as SAC$^{\text{TorchRL}}$ using a 1000:1000 update ratio, SAC$^{\text{SB3}}$ using scaled actions for training, and the different formulas as shown in Boxes 6 and 7 are important differences.



(a) HalfCheetah-v4 $\mathcal{H}^{\text{SB3}}$ (b) HalfCheetah-v4 $\mathcal{H}^{\text{TorchRL}}$ (c) Swimmer-v4 $\mathcal{H}^{\text{CleanRL}}$



(d) Swimmer-v4 $\mathcal{H}^{\text{sac}}$ (e) Ant-v4 $\mathcal{H}^{\text{CleanRL}}$ (f) Ant-v4 $\mathcal{H}^{\text{sac}}$

Figure 8: A highlight of inconsistent behaviour in SAC implementations by Stable Baselines 3, CleanRL, and TorchRL, using a similar hyperparameter configuration. The top plots show the mean and standard deviation return over 30 evaluation episodes. Bottom plots show the 95% confidence interval over the last reported evaluation mean and standard deviation.

With the aforementioned differences, we also observe a significant difference between implementations using $\mathcal{H}^{\text{SB3}}$ and $\mathcal{H}^{\text{TorchRL}}$. The only difference between these hyperparameter configurations is the cold start delay. We identified differences in action selection (see Box 5), and the different domains within which $\log \sigma_\theta$ is constrained. These differences in when and where to explore, different formulations, and different learning rates are likely confounding factors for the differences observed in Figure 8.

6.3 Relative ranking

All the results involving the relative ranking of PPO, TD3, and SAC under specific hyperparameter configurations can be found in Appendix E: Figures 21, 22, 23, 24, and 25.

In general, the relative ranking of the algorithms remains largely preserved. We used the default hyperparameters provided by each framework. We observe that for the majority of environments, the relative ranking of the implementations is similar. Furthermore, this generally applies to experiments where $\mathcal{H}^{\text{TorchRL}}$ or $\mathcal{H}^{\text{OG}}$ are used.

There are a few exceptions to this statement. First, we observe that the magnitude of the performance differences between implementations may not be preserved. Figure 9 shows an

example where it can be argued whether the relative ranking of the implementations is still preserved. This occurs most often in experiments using $\mathcal{H}^{\text{CleanRL}}$.

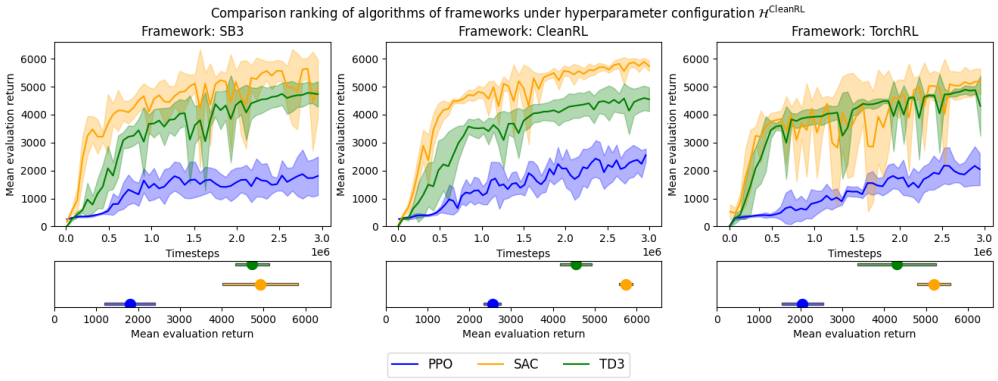


Figure 9: Relative ranking of algorithms of different baseline frameworks under the default hyperparameters of $\mathcal{H}^{\text{CleanRL}}$, with the top plot showing the mean return during evaluation and the bottom plot showing the 95% CI mean at the end of training. Although the relative ranking is preserved, the magnitude of the performance differences between implementations varies across frameworks.

Furthermore, we can also see that in some environments where we have used the default hyperparameters from a single framework, the relative ranking is not preserved. This occurs more often in environments Swimmer-v4 and Walker2d-v4. For example, in Figure 10, we can see a clear segregation in implementation performance. This allows users to choose a specific algorithm of a specific framework to use as a baseline.

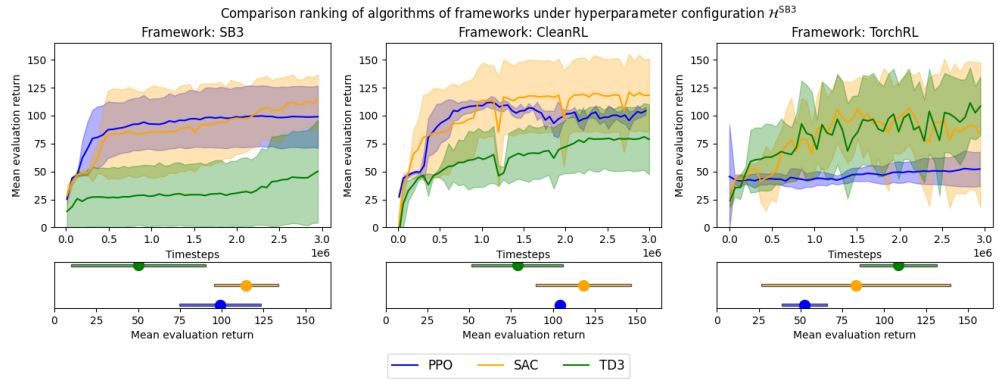


Figure 10: Relative ranking of algorithms of different baseline frameworks under the default hyperparameters of $\mathcal{H}^{\text{SB3}}$, with the top plot showing the mean return during evaluation and the bottom plot showing the 95% CI mean at the end of training. Here, the relative ranking of algorithms is not preserved across different baseline frameworks.

Lastly, we compare the relative ranking of implementations within each baseline framework, using the default hyperparameters provided by that framework. Even under these conditions, we observe large differences. In half of the experiments, it may be argued that the relative ranking between algorithms is not preserved. We acknowledge that there are large differences in the

hyperparameter configurations across baseline frameworks [5]. As stated, baseline frameworks have different design motivations. So the provided hyperparameter configurations may be intended to reach state-of-the-art performance or to generally perform well across tasks. Figure 11 shows an extreme case where the relative ranking of implementations is not preserved across frameworks. Reproducing experiments may be difficult without specifying the source of the algorithm implementation.

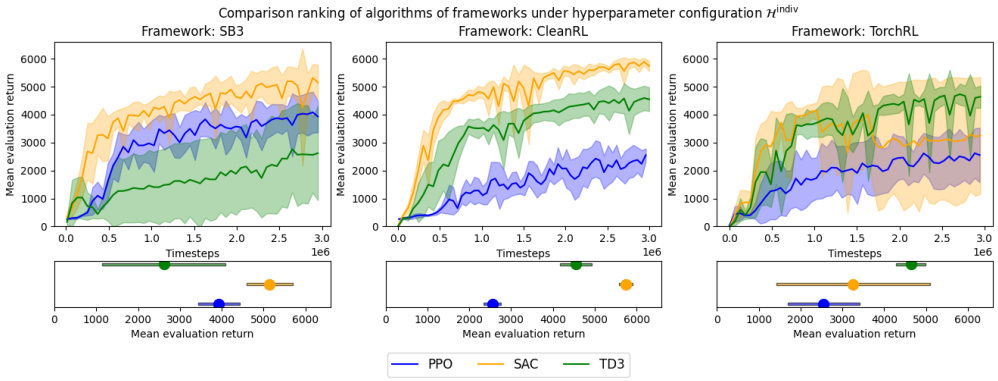


Figure 11: Relative ranking of algorithms of different baseline frameworks under their provided default hyperparameters, with the top plot showing the mean return during evaluation and the bottom plot showing the 95% CI mean at the end of training. The relative ranking is clearly not preserved across baseline frameworks.

7 Discussion

In this section, we discuss the limitations, possible extensions, and conclusions of our work.

7.1 Limitations

In this work, we used a more recent version of MuJoCo, compared to the algorithm papers. The benefit of using a more recent version of MuJoCo is that our results are up to date with current research. However, our results do not fully reproduce the original results. The MuJoCo environment versions v1/v2 and v4 differ in reward functions, weighted penalties, and numerical differences in the physics engine. While we have attempted to perform these experiments on their respective original environment versions, we encountered difficulty running the older `mujoco-py` binary files on the HPC. However, our results still give insight into the differences between implementations.

This work utilizes three popular baseline frameworks and three state-of-the-art recent algorithms. Although these algorithms and baseline frameworks are highly influential, this work remains limited. Implementations using different machine learning backends, or originating from a differently motivated baseline framework, may show additional implementation differences. Furthermore, considering a broader set of algorithms could reveal whether differences

[5] Furthermore, we experimented on the same environment setup, compared to the custom environment setup in $\text{PPO}^{\text{CleanRL}}$

are more frequent in a specific class, like value-based methods or policy-based methods.

Furthermore, we acknowledge that using 5 seeds to show significant differences in performance is the minimum number. Our experiments were limited to 8 environments, specifically the Gymnasium MuJoCo suite. To reduce uncertainty, additional repetitions of each experiment must be required. However, at this scale, repeating experiments is costly. We are using 3 baseline frameworks, 3 algorithms, 8 environments, 4 hyperparameter settings, and 5 repetitions, bringing the total number of experiments to $3 \times 3 \times 8 \times 4 \times 5 = 1440$. Due to time constraints, we were unable to perform an additional 1440 experiments.

We have performed an analysis of code-level differences across implementations from different baseline frameworks. However, we have not thoroughly experimented with the impact of individual differences on performance. Appendix C highlights some of the effects of implementation details. We limited the scope of this work to exposing these differences and will defer experiments on the magnitude of these differences to future work.

In some cases, the results show apparent differences in performance between implementations. This is partly due to the way we have aligned hyperparameters. As each algorithm implementation by a baseline framework could offer a different collection of hyperparameters, we have to distinguish between what is kept constant and what may vary. The assumption that the most important hyperparameters are available to be adjusted in each implementation offers enough flexibility to align these implementations correctly. However, this assumption may not scale in larger settings.

7.2 Future work

There are several options for extensions to this work. For a start, this work is limited to three baseline frameworks and three algorithms. A wider selection of baseline frameworks and algorithms may be evaluated to determine whether differences are prevalent. The scope could be enlarged by using other classes of algorithms and baseline frameworks motivated by different design principles.

In this work, we have experimented with the Gymnasium MuJoCo suite, limiting our research to 8 algorithms. In contrast, the work of Hundal et al. (2025) experimented on the full ALE suite. To expose differences between implementations, a larger, more varied collection of environments may be used.

Lastly, we limited the scope of this work to exposing inconsistent results among algorithms via code-level differences. We did not focus on testing the impact of individual differences; however, we only hypothesized how they may affect algorithm performance. Future work may focus on exposing the relevance of these differences in more detail via an ablation study.

7.3 Conclusion

In this work, we investigated the consistency of the performance of implementations by different baseline frameworks. We have chosen three influential baseline frameworks: Stable Baselines 3, CleanRL, and TorchRL. Furthermore, we have chosen three state-of-the-art algo-

rithms, PPO, TD3, and SAC. We questioned whether the performance of the implementations matches the original results. Additionally, we examined the consistency of the performance implementations under the same hyperparameter configuration. Lastly, we questioned how the relative ranking of implementations by a single baseline framework is preserved across other baseline frameworks.

We conducted a thorough code-level analysis on the implementations to uncover differences. Furthermore, we have performed a reproduction study using the original hyperparameters on the implementations by different baseline frameworks. Additionally, we applied the default hyperparameters provided by the implementations and applied each one to others.

From our analysis, we found many code-level differences. These include differences in methods, adjusted formulas, incorrect methodologies, and variations in the availability of hyperparameters. We used this analysis to clarify differences in performance throughout this work. Furthermore, based on our results, implementations across baseline frameworks are mostly able to match the original results. For the results where implementations perform statistically better or worse, we are able to identify underlying causes. Additionally, the results obtained from implementations of algorithms across baseline frameworks are not always consistent. We observe that in some environments, the implementations produce different results under the same hyperparameter configuration. The largest variability of results are observed across PPO implementations. Similarly, due to the unavailability of hyperparameters, all implementations across baselines slightly differ. Lastly, the relative ranking of implementations by a single baseline framework against other baseline frameworks is mostly preserved. In some cases, we observe a difference in the magnitude of the differences of implementation performance across baseline frameworks.The relative ranking of implementations within each baseline framework, using the default hyperparameters provided by that framework, is not preserved. This suggests that reproduction without access to the source of the implementation becomes unreliable.

The paper by Henderson et al. (2018) has stated that the baseline used to compare the newly implemented algorithms should have matching performance against the original algorithm. We can conclude that the source of the implementation must be included. The assumption that implementations across baseline frameworks are interchangeable does not hold, which is in line with Hundal et al. (2025). Due to small variations between algorithm implementations, even if they are provided by well-established baseline frameworks, they may contain differences that could make relative comparisons unreliable. A mention of the implementation used for development and comparison of algorithms must be included to ensure reliability and consistency.

Lastly, we empirically found aligning hyperparameters to be error-prone. It cannot be assumed that the functionality of a hyperparameter is intuitively apparent. Methods and hyperparameters must have a clear objective that must be apparent to the user. We advise careful experimentation in future work.

Acknowledgements

I would like to thank my supervisor, Álvaro Serra-Gómez, for providing this opportunity, his critical perspective and feedback, and his overall support. Similarly, I would like to thank Felix Kleuker for his input and feedback. Additionally, I would like to thank the ALICE team for providing the computational and storage resources for this work. Last but definitely not least, I would like to thank my family and friends for their insightful comments and discussions, love and care, and endless support throughout this journey.

References

M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: A system for large-scale machine learning, 2016. URL `https://arxiv.org/abs/1605.08695`.

J. Achiam. Spinning Up in Deep Reinforcement Learning. 2018.

A. O. Akman, A. Psarou, Ł. Gorczyca, Z. G. Varga, G. Jamróz, and R. Kucharski. Routerl: Multi-agent reinforcement learning framework for urban route choice with autonomous vehicles. *arXiv preprint arXiv:2502.20065*, 2025.

ALICE - SHARK HPC team. ALICE - SHARK HPC wiki - HPC wiki, 2025. URL `https://pubappslu.atlassian.net/wiki/spaces/HPCWIKI/overview`.

M. Andrychowicz, A. Raichuk, P. Stańczyk, M. Orsini, S. Girgin, R. Marinier, L. Hussenot, M. Geist, O. Pietquin, M. Michalski, et al. What matters for on-policy deep actor-critic methods? a large-scale study. In *International conference on learning representations*, 2021.

M. Babic. *Analysis and Evaluation of Reinforcement Learning Algorithms for a Continuous Control Problem*. PhD thesis, Hochschule für Angewandte Wissenschaften Hamburg, 2024.

M. Baker. 1,500 scientists lift the lid on reproducibility, 2016.

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

J. Benad, F. Röder, and M. Eppe. Scilab-rl: A software framework for efficient reinforcement learning and cognitive modeling research. *SoftwareX*, 29:102064, 2025. ISSN 2352-7110. doi: https://doi.org/10.1016/j.softx.2025.102064. URL `https://www.sciencedirect.com/science/article/pii/S2352711025000317`.

F. Berto, C. Hua, J. Park, L. Luttmann, Y. Ma, F. Bu, J. Wang, H. Ye, M. Kim, S. Choi, et al. Rl4co: an extensive reinforcement learning for combinatorial optimization benchmark. *arXiv preprint arXiv:2306.17100*, 2023.

L. Biewald. Experiment tracking with weights and biases, 2020. URL `https://www.wandb.com/`. Software available from wandb.com.

A. Bou, M. Thomas, S. Dittert, C. N. Ramírez, M. Majewski, Y. Wang, S. Patel, G. Tresadern, M. Ahmad, V. Moens, et al. Acegen: a torchrl-based toolkit for reinforcement learning in generative chemistry. In *ICLR 2024 Workshop on Generative and Experimental Perspectives for Biomolecular Design*.

A. Bou, M. Bettini, S. Dittert, V. Kumar, S. Sodhani, X. Yang, G. D. Fabritiis, and V. Moens. Torchrl: A data-driven decision-making library for pytorch, 2023. URL `https://arxiv.org/abs/2306.00577`.

J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL `http://github.com/jax-ml/jax`.

I. Caspi, G. Leibovich, G. Novik, and S. Endrawis. Reinforcement learning coach, Dec. 2017. URL `https://doi.org/10.5281/zenodo.1134899`.

P. S. Castro, S. Moitra, C. Gelada, S. Kumar, and M. G. Bellemare. Dopamine: A research framework for deep reinforcement learning. *arXiv preprint arXiv:1812.06110*, 2018.

S. Černỳ. Morphological analyser implemented as fsas. 2008.

T. F. Chan, G. H. Golub, and R. J. LeVeque. Updating formulae and a pairwise algorithm for computing sample variances. In *COMPSTAT 1982 5th Symposium held at Toulouse 1982: Part I: Proceedings in Computational Statistics*, pages 30–41. Springer, 1982.

F. Christianos, L. Schäfer, and S. Albrecht. Shared experience actor-critic for multi-agent reinforcement learning. *Advances in neural information processing systems*, 33:10707–10717, 2020.

C. Colas, O. Sigaud, and P.-Y. Oudeyer. How many random seeds? statistical power analysis in deep reinforcement learning experiments. *arXiv preprint arXiv:1806.08295*, 2018.

H. Dai, X. Peng, X. Shi, L. He, Q. Xiong, and H. Jin. Reveal training performance mystery between tensorflow and pytorch in the single gpu environment. *Science China Information Sciences*, 65(1):112103, 2022.

N. De La Fuente and D. A. V. Guerra. A comparative study of deep reinforcement learning models: Dqn vs ppo vs a2c. *arXiv preprint arXiv:2407.14151*, 2024.

C. D'Eramo, D. Tateo, A. Bonarini, M. Restelli, and J. Peters. Mushroomrl: Simplifying reinforcement learning research. *Journal of Machine Learning Research*, 22(131):1–5, 2021. URL `http://jmlr.org/papers/v22/18-056.html`.

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

S. Dittert, V. Moens, and G. De Fabritiis. Bricksrl: A platform for democratizing robotics and reinforcement learning research and education with lego. *Advances in Neural Information Processing Systems*, 37:75199–75228, 2024.

L. Engstrom, A. Ilyas, S. Santurkar, D. Tsipras, F. Janoos, L. Rudolph, and A. Madry. Implementation matters in deep rl: A case study on ppo and trpo. In *International conference on learning representations*, 2019.

L. Espeholt, H. Soyer, R. Munos, K. Simonyan, V. Mnih, T. Ward, Y. Doron, V. Firoiu, T. Harley, I. Dunning, et al. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. In *International conference on machine learning*, pages 1407–1416. PMLR, 2018.

L. Fan, Y. Zhu, J. Zhu, Z. Liu, O. Zeng, A. Gupta, J. Creus-Costa, S. Savarese, and L. Fei-Fei. Surreal: Open-source reinforcement learning framework and robot manipulation benchmark. In *Conference on robot learning*, pages 767–782. PMLR, 2018.

B. Flowers and S. Dey. Bric: Bottom-up residual vector quantization for learned image compression. *IEEE Access*, 2024.

J. Z. Forde and M. Paganini. The scientific method in the science of machine learning. *arXiv preprint arXiv:1904.10922*, 2019.

S. Fujimoto, H. Hoof, and D. Meger. Addressing function approximation error in actor-critic methods. In *International conference on machine learning*, pages 1587–1596. PMLR, 2018.

Y. Fujita, P. Nagarajan, T. Kataoka, and T. Ishikawa. Chainerrl: A deep reinforcement learning library, 2021a. URL `https://arxiv.org/abs/1912.03905`.

Y. Fujita, P. Nagarajan, T. Kataoka, and T. Ishikawa. Chainerrl: A deep reinforcement learning library. *Journal of Machine Learning Research*, 22(77):1–14, 2021b. URL `http://jmlr.org/papers/v22/20-376.html`.

T. garage contributors. Garage: A toolkit for reproducible reinforcement learning research. `https://github.com/rlworkgroup/garage`, 2019.

K. M. Getzandanner and J. R. Martin. Reinforcement learning for spacecraft navigation & environment characterization in the planar-restricted two-body problem. In *2025 AAS/AIAA Spaceflight Mechanics Meeting*, number AAS 25-285, 2025.

S. Guadarrama, A. Korattikara, O. Ramirez, P. Castro, E. Holly, S. Fishman, K. Wang, E. Gonina, N. Wu, E. Kokiopoulou, L. Sbaiz, J. Smith, G. Bartók, J. Berent, C. Harris, V. Vanhoucke, and E. Brevdo. TF-Agents: A library for reinforcement learning in tensorflow. `https://github.com/tensorflow/agents`, 2018. URL `https://github.com/tensorflow/agents`. [Online; accessed 25-June-2019].

O. E. Gundersen and S. Kjensmo. State of the art: Reproducibility in artificial intelligence. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.

T. Haarnoja, H. Tang, P. Abbeel, and S. Levine. Reinforcement learning with deep energy-based policies. In *International conference on machine learning*, pages 1352–1361. PMLR, 2017.

T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*, pages 1861–1870. Pmlr, 2018.

T. Haarnoja, A. Zhou, K. Hartikainen, G. Tucker, S. Ha, J. Tan, V. Kumar, H. Zhu, A. Gupta, P. Abbeel, and S. Levine. Soft actor-critic algorithms and applications, 2019. URL https://arxiv.org/abs/1812.05905.

H. Hasselt. Double q-learning. In J. Lafferty, C. Williams, J. Shawe-Taylor, R. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems*, volume 23. Curran Associates, Inc., 2010. URL https://proceedings.neurips.cc/paper_files/paper/2010/file/091d584fced301b442654dd8c23b3fc9-Paper.pdf.

P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger. Deep reinforcement learning that matters. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.

A. Hill, A. Raffin, M. Ernestus, A. Gleave, A. Kanervisto, R. Traore, P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, and Y. Wu. Stable baselines. https://github.com/hill-a/stable-baselines, 2018.

M. W. Hoffman, B. Shahriari, J. Aslanides, G. Barth-Maron, N. Momchev, D. Sinopalnikov, P. Stańczyk, S. Ramos, A. Raichuk, D. Vincent, et al. Acme: A research framework for distributed reinforcement learning. *arXiv preprint arXiv:2006.00979*, 2020.

S. Huang, R. F. J. Dossa, A. Raffin, A. Kanervisto, and W. Wang. The 37 implementation details of proximal policy optimization. In *ICLR Blog Track*, 2022a. URL https://iclr-blog-track.github.io/2022/03/25/ppo-implementation-details/. https://iclr-blog-track.github.io/2022/03/25/ppo-implementation-details/.

S. Huang, R. F. J. Dossa, C. Ye, J. Braga, D. Chakraborty, K. Mehta, and J. G. Araújo. Cleanrl: High-quality single-file implementations of deep reinforcement learning algorithms. *Journal of Machine Learning Research*, 23(274):1–18, 2022b. URL http://jmlr.org/papers/v23/21-1342.html.

S. Huang, J. Weng, R. Charakorn, M. Lin, Z. Xu, and S. Ontañón. Cleanba: A reproducible and efficient distributed reinforcement learning platform. *arXiv preprint arXiv:2310.00036*, 2023.

R. S. Hundal, Y. Xiao, X. Cao, J. S. Dong, and M. Rigger. On the mistaken assumption of interchangeable deep reinforcement learning implementations. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 2225–2237, 2025. doi: 10.1109/ICSE55347.2025.00222.

M. Hutson. Artificial intelligence faces reproducibility crisis. *Science*, 359(6377):725–726, 2018. doi: 10.1126/science.359.6377.725. URL https://www.science.org/doi/abs/10.1126/science.359.6377.725.

R. Islam, P. Henderson, M. Gomrokchi, and D. Precup. Reproducibility of benchmarked deep reinforcement learning tasks for continuous control. *arXiv preprint arXiv:1708.04133*, 2017.

W. L. Keng and L. Graesser. Slm lab. https://github.com/kengz/SLM-Lab, 2017.

K. Khetarpal, Z. Ahmed, A. Cianflone, R. Islam, and J. Pineau. Re-evaluate: Reproducibility in evaluating reinforcement learning algorithms. 2018.

S. Kolesnikov. Accelerated rl. https://github.com/catalyst-team/catalyst-rl, 2018.

A. Kuhnle, M. Schaarschmidt, and K. Fricke. Tensorforce: a tensorflow library for applied reinforcement learning. Web page, 2017. URL https://github.com/tensorforce/tensorforce.

S. Kullback and R. A. Leibler. On Information and Sufficiency. *The Annals of Mathematical Statistics*, 22(1):79 – 86, 1951. doi: 10.1214/aoms/1177729694. URL https://doi.org/10.1214/aoms/1177729694.

H. Küttler, N. Nardelli, T. Lavril, M. Selvatici, V. Sivakumar, T. Rocktäschel, and E. Grefenstette. Torchbeast: A pytorch platform for distributed rl, 2019. URL https://arxiv.org/abs/1910.03552.

M. Lanctot, E. Lockhart, J.-B. Lespiau, V. Zambaldi, S. Upadhyay, J. Pérolat, S. Srinivasan, F. Timbers, K. Tuyls, S. Omidshafiei, D. Hennes, D. Morrill, P. Muller, T. Ewalds, R. Faulkner, J. Kramár, B. D. Vylder, B. Saeta, J. Bradbury, D. Ding, S. Borgeaud, M. Lai, J. Schrittwieser, T. Anthony, E. Hughes, I. Danihelka, and J. Ryan-Davis. Openspiel: A framework for reinforcement learning in games, 2020. URL https://arxiv.org/abs/1908.09453.

E. Liang, R. Liaw, R. Nishihara, P. Moritz, R. Fox, K. Goldberg, J. Gonzalez, M. Jordan, and I. Stoica. Rllib: Abstractions for distributed reinforcement learning. In *International conference on machine learning*, pages 3053–3062. PMLR, 2018.

T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

X.-Y. Liu, Z. Li, Z. Yang, J. Zheng, Z. Wang, A. Walid, J. Guo, and M. I. Jordan. Elegantrl-podracer: Scalable and elastic library for cloud-native deep reinforcement learning. *NeurIPS, Workshop on Deep Reinforcement Learning*, 2021a.

X.-Y. Liu, H. Yang, J. Gao, and C. D. Wang. Finrl: Deep reinforcement learning framework to automate trading in quantitative finance. In *Proceedings of the second ACM international conference on AI in finance*, pages 1–9, 2021b.

C. I. Lu. Evaluation of deep reinforcement learning algorithms for portfolio optimisation. *arXiv preprint arXiv:2307.07694*, 2023.

M. C. Machado, M. G. Bellemare, E. Talvitie, J. Veness, M. Hausknecht, and M. Bowling. Revisiting the arcade learning environment: evaluation protocols and open problems for general agents. *Journal of Artificial Intelligence Research*, 61:523–562, Mar. 2018. doi: 10.1613/jair.5699. URL https://jair.org/index.php/jair/article/view/11182.

G. Melis, C. Dyer, and P. Blunsom. On the state of the art of evaluation in neural language models. *arXiv preprint arXiv:1707.05589*, 2017.

V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous methods for deep reinforcement learning, 2016. URL https://arxiv.org/abs/1602.01783.

T. Moerland. Continuous markov decision process and policy search. *Lecture notes for the course reinforcement learning, Leiden University*, 28(76):53, 2021.

P. Nagarajan, G. Warnell, and P. Stone. Deterministic implementations for reproducibility in deep reinforcement learning, 2019. URL https://arxiv.org/abs/1809.05676.

Y. Niu, J. Xu, Y. Pu, Y. Nie, J. Zhang, S. Hu, L. Zhao, M. Zhang, and Y. Liu. Di-engine: A universal ai system/engine for decision intelligence. https://github.com/opendilab/DI-engine, 2021.

C. Nota. The autonomous learning library. https://github.com/cpnota/autonomous-learning-library, 2020.

P. S. Nouwou Mindom, A. Nikanjam, and F. Khomh. A comparison of reinforcement learning frameworks for software testing tasks. *Empirical Software Engineering*, 28(5):111, 2023.

O.-C. Novac, M. C. Chirodea, C. M. Novac, N. Bizon, M. Oproescu, O. P. Stan, and C. E. Gordan. Analysis of the application efficiency of tensorflow and pytorch in convolutional neural network. *Sensors*, 22(22):8872, 2022.

F. Pardo, A. Tavakoli, V. Levdik, and P. Kormushev. Time limits in reinforcement learning. *CoRR*, abs/1712.00378, 2017. URL http://arxiv.org/abs/1712.00378.

R. Pascanu, T. Mikolov, and Y. Bengio. On the difficulty of training recurrent neural networks. In *International conference on machine learning*, pages 1310–1318. Pmlr, 2013.

A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019. URL https://arxiv.org/abs/1912.01703.

A. Petrenko, Z. Huang, T. Kumar, G. Sukhatme, and V. Koltun. Sample factory: Egocentric 3d control from pixels at 100000 fps with asynchronous reinforcement learning. In *International Conference on Machine Learning*, pages 7652–7662. PMLR, 2020.

J. Pineau, P. Vincent-Lamarre, K. Sinha, V. Larivière, A. Beygelzimer, F. d'Alché Buc, E. Fox, and H. Larochelle. Improving reproducibility in machine learning research (a report from the neurips 2019 reproducibility program). *Journal of machine learning research*, 22(164):1–20, 2021a.

J. Pineau, P. Vincent-Lamarre, K. Sinha, V. Larivière, A. Beygelzimer, F. d'Alché Buc, E. Fox, and H. Larochelle. Improving reproducibility in machine learning research (a report from the neurips 2019 reproducibility program). *Journal of machine learning research*, 22(164):1–20, 2021b.

M. Plappert. keras-rl. https://github.com/keras-rl/keras-rl, 2016.

B. Polyak. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1–17, 1964. ISSN 0041-5553. doi: https://doi.org/10.1016/0041-5553(64)90137-5. URL https://www.sciencedirect.com/science/article/pii/0041555364901375.

M. L. Puterman. *Markov Decision processes*. 4 1994. doi: 10.1002/9780470316887. URL https://doi.org/10.1002/9780470316887.

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

A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of machine learning research*, 22 (268):1–8, 2021.

R. Ring. Reaver: Modular deep reinforcement learning framework. https://github.com/inoryy/reaver, 2018.

G. Schäfer, M. Schirl, J. Rehrl, S. Huber, and S. Hirlaender. Python-based reinforcement learning on simulink models. In *International Conference on Soft Methods in Probability and Statistics*, pages 449–456. Springer, 2024.

C. A. Schiller. Virtual augmented reality for atari reinforcement learning. *arXiv preprint arXiv:2310.08683*, 2023.

J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897. PMLR, 2015a.

J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015b.

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

H. Semmelrock, S. Kopeinik, D. Theiler, T. Ross-Hellauer, and D. Kowald. Reproducibility in machine learning-driven research. *arXiv preprint arXiv:2307.10320*, 2023.

C. E. Shannon. A mathematical theory of communication. *The Bell system technical journal*, 27(3):379–423, 1948.

D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller. Deterministic policy gradient algorithms. In *International conference on machine learning*, pages 387–395. Pmlr, 2014.

A. Silvestri, D. Coraci, D. Wu, E. Borkowski, and A. Schlueter. Comparison of two deep reinforcement learning algorithms towards an optimal policy for smart building thermal control. In *Journal of Physics: Conference Series*, volume 2600, page 072011. IOP Publishing, 2023.

A. Stooke and P. Abbeel. rlpyt: A research code base for deep reinforcement learning in pytorch. *arXiv preprint arXiv:1909.01500*, 2019.

J. Suarez. Pufferlib: Making reinforcement learning libraries and environments play nice, 2024. URL https://arxiv.org/abs/2406.12905.

J. Suarez, D. Bloomin, K. W. Choe, H. X. Li, R. Sullivan, N. Kanna, D. Scott, R. Shuman, H. Bradley, L. Castricato, et al. Neural mmo 2.0: A massively multi-task addition to massively multi-agent learning. *Advances in Neural Information Processing Systems*, 36:50094–50104, 2023.

R. S. Sutton. Learning to predict by the methods of temporal differences. *Machine learning*, 3:9–44, 1988.

R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation. *Advances in neural information processing systems*, 12, 1999.

S. Thrun and A. Schwartz. Issues in using function approximation for reinforcement learning. In *Proceedings of the 1993 connectionist models summer school*, pages 255–263. Psychology Press, 2014.

Y. Tian, Q. Gong, W. Shang, Y. Wu, and C. L. Zitnick. Elf: An extensive, lightweight and flexible research platform for real-time strategy games. *Advances in Neural Information Processing Systems*, 30, 2017.

D. Tiapkin, N. Morozov, A. Naumov, and D. P. Vetrov. Generative flow networks as entropy-regularized rl. In *International Conference on Artificial Intelligence and Statistics*, pages 4213–4221. PMLR, 2024.

E. Todorov, T. Erez, and Y. Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033, 2012. doi: $10.1109/\mathrm{IROS}.2012.6386109$.

M. Towers, A. Kwiatkowski, J. Terry, J. U. Balis, G. De Cola, T. Deleu, M. Goulão, A. Kallinteris, M. Krimmel, A. KG, et al. Gymnasium: A standard interface for reinforcement learning environments. *arXiv preprint arXiv:2407.17032*, 2024.

M. Towers, A. Kwiatkowski, J. Terry, J. U. Balis, G. De Cola, T. Deleu, M. Goulão, A. Kallinteris, M. Krimmel, A. KG, et al., Feb. 2025. URL `https://farama.org/Vector-Autoreset-Mode`.

G. Tucker, S. Bhupatiraju, S. Gu, R. Turner, Z. Ghahramani, and S. Levine. The mirage of action-dependent baselines in reinforcement learning. In *International conference on machine learning*, pages 5015–5024. PMLR, 2018.

C. P. Wan, T. Li, and J. M. Wang. Rlor: A flexible framework of deep reinforcement learning for operation research. *arXiv preprint arXiv:2303.13117*, 2023.

Q. Wang, J. Xiong, L. Han, M. Fang, X. Sun, Z. Zheng, P. Sun, and Z. Zhang. Arena: a toolkit for multi-agent reinforcement learning, 2019. URL `https://arxiv.org/abs/1907.09467`.

B. P. Welford. Note on a method for calculating corrected sums of squares and products. *Technometrics*, 4(3):419–420, 1962. doi: $10.1080/00401706.1962.10490022$. URL `https://www.tandfonline.com/doi/abs/10.1080/00401706.1962.10490022`.

J. Weng, H. Chen, D. Yan, K. You, A. Duburcq, M. Zhang, Y. Su, H. Su, and J. Zhu. Tianshou: a highly modularized deep reinforcement learning library, 2022a. URL https://arxiv.org/abs/2107.14171.

J. Weng, M. Lin, S. Huang, B. Liu, D. Makoviichuk, V. Makoviychuk, Z. Liu, Y. Song, T. Luo, Y. Jiang, Z. Xu, and S. Yan. Envpool: A highly parallel reinforcement learning environment execution engine, 2022b. URL https://arxiv.org/abs/2206.10558.

R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3–4):229–256, May 1992. doi: 10.1007/bf00992696. URL https://link.springer.com/article/10.1007/BF00992696.

M. Wolk, A. Applebaum, C. Dennler, P. Dwyer, M. Moskowitz, H. Nguyen, N. Nichols, N. Park, P. Rachwalski, F. Rau, et al. Beyond cage: Investigating generalization of learned autonomous network defense policies. *arXiv preprint arXiv:2211.15557*, 2022.

Y. Wu, E. Mansimov, R. B. Grosse, S. Liao, and J. Ba. Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation. *Advances in neural information processing systems*, 30, 2017.

R. Young and N. Pugeault. Enhancing robustness in deep reinforcement learning: A lyapunov exponent approach. *Advances in Neural Information Processing Systems*, 37:86102–86123, 2024.

L. Zhang, Z. Ji, and B. Chen. Crew: Facilitating human-ai teaming research. *arXiv preprint arXiv:2408.00170*, 2024.

S. Zhang, S. Nandakumar, Q. Pan, E. Yang, R. Migne, and L. Subramanian. Benchmarking reinforcement learning algorithms on island microgrid energy management. In *2021 IEEE PES Innovative Smart Grid Technologies - Asia (ISGT Asia)*, pages 1–5, 2021. doi: 10.1109/ISGTAsia49270.2021.9715570.

L. Zheng, J. Yang, H. Cai, W. Zhang, J. Wang, and Y. Yu. Magent: A many-agent reinforcement learning platform for artificial collective intelligence, 2017. URL https://arxiv.org/abs/1712.00600.

J. B. D. J. Y. W. Y. E. R. X. L. W. H. G. A. N. D. K. U. D. F. C. Z. W. W. X. Zheqing Zhu, Rodrigo de Salvo Braz. Pearl: A production-ready reinforcement learning agent. 2023.

J. Zhi, R. Wang, J. Clune, and K. O. Stanley. Fiber: A platform for efficient development and distributed training for reinforcement learning and population-based methods, 2020. URL https://arxiv.org/abs/2003.11164.

B. D. Ziebart. *Modeling purposeful adaptive behavior with the principle of maximum causal entropy*. Carnegie Mellon University, 2010.

A Notation

Symbol	Explanation
$\mathcal{S}$	The state space
$\mathcal{A}$	The action space
P	The transition function
r	A reward
p_0	The starting state distribution
s	A state within the statespace $\mathcal{S}$
$\dim \mathcal{S}$	The dimensionality of the state space
a	An action within the action space $\mathcal{A}$
$\dim \mathcal{A}$	The dimensionality of the action space
$p(\mathcal{S})$	A distribution over the state space
t	Time or timestep
s_0	The starting state
τ	A trace or episode
T	The final timestep or total amount of timesteps
$R(\tau)$	The return: the discounted cumulative rewards of a trace τ
γ	The discount factor
$G_t(\tau)$	The return starting from time t
$J(\theta)$	The objective function
$p_{\pi_\theta(\tau)}$	The distribution of traces under policy π_θ
π	A policy
θ	Parameters defining the policy
$\pi_\theta(a\|s)$	A stochastic policy parameterized by θ
$\pi_\theta(s)$	A deterministic policy parameterized by θ
$\mathcal{N}(\mu, \sigma^2)$	A Gaussian distribution with mean μ and standard deviation σ^2
a_i	An action value in dimension i, where $0 \leq i \leq \mathcal{A} - 1$
$\mathcal{A}_{\text{low}}$	The lower bound of the action space
$\mathcal{A}_{\text{high}}$	The higher bound of the action space
$\text{clip}(x, l, u)$	Clipping function, defined by $\min(\max(a, l), u)$
V^ϕ	A state-value function, parameterized by ϕ
ϕ	Parameters defining the state-value function
δ_t	Temporal difference between the cumulative expected return via the state-value function, and the state-value of the current state.
Q^ϕ	A state-action value function, parameterized via ϕ
$A_t(s_t, a_t)$	The advantage function, which is defined as $Q(s, a) - V(s)$
$H(s)$	The entropy function, defined as $\mathbb{E}_{a \sim \pi_\theta(a\|s)}[-\log \pi_\theta(a\|s)\pi_\theta(a\|s)]$
$\mathcal{H}^{FW}$	A hyperparameter configuration from a framework FW. The algorithm is implicitly tied to the hyperparameter configuration.
ALG^{FW}	An algorithm [ALG] implemented by a specific framework FW
Ψ_t	A performance measure
$r(\theta)$	The probability ratio between current and old policies π_θ and $\pi_{\theta_{\text{old}}}$
KL	Kullback-Leibler Divergence
c	A threshold concerning the constraint in TRPO
β	Alternative parameter weighing the penalty of the constraint
ϵ	A policy clipping parameter used in PPO

B	A batch
$\|B\|$	Size of the batch
A^{GAE}	Advantage via Generalized Advantage Estimation
λ	Bias-variance trade-off hyperparameter in GAE
α_π	The policy network learning rate
M	A minibatch
$\|M\|$	Size of the minibatch
L^π	Policy loss
L^V	State-value loss
ϵ_V	A value clipping parameter used in PPO
L_H	Entropy loss
c_1	State-value loss weight
c_2	Entropy loss weight
$\mathcal{D}$	Replay buffer
$\|\mathcal{D}\|$	Size of the replay buffer
$Q^{\phi'}$	A target state-action value function, parameterized via ϕ'
σ^2	The standard deviation term of the Gaussian noise added in action selection
$\sigma^{2'}$	Target noise scale Gaussian standard deviation
b	The target noise boundary
y	The TD target
τ_p	The Polyak averaging tau
$f_\theta(\epsilon, s)$	The reparameterization trick, via $\mu_\theta(s) + \sigma_\theta(s) \cdot \epsilon$
α	The temperature parameter
α_V	The state-value network learning rate
α_H	The temperature network learning rate
L^α	The entropy loss

B Experimentation codebase

In this section, we will concisely explain the API that allowed us to experiment on a large scale. The code is published in its entirety on this GitHub repository.

Every framework has its own requirements, and we adhered to these requirements in this work. This means that we cannot use a single collection of packages and one Python version. Instead, we used the environment management system Conda. To set up the environments, simply run

```
bash setup.sh -all
```

which sets up frameworks Stable Baselines 3, CleanRL, and TorchRL in Python 3.9.0, Python 3.8.3, and Python 3.9.21, respectively. Furthermore, it installs the required packages provided by the installation guide for each framework. The sequential use of `yml` files and `sh` files allows us to set up the frameworks completely according to their installation guide, along with an accessible way to include frameworks beyond the scope of this research. The correct environment is automatically used for experimentation. To make a specific algorithm work with our framework, a stub between the automatically called file `train_[alg].py` and the

baseline framework's implementation of said algorithm has to be hand-coded.

Through SLURM jobs, we have access to perform repeated experiments in parallel. We included a simple file to set up SLURM jobs. To run a job, enter

```
bash job.sh [--fw FW] [--alg ALG] [--env ENV] [--steps STEPS]
    [--rep REP] [--time TIME] [--par PAR] [--mem MEM] [--hps HPS]
```

The following parameters can be supplied:

- The framework `fw` $\in$ {SB3, CleanRL, TorchRL};
- The algorithm `alg` $\in$ {ppo, td3, sac};
- The environment `env`, which must be compatible with the Gymnasium framework. Some suites, like ALE, require additional setup. A version of the environment must be provided, e.g. `env=HalfCheetah-v4`;
- The total training timesteps `steps`;
- The number of repetitions `rep`, which is denoted by an SLURM array job [1-`rep`];
- The total allowed computation time `time`. We suggest 24:00:00, 48:00:00, and 72:00:00 for the algorithms ppo, td3, and sac, respectively;
- The computation partition `par`;
- The total allowed memory usage `mem`. We suggest `mem=4G`;
- The hyperparameter configuration `hps`, which is a path to a YAML file.

The flexibility of providing the path of the hyperparameters instead of the individual hyperparameters allows for high maintenance. Furthermore, it keeps a clear overview of what hyperparameters are being used in each experiment.

The results of each experiment can be found within `./[fw]/results/[alg]_[env]_[steps]/[hps]`, where each repetition of an experiment is placed within a different folder. Each SLURM output file contains the list of hyperparameters used, such that each experiment can easily be identified. A collection of jobs can be found in `./exps`. As a sidenote, each experiment can be run individually and outside of the SLURM job context via

```
conda activate [fw]
python ./[fw]/train_[alg].py [--env ENV] [--steps STEPS] [--hps HPS]
```

where the parameters within brackets are similar to the aforementioned parameters.

We provide some utility whilst experimenting, such as formatting YAML hyperparameter files, environment wrapper functions, and job generators. Furthermore, HPCs may fail or re-queue jobs, which may stay unnoticed during large-scale experimentation. The file `checkruns.py` brute-force reads out each experiment to validate it.

The plots of the results within this research can be generated via the respective plotting files. A cache is saved when reading out an experiment, which allows faster plot generation. Beware that if experiments are rerun, this cache must be manually removed, or it will use the old results when plotting.

C Analysis code-level differences

This section explores the effect of some of the code-level differences we found across frameworks. Since the emphasis is on exposing differences rather than measuring their influence on algorithm performance, this subject is not the main focus of this work. However, we would like to emphasize some of the differences and their effects to illustrate how they contribute to our results.

C.1 Wrappers

Environment wrappers may help stabilize training via feedback transformations, but can also be utilized for input transformation and statistical logging. In this section, we will cover the feedback-altering wrappers that are applied in PPO$^{\text{CleanRL}}$ and how they affect performance.

Action clipping Action clipping is an input-altering wrapper for environments to ensure $a \sim \pi_\theta(a|s)$ is bounded $[\mathcal{A}_{\text{low}}, \mathcal{A}_{\text{high}}]$. This is done via $\text{clip}(a, \mathcal{A}_{\text{low}}, \mathcal{A}_{\text{high}})$.

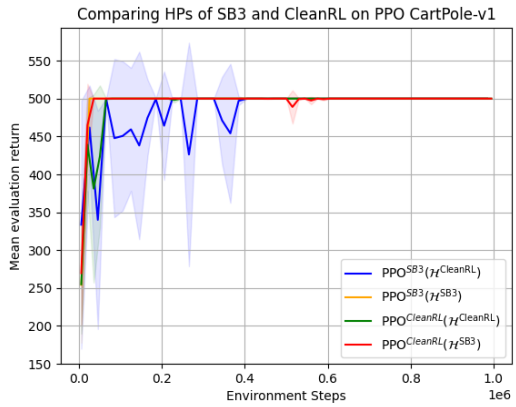
Normalization Normalization wrappers ensure that observations and/or rewards are normalized with unit variance. An internal distribution is tracked, and the mean and standard deviation are updated with each instance (Welford, 1962; Chan et al., 1982). This distribution is also used during the evaluation of $\pi_\theta(a|s)$. A copy of the distribution is used such that it is not updated during evaluation. Rewards are not normalized during evaluation.

Transformation Transformation wrappers apply a mapping $f : \mathbb{R}^{\dim x} \to \mathbb{R}^{\dim x}$ to some instance x. For instance, x could be clipped or scaled to some domain.

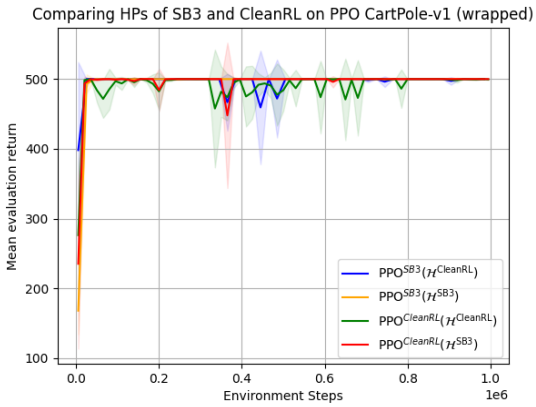
PPO$^{\text{CleanRL}}$ comes equipped with action clipping, observation and reward normalization, and observation and reward transformation wrappers. To illustrate the effects of these wrappers, we performed experiments where we consistently applied these wrappers across PPO$^{\text{SB3}}$ and PPO$^{\text{CleanRL}}$. We use both unwrapped and wrapped environment setups across three environments that vary in difficulty. These can be observed in Figure 12. We can see that the impact of wrappers is more significant, the more difficult the environment. Furthermore, especially apparent in Ant-v4, it illustrates how the ranking of the implementations is affected.

C.2 Vectorized environments

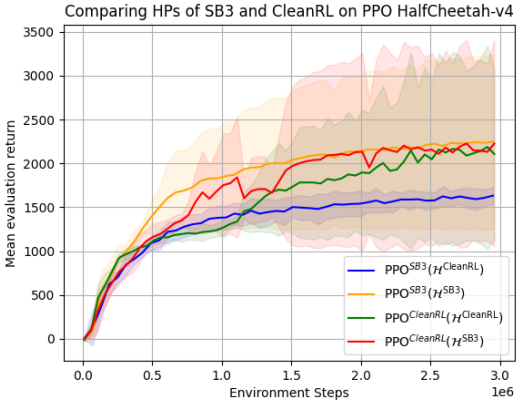
Environment vectorization refers to the process of using multiple environments in parallel. This speeds up training by more efficient use of computational resources (Liang et al., 2018; Raffin et al., 2021). Furthermore, the support of asynchronous vectorization speeds up computation by eliminating the requirement for all environments to finish (Weng et al., 2022b). However, there is a trade-off between the number of parallel environments and performance. Using



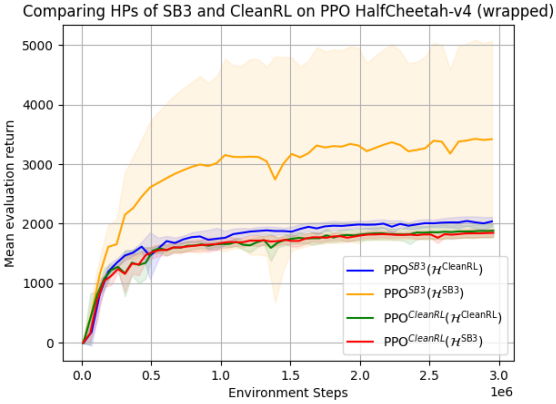
(a) Unwrapped CartPole-v1



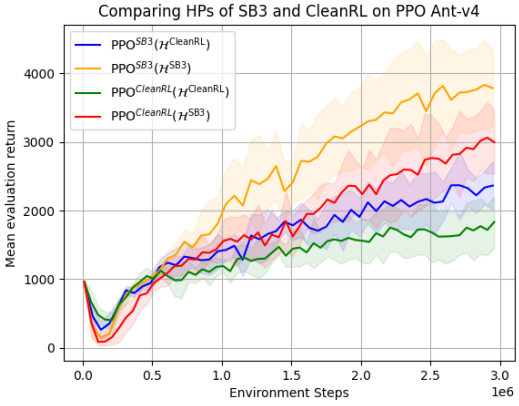
(b) Wrapped CartPole-v1



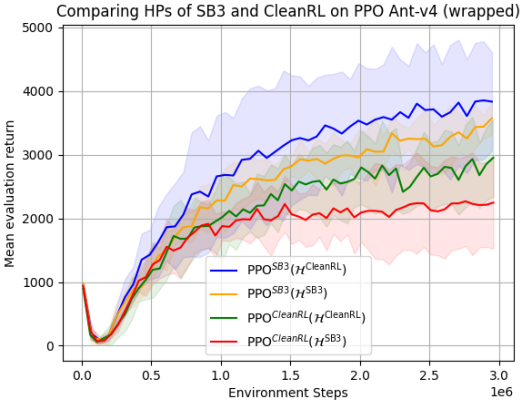
(c) Unwrapped HalfCheetah-v4



(d) Wrapped HalfCheetah-v4



(e) Unwrapped Ant-v4



(f) Wrapped Ant-v4

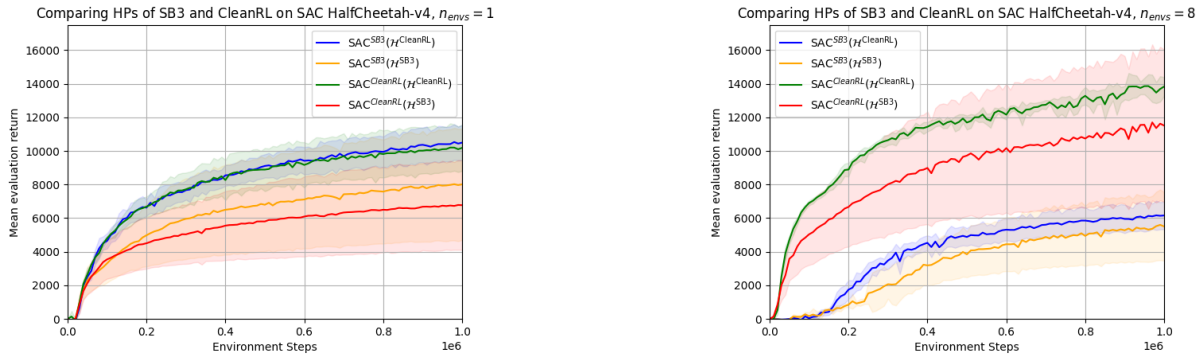
Figure 12: Pairwise PPO^{SB3} and $\text{PPO}^{\text{CleanRL}}$ with $\mathcal{H}^{\text{SB3}}$ and $\mathcal{H}^{\text{CleanRL}}$ applied on CartPole-v1 (ab), HalfCheetah-v4 (cd), and Ant-v4 (ef). The environments in plots a, c, and e use no feedback-altering wrappers. The environments in plots b, d, and f use observation normalization and transformation ($\text{clip}(s, -10, 10)$) and reward normalization and transformation ($\text{clip}(r, -10, 10)$).

fewer parallel environments is often more stable, but it is slow. In contrast, more parallel

environments yield non-diverse data, but cover more of the statespace (Mnih et al., 2016). A collection of algorithms is designed to approach this problem, for instance, via V-trace (Espeholt et al., 2018) or via importance sampling (Christianos et al., 2020).

Vectorization may affect plug-and-play usage, limiting the set of supported environments (Petrenko et al., 2020). Furthermore, differences in data collection may appear. Depending on how vectorization is implemented, the last transition of an episode might differ in what next state is returned: s_0 or s_T (Towers et al., 2025). Huang et al. (2022b) has pointed out that the results produced at the time with their EnvPool (Weng et al., 2022b) implementation are inconsistent with Gymnasium (Towers et al., 2024), but they claim to believe it has not had much impact.

The definition of vectorization should be similar across implementations. During experimentation, we empirically found that vectorization did not yield any positive time difference in CleanRL: both unvectorized and vectorized environments took roughly the same time to train, considering that every other influential parameter is constant between the two experiments. This led to the experiment seen in Figure 13, where we only differed in the usage of vectorization applied in SAC. Due to SAC$^{\text{PPO}}$ not accounting for the multiplied amount of samples seen per timestep, it processes a factor n_{envs} more samples.



(a) No vectorization ($n_{envs} = 1$)

(b) Vectorization with $n_{envs} = 8$

Figure 13: Task HalfCheetah-v4 applied on SAC with its respective hyperparameters. Plot (a) shows this experiment with a single environment whilst training, whilst plot (b) shows this experiment using 8 parallel environments. CleanRL does not account for more sampling if using $n_{envs} > 1$: it processes 8 times more samples per accounted timestep.

C.3 Action selection

We found that action space enforcement differs among all implementations. The two methods, clipping and $\tanh$ with upscaling, are assumed to be interchangeable. This has led us to look into the effects of the action distributions that both methods yield.

Given a Gaussian distribution $\mathcal{N}(\mu_\theta(s), \sigma^2_\theta(s))$ with parameterized μ_θ and σ^2_θ dependent on state s as the untransformed distribution, we can note drawing action a from this distribution as

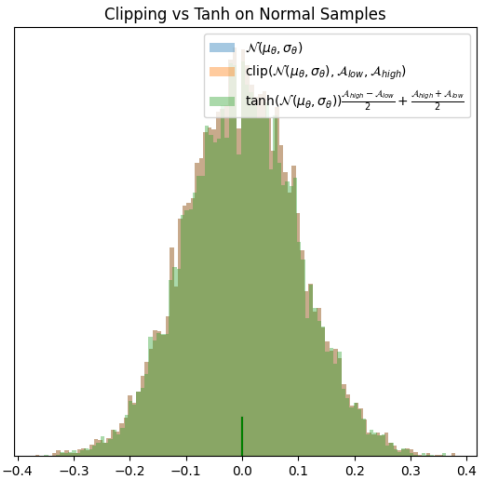
$$a \sim \mathcal{N}(\mu_\theta(s), \sigma^2_\theta(s))$$

Furthermore, we can define the Gaussian distributions to follow the aforementioned methods as

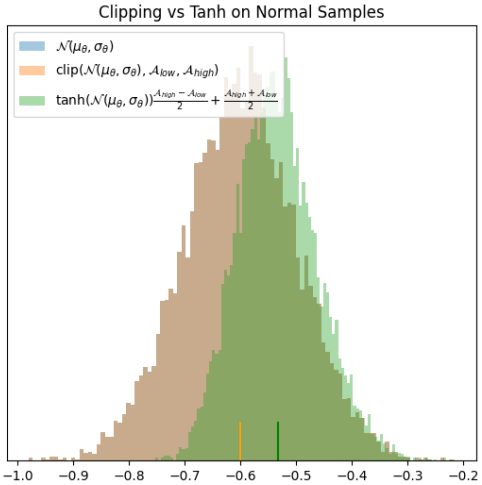
$$\mathcal{N}^{\text{clip}} := \text{clip}(\mathcal{N}(\mu_\theta(s), \sigma_\theta^2(s)), \mathcal{A}_{\text{low}}, \mathcal{A}_{\text{high}})$$
$$\mathcal{N}^{\tanh} := \tanh(\mathcal{N}(\mu_\theta(s), \sigma_\theta^2(s)))\frac{\mathcal{A}_{\text{high}} - \mathcal{A}_{\text{low}}}{2} + \frac{\mathcal{A}_{\text{high}} + \mathcal{A}_{\text{low}}}{2}$$

We may assume that a policy becomes more deterministic during training. This leads the value of $\sigma_\theta^2(s)$ to approach 0. With small $\sigma_\theta^2(s)$, we notice that the distributions $\mathcal{N}^{\text{clip}}$ and $\mathcal{N}^{\tanh}$ remain similar around $\mu_\theta = 0$. However, the distributions differ when μ_θ approaches the action space bounds $[\mathcal{A}_{\text{low}}, \mathcal{A}_{\text{high}}]$.

Figure 14 shows both distributions using $\mu_\theta = 0$ and $\mu_\theta = -0.6$ for small $\sigma_\theta^2 = 0.1$. It illustrates that the distributions may differ in both distribution and mean, with the latter influencing the deterministic behavior of the policy. This could impact training time when the optimal $\mu_\theta^*(s)$ is close to the action space boundaries. Furthermore, it may affect exploration at the start of training, as $\mathcal{N}^{\text{clip}}$ favors actions near the action space bounds, as opposed to a uniform exploration via $\mathcal{N}^{\tanh}$. Lastly, it shows that implementations where a different action space enforcement method is used have non-interchangeable network parameters.



(a) $\mu_\theta = 0, \sigma_\theta^2 = 0.1, [\mathcal{A}_{\text{low}}, \mathcal{A}_{\text{high}}] = [-1, 1]$



(b) $\mu_\theta = -0.6, \sigma_\theta^2 = 0.1, [\mathcal{A}_{\text{low}}, \mathcal{A}_{\text{high}}] = [-1, 1]$

Figure 14: The action distributions under the aforementioned actionspace-bounding methodologies. Assuming a progressively more deterministic policy $\pi_\theta(a|s)$, we illustrate how the action distributions across both methods differ. The mean of the distributions is given by the darker vertical line extending from the x-axis. Denote the original distribution as exactly overlapping with $\mathcal{N}^{\text{clip}}$.

D Hyperparameter definitions

Hyperparameter	Symbol	Equivalent						ppo
		SB3		CleanRL		TorchRL		
Learning rate in Adam optimizer	α	`learning_rate`	0.0003	`learning_rate`	0.0003	`lr`	0.0003	0.0003
Learning rate linear annealing		-*	-	`anneal_lr`	True	`anneal_lr`	True	False
Number of steps per environment update	$\|B\|$	`n_steps`	2048	`num_steps`	2048	`frames_per_batch`	2048	2048
Minibatch size	$\|M\|$	`batch_size`	64	-	-	`mini_batch_size`	64	64
Number of minibatches		-	-	`num_minibatches`	32	-	-	-
Updating epochs		`n_epochs`	10	`update_epochs`	10	`ppo_epochs`	10	10
Discount factor	γ	`gamma`	0.99	`gamma`	0.99	`gamma`	0.99	0.99
GAE-lambda factor	λ	`gae_lambda`	0.95	`gae_lambda`	0.95	`gae_lambda`	0.95	0.95
Clipping parameter	ϵ	`clip_range`	0.2	`clip_coef`	0.2	`clip_epsilon`	0.2	0.2
Value clipping parameter	ϵ_V	`clip_range_vf`	None	`clip_vloss`	True	`clip_value`**	False	False
Normalize advantages		`normalize_advantage`	True	`norm_adv`	True	-	True	True
Entropy loss coefficient	c_2	`ent_coef`	0.0	`ent_coef`	0.0	`entropy_coef`	0.0	0.0
Value loss coefficient	c_1	`vf_coef`	0.5	`vf_coef`	0.5	`critic_coef`	0.25	0.5
Maximum gradient norm		`max_grad_norm`	0.5	`max_grad_norm`	0.5	-	-	0.5
Early stopping target KL-divergence		`target_kl`	None	`target_kl`	None	-	-	None

Table 6: PPO hyperparameters

* a linear annealing schedule is used to match linear annealing when using other hyperparameter configurations.
** not provided in the `config_ppo.yml` file, but is part of the PPO class.

In terms of minibatch size calculation in CleanRL, this is done at runtime to correctly handle parallel environments. Using just a single environment whilst training, this is calculated via $\texttt{minibatch\_size} = \frac{\texttt{num\_steps}}{\texttt{num\_minibatches}}$

The network architecture used is similar across frameworks: a neural network with 2 hidden layers, each consisting of 64 nodes. $\tanh$ activation is used between the layers. Orthogonal initialization is also used (as a constant) among all frameworks, although initial scales and biases may differ (see Section 5.1).

All hyperparameters in *italics* are considered constants: they are left at their respective default values.

Hyperparameter	Symbol	Equivalent SB3		Equivalent CleanRL		Equivalent TorchRL		td3
Learning rate in Adam optimizer	α	`learning_rate`	0.0003	`learning_rate`	0.0003	`lr`	0.0003	0.0003
Batch size for updating	N	`batch_size`	256	`batch_size`	256	`batch_size`	256	256
The size of the replay buffer	$\|\mathcal{D}\|$	`buffer_size`	10^6	`buffer_size`	10^6	`size`	10^6	10^6
Transitions collected before learning starts		`learning_starts`	100	`learning_starts`	25000	`init_random_frames`	25000	25000
Polyak update coefficient	τ_p	`tau`	0.005	`tau`	0.005	`tau`	0.995*	0.005
Discount factor	γ	`gamma`	0.99	`gamma`	0.99	`gamma`	0.99	0.99
Transitions collected before updating		`train_frequency`	1	-	-	`frames_per_batch`	1000	1
Number of gradient updates after rollout		`gradient_steps`	1	-	-	-**	-	1
Exploration noise	σ	`action_noise`	None	`exploration_noise`	0.1	-	0.1	0.1
Delay policy and target networks update		`policy_delay`	2	`policy_frequency`	2	`policy_update_delay`	2	2
Standard deviation of noise applied to target actor	σ'	`target_policy_noise`	0.2	`policy_noise`	0.2	`policy_noise`	0.2	0.2
Absolute value for smoothing target actor noise	c	`target_noise_clip`	0.5	`noise_clip`	0.5	`noise_clip`	0.5	0.5
Ratio data collection to gradient steps						`utd_ratio`	1	
Network architecture		`net_arch`	400,300	-	256,256	`hidden_layers`	256,256	256,256

Table 7: TD3 hyperparameters

* The polyak update parameter in TorchRL's implementation is flipped, such that the actual usage is $1 - \tau$. TorchRL needs to retrieve this value to align with the other frameworks. This is taken into account when aligning the hyperparameters accordingly.

** The number of gradient steps is determined by `frames_per_batch` and `utd_ratio`. Since this is 1:1, it implicitly means that `frames_per_batch` number of gradient update steps are done per `frames_per_batch` transitions collected.

All hyperparameters in *italics* are considered constants: they are left at their respective default values.

Hyperparameter	Symbol	Equivalent SB3		Equivalent CleanRL		Equivalent TorchRL		sac
Policy learning rate in Adam optimizer	α_π	`learning_rate`	0.0003	`policy_lr`	0.0003	`lr`	0.0003	0.0003
Q-value learning rate in Adam optimizer	α_V	`learning_rate`	0.0003	`q_lr`	0.001	`lr`	0.0003	0.0003
Temperature learning rate in Adam optimizer	α_H	`ent_coef`	$1 - \frac{t}{T}$	`q_lr`	0.001	-	0.0003	0.0003
Batch size for updating	N	`batch_size`	256	`batch_size`	256	`batch_size`	256	256
Size of the replay buffer	$\lvert\mathcal{D}\rvert$	`buffer_size`	10^6	`buffer_size`	10^6	`size`	10^6	10^6
Transitions collected before learning starts		`learning_starts`	100	`learning_starts`	5000	`init_random_frames`	25000	0
Polyak update coefficient	τ_p	`tau`	0.005	`tau`	0.005	`tau`	0.995*	0.005
Discount factor	γ	`gamma`	0.99	`gamma`	0.99	`gamma`	0.99	0.99
Transitions collected before updating		`train_frequency`	1	-	-	`frames_per_batch`	1000	1
Number of gradient updates after rollout		`gradient_steps`	1	-	-	-**	-	1
Delay policy updates		-	-	`policy_frequency`	2	-	-	-
Entropy regularization coefficient initial value		`ent_coef`	1.0	`alpha`	1.0	`alpha_init`	1.0	1.0
Target network update interval		`target_update_interval`	1	`target_network_frequency`	1	-	1	1
Target entropy		`target_entropy`	$-\dim \mathcal{A}$	-	$-\dim \mathcal{A}$	-	-	$-\dim \mathcal{A}$

Table 8: SAC hyperparameters

* The polyak update parameter in TorchRL's implementation is flipped, such that the actual usage is $1 - \tau$. TorchRL needs to retrieve this value to align with the other frameworks. This is taken into account when aligning the hyperparameters accordingly.

** The number of gradient steps is determined by `frames_per_batch` and `utd_ratio`. Since this is 1:1, it implicitly means that `frames_per_batch` number of gradient update steps are done per `frames_per_batch` transitions collected. Furthermore, the original implementation of SAC uses reward scaling. This is not used throughout experimentation.

All hyperparameters in *italics* are considered constants: they are left at their respective default values.

E All experimental results

HPS	FW	HalfCheetah-v4	Ant-v4	Hopper-v4	Walker2d-v4
SB3	SB3	2241.9±873.1	3889.4±606.8	2626.3±360.1	**4034.8±450.3**
SB3	CleanRL	2242.9±1060.0	3059.1±481.3	2681.0±291.1	2728.3±559.2
SB3	TorchRL	3485.0±1229.9	3469.6±733.9	2701.7±477.2	2658.4±688.0
CleanRL	SB3	1646.4±83.6	2445.7±274.6	2785.2±459.2	1711.8±514.9
CleanRL	CleanRL	2100.4±834.3	1771.9±339.4	**1457.5±687.3**	2329.2±302.1
CleanRL	TorchRL	**3889.0±1086.1**	**3763.7±618.9**	2736.0±490.0	2073.6±500.0
TorchRL	SB3	2114.7±840.1	3688.3±607.2	3171.3±417.6	2406.9±957.6
TorchRL	CleanRL	1702.0±125.2	**2607.0±523.1**	**2216.7±451.0**	2803.7±662.0
TorchRL	TorchRL	3129.6±1564.2	3715.5±932.4	3272.4±368.3	2770.4±766.1
ppo	SB3	2839.1±1087.6	**2709.6±394.1**	1626.0±389.8	1736.8±538.0
ppo	CleanRL	1947.1±709.0	1960.9±410.9	2792.9±852.0	2471.4±763.8
ppo	TorchRL	3093.9±1124.9	1256.8±520.4	1557.3±751.3	1459.9±815.7

HPS	FW	Inv.Pend.-v4	Inv.Doub.Pend.-v4	Swimmer-v4	Reacher-v4
SB3	SB3	1000.0±0.0	7537.4±1024.4	98.7±23.8	-4.5±0.5
SB3	CleanRL	1000.0±0.0	**9351.9±13.1**	104.6±2.4	-6.0±1.0
SB3	TorchRL	1000.0±0.0	7709.9±1540.6	**51.9±13.4**	-5.4±1.0
CleanRL	SB3	987.5±22.0	9354.1±9.3	47.0±2.2	-4.2±0.4
CleanRL	CleanRL	971.7±49.6	9349.0±11.4	50.1±1.5	-5.1±0.7
CleanRL	TorchRL	1000.0±0.0	8693.6±459.2	**78.0±21.2**	-5.6±1.0
TorchRL	SB3	1000.0±0.0	7598.6±1400.8	81.6±27.6	**-4.4±0.2**
TorchRL	CleanRL	1000.0±0.0	**9339.1±15.5**	68.9±19.3	-5.6±0.8
TorchRL	TorchRL	1000.0±0.0	8005.3±1115.7	61.2±15.2	-5.8±0.6
ppo	SB3	1000.0±0.0	1642.9±1446.2	**121.1±4.1**	-4.1±0.3
ppo	CleanRL	1000.0±0.0	**9357.8±2.8**	86.6±17.2	-4.9±0.3
ppo	TorchRL	1000.0±0.0	4044.5±2883.7	53.7±23.2	-5.6±1.3

Table 9: Mean and CI 95%. of PPO of the last 30 evaluation episodes, in line with Figures 15 and 18.

HPS	FW	HalfCheetah-v4	Ant-v4	Hopper-v4	Walker2d-v4
SB3	SB3	**4804.3±2282.7**	3690.6±290.0	1566.4±1520.6	**2618.0±1580.9**
SB3	CleanRL	9851.9±3300.1	3013.3±1050.0	2328.2±1119.2	4910.6±344.7
SB3	TorchRL	9037.4±3041.8	4470.5±954.3	**3405.3±452.7**	5446.7±298.0
CleanRL	SB3	**12030.0±425.0**	4956.5±2301.8	**3006.3±1076.8**	4359.0±922.8
CleanRL	CleanRL	13213.2±664.8	**5390.3±967.1**	3683.3±36.1	4548.4±372.0
CleanRL	TorchRL	14168.3±546.5	6676.4±221.3	3667.9±27.7	4845.0±168.0
TorchRL	SB3	**12056.9±1087.8**	6054.5±202.7	**3617.4±33.8**	5153.5±565.1
TorchRL	CleanRL	13228.6±507.6	**5392.3±819.7**	2931.9±1131.7	5136.1±217.8
TorchRL	TorchRL	13805.6±728.4	6397.6±373.4	2310.0±1015.6	4710.6±405.4
td3	SB3	**11984.2±741.5**	6498.1±299.8	3535.7±135.9	5035.5±443.7
td3	CleanRL	13554.0±665.4	**5676.5±915.0**	3608.0±81.4	4911.7±324.5
td3	TorchRL	13314.5±664.3	6511.8±235.3	3649.4±71.8	4547.1±148.3

HPS	FW	Inv.Pend.-v4	Inv.Doub.Pend.-v4	Swimmer-v4	Reacher-v4
SB3	SB3	800.6±349.6	4792.2±3646.6	50.1±40.2	-4.7±0.6
SB3	CleanRL	860.2±245.1	4055.0±3792.2	78.8±27.4	-3.7±0.2
SB3	TorchRL	1000.0±0.0	7491.9±3268.0	105.3±26.6	-3.8±1.3
CleanRL	SB3	1000.0±0.0	7462.1±3256.6	125.0±12.1	-4.4±0.3
CleanRL	CleanRL	1000.0±0.0	5716.5±3897.6	135.6±2.2	-3.7±0.2
CleanRL	TorchRL	838.6±282.9	9345.0±8.4	134.8±3.2	-3.3±1.3
TorchRL	SB3	962.1±66.4	9319.2±0.5	121.3±12.1	-4.8±0.4
TorchRL	CleanRL	1000.0±0.0	6200.7±3471.3	133.6±11.8	-3.5±0.1
TorchRL	TorchRL	1000.0±0.0	7598.3±3052.3	128.7±12.4	-4.2±0.9
td3	SB3	1000.0±0.0	9318.4±1.5	127.3±7.1	**-5.0±0.8**
td3	CleanRL	1000.0±0.0	7482.5±3266.2	110.7±27.1	-3.6±0.2
td3	TorchRL	1000.0±0.0	9336.9±8.2	128.4±7.5	-3.3±1.1

Table 10: Mean and CI 95%. of TD3 of the last 30 evaluation episodes, in line with Figures 16 and 19.

HPS	FW	HalfCheetah-v4	Ant-v4	Hopper-v4	Walker2d-v4
SB3	SB3	**4366.1±2997.6**	5746.4±748.5	2595.6±880.8	5063.2±719.6
SB3	CleanRL	10856.9±3683.7	5286.2±931.8	3026.4±637.4	5504.1±389.7
SB3	TorchRL	12691.1±333.4	5548.0±1864.5	2947.9±773.2	4783.6±1759.8
CleanRL	SB3	15949.7±133.7	4245.0±2646.6	2509.1±600.6	5706.4±350.7
CleanRL	CleanRL	14899.1±733.4	5297.1±849.5	2858.5±382.7	5736.4±164.1
CleanRL	TorchRL	9985.1±4420.4	**6796.0±249.1**	3061.5±768.7	**5233.9±461.7**
TorchRL	SB3	15047.3±423.6	4924.8±1152.4	2055.9±585.3	**5580.5±217.8**
TorchRL	CleanRL	15867.5±208.3	5157.0±1146.1	3071.5±515.9	4393.2±914.1
TorchRL	TorchRL	**11216.3±1967.5**	4853.8±3448.7	2997.7±1081.9	3719.3±1442.6
sac	SB3	11219.1±3868.6	5304.8±616.4	2417.8±546.6	**5600.1±208.8**
sac	CleanRL	10901.1±3786.6	5221.5±1142.5	2731.7±585.4	4523.7±1042.2
sac	TorchRL	10443.0±3447.9	**6817.6±232.2**	3276.6±871.9	3143.3±2290.5

HPS	FW	Inv.Pend.-v4	Inv.Doub.Pend.-v4	Swimmer-v4	Reacher-v4
SB3	SB3	1000.0±0.0	9359.3±0.6	114.4±20.4	-3.7±0.2
SB3	CleanRL	1000.0±0.0	9353.2±7.3	118.2±28.5	-3.5±0.1
SB3	TorchRL	800.6±349.6	7497.9±3257.9	80.0±52.8	-3.7±0.6
CleanRL	SB3	1000.0±0.0	9359.4±0.4	**108.3±10.1**	-3.6±0.2
CleanRL	CleanRL	1000.0±0.0	9352.4±7.1	71.3±24.5	-3.6±0.2
CleanRL	TorchRL	1000.0±0.0	9357.4±1.9	67.1±39.4	-3.4±0.8
TorchRL	SB3	1000.0±0.0	9357.1±1.4	108.3±27.0	-3.6±0.2
TorchRL	CleanRL	859.5±246.3	9353.7±2.2	71.7±32.9	-3.5±0.2
TorchRL	TorchRL	800.6±349.6	9357.3±1.2	93.2±31.4	-4.0±0.6
sac	SB3	1000.0±0.0	9358.6±0.8	**114.7±26.6**	-3.4±0.3
sac	CleanRL	1000.0±0.0	9354.3±4.5	74.3±24.5	-3.5±0.1
sac	TorchRL	1000.0±0.0	7492.8±3268.3	43.4±37.6	-2.7±1.5

Table 11: Mean and CI 95%. of SAC of the last 30 evaluation episodes, in line with Figures 17 and 20.

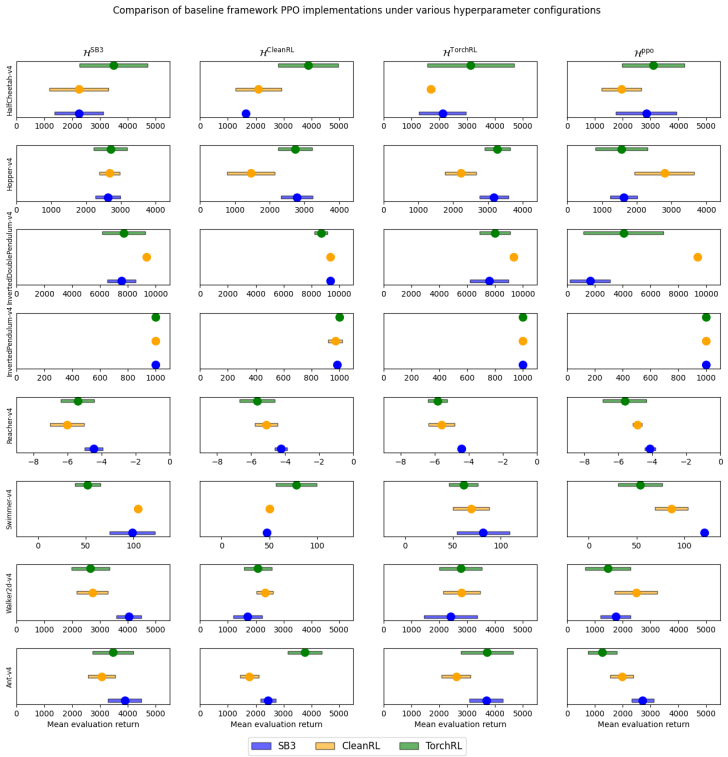


Figure 15: The 95% confidence interval for the mean performance for PPO under different hyperparameter configurations and various frameworks.

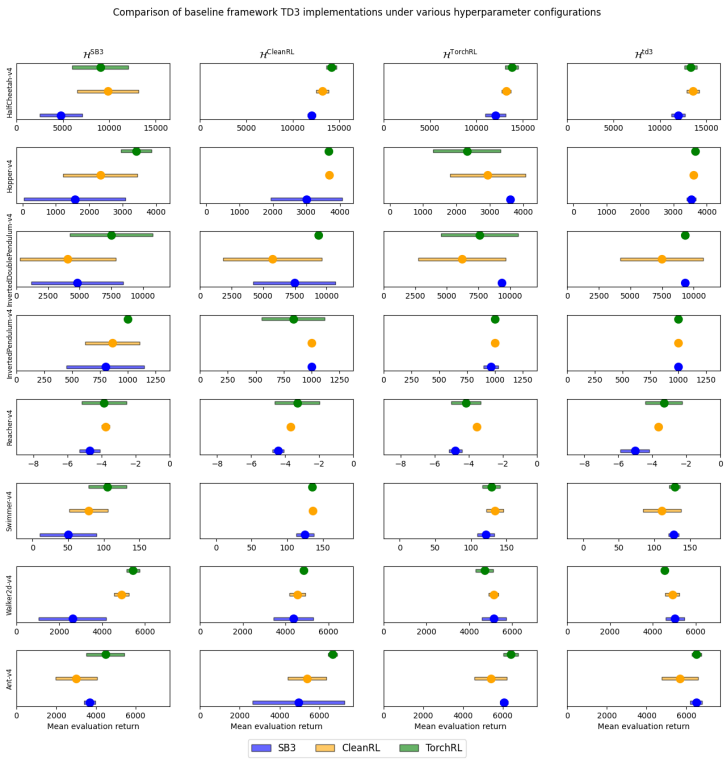


Figure 16: The 95% confidence interval for the mean performance for TD3 under different hyperparameter configurations and various frameworks.

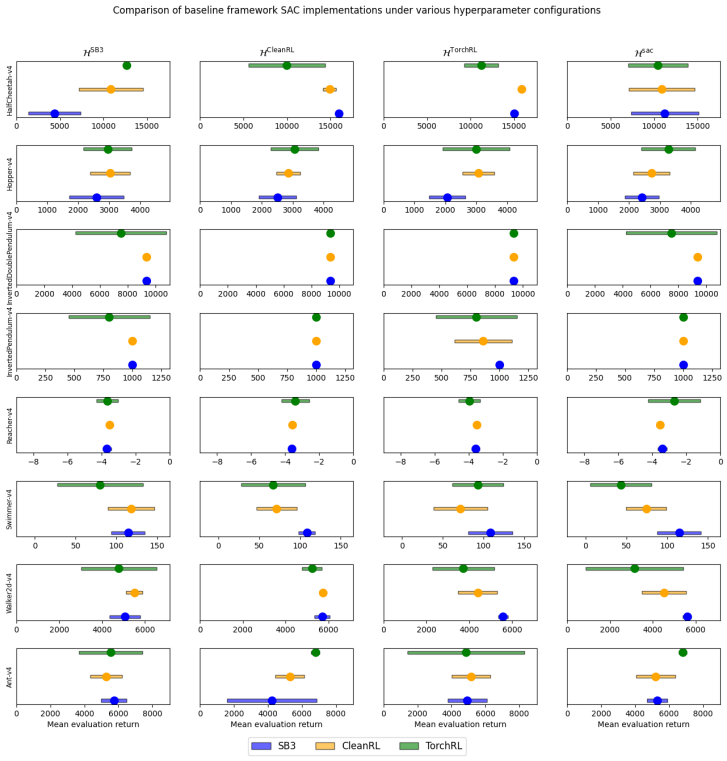


Figure 17: The 95% confidence interval for the mean performance for SAC under different hyperparameter configurations and various frameworks.

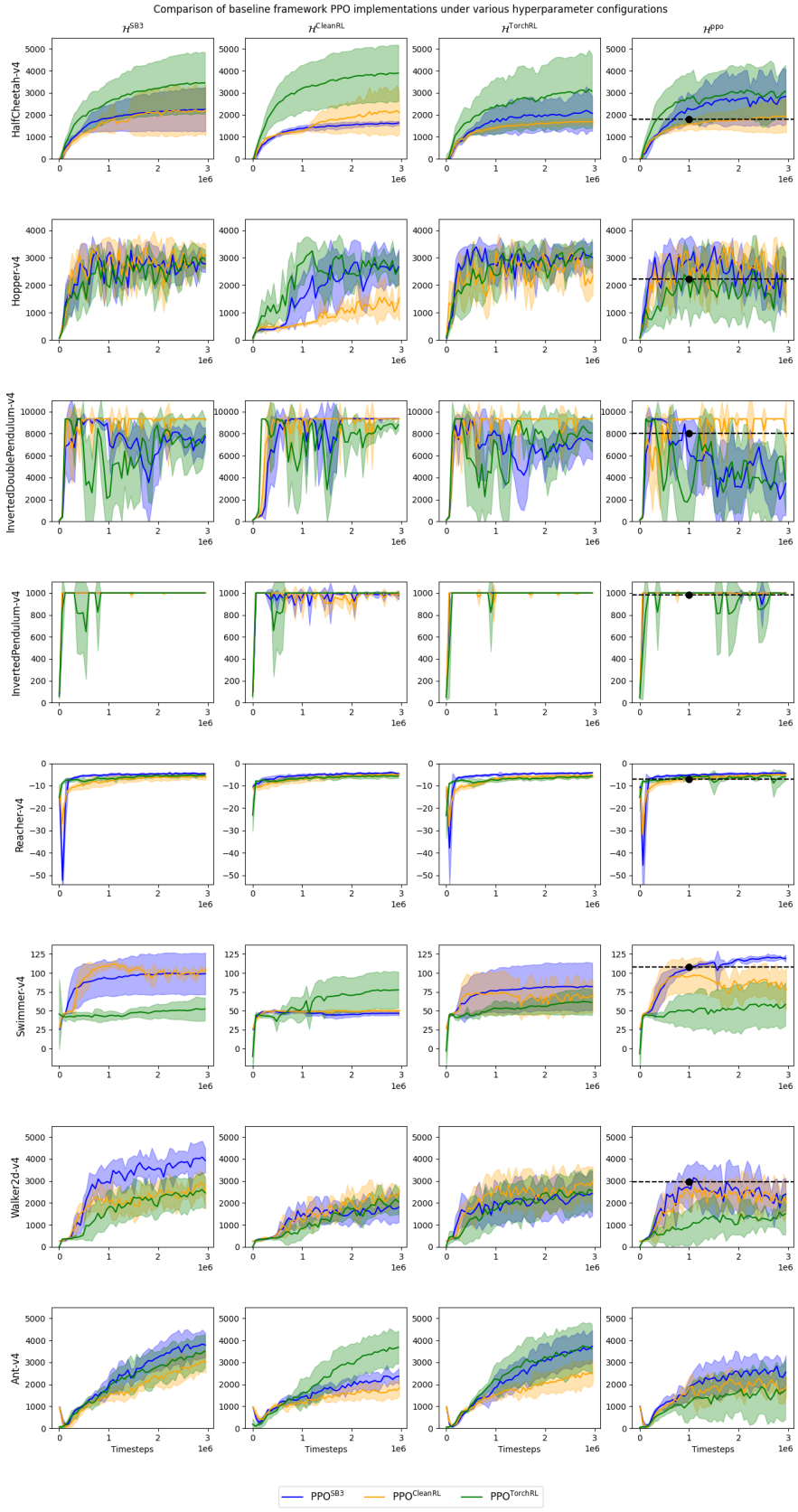


Figure 18: A comparison between different implementations of PPO by baseline frameworks, using the default hyperparameter set (columns) of each baseline framework's implementation of PPO, with the additional original hyperparameter configuration as given by Schulman et al. (2017).

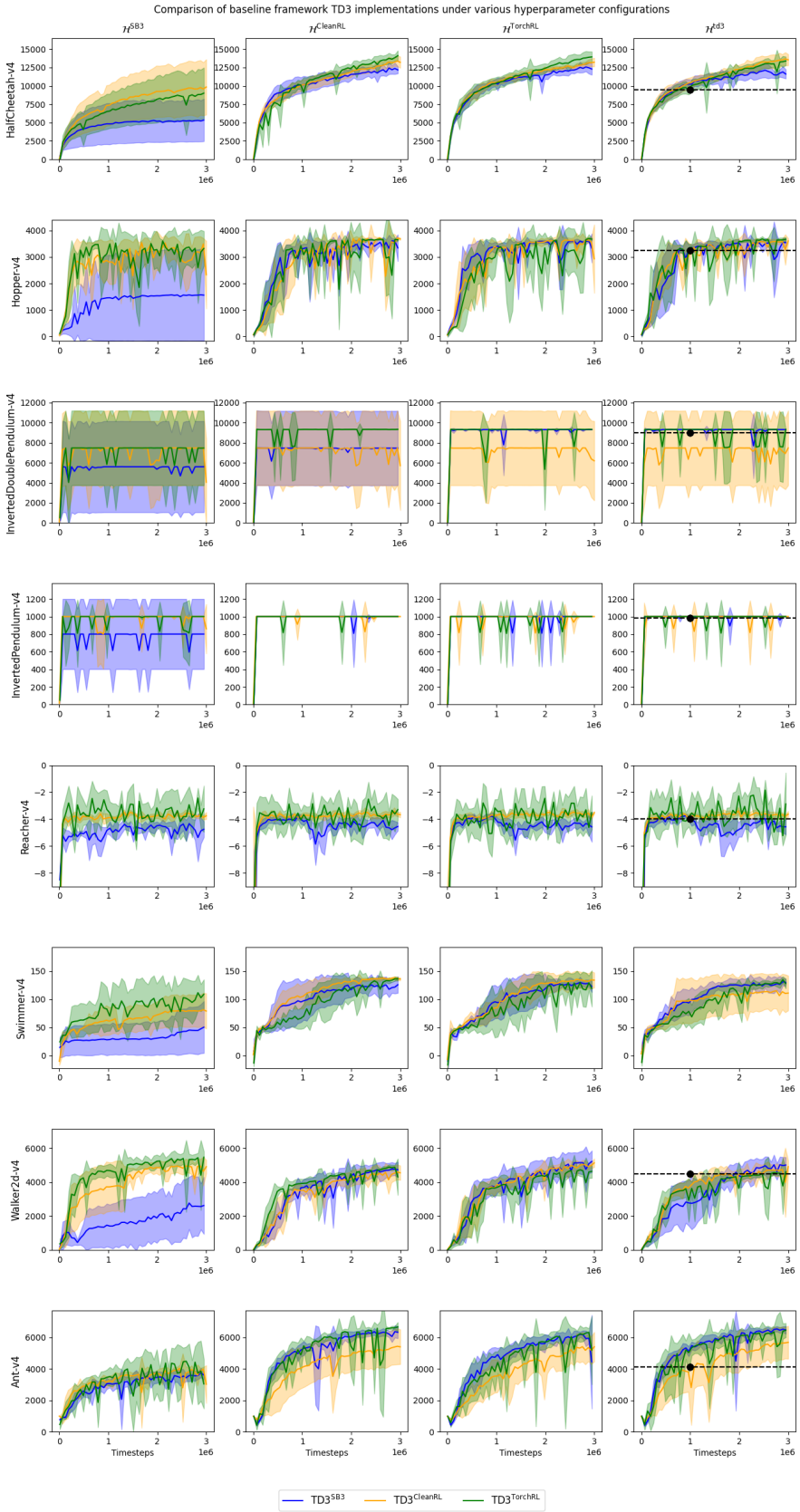


Figure 19: A comparison between different implementations of TD3 by baseline frameworks, using the default hyperparameter set (columns) of each baseline framework's implementation of TD3, with the additional original hyperparameter configuration as given by Fujimoto et al. (2018).

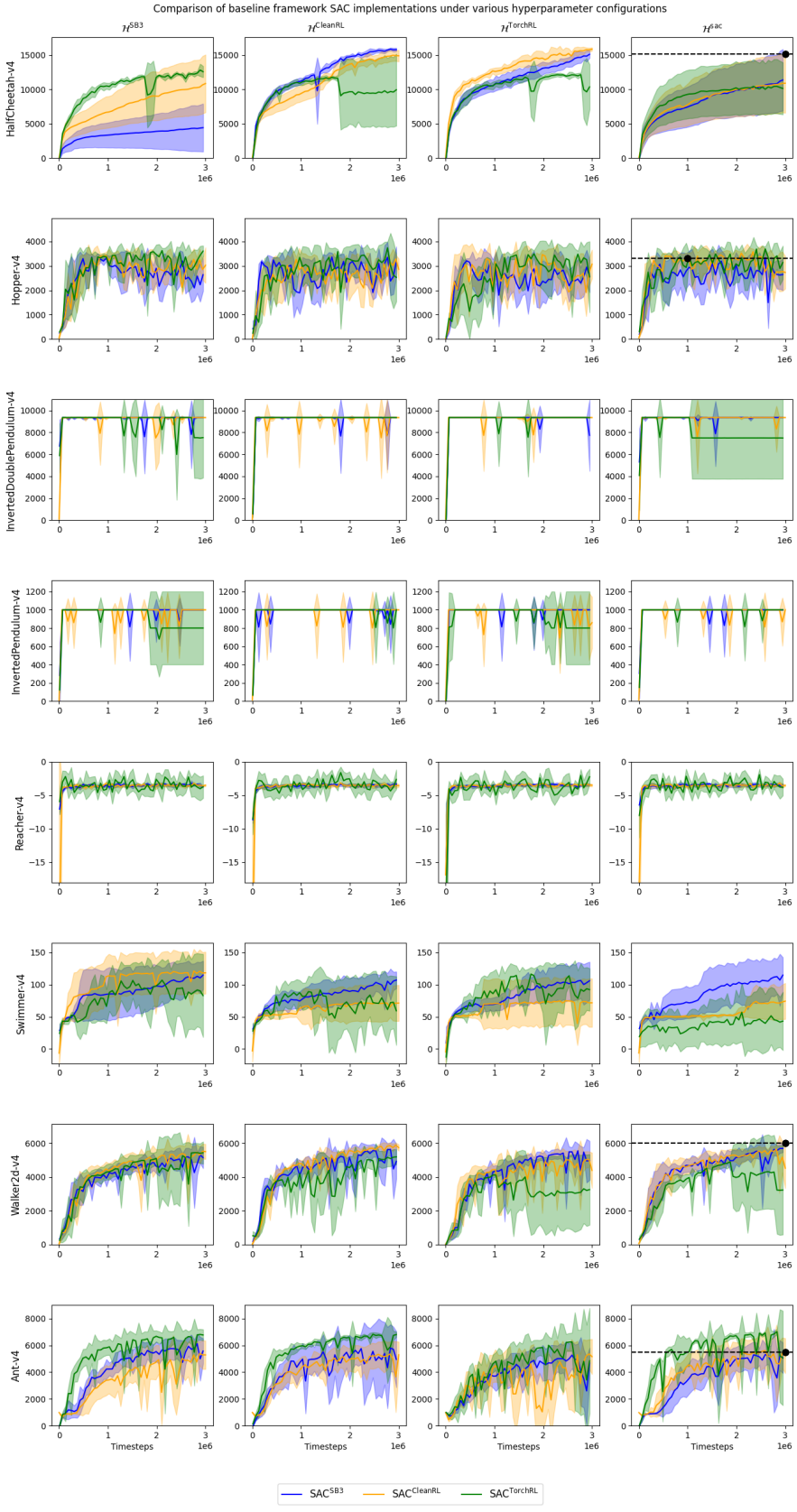


Figure 20: A comparison between different implementations of SAC by baseline frameworks, using the default hyperparameter set (columns) of each baseline framework's implementation of SAC, with the additional original hyperparameter configuration as given by Haarnoja et al. (2019).

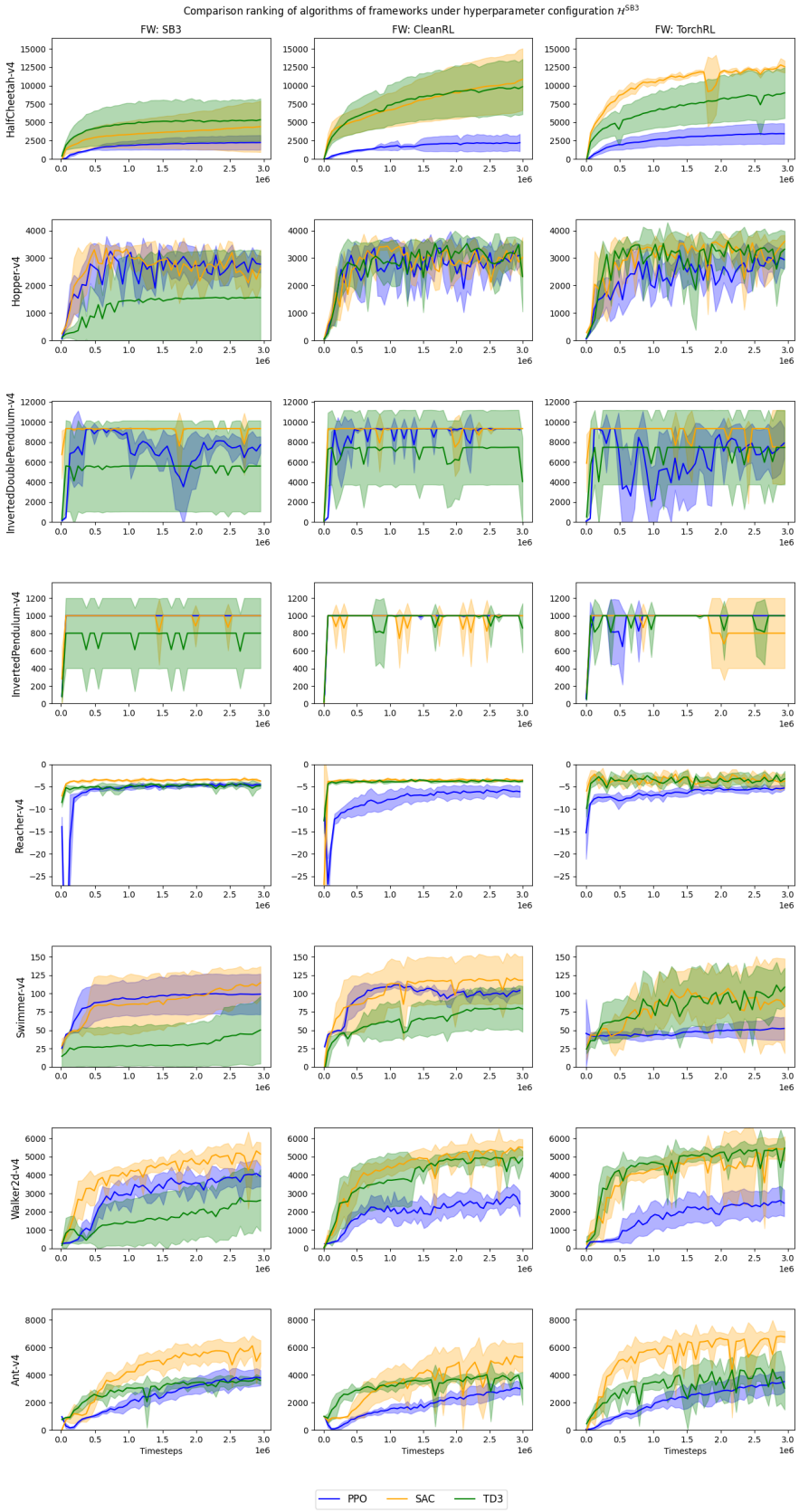


Figure 21: A comparison between the relative ranking of the algorithms PPO, TD3, and SAC, by each baseline framework (columns) under the default hyperparameters of Stable Baselines 3 for each algorithm.

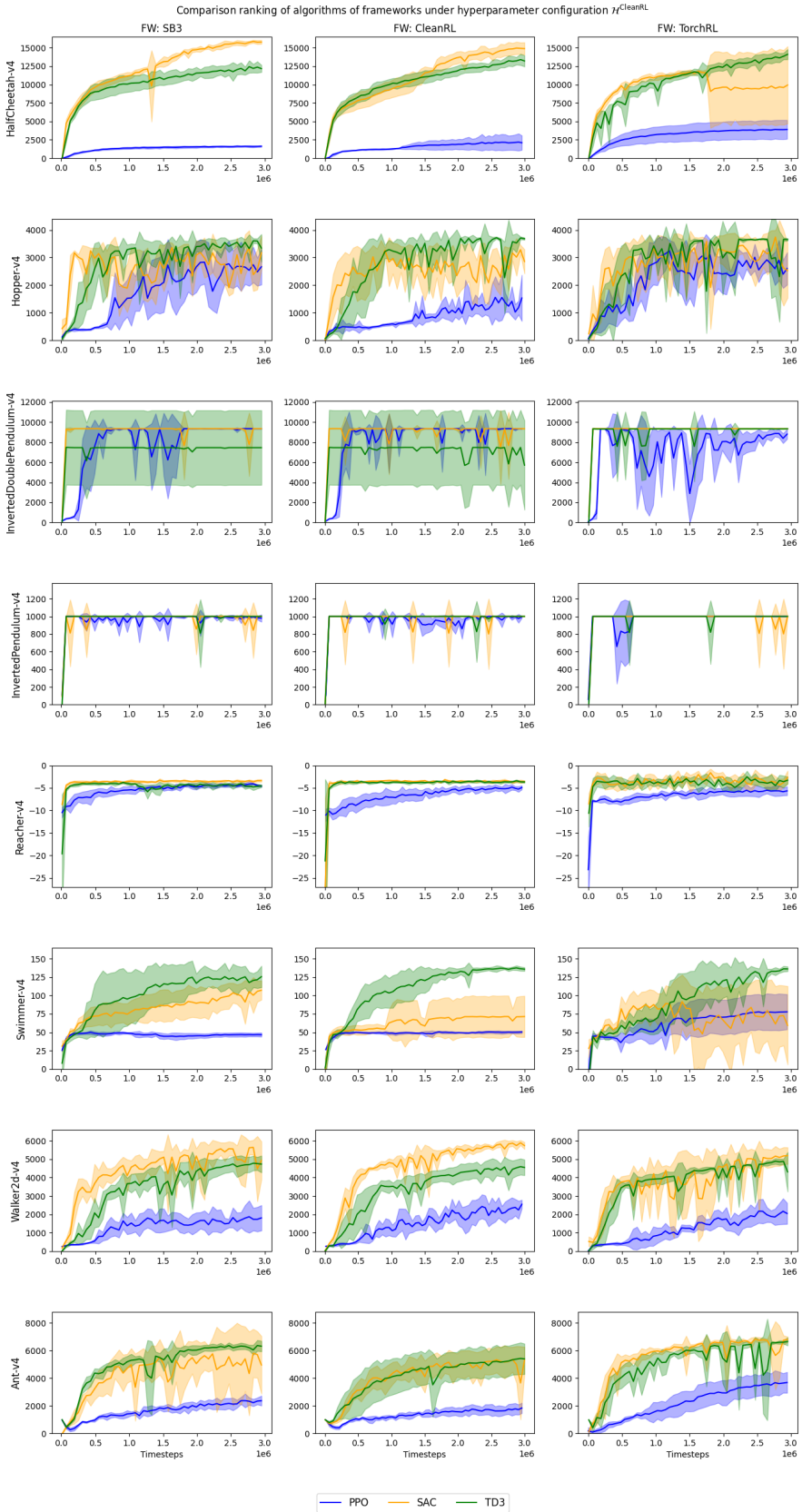


Figure 22: A comparison between the relative ranking of the algorithms PPO, TD3, and SAC, by each baseline framework (columns) under the default hyperparameters of CleanRL for each algorithm.

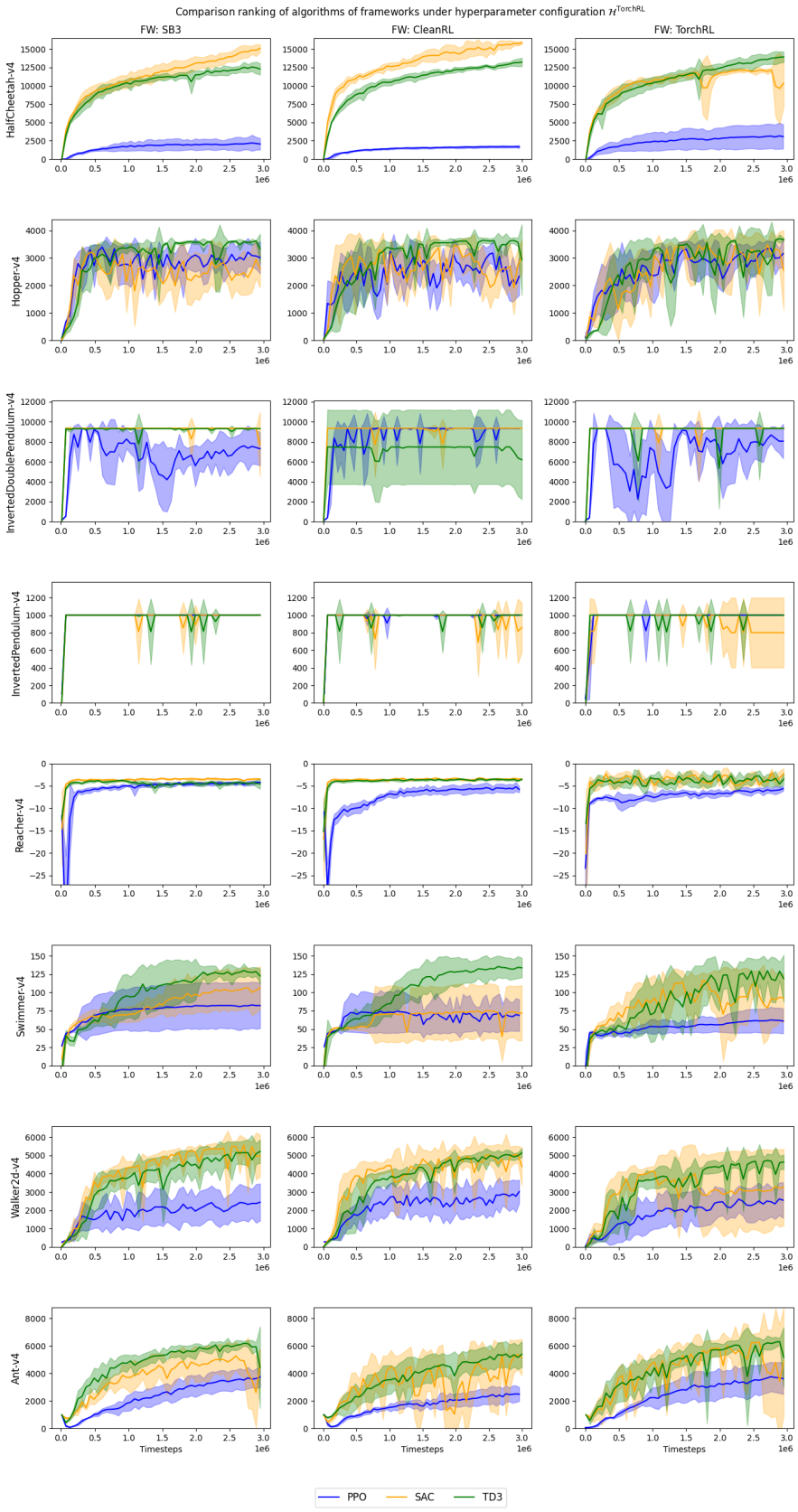


Figure 23: A comparison between the relative ranking of the algorithms PPO, TD3, and SAC, by each baseline framework (columns) under the default hyperparameters of TorchRL for each algorithm.

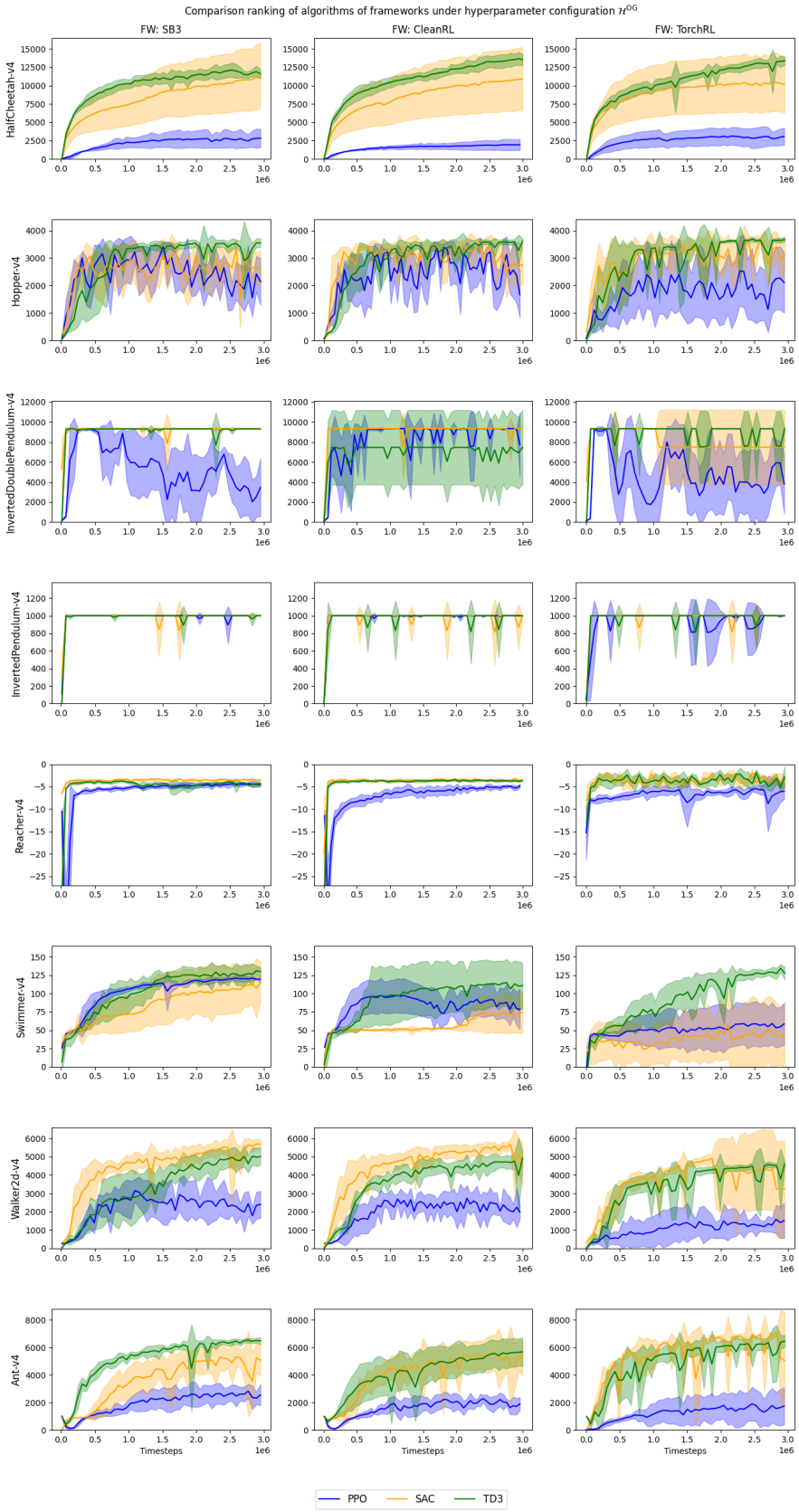


Figure 24: A comparison between the relative ranking of the algorithms PPO, TD3, and SAC, by each baseline framework (columns) under the hyperparameters reported by the original algorithm papers for each algorithm.

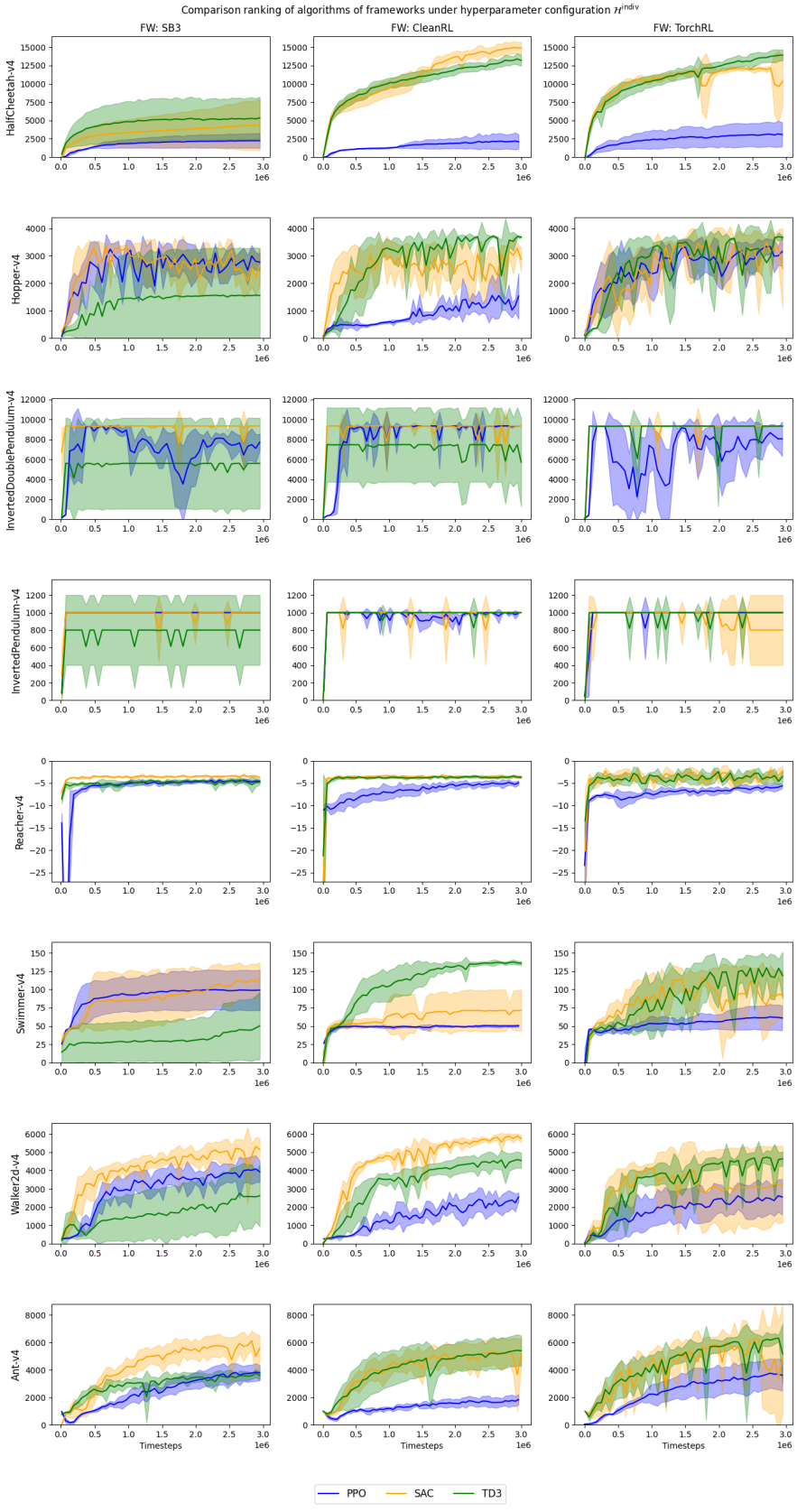


Figure 25: A comparison between the relative ranking of the algorithms PPO, TD3, and SAC, by each baseline framework (columns) under the default hyperparameters of the same baseline framework.