Isaac Sim Integration in the MushroomRL Library for Locomotion Learning

Isaac Sim Integration in die MushroomRL Bibliothek für Fortbewegungslernen
Bachelor thesis in the department of Computer Science by Bjarne Freund
Date of submission: March 19, 2025

1. Review: Davide Tateo
2. Review: Jan Peters
Darmstadt

TECHNISCHE UNIVERSITÄT DARMSTADT

Computer Science Department
Intelligent Autonomous Systems

Erklärung zur Abschlussarbeit gemäß § 22 Abs. 7 APB TU Darmstadt

Hiermit erkläre ich, *Bjarne Freund*, dass ich die vorliegende Arbeit gemäß § 22 Abs. 7 APB TU Darmstadt selbstständig, ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt habe. Ich habe mit Ausnahme der zitierten Literatur und anderer in der Arbeit genannter Quellen keine fremden Hilfsmittel benutzt. Die von mir bei der Anfertigung dieser wissenschaftlichen Arbeit wörtlich oder inhaltlich benutzte Literatur und alle anderen Quellen habe ich im Text deutlich gekennzeichnet und gesondert aufgeführt. Dies gilt auch für Quellen oder Hilfsmittel aus dem Internet.

Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Falle eines Plagiats (§38 Abs.2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

English translation for information purposes only:

Thesis Statement pursuant to § 22 paragraph 7 of APB TU Darmstadt

I herewith formally declare that I, Bjarne Freund, have written the submitted thesis independently pursuant to § 22 paragraph 7 of APB TU Darmstadt without any outside support and using only the quoted literature and other sources. I did not use any outside support except for the quoted literature and other sources mentioned in the paper. I have clearly marked and separately listed in the text the literature used literally or in terms of content and all other sources I used for the preparation of this academic work. This also applies to sources or aids from the Internet.

This thesis has not been handed in or published before in the same or similar form.

I am aware, that in case of an attempt at deception based on plagiarism (§38 Abs. 2 APB), the thesis would be graded with 5,0 and counted as one failed examination attempt. The thesis may only be repeated once.

For a thesis of the Department of Architecture, the submitted electronic version corresponds to the presented model and the submitted architectural plans.

Datum / Date:	Unterschrift/Signature:
19.03.25	[illegible]

Abstract

Deep Reinforcement Learning (DRL) has significantly advanced robotics, particularly in autonomous locomotion. However, traditional CPU-based environments limit scalability, leading to long training times. The introduction of massively parallelized simulations has transformed the field, enabling efficient training on GPU-accelerated platforms. However, MushroomRL, a RL framework, lacks support for parallelized environments, restricting its use in large-scale research. To address this, we developed an interface for Nvidia Isaac Sim within MushroomRL, allowing seamless integration of massively parallelized robotic simulations. This interface abstracts simulator complexities, enabling researchers to focus on reward design and algorithm development while optimizing performance for efficient parallel execution. To validate our approach, we implemented locomotion environments for the quadrupedal robots Unitree A1, Honey Badger, and Silver Badger. These environments replicate existing works. By extending MushroomRL with parallelized environment support, we provide an accessible framework for large-scale reinforcement learning.

Contents

1 Introduction

Deep Reinforcement Learning (DLR) has become a powerful tool for modern robotics research, in particular in the development of autonomous agents capable of sophisticated locomotion. By leveraging deep neural networks, Reinforcement Learning (RL) techniques enable robots to develop policies that allow them to navigate complex terrains, adapt to dynamic environments, and respond robustly to external perturbations. Traditionally, reinforcement learning relied on CPU-based environments that constrained the scalability due to the sequential execution of the environment interaction. This resulted into longer and longer training times for complex tasks and limited the progress in the field. The introduction of massively parallelized reinforcement learning environments has fundamentally transformed this landscape, enabling training at an unprecedented scale and efficiency. By utilizing GPUs and TPUs, researcher can simulate thousands of parallel agents simultaneously, which has lead to significant breakthroughs, in particular in locomotions. Studies such as those by Rudin et al. have demonstrated that by running thousands of concurrent environment instances on a GPU, quadrupedal locomotion policies can be learned in a fraction of the time required by conventional approaches. This drastic reduction in training time has driven a shift toward parallel RL, making massively parallelized environments the standard in modern locomotion research. As demonstrated by numerous examples that have achieved remarkable results in locomotion while using parallelized environments such as running on deformable objects [6], mastering an obstacle course [3] and leaping over large gaps [32].

MushroomRL [7] is a Python library designed for conducting Reinforcement Learning (RL) experiments. Its main focus is to provide a modular framework where all essential RL components are represented. This approach makes it easy to understand and to extend it. The library includes a variety of already implemented classical and deep RL algorithms, along with several environments such as MuJoCo or gymnasium. The modular structure, with common interfaces, enables users to seamlessly experiment with different algorithms or integrate custom environments. However, its lack of support for parallelized environments limits its usefulness in cutting-edge research, particularly in locomotion learning, which heavily relies on massive parallelization.

To address this limitation, we implemented an interface for Nvidia Isaac Sim into MushroomRL. Nvidia Isaac Sim is a robotic simulation platform built on Nvidia Omniverse. It uses the GPU-based PhysX engine, and offers high-fidelity physics, photo-realistic rendering and sensor simulation in virtual worlds. And most important, Isaac Sim is capable of running simulations with thousands of robots in parallel. The implemented interface provides an easy way for users to develop environments that utilize Isaac Sim and thereby leverage massively parallelization. The interface is designed to streamline the development process and minimize the need for direct interaction with the underlying simulator. This allows developers to concentrate on the design of the reward function or other RL specific parts when implementing a new environment. Another key focus in developing the interface was maximizing its performance by fully leveraging Isaac Sim's capabilities for speed and efficiency.

Additionally, we implemented three specialized environments that utilize the new Isaac Sim integration and enable locomotion learning for the quadrupeds Unitree A1, Honey Badger and Silver Badger. The environment that we implemented for the Unitree A1 robot closely resembles the one used by Rudin et al. [28], most importantly the used reward function and domain randomization are identical. For the Honey Badger environment, we adapt the Honey Badger environment used by Bohlinger et al. [2]. The environment we implemented for Honey Badger is also used for Silver Badger, with the only difference being adaptions for the one additional joint of Silver Badger.

To conclude, the thesis makes several contributions. First, we implemented a parallelized version of MushroomRLs core logic to support large-scale reinforcement learning experiments. Second, we developed a generic interface a for integrating Isaac Sim environments into MushroomRL, making it easier to design and implement RL tasks with GPU-accelerated simulation. Hereby, we investigate the factors that influence the simulation efficiency, identifying bottlenecks and optimizations that can enhance the learning process. Third, we implemented three environments for three different robots, that demonstrate the capabilities of the implemented Isaac Sim interface. Finally, using the three environments, we perform an analysis of the learning performance and the impact of the number of parallel environments.

2 State of the Art

Nicolas Heess et al. [13] demonstrated that deep reinforcement learning can enable the emergence of complex locomotion behaviors, even when using simple reward functions as long as the training takes place in rich environments so very diverse environments with many different challenges and obstacles are used. For this they used CPU parallelization where multiple instances of the simulation were running in parallel. However, this application of parallelization in the environment doesn't focus on improving learning speed and instead was used for averaging the gradient between the instances.

Daniel Freeman et al. [9] have achieved great results by utilizing TPUs for parallelizing the environment. To accomplish this, they developed a new differentiable physics engine that is optimized to run on TPUs. With this new physics engine, they successfully trained policies for multiple classic locomotion tasks such as Cheetah or Ant in under 10 minutes using optimized versions of Proximal Policy Optimization (PPO)[29] and Soft Actor-Critic (SAC) [11].

Further advancing the field of high-speed RL training, Rudin et al. [28] demonstrated the effectiveness of massively parallelized training environments combined with highly optimized RL algorithms. Their study demonstrated that utilizing a massively parallelized training framework enables extremely fast policy learning that is multiple times faster than previous works. Specifically, they trained a walking policy for the quadrupedal robot ANYmal in under 4 minutes on flat terrain and in less than 20 minutes on rough terrain, all on a single GPU. To achieve these results, they used the simulation tool NVIDIA Isaac Gym and a customized version of PPO.

The success of parallelized environments has led to their application in solving a wide variety of tasks. In high-agility locomotion, substantial progress has been made. For example, [6] utilizes parallelized environments to enhance training effectiveness for locomotion tasks on deformable terrains, such as running on soft surfaces. Additionally, [22] showcased highly agile behaviors such as sprinting and high-speed turning, while [15] focused on sprinting on slippery surfaces. Similarly, [3, 32] demonstrated that quadrupedal robots could develop complex locomotion skills, including parkour-style movements and obstacle course navigation, through parallelized learning frameworks.

Parallelized environments have also been employed in perception-based tasks. Agarwal et al. [1] presented a robust locomotion controller that utilizes vision to traverse stairs and other challenging terrains. Furthermore, Gangapurwala et al. [10] developed a controller that prioritizes stability through elevation mapping. In navigation tasks, end-to-end policies have been trained to reach targets within a given time while autonomously selecting optimal paths [27].

Parallelized environments also contribute to robust generalization. [23] showed that a single policy could encode diverse locomotion strategies, enhancing adaptability across various environments. Parallelization has further enabled advancements in generalizing locomotion across multiple robot morphologies. [30, 8, 19] successfully developed locomotion controllers capable of achieving robust locomotion across different quadrupedal robots.

Moreover, parallelized environments have helped in developing policies that are in particularly successful in the real-world , as well as techniques that ease the deployment of RL-trained policies in real-world scenarios. Kumar et al. [17] introduced an approach with excellent adaptability to unforeseen real-world conditions, such as varying terrains and payloads. [4] shows that good real word performance with minimal dynamics randomization is possible by utilizing random forces to emulate dynamics randomization.

Sparse reward problems, which are common in locomotion tasks, have also been addressed through hybrid architectures. [14] proposed a framework that integrates trajectory optimization with inverse dynamics and RL to tackle sparse reward settings effectively.

Parallelized environments are also valuable when constrains are addressed during training. [18] successfully incorporated physical constraints into the learning process, while [16] showed that leveraging physical constraints reduces the need for extensive reward engineering while maintaining high locomotion performance. Additionally, in safe locomotion, parallelized environments have facilitated progress. [12] developed methods for safe, high-speed locomotion that avoid collisions, ensuring reliable real-world deployment while utilizing a massively parallelized simulator.

Beyond quadrupedal locomotion, parallelized environments have also been applied to humanoid locomotion. Radosavovic et al. [26] developed RL-based approach for humanoid robot locomotion. Cheng et al.[5] demonstrated expressive whole-body movements in humanoid robots, and Luo et al. [20] presented human-like motion capabilities in humanoid systems.

3 Methodology

3.1 MushroomRLs Vectorized Core Logic

The core components of Mushroom RL are the interface Environment and Agent and the class Core that ties them together and controls the learning loop. The environment interface is similar to the environment interface from gymnasium, most importantly, it provides the methods step and reset. The method step is intended to update the environment with a given action and returns the resulting observation, the resulting reward, a flag if the state is absorbing and additional information. The reset function resets the environment to its initial state and returns the resulting observation and additional information that may exist. The agent implements the learning algorithm and contains policy.

The environment and core each have a vectorized variant. The vectorized environment is a wrapper that allows multiple instances of an environment to run in parallel, allowing a RL agent to interact with several environments simultaneously. Additionally to the environment interface, it provides the methods step_all and reset_all which allow for a specification of which environments of the vectorized environment should be used. The vectorized core is a variant of the normal core class that can deal with vectorized environments.

The vectorized core provides two fundamental functionalities: learning and evaluation. For learning, the vectorized core collects data from the vectorized environment and fits the policy using the passed algorithm. For evaluation, it collects data from the environment and returns the dataset without modifying the policy. Both the learning and evaluation run for specified number of steps or episodes. Additionally, for learning, the user can also specify the frequency (number of steps or episodes) at which the policy should be updated.

A learning run begins with the creation of an empty Dataset with a fixed size, designed to store the data needed for one fit. The dataset stores for each step the state at the beginning of the step, the action taken, the resulting state and two flags that indicate whether the resulting state is absorbing and whether the environment must be reset before the next step. However, in vectorized environments, the dataset can not store any additional information of each step.

A learning run consists of two alternating phases that are repeated until the specified total number of steps or episodes is reached. The first phase is the data collection phase. The core first checks whether an environment needs to be reset, which occurs either when it was flagged as absorbing by the vectorized environment or when the episode horizon is reached. If resets are required, the core calls the reset function of the vectorized environment with a specification of the affected sub environments and stores the resulting observation and additional info in the dataset. Next, the vectorized environment step function is called, and the resulting data is stored in the dataset. The data collection repeats until the specified number of episodes or steps required for the next policy update are reached. Once the condition for the next fit is met, the dataset will be passed to the agent for the fit. After the update, all data used for the training is removed from the dataset and the core resumes the data collection for the next fit. This process continues until the total number of steps or episode specified for the learning run is reached.

During data collection, a mask generation algorithm regulates the number of data points generated per step, adapting to whether the fit frequency and the learning run length are defined by steps or episodes. If it is determined by steps, the generated mask will be always active for all parallel environments. Data points that produced, without being needed for the next fit, will be passed to the next data collection phase. This procedure allows for maximum efficiency without losing any data points. If the fit frequency is determined by episodes, the mask remains active for all parallel environments when the number of remaining episodes for the next fit exceeds the number of parallel environments, otherwise, it matches the number of remaining episodes. Additionally, the masking logic ensures that the total training run length is not exceeded, preventing the number of steps from surpassing the chosen steps per run or the number of episodes from exceeding the chosen episodes per run. This approach offers full flexibility in configuring fit frequency and epoch length, even allowing combinations of fixed steps per fit with fixed episodes per epoch, or vice versa. This flexibility ensures that the training process can be tailored precisely to the desired objectives, like optimizing for fast convergence or aligning for certain hardware constrains.

However, the preexisting version did not function as intended due to two major issues.

First, the dataset length was not updated correctly after each fit, as it was always reset to zero, leading to unnecessary data generation and an unpredictable dataset size. This issue was resolved during the thesis by ensuring that the dataset length was correctly updated after each fit. The preexisting implementation miscalculated the necessary dataset shape, which hindered the effectiveness of the masking feature. To understand this, it is important to note that the dataset is two-dimensional: The first dimension represents the current data collection step and the second dimension corresponds to the number of parallel environments. In the faulty implementation, the dataset shape was defined as follows, where N_{envs} represents the number of parallel environments, N_{steps} denotes the number of steps and N_{episodes} denotes the number of episodes:

$$(N_{\text{steps}}, N_{\text{envs}})$$

or:

$$(N_{\text{episodes}} \cdot \text{horizon}, N_{\text{envs}})$$

However, this formulation was incorrect because it allocated N_{envs} times more space than necessary. The correct formulation is

$$\left(\left\lceil \frac{N_{\text{steps}}}{N_{\text{envs}}} \right\rceil + 1, N_{\text{envs}} \right)$$

or, alternatively:

$$\left(\left\lceil \frac{N_{\text{episodes}}}{N_{\text{envs}}} \right\rceil \cdot \text{horizon}, \min(N_{\text{envs}}, N_{\text{episodes}}) \right)$$

The additional $+1$ accounts for data points carried over from the previous data collection phase.

Beyond correcting the dataset allocation, this adjustment introduces an additional optimization. When evaluation runs are defined by the number of episodes, or when the fit frequency is based on an episode count and the used number of episodes is smaller than the number of parallel environments, memory usage is significantly reduced. Implementing this optimization required minor modifications to how data is appended to the dataset. Together, these changes (correct shape calculation and special variant if number of episodes is less than number of environments) lead to memory savings by several orders of magnitude.

3.2 General Isaac Sim Environment

The Isaac Sim environment class follows a similar design approach to the pre-existing MuJoCo environment. However, it differs among other things in its vectorized implementation and its reliance on Isaac Sim as the simulator.

At a minimum, the environment must be provided with several essential configurations. These include a USD file, a file format used by Isaac Sim to store object descriptions and geometries, in this case it should specify the robot and its components, a definition of which joints are to be controlled, a specification of which properties are to be included in the observation, and the number of parallel environments, i.e. the count of robot instances that should be simulated concurrently.

The observation specification is a structured list where each entry defines a property of the robot that should be part of the observation. Each property is described by specifying the path to the corresponding object in the USD file (e.g. a robot's foot), the property identifier (e.g. global position), and, if necessary, sub-entities required to fully describe the property. Sub-entities are necessary for certain cases. For example, when reading joint positions, the main object is the robot itself, and the sub-entities are the specific joints whose positions must be observed. During both environment steps and resets, this observation specification specifies the construction of the observation. If there are specific values that cannot be retrieved directly from the simulator but still need to be part of the observation, users can define custom observations. These custom observations require a unique name, the number of values, and the minimum and maximum possible values. Space for these custom observations is reserved within the observation during the observation construction, allowing users to easily insert their custom data.

The provided `write_data` and `read_data` methods allow users to read and write properties defined within the observation specification directly to and from the simulation. Furthermore, if there is a need to access properties that should not be part of the observation, an additional data specification, that is structured identically to the observation specification, can be passed to the Isaac Sim environment. All properties defined in this specification can be used for the `write_data` and `read_data` methods, but they are ignored during the observation creation.

3.2.1 Initialization

During the initialization, Isaac Sim is launched, and the simulation scene is created. Initially, the robot described in the provided USD file is loaded into the simulation scene. This robot is then cloned according to the user-specified number of parallel environments. All robot instances are arranged systematically in a grid pattern, maintaining a fixed, user-defined distance between individual robots. Users can additionally specify whether collision detection between separate robot environments should be activated or deactivated. Furthermore, a ground plane is automatically added to the simulation scene.

Subsequently, for each unique object described by a path in the observation specification and the additional data specification, a `RigidPrim` is registered in the simulation scene. A `RigidPrim` is a wrapper provided by Isaac Sim, capable of grouping multiple rigid bodies into one manageable entity, thereby facilitating efficient data reading and writing operations [25]. In the context of the Isaac Sim environment, one RigidPrim contains the same component of the robot for every cloned robot. For example, a single `RigidPrim` might represent the front right foot across all robot instances. After the registration process is complete, these `RigidPrims` enable efficient interaction with all properties defined within the observation and additional data specifications.

After completing the robot instantiation process, an object containing all essential environmental information is created. This object is required by algorithms implemented within Mushroom-RL. It includes the horizon length, the discount factor (γ), the step frequency, and the minimum and maximum allowable values for each element of both the observation and action spaces. For each observation and action element, the corresponding minimum and maximum values are obtained directly from the USD file if available (e.g., predefined joint range limits). If such values are not specified in the USD file, default bounds of negative and positive infinity are assigned. For user-defined custom observation elements, the minimum and maximum values explicitly provided by the user are utilized.

3.2.2 Environment Step

The step function takes as input the action for each parallel environment and a mask indicating which parallel environments are actively being simulated. The process begins with an optional action preprocessing phase, where users can scale or limit the actions to their specific requirements. Following preprocessing, the function executes a sequence of intermediate steps. During each step, actions are applied to the controlled joints, and a simulation step advances the physical simulation. Actions can be recomputed during each intermediate step, enabling implementation of control algorithms such as PD-controllers. Before and after each simulation step, overwritable functions are provided, allowing for custom operations like state adjustments, logging, or monitoring.

Following these intermediate steps, the system automatically constructs the observation according to the observation specification. It reads the properties from each robot using the RigidPrims registered during the initialization. The resulting observation may include dedicated space for custom observation elements, that the user registered. An overwritable function allows users to populate this dedicated space in the observation. The system then performs two key evaluations, each requiring custom user implementation: it determines for all parallel environments whether the current state is absorbing (terminal) through one user-implemented function, and calculates the reward for all parallel environments through another user-implemented function. In addition, the system generates additional step information if the user has implemented the corresponding overwritable function. Before finalizing the process, a final overwritable function allows users to modify the observation if needed. The step function then returns four elements: the observation, the calculated reward, any additional step information, and a flag indicating which parallel environments are currently active and have reached absorbing states.

3.2.3 Environment Reset

The reset function accepts two parameters: a mask indicating which environments should be reset and an optional target state. By default, robots in the masked parallel environments will reset to their default states as defined in the USD file. However, users can customize this behavior through a mandatory user-implemented function called `setup`, in which the user can implement custom reset behaviors using the `write_data` method to manipulate robot properties directly. For example, a common application of the `setup` function is implementing randomized initial joint positions, which enhances training robustness by exposing the learning algorithm to varied starting conditions.

If the environment needs to support algorithms that reset to specific states, the setup function must be capable of resetting the robot to the given state. After this reset process, the observation construction process follows the same procedure as in the step function: reading robot properties, constructing the observation and populating any custom observation components. As with the step function, additional information may be generated if the user has implemented the associated function. The reset function then returns both the constructed observation and any additional information. Notably, observations are generated for all parallel environments, regardless of whether a specific parallel environment was reset or not. This comprehensive observation generation occurs because the `VectorCore` filters which observations are relevant to the algorithm.

3.2.4 Collision Detection

The environment allows the collision detection between user defined collision groups. Each collision group is specified by a name and a list of paths to objects that should be included. The specification for all collision groups must be passed as a parameter during environment initialization. During the environment's initialization phase, before the robot is cloned, the system checks whether the objects specified in the collision groups possess the necessary APIs so that Isaac Sim can detect collisions. If required APIs are missing from any object, they are automatically applied. This verification is crucial prior to cloning, as failure to properly configure physics components would prevent physics cloning, resulting in significant performance degradation. After the robots are cloned, the environment registers a `RigidContactView` for each collision group. `RigidContactView` is a wrapper provided by Isaac Sim that combines multiple rigid bodies and allows efficient contact tracking [24]. The environment offers two collision detection approaches: the first method, `get_collision_force`, detects collisions between two specified collision groups; the second method, `get_net_collision_forces`, provides unfiltered contact forces for each object in a collision group, functioning similarly to a pressure sensor. Building upon `get_collision_force`, the environment also provides two helper functions: One that checks whether a collision occurred between two collision groups and another that counts the number of collisions between two collision groups.

3.2.5 Optimizations

The primary factor influencing the simulation speed of Isaac Sim is how the collision detection is implemented in the interface. There are three main methods for detecting collisions in Isaac Sim: using RigidContactViews, using RigidPrims, and directly accessing the ContactReportAPI. Among these, RigidContactViews is the most efficient. Although RigidPrim internally utilizes a RigidContactView, it introduces significant overhead, making it slower. Directly accessing the ContactReportAPI is the least efficient approach, as it can only detect one collision at a time, leading to a sequential and consequently slow detection process.

However, the way RigidContactViews are used also plays a crucial role in performance. As previously mentioned, the PhysxContactReportAPI, which is required for RigidContactViews to function, should be applied to every object in a collision group before cloning. Failing to do so results in each robot being treated as a separate object, significantly reducing performance. Additionally, the `prepare_contact_sensors` option should be disabled when creating RigidContactViews, as enabling it drastically increases startup time. The number of registered RigidContactViews is another important consideration. Each registered RigidContactView slows down the simulation, regardless of how many objects it includes. For example, using two RigidContactViews, one containing all feet of all robots and another containing all calves, results in slower performance compared to using a single RigidContactView that includes both feet and calves. That's why, the implemented interface uses one RigidContactView per collision group, as this is the minimal number required without reducing functionality. If maximizing simulation speed is a priority, users should aim to minimize the number of collision groups needed for their application.

The second most critical performance optimization involves setting low values for the position solver iteration count and the velocity solver iteration count. These values determine how accurately contacts, drives, and limits are resolved. Since they impact simulation accuracy, the ideal values may vary depending on the use case. To provide flexibility, the interface allows users to specify these values via the constructor. Users should choose the lowest values that still maintain sufficient accuracy for their application.

Finally, we use many different settings of Isaac Sim to optimize its performance. While each individual setting provides only a small improvement, their combined effect can lead to significant speed gains. The most important settings include disabling scene query support, disabling contact processing, disabling custom geometries for collision detection of cones and cylinders, and optimizing rendering settings. Despite its name, disabling contact processing doesn't interfere with the collision detection.

3.3 Unitree A1 Environment

This section describes the learning environment developed for training the Unitree A1 robot [31] to walk, implemented using the above introduced Isaac Sim environment for MushroomRL. The environment replicates the training conditions from Rudin et al. [28], which has demonstrated excellent performance for the A1 robot when paired with a heavily parallelized simulation framework. By leveraging these established design principles, the environment aims to demonstrate that the integration of Isaac Sim with MushroomRL can effectively yield robust locomotion behaviors.

The observation space provided to the agent during the training consists of the joint positions and velocities of all joints of the robot, a projected gravity vector representing the direction of gravity, expressed in the robot's local reference frame and the commanded velocities, composed of the linear velocities along the x- and y-axis and the yaw velocity. Additionally, an observation includes the most recently applied action, as well as the robot's linear and angular velocities expressed in the global reference frame. The individual observation components are normalizes and all except the most recently applied action and the commands incorporate some noise. The episodes terminate upon entering an absorbing state, defined as the state in which the robot's trunk contacts the ground plane, indicating a fall or instability.

The commands have a probability of 0.2% of being resampled at every environment step. Both commanded linear velocities (along the x- and y-axes) are randomly sampled from a uniform distribution between $-1\,\mathrm{m\,s^{-1}}$ and $1\,\mathrm{m\,s^{-1}}$. However, values whose magnitudes are less than $0.2\,\mathrm{m\,s^{-1}}$ are rounded to 0. Additionally, a target heading angle is simultaneously sampled uniformly between $-\pi$ and π. At the end of each step, the commanded yaw velocity is computed based on the difference between the robot's current heading and the target heading angle. This yaw velocity command is then clipped within the range $-1\,\mathrm{rad\,s^{-1}}$ to $1\,\mathrm{rad\,s^{-1}}$.

To ensure robust performance across varied scenarios, the environment incorporates several domain randomization techniques. These include randomizing friction coefficients at the robot's feet, randomizing initial joint positions slightly around the default robot pose, and applying random external pushes to the robot to simulate disturbances. Such randomizations help the trained policy generalize effectively to real-world uncertainties and unforeseen disturbances.

The robot's motion is controlled by an action vector, where each value indicates the target position for an individual joint. Each environment step includes four intermediate control updates, in which the target positions indicated by the action are translated into torques and that are applied to the robot's joints. The target positions are translated using a Proportional-Derivative (PD) controller, which calculates the required torques based on the difference between the target positions and the robot's current joint positions and velocities.

The reward function is designed to encourage a behavior that is closely aligned with the commanded velocities and stability criteria. Specifically, rewards are granted for matching the commanded linear velocities in the x and y directions, as well as matching the commanded yaw velocity. Conversely, penalties are applied to discourage undesirable behaviors, including linear velocity along the z-axis, angular velocity around the x and y axes, collisions involving the robot's thighs and calves, and joint positions approaching mechanical limits. Additionally, the reward function imposes penalties to minimize torque usage, joint acceleration and action changes. Finally, it incentivizes longer steps, promoting more efficient locomotion strategies.

3.4 Honey and Silver Badger Environment

This section describes the learning environments for training the Honey Badger Robot and the Silver Badger Robot [21] to walk, implemented using the introduced Isaac Sim environment for Mushroom RL. The environment resembles the setup that was introduced by Bohlinger et al. for the Honey Badger Robot for his work in developing one policy that runs multiples robots[2]. Silver Badger and Honey Badger are identical robots with one exception: the Silver Badger's upper body consists of two parts which are connected by an additional joint. Consequently, the learning environments for both robots are nearly identical, with minor modifications to the Silver Badger environment to accommodate its additional joint and this additional body part.

The observation space, command structure, and robot motion control closely mirror those used for the A1 environment, with only minor differences. The observation space additionally contains the robot's z-position. For the commands, the commanded yaw velocity is directly randomly sampled from a uniform distribution between $-1\,\mathrm{rad\,s^{-1}}$ and $1\,\mathrm{rad\,s^{-1}}$ together with the linear velocity commands. This skips the additional step used by the A1 environment of first sampling a target heading. The robot's motion control remains consistent with the A1 environment's approach, with the key distinction that the PD-controller incorporates additional parameters subject to domain randomization, as detailed in the following section.

The environment implements extensive domain randomization techniques. Similar to the A1 Environment, the system randomizes the friction coefficients on all robot feet, uses randomized initial joint positions, and applies random external pushes to the robot. Additionally, the environment introduces slight variations in several physical parameters: trunk mass, center of mass positioning, joint torque limits, maximum joint velocities, and foot size. Furthermore, the environment randomly varies the damping, stiffness, armature, and friction loss values across all joints. As previously mentioned, the environment also introduces slight random variations in the PD controller parameters, specifically: the P-gain, D-gain, action scaling factor (applied before the PD controller processes the actions), and torque scaling multiplier. Most of these randomized parameters are incorporated into the observation space with some added noise, enabling the agent to adapt to different scenarios.

The reward function incorporates the same components as the reward function used for the A1 environment, with two notable additions. First, it penalizes deviations between the robot's z-position and a goal height of 0.3. Second, the reward function includes penalties when both front feet or both rear feet simultaneously lose ground contact.

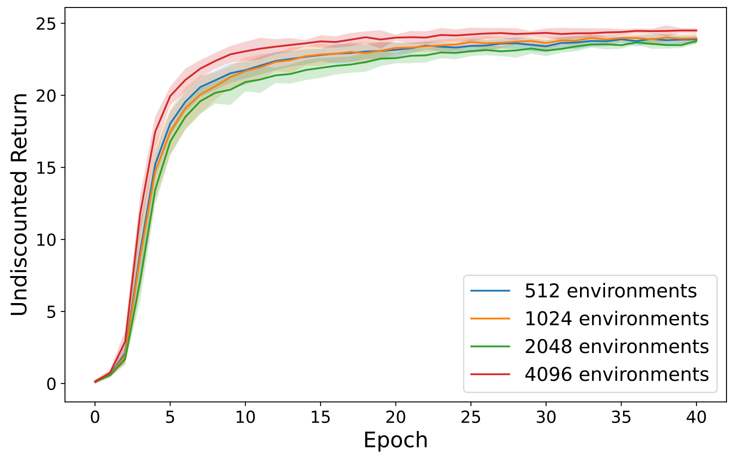
4 Experiments

For evaluation, we tested the implemented learning environments on the Unitree A1, Honey Badger, and Silver Badger robots, analyzing the impact of the number of parallel environments and the number of steps per fit on both the return and the time required to achieve a satisfactory return. The reported times are measured after the agent is evaluated for an epoch, encompassing both the learning and evaluation durations. We used a modified version of the PPO algorithm implemented in MushroomRL, which applies gradient norm clipping to a predefined value and dynamically adjusts the learning rate based on the Kullback-Leibler (KL) divergence between the old and new policies after each policy update. Additionally, we tested each robot with MushroomRL's default PPO algorithm to verify its compatibility with the respective robot. All the experiments are conducted on five different seeds, and additional plots for each experiment can be found in the appendix.

4.1 Results for Unitree A1

For the Unitree A1 experiments, we used the hyperparameters shown in Figure 5.1, which are largely based on those used by Rudin et al. [28] in their implementation of a learning environment for the A1 robot. To measure training time, we defined a return of 21 as a satisfactory threshold, as training typically slows down around this point for most configurations. However, this value does not carry any deeper significance, and any similar value in this range could have been used.

To examine the impact of the number of parallel environments used, we kept all hyperparameters unchanged except for the number of parallel environments, which we varied across four values: 512, 1024, 2048, and 4096. Due to hardware limitations, we could not test configurations beyond 4096 environments, as the 8GB VRAM of the RTX 3060 Ti on which the A1 experiments were run, constrained the simulation capacity.



(a) Undiscounted return after each epoch.

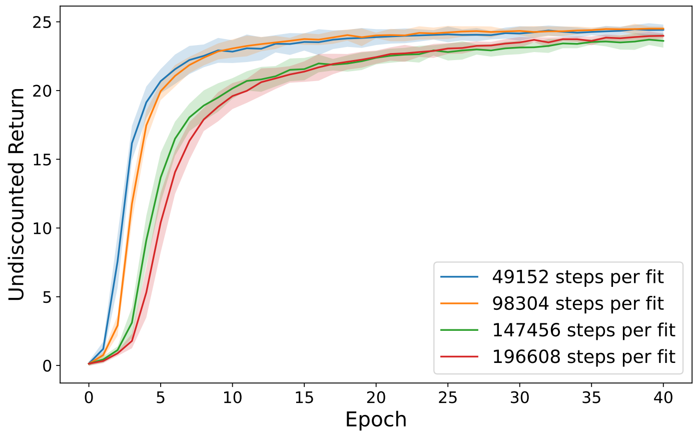
Number of envs	Time for return > 21
512 Envs	2054 s
1024 Envs	1639 s
2048 Envs	1526 s
4096 Envs	888 s

(b) Wall-clock time required to reach an undiscounted return of over 21.

Figure 4.1: Experiment results for the A1 robot using varying numbers of parallel environments, executed on an RTX 3060 Ti GPU and a Ryzen 5800X3D CPU. Each graph presents the mean and the 95% confidence interval. A modified version of MushroomRL's PPO algorithm was used. Each configuration was run on five different seeds.

As shown in Figure 4.1a, the learning successfully converges across all tested configurations, confirming that the learning environment is functioning correctly. We also verified that the converging and a high return does translate into the intended behavior. Among the tested setups, 4096 parallel environments achieved the best performance in terms of both maximum return and training speed, as illustrated in Figure 4.1b. Interestingly, while the other configurations reached similar maximum returns after 40 epochs, the 2048-environment setup converged more slowly and underperformed compared to the 512- and 1024-environment configurations for most of the training process. This result is somewhat counterintuitive, as one would expect 2048 environments to behave more similarly to 4096 rather than to the lower-count configurations.

To examine the impact of the number of steps per fit, we kept all hyperparameters unchanged, as shown in Figure 5.1, except for the number of steps per fit, which we varied across four values: 49,152, 98,304, 147,456, and 196,608. These values correspond to 12, 24, 36, and 48 steps per parallel environment.



(a) Undiscounted return after each epoch.

Number of steps per fit	Time for return > 21
49152 steps	862 s
98304 steps	888 s
147456 steps	2105 s
196608 steps	2336 s

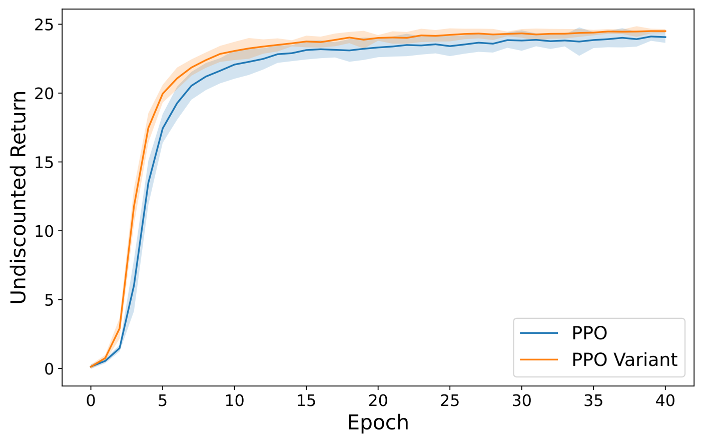
(b) Wall-clock time required to reach an undiscounted return of over 21.

Figure 4.2: Experiment results for the A1 robot using varying numbers of steps per fit, executed on an RTX 3060 Ti GPU and a Ryzen 5800X3D CPU. Each graph presents the mean and the 95% confidence interval. A modified version of MushroomRL's PPO algorithm was used. Each configuration was run on five different seeds.

As shown in Figure 4.2a, the learning process successfully converges for all tested configurations, and the final return after 40 epochs is approximately the same across all setups. However, there are significant differences in performance before epoch 20. The configurations with 49,152 and 98,304 steps per fit show similar performance, while those with 147,456 and 196,608 steps per fit also behave similarly. However, there is a clear gap between these two groups, with the latter configurations performing significantly worse. This trend is also evident in training speed, where the setups with 49,152 and 98,304 steps per fit require less than half the training time compared to those with 147,456 and 196,608 steps per fit to reach a return of 21.

Lastly, we examine the difference in learning performance between MushroomRL's default PPO and the modified variant. For this, we use the same hyperparameters as in the previous experiments, except that the learning rate for MushroomRL's PPO is set to 1×10^{-4}, as it becomes unstable at 1×10^{-3}.

Overall, both versions of PPO exhibit similar performance, but the modified variant



(a) Undiscounted return after each epoch.

Number of envs	Time for return > 21
PPO	913 s
PPO Variant	888 s

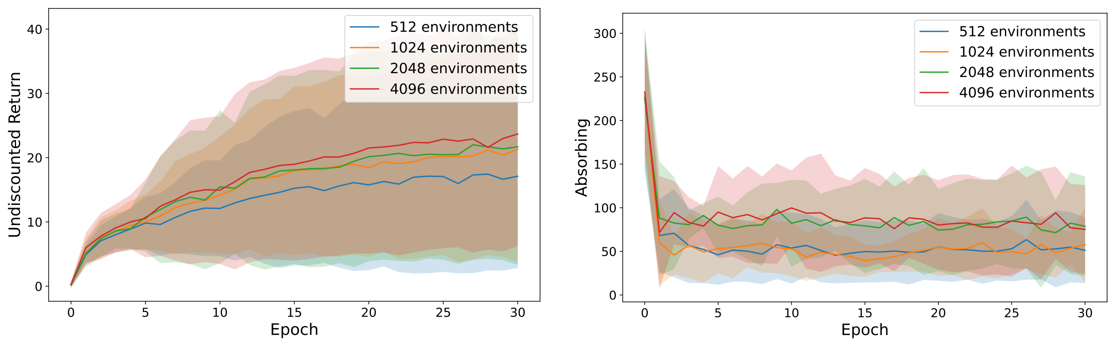
(b) Wall-clock time required to reach an undiscounted return of over 21.

Figure 4.3: Experiment results for the A1 robot, comparing its performance when trained with MushroomRL's default PPO versus a modified version incorporating gradient clipping and learning rate adaptation. The experiment is executed on an RTX 3060 Ti GPU and a Ryzen 5800X3D CPU. Each graph presents the mean and the 95% confidence interval. Each configuration was run on five different seeds.

consistently outperforms the default implementation. It converges faster and achieves a slightly higher return after 40 epochs, as shown in Figure 4.3a. However, this advantage shrinks when considering training speed. While the modified PPO reaches a return of 21 after six epochs, the default PPO requires eight epochs. In terms of wall-clock time, this difference amounts to only 20 seconds, likely due to the additional computational cost introduced by calculating the Kullback-Leibler divergence, which slows down the modified PPO.

4.2 Results for Honey Badger

For the Honey Badger experiments, we used the hyperparameters shown in Figure 5.1. To examine the impact of the number of parallel environments, we kept all hyperparameters unchanged except for this variable, which we tested at four different values: 512, 1024, 2048, and 4096.



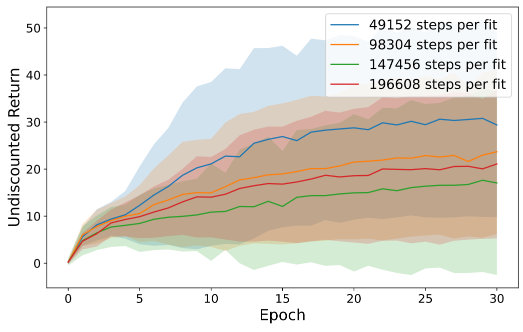
(a) Undiscounted return after each epoch. (b) Terminations per epoch.

Figure 4.4: Experiment results for the Honey Badger robot using varying numbers of parallel environments, executed on an RTX 2080 Ti GPU and six CPU-Cores of Ryzen 3950X. Each graph presents the mean and the 95% confidence interval. A modified version of MushroomRL's PPO algorithm was used. Each configuration was run on five different seeds.

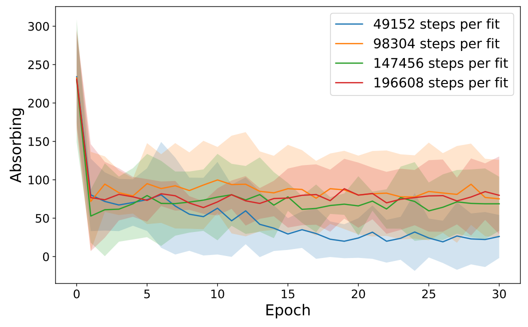
As shown in Figure 4.4a, the learning process does not reliably converge for any tested configuration. In every case, some runs failed to improve and achieved only very low performance. Even the runs that reached a high return did not translate into the intended behavior and remained unstable. As illustrated in Figure 4.4b, the robot frequently terminates episodes due to falls.

This issue persists when analyzing the effect of the number of steps per fit. As before, we kept all hyperparameters unchanged and varied only the number of steps per fit across four values: 49,152, 98,304, 147,456, and 196,608. As shown in Figure 4.5a, performance remained poor across all tested configurations. The setup with 49,152 steps per fit performed noticeably better than the others, but even this configuration included runs that failed to learn effectively. Similar to the A1 experiments, we also compared MushroomRL's default PPO with the modified version. As seen in Figure 4.5c, MushroomRL's PPO performed even worse in this scenario.

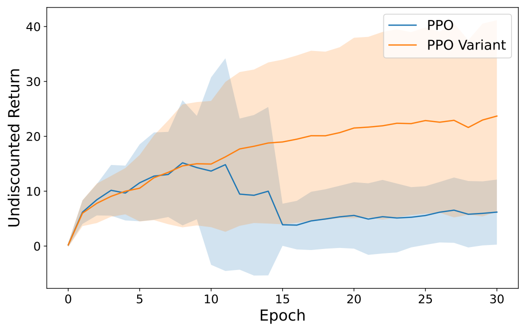
These results suggest that the learning environment for the Honey Badger requires further tuning, particularly in the reward function or termination conditions, to improve performance.



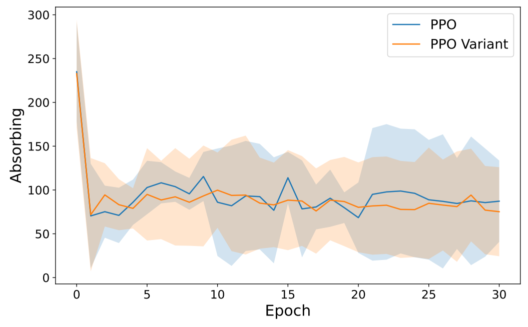
(a) Undiscounted return after each epoch.



(b) Terminations per epoch.



(c) Undiscounted return after each epoch.



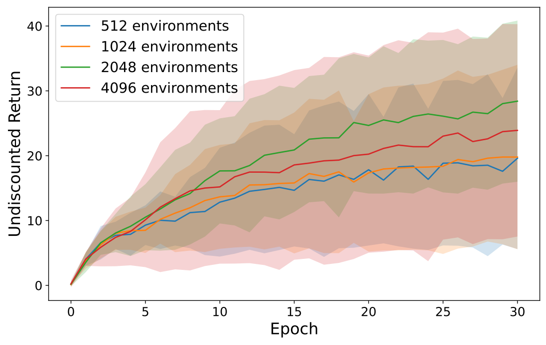
(d) Terminations per epoch.

Figure 4.5: Experiment results for the Honey Badger robot using varying values for steps per fit (a - b) and comparing MushroomRL's PPO with the modified PPO.

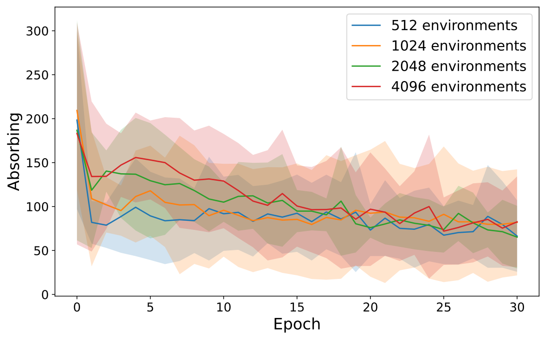
4.3 Results for Silver Badger

For the Silver Badger experiments, we use the same hyperparameters as in the Honey Badger experiments. Since the learning environment remains nearly identical, we do not expect significant differences in performance between Silver Badger and Honey Badger.

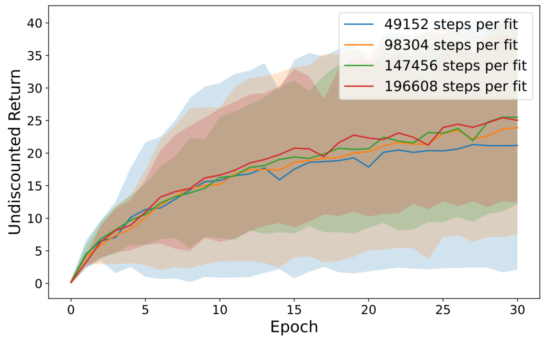
As shown in Figure 4.6a, the performance exhibits a similar level of unreliability as observed in the Honey Badger experiments. This trend persists in the analysis of steps per fit (Figure 4.6c) and in the comparison between MushroomRL's PPO and the modified PPO (Figure 4.6e). In conclusion, as expected, the Silver Badger learning environment does not perform any better, given that it is essentially the same as the Honey Badger environment.



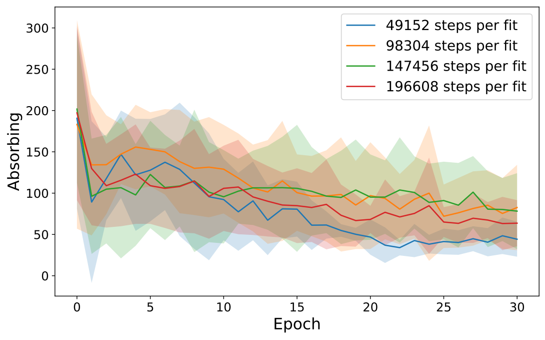
(a) Undiscounted return after each epoch.



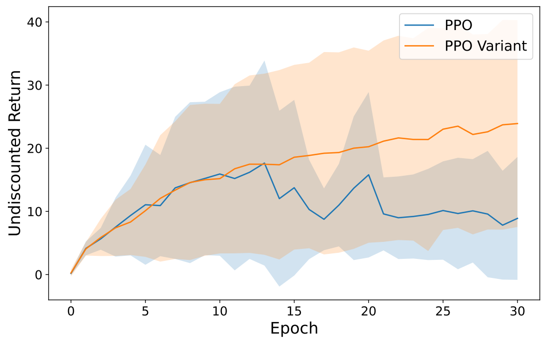
(b) Terminations per epoch.



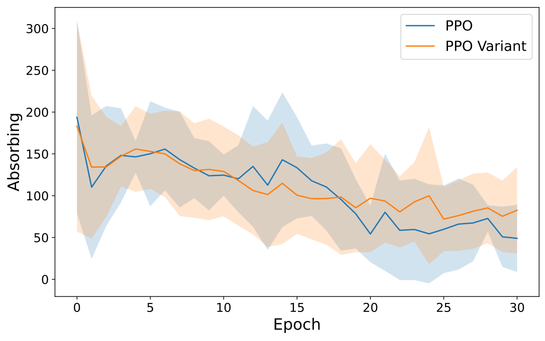
(c) Undiscounted return after each epoch.



(d) Terminations per epoch.



(e) Undiscounted return after each epoch.



(f) Terminations per epoch.

Figure 4.6: Experiment results for the Silver Badger robot using varying numbers of parallel environments (a-b), varying steps per fit (c-d) and comparing MushroomRL's PPO with the modified PPO (e-f), executed on an RTX 2080 Ti GPU and six CPU-Cores of Ryzen 3950X. Each graph presents the mean and the 95% confidence interval. Each configuration was run on five different seeds.

5 Conclusion

In conclusion, we developed an easy-to-use and efficient interface for IsaacSim within MushroomRL, along with three environments for different robots utilizing this interface. Using the A1 environment, an agent successfully learns a robust policy for the A1 robot, demonstrating the capabilities of the IsaacSim interface. However, the Honey Badger and Silver Badger learning environments will need further tuning of the reward function or the termination condition.

With the new interface, MushroomRL now supports massively parallelized environments, significantly enhancing its usability for state-of-the-art robotics research. However, there are still many features that could be added to further improve the IsaacSim interface. First, enabling the training of multiple different robots within the same environment simultaneously would unlock a range of new applications, such as multi-morphology research. To achieve this, the initialization process would need to be adapted to support the cloning of different robot models, and the automatic observation generation process would likely require modifications. Additionally, other useful extensions could include support for camera sensors and deformable objects, which are features available in IsaacSim that are not yet integrated into the interface.

Bibliography

[1] Ananye Agarwal et al. “Legged Locomotion in Challenging Terrains using Egocentric Vision”. In: *Proceedings of The 6th Conference on Robot Learning*. Ed. by Karen Liu, Dana Kulic, and Jeff Ichnowski. Vol. 205. Proceedings of Machine Learning Research. PMLR, 14–18 Dec 2023, pp. 403–415. URL: `https://proceedings.mlr.press/v205/agarwal23a.html`.

[2] Nico Bohlinger et al. *One Policy to Run Them All: an End-to-end Learning Approach to Multi-Embodiment Locomotion*. 2024. arXiv: `2409.06366 [cs.RO]`. URL: `https://arxiv.org/abs/2409.06366`.

[3] Ken Caluwaerts et al. *Barkour: Benchmarking Animal-level Agility with Quadruped Robots*. 2023. arXiv: `2305.14654 [cs.RO]`. URL: `https://arxiv.org/abs/2305.14654`.

[4] Luigi Campanaro et al. *Learning and Deploying Robust Locomotion Policies with Minimal Dynamics Randomization*. 2023. arXiv: `2209.12878 [cs.RO]`. URL: `https://arxiv.org/abs/2209.12878`.

[5] Xuxin Cheng et al. “Expressive whole-body control for humanoid robots”. In: *arXiv preprint arXiv:2402.16796* (2024).

[6] Suyoung Choi et al. “Learning quadrupedal locomotion on deformable terrain”. In: *Science Robotics* 8.74 (2023), eade2256. DOI: `10.1126/scirobotics.ade2256`. eprint: `https://www.science.org/doi/pdf/10.1126/scirobotics.ade2256`. URL: `https://www.science.org/doi/abs/10.1126/scirobotics.ade2256`.

[7] Carlo D’Eramo et al. “MushroomRL: Simplifying Reinforcement Learning Research”. In: *Journal of Machine Learning Research* 22.131 (2021), pp. 1–5. URL: `http://jmlr.org/papers/v22/18-056.html`.

[8] Gilbert Feng et al. “GenLoco: Generalized Locomotion Controllers for Quadrupedal Robots”. In: *Proceedings of The 6th Conference on Robot Learning*. Ed. by Karen Liu, Dana Kulic, and Jeff Ichnowski. Vol. 205. Proceedings of Machine Learning Research. PMLR, 14–18 Dec 2023, pp. 1893–1903. URL: https://proceedings.mlr.press/v205/feng23a.html.

[9] C. Daniel Freeman et al. *Brax – A Differentiable Physics Engine for Large Scale Rigid Body Simulation*. 2021. arXiv: 2106.13281 [cs.RO]. URL: https://arxiv.org/abs/2106.13281.

[10] Siddhant Gangapurwala et al. “RLOC: Terrain-Aware Legged Locomotion Using Reinforcement Learning and Optimal Control”. In: *IEEE Transactions on Robotics* 38.5 (2022), pp. 2908–2927. DOI: 10.1109/TRO.2022.3172469.

[11] Tuomas Haarnoja et al. “Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor”. In: *International conference on machine learning*. Pmlr. 2018, pp. 1861–1870.

[12] Tairan He et al. “Agile but safe: Learning collision-free high-speed legged locomotion”. In: *arXiv preprint arXiv:2401.17583* (2024).

[13] Nicolas Heess et al. *Emergence of Locomotion Behaviours in Rich Environments*. 2017. arXiv: 1707.02286 [cs.AI]. URL: https://arxiv.org/abs/1707.02286.

[14] Fabian Jenelten et al. “DTC: Deep Tracking Control”. In: *Science Robotics* 9.86 (2024), eadh5401. DOI: 10.1126/scirobotics.adh5401. eprint: https://www.science.org/doi/pdf/10.1126/scirobotics.adh5401. URL: https://www.science.org/doi/abs/10.1126/scirobotics.adh5401.

[15] Gwanghyeon Ji et al. “Concurrent Training of a Control Policy and a State Estimator for Dynamic and Robust Legged Locomotion”. In: *IEEE Robotics and Automation Letters* 7.2 (2022), pp. 4630–4637. DOI: 10.1109/LRA.2022.3151396.

[16] Yunho Kim et al. “Not Only Rewards but Also Constraints: Applications on Legged Robot Locomotion”. In: *IEEE Transactions on Robotics* 40 (2024), pp. 2984–3003. DOI: 10.1109/TRO.2024.3400935.

[17] Ashish Kumar et al. “Rma: Rapid motor adaptation for legged robots”. In: *arXiv preprint arXiv:2107.04034* (2021).

[18] Joonho Lee et al. “Evaluation of constrained reinforcement learning algorithms for legged locomotion”. In: *arXiv preprint arXiv:2309.15430* (2023).

[19] Zeren Luo et al. "MorAL: Learning Morphologically Adaptive Locomotion Controller for Quadrupedal Robots on Challenging Terrains". In: *IEEE Robotics and Automation Letters* 9.5 (2024), pp. 4019–4026. DOI: 10.1109/LRA.2024.3375086.

[20] Zhengyi Luo et al. "Universal humanoid motion representations for physics-based control". In: *arXiv preprint arXiv:2310.04582* (2023).

[21] MAB Robotics. *MAB Robotics - Robotics and Automation Solutions*. Accessed: 2025-03-14. 2025. URL: https://www.mabrobotics.pl/.

[22] Gabriel B Margolis et al. "Rapid locomotion via reinforcement learning". In: *The International Journal of Robotics Research* 43.4 (2024), pp. 572–587.

[23] Gabriel B. Margolis and Pulkit Agrawal. "Walk These Ways: Tuning Robot Control for Generalization with Multiplicity of Behavior". In: *Proceedings of The 6th Conference on Robot Learning*. Ed. by Karen Liu, Dana Kulic, and Jeff Ichnowski. Vol. 205. Proceedings of Machine Learning Research. PMLR, 14–18 Dec 2023, pp. 22–31. URL: https://proceedings.mlr.press/v205/margolis23a.html.

[24] NVIDIA Corporation. *Isaac Sim Core API Documentation - RigidContactView*. Accessed: 2025-03-17. 2025. URL: https://docs.isaacsim.omniverse.nvidia.com/latest/py/source/extensions/isaacsim.core.api/docs/index.html#isaacsim.core.api.sensors.RigidContactView.

[25] NVIDIA Corporation. *Isaac Sim Core API Documentation - RigidPrim*. Accessed: 2025-03-17. 2025. URL: https://docs.isaacsim.omniverse.nvidia.com/latest/py/source/extensions/isaacsim.core.prims/docs/index.html#isaacsim.core.prims.RigidPrim.

[26] Ilija Radosavovic et al. "Real-world humanoid locomotion with reinforcement learning". In: *Science Robotics* 9.89 (2024), eadi9579.

[27] Nikita Rudin et al. "Advanced Skills by Learning Locomotion and Local Navigation End-to-End". In: *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2022, pp. 2497–2503. DOI: 10.1109/IROS47612.2022.9981198.

[28] Nikita Rudin et al. "Learning to Walk in Minutes Using Massively Parallel Deep Reinforcement Learning". In: *Proceedings of the 5th Conference on Robot Learning*. Ed. by Aleksandra Faust, David Hsu, and Gerhard Neumann. Vol. 164. Proceedings of Machine Learning Research. PMLR, Aug. 2022, pp. 91–100. URL: https://proceedings.mlr.press/v164/rudin22a.html.

[29] John Schulman et al. "Proximal policy optimization algorithms". In: *arXiv preprint arXiv:1707.06347* (2017).

[30] Milad Shafiee, Guillaume Bellegarda, and Auke Ijspeert. “ManyQuadrupeds: Learning a Single Locomotion Policy for Diverse Quadruped Robots”. In: *2024 IEEE International Conference on Robotics and Automation (ICRA)*. 2024, pp. 3471–3477. DOI: 10.1109/ICRA57147.2024.10610155.

[31] Unitree Robotics. *Unitree A1 - The High-Performance Quadruped Robot*. Accessed: 2025-03-13. 2025. URL: https://unitreerobotics.net/robotdog/unitree-a1/.

[32] Ziwen Zhuang et al. *Robot Parkour Learning*. 2023. arXiv: 2309.05665 [cs.RO]. URL: https://arxiv.org/abs/2309.05665.

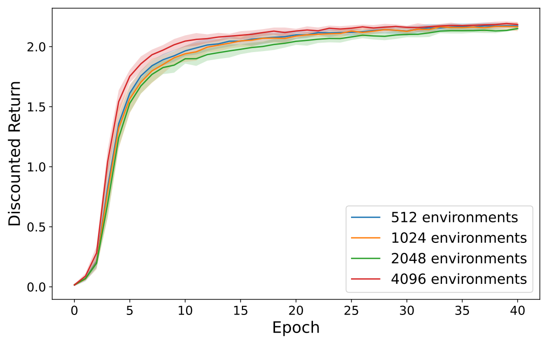
Appendix

Hyperparameters for Experiments

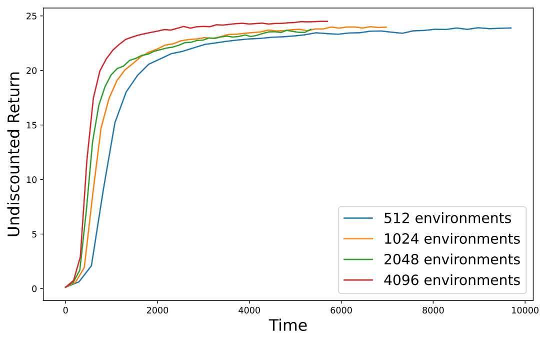
Parameter	Unitree A1	Honey / Silver Badger
Number of epochs	40	30
Steps per epoch	204800	409600
Steps per fit	98304	98304
Episodes in evaluation	256	256
Mini-batch size	6144	3072
Learning rate	1×10^{-3}	1×10^{-4}
Discount factor	0.99	0.99
GAE λ	0.95	0.95
Entropy coefficient	0.01	0
Epsilon	0.2	0.2
Number of epochs policy	5	5
Gradient norm clipping	-	1.0
Desired kl	-	0.01

Figure 5.1: Table containing the PPO hyperparameters used in all experiments.

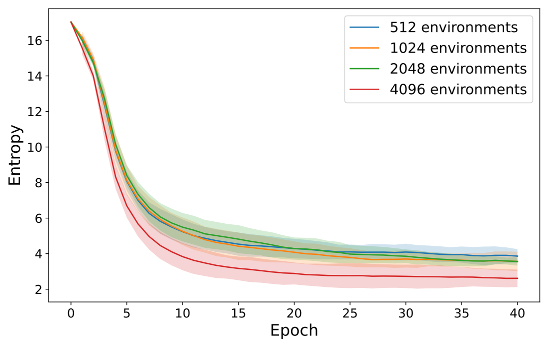
Additional results for the unitree a1 robot



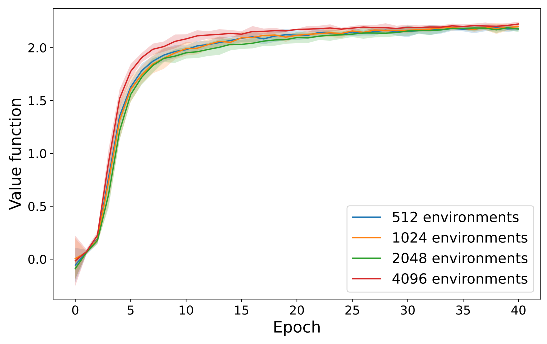
(a) Discounted return after each epoch.



(b) Time needed to achieve return.

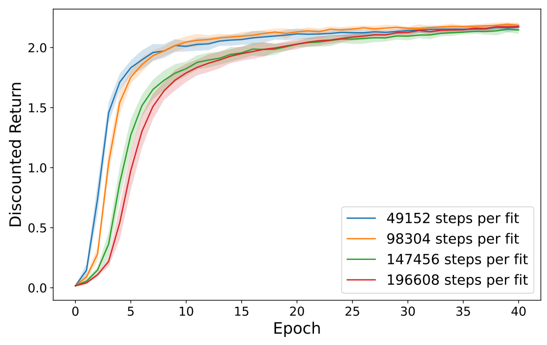


(c) Entropy after each epoch.

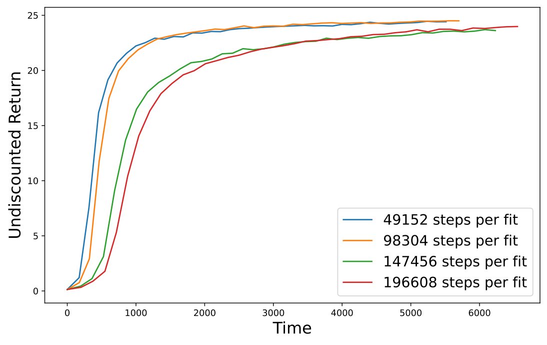


(d) Value function estimate for the initial states.

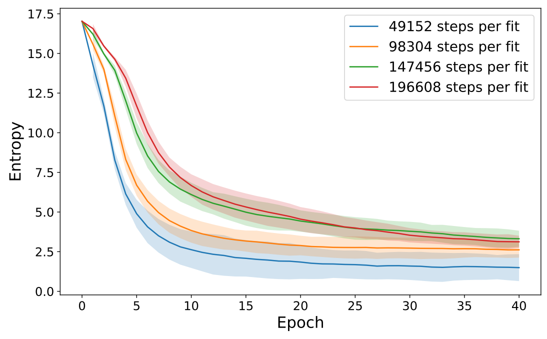
Figure 5.2: Experiment results for the A1 robot using varying numbers of parallel environments, executed on an RTX 3060 Ti GPU and a Ryzen 5800X3D CPU. Each graph presents the mean and the 95% confidence interval. Graph b only shows the mean values. A modified version of MushroomRL's PPO algorithm was used. Each configuration was run five different seeds.



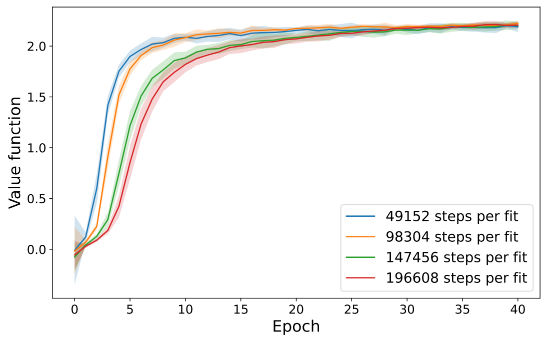
(a) Discounted return after each epoch.



(b) Time needed to achieve return.

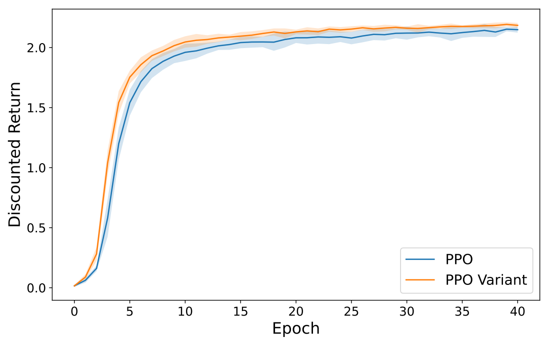


(c) Entropy after each epoch.

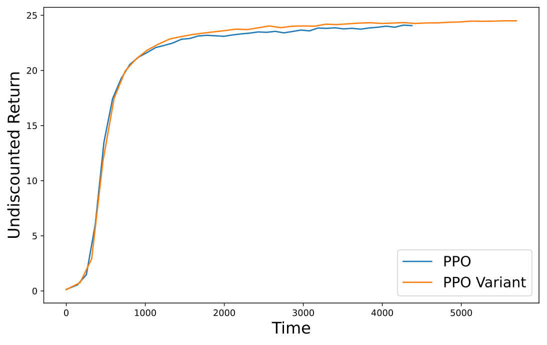


(d) Value function estimate for the initial states.

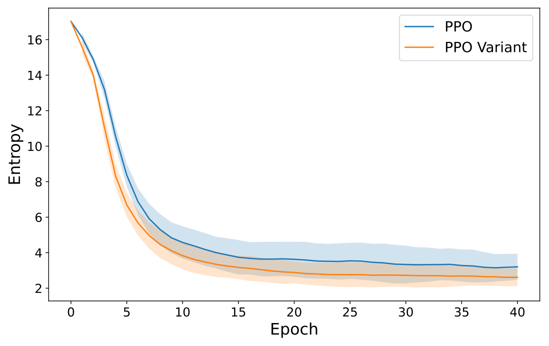
Figure 5.3: Experiment results for the A1 robot using varying values for steps per fit, executed on an RTX 3060 Ti GPU and a Ryzen 5800X3D CPU. Each graph presents the mean and the 95% confidence interval. Graph b only shows the mean values. A modified version of MushroomRL's PPO algorithm was used. Each configuration was run five different seeds.



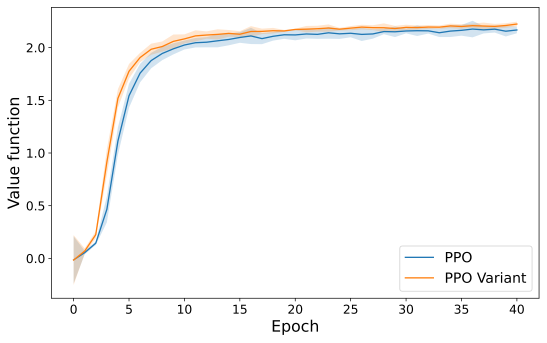
(a) Discounted return after each epoch.



(b) Time needed to achieve return.



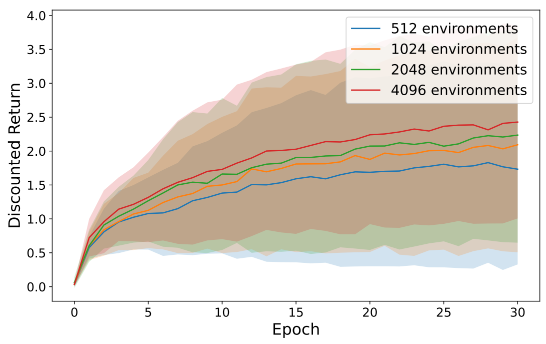
(c) Entropy after each epoch.



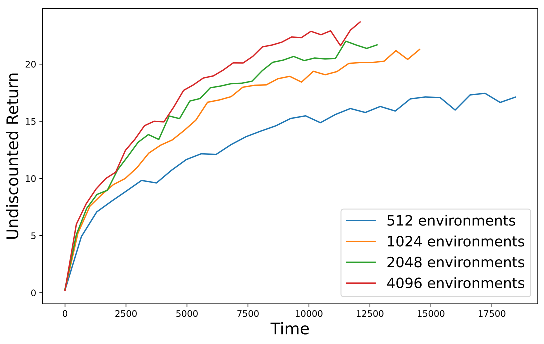
(d) Value function estimate for the initial states.

Figure 5.4: Experiment results for the A1 robot, comparing its performance when trained with MushroomRL's default PPO versus a modified version incorporating gradient clipping and learning rate adaptation. The experiment is executed on an RTX 3060 Ti GPU and a Ryzen 5800X3D CPU. a, c, d present the mean and the 95% confidence interval. A modified version of MushroomRL's PPO algorithm was used. Each configuration was run five different seeds.

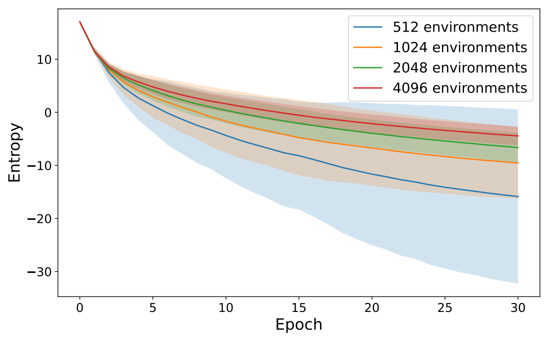
Additional results for the honey badger robot



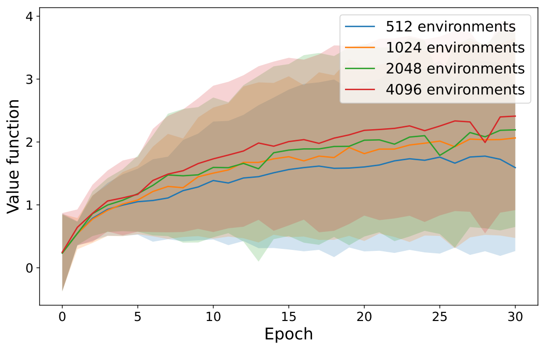
(a) Discounted return after each epoch.



(b) Time needed to achieve return.

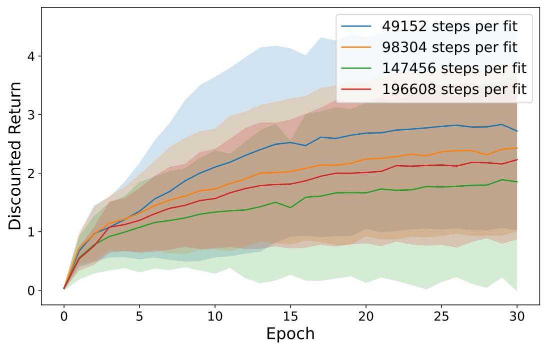


(c) Entropy after each epoch.

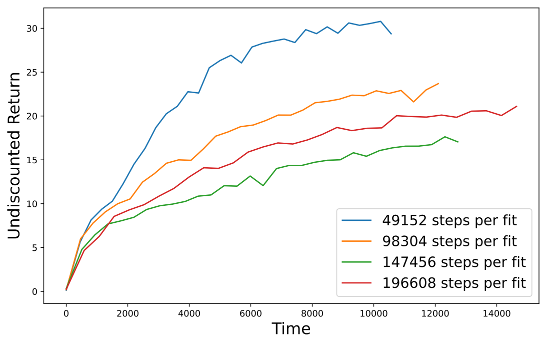


(d) Value function estimate for the initial states.

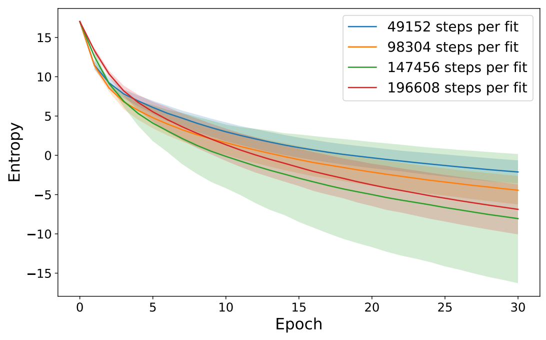
Figure 5.5: Experiment results for the Honey Badger robot using varying numbers of parallel environments, executed on an RTX 2080 Ti GPU and six CPU-Cores of Ryzen 3950X. Each graph presents the mean and the 95% confidence interval. Graph b only shows the mean values. A modified version of MushroomRL's PPO algorithm was used. Each configuration was run five different seeds.



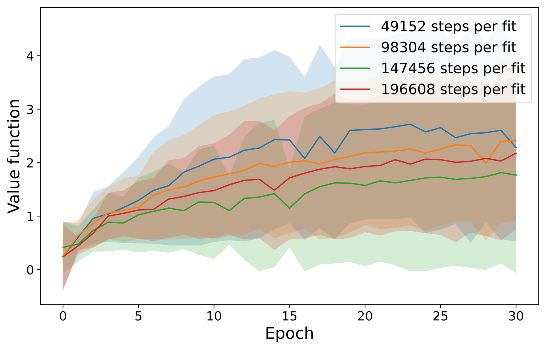
(a) Discounted return after each epoch.



(b) Time needed to achieve return.

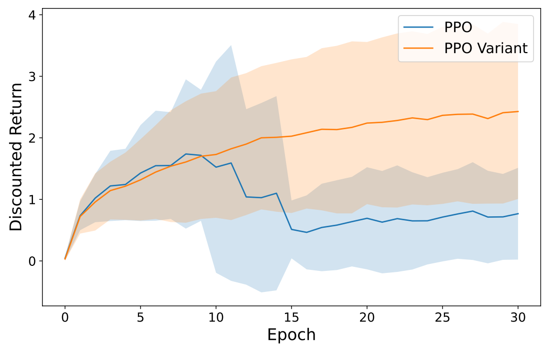


(c) Entropy after each epoch.

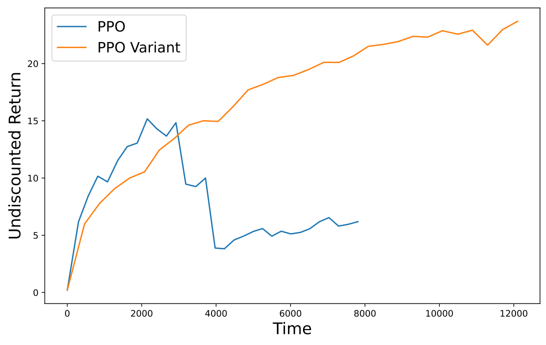


(d) Value function estimate for the initial states.

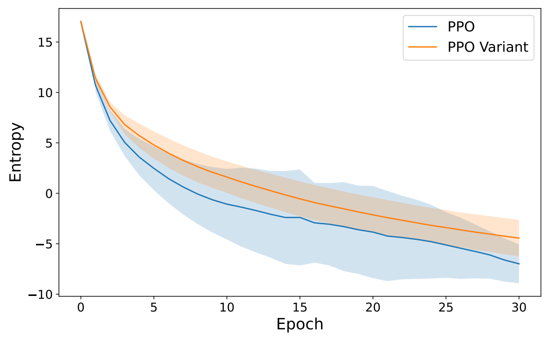
Figure 5.6: Experiment results for the Honey Badger robot using varying values for steps per fit, executed on an RTX 2080 Ti GPU and six CPU-Cores of Ryzen 3950X. Each graph presents the mean and the 95% confidence interval. Graph b only shows the mean values. A modified version of MushroomRL's PPO algorithm was used. Each configuration was run five different seeds.



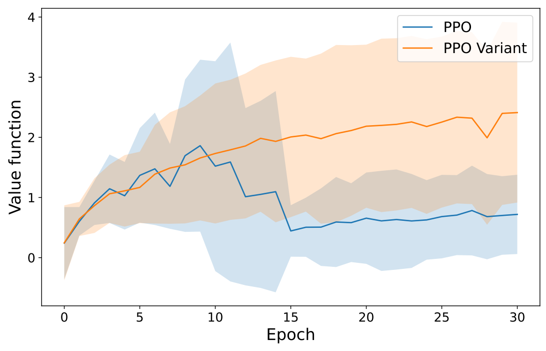
(a) Discounted return after each epoch.



(b) Time needed to achieve return.



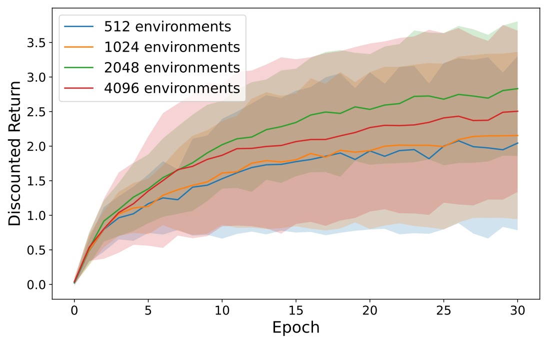
(c) Entropy after each epoch.



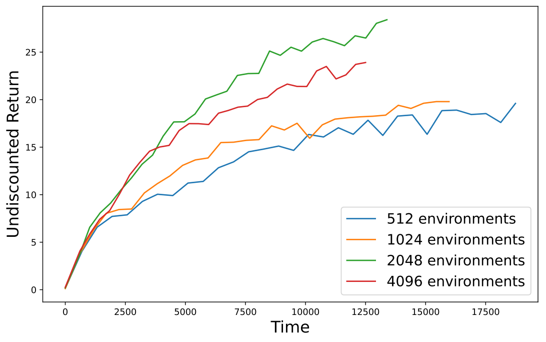
(d) Value function estimate for the initial states.

Figure 5.7: Experiment results for the Honey Badger robot, comparing its performance when trained with MushroomRL's default PPO versus a modified version incorporating gradient clipping and learning rate adaptation. The experiment is executed on an RTX 2080 Ti GPU and six CPU-Cores of Ryzen 3950X. Each graph presents the mean and the 95% confidence interval. Graph b only shows the mean values. Each configuration was run five different seeds.

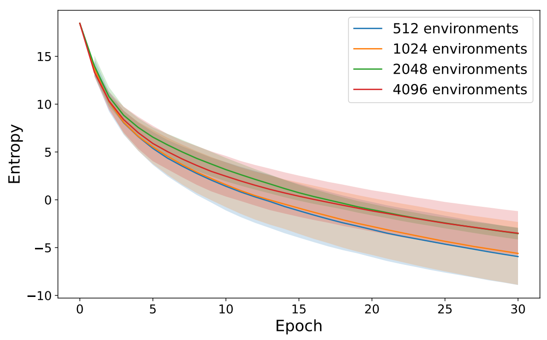
Additional results for the silver badger robot



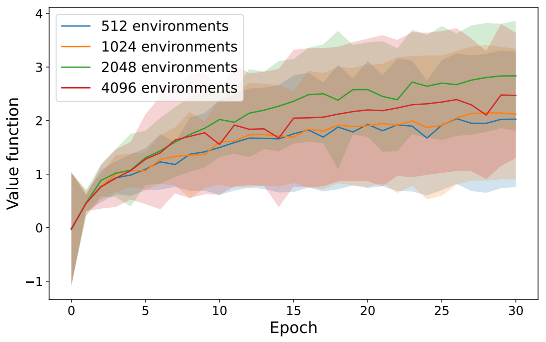
(a) Discounted return after each epoch.



(b) Time needed to achieve return.

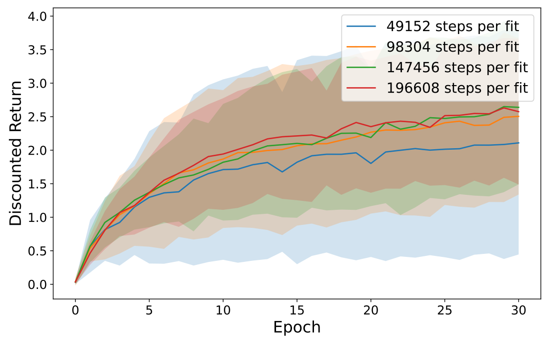


(c) Entropy after each epoch.

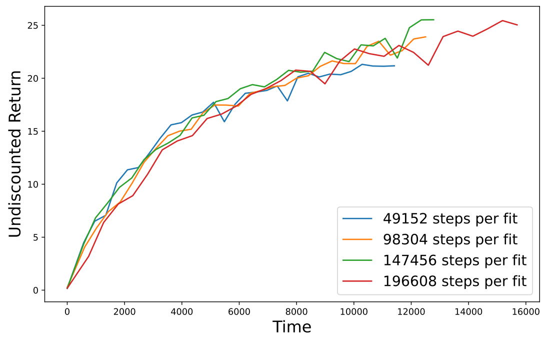


(d) Value function estimate for the initial states.

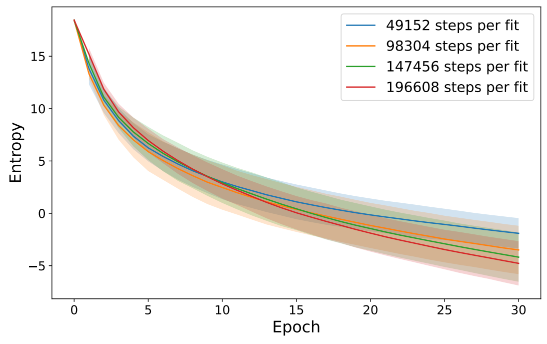
Figure 5.8: Experiment results for the Silver Badger robot using varying numbers of parallel environments, executed on an RTX 2080 Ti GPU and six CPU-Cores of Ryzen 3950X. Each graph presents the mean and the 95% confidence interval. Graph b only shows the mean values. A modified version of MushroomRL's PPO algorithm was used. Each configuration was run five different seeds.



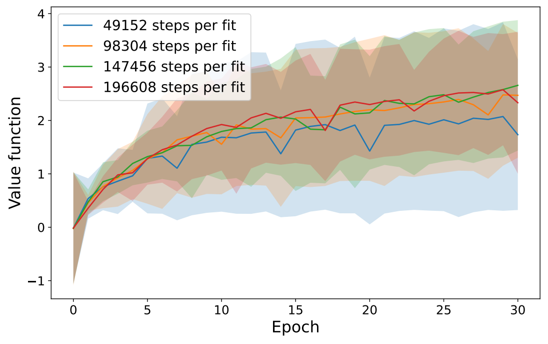
(a) Discounted return after each epoch.



(b) Time needed to achieve return.

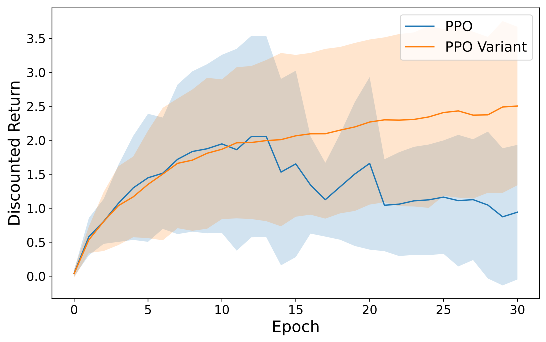


(c) Entropy after each epoch.

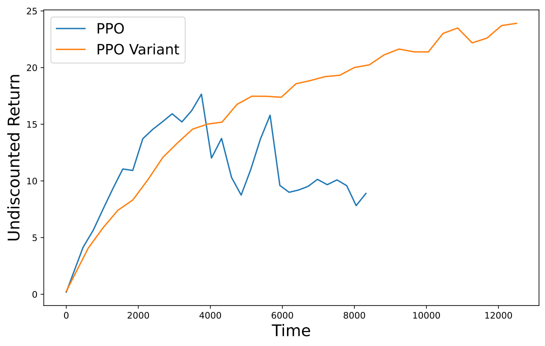


(d) Value function estimate for the initial states.

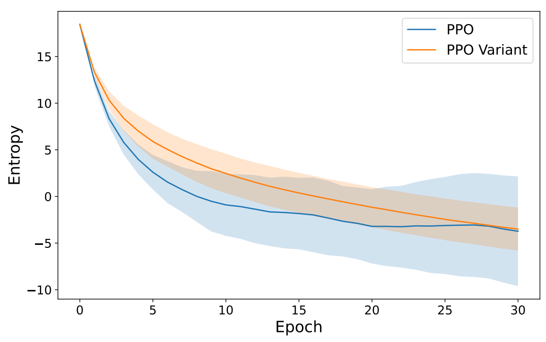
Figure 5.9: Experiment results for the Silver Badger robot using varying values for steps per fit, executed on an RTX 2080 Ti GPU and six CPU-Cores of Ryzen 3950X. Each graph presents the mean and the 95% confidence interval. Graph b only shows the mean values. A modified version of MushroomRL's PPO algorithm was used. Each configuration was run five different seeds.



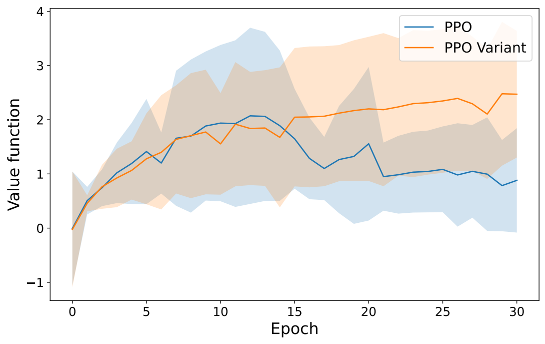
(a) Discounted return after each epoch.



(b) Time needed to achieve return.



(c) Entropy after each epoch.



(d) Value function estimate for the initial states.

Figure 5.10: Experiment results for the Silver Badger robot, comparing its performance when trained with MushroomRL's default PPO versus a modified version incorporating gradient clipping and learning rate adaptation. The experiment is executed on an RTX 2080 Ti GPU and six CPU-Cores of Ryzen 3950X. Each graph presents the mean and the 95% confidence interval. Graph b only shows the mean values. A modified version of MushroomRL's PPO algorithm was used. Each configuration was run five different seeds.