Exploration of Reinforce Learning Algorithms to Train a Simulated Robot in Various Domains

Lorenzo Suppa, Vito Perucci, Tanguy Dugas du Villard
Politecnico di Torino
Corso Duca degli Abruzzi, 24, Torino, Italy
https://github.com/Tanguy-ddv/rl_project

Abstract

This project investigates the application of reinforcement learning techniques to impart the ability to jump to a simulated hopper. We conduct experiments with various algorithms and methodologies, including REINFORCE, Actor-Critic, and Proximal Policy Optimization, to augment the hopper's learning efficiency and performance within a simulated environment. Additionally, we explore strategies aiming at mitigating the sim2real gap. The implementations of Active Domain Randomization, Uniform Domain Randomization, and Gaussian Domain Randomization produce favorable results regarding the robustness and the performance of the hopper in different environments. This research lays the groundwork for more efficacious real-world applications of reinforcement learning in robotic systems.

1. Introduction

Reinforcement Learning (RL) is a sophisticated machine learning paradigm that draws inspiration from the human learning process, particularly through iterative 'trial-and-error' interactions between an agent and a dynamic environment. RL is based on the *Markov Decision Process* (MDP) model, which represents sequential decision-making problems under uncertainty [10]. An MDP consists of:

- **States** S: The possible situations in which the agent can find itself. A certain state is represented as s and of course $s \in S$.
- **Actions** A: The set of all possible actions the agent can take. A certain action is represented as a and of course $a \in A$.
- **Transition Functions** $P(s'|s,a)$: The probabilities of moving from one state to another, given an action.
- **Reward Functions** $R(s,a)$: The immediate reward received after taking an action in a state.
- **Discount Factor** γ: A factor used to discount future rewards, balancing immediate and long-term rewards.

In RL, an agent operates autonomously, making decisions and striving to achieve specific objectives without human supervision.

A central element in RL is the policy that guides the agent's actions within the environment. The **Policy**, denoted as $\pi(a|s)$, is a strategy used by the agent to decide which action to take based on the current state s. The connection between an agent and its policy is fundamental to how the agent learns and adapts over time. The policy can be deterministic or stochastic, and it is often represented by a neural network in complex environments.
Solving an MDP involves finding a policy that maximizes the expected cumulated reward. RL extends the MDP framework to handle situations where the environment model or the reward function is unknown. Through trial and error interactions with the environment, RL agents aim to learn optimal policies without prior knowledge of the environment dynamics. Each iteration in RL, called an episode, ends when a termination state is reached, which can be a wrong or dangerous position, or a predefined time limit. During each episode, the agents interacts with the enviroment by observing its current *state*, selecting and executing an *action*, and then receiving a *feedback* in the form of rewards. This feedback is used to update the agent's decision-making strategy, with the aim of maximizing the cumulative reward obtained over time.

2. Related Works

Recent advancements in reinforcement learning (RL) have significantly impacted the field of sim-to-real transfer, where models trained in simulated environments are deployed in real-world scenarios. Notable works have addressed the challenge of the sim-to-real gap, which arises due to discrepancies between simulated and real-world environments. One of the most notable advancements is the Cherry-Picking paper [14] demonstrating the effectiveness of pre-training in simulation followed by fine-tuning in the real world. Tobin et al. (2017) [13] introduced domain randomization, a technique where the simulator is varied extensively to expose the model to a wide range of conditions, thereby improving its robustness in the real world. OpenAI et al. (2019) extended this concept by training a robotic hand to solve a Rubik's Cube [9], demonstrating the potential of domain randomization in complex manipulation tasks.

Other approaches focus on reducing the sim-to-real gap through improved simulation fidelity and transfer learning. Rusu et al. (2017) proposed Progressive Neural Networks to leverage knowledge from multiple tasks [2], facilitating transfer learning across different domains. Additionally, Andrychowicz et al. (2017) introduced Goal-Conditioned RL and Hindsight Experience Replay (HER) [6], which have shown promise in enabling agents to generalize from simulated to real environments by learning from both successes and failures.

Further advancements have explored the integration of meta-learning techniques to adaptively adjust to new environments. Obamuyide et al. (2019) presented the concept of Model-Agnostic Meta-Learning (MAML) in the context of RL [8], allowing agents to quickly adapt to new tasks with minimal data, thus enhancing sim-to-real performance. This body of work collectively underscores the importance of bridging the sim-to-real gap through diverse and innovative methodologies, paving the way for more reliable and robust RL applications in real-world settings.

3. The Environment

The robot trained in this study is the gym's hopper [3], simulated using the MuJoCo physics engine [12]. The Hopper is composed of four cylindrical components: a torso, a thigh, a leg, and a foot, interconnected by three articulations. We will use two different variants of the same environment, named *Source* and *Target*. For the first configuration, the masses are: torso - 2.53 kg, thigh - 3.93 kg, leg - 2.71 kg, foot - 5.09 kg; For the Target one the masses are the same of the Source with the exception of the torso that is one kilogram heavier (3.53 kg). This modification aims to assess how algorithms trained in a certain environment are likely to perform worse even in a nearly identical settings, thereby supporting the thesis that sim-to-real transfer is particularly challenging. The Hopper operates on a flat floor with movement restricted to the bottom-top and left-right directions. During training, the objective for the robot is to learn to move towards the right by jumping. The articulations are actuated by motors that apply forces to control their movements. These forces constitute the actions in the reinforcement learning process, while the states are characterized by the angular positions and velocities of the robot's articulations, as well as the relative position of the robot to the ground. The environment features a continuous state space where the angular positions and velocities of the articulations, the relative position of the robot to the ground, and other relevant variables describing the robot's orientation and dynamics are indeed continuous. Similarly, the action space consists of continuous ranges of force values that agents can apply to each articulation.
The primary reward is given when the robot successfully moves to the right and maintains an upright stance. Episodes terminate either when the robot falls or when a maximum number of steps is reached. A screenshot of the hopper in its environment is show Fig. 1

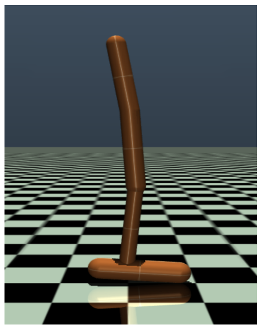
Figure 1. The hopper in its environment

4. Basic Algorithms

In this section, we focus on the algorithms used in our Reinforcement Learning framework to train the Hopper agent, particularly policy gradient methods [10]. These methods use various episodes to learn a parameterized policy, that is updated based on the data collected during each step of the episode. We record the state, the action taken, and the reward obtained; these recordings are used to compute $J(\theta)$, the performance measure. The policy's parameters are not updated continuously but only at the end of each episode using the collected data.

The policy's parameter vector is $\theta \in \mathbb{R}^d$. We represent the policy as $\pi(a \mid s, \theta) = \Pr\{A_t = a \mid S_t = s, \theta_t = \theta\}$, the probability of taking action a at time t given state s and parameter θ.

We consider methods for learning the policy parameters based on the gradient of $J(\theta)$:

$$\theta_{t+1} = \theta_t + \alpha \nabla J(\theta_t), \tag{1}$$

where $\nabla J(\theta_t) \in \mathbb{R}^d$ is a stochastic estimate approximating the gradient of the performance measure and α is the learning rate.

The Policy Gradient Theorem is crucial in these methods, providing a foundation for computing the gradient of the expected reward with respect to the policy parameters. This theorem guides the updates to increase the expected cumulative reward, making it essential for policy gradient methods. Next, we will examine how these algorithms function and contribute to the training of the Hopper.

4.1. REINFORCE

The REINFORCE [10] algorithm is the simplest policy gradient method in RL. To be more detailed, the update rule for this algorithm is given by:

$$\theta_{t+1} = \theta_t + \alpha G_t \nabla_\theta \log \pi(A_t \mid S_t, \theta_t) \tag{2}$$

where θ_t is the policy parameter vector at time t, α is the learning rate, $\nabla_\theta \log \pi(A_t \mid S_t, \theta_t)$ is the gradient of the log-probability of the taken action A_t given the state S_t and policy parameters θ_t and G_t is defined as:

$$G_t = \sum_{i=t}^{T} \gamma^i r_i \tag{3}$$

which is the bootstrapped cumulative reward (return) obtained after time t. Here the episode is composed of T steps, all of them rewarding r_i. γ is the discount factor, generally close to 1. By introducing bootstrapped cumulative returns, REINFORCE can leverage value estimates to provide more frequent updates and reduce the variance of the returned estimates. The term $\nabla_\theta \log \pi(A_t \mid S_t, \theta_t)$ serves as an indicator of how the probability of the chosen action changes with respect to the policy parameters. By scaling this gradient by the cumulative reward G_t, the update rule reinforces actions that lead to higher rewards and discourages actions that result in lower rewards. In this way, the policy parameters are adjusted to increase the likelihood of actions that yield higher returns.

The simplicity and directness of the REINFORCE algorithm make it a foundational method in policy gradient approaches. However, it is important to note that REINFORCE can suffer from high variance in the gradient estimates, which can lead to slow learning. Various improvements and extensions, such as baselines, are often employed to address these issues.

4.1.1 With constant baseline

A small modification to the REINFORCE algorithm involves adding a baseline. This baseline can be either a constant or a function that depends only on the observation, for example, the value-function [10].

$$\theta_{t+1} = \theta_t + \alpha \cdot \nabla_\theta \log \pi_\theta(a_t \mid s_t) \cdot (G_t - b(s_t)) \tag{4}$$

With $b = 0$, we have the previous REINFORCE algorithm.

4.1.2 Results and considerations

Results in Fig. 2 highlight the importance of selecting the appropriate baseline in the REINFORCE algorithm. A

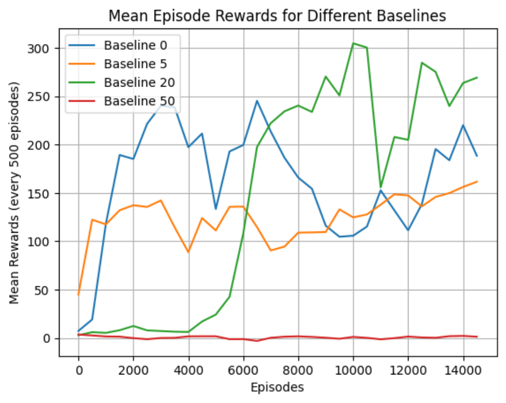

Figure 2. Training of a REINFORCE agent

well-chosen baseline significantly reduces the variance of policy gradient estimates, leading to more stable updates and improved convergence rates. Importantly, incorporating a baseline does not introduce bias but enhances the practical performance of the REINFORCE algorithm.

However, as shown by the results with a baseline of 50, an excessively high baseline can cause many actual returns to fall below it, resulting in negative advantage estimates. This decreases the probability of beneficial actions, leading to suboptimal policy updates and poor performance. A high baseline reduces the perceived difference between good and bad actions, diminishing the incentive to optimize, causing slow learning and erratic updates.

High rewards for a hopping robot typically indicate prolonged upright stability. In our experiment, a baseline of 50 resulted in a total runtime of 207.290 seconds, while a baseline of 20 required 1604.660 seconds. When visualizing the robot's operation, it attempts to remain standing and only slightly moves to the right. Even if it finds a way to earn rewards, this behavior is unacceptable as we want it to jump.

4.2. ACTOR-CRITIC

The Actor-Critic [10] algorithm represents a significant advancement in Reinforcement Learning, combining elements of both policy-based and value-based approaches. Unlike REINFORCE, which updates the policy based solely on the rewards received, Actor-Critic introduces a separate value function to provide additional information about the quality of actions taken by the agent. This combination allows for more efficient learning by leveraging both policy optimization and value estimation.

4.2.1 Algorithm

In the Actor-Critic algorithm, the agent consists of two components: the Actor and the Critic. The Actor selects

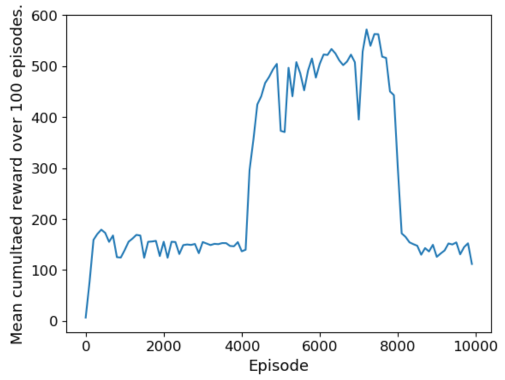

Figure 3. Training of an actor-critic agent

actions according to the current policy, which is typically parameterized by a neural network. This network takes the state as input and outputs a probability distribution over actions. The actor's objective is to maximize the expected return by updating its parameters based on the feedback received from the critic. The Critic, on the other hand, estimates the expected return of the actions taken by the actor by maintaining a value function. This function predicts the future cumulative reward starting from a given state-action pair. The Critic's objective is to minimize the error between its predicted values and the actual returns received from the environment. Formally the updates are:
Actor Update: The actor (policy network) updates its parameters θ by gradient ascent on the expected advantage:

$$\theta_{t+1} = \theta_t + \alpha_\theta \cdot \nabla_\theta \log \pi_\theta(a_t \mid s_t) \cdot A(s_t, a_t) \quad (5)$$

where $\pi_\theta(a_t \mid s_t)$ is the policy's probability of selecting action a_t in state s_t, and $A(s_t, a_t) = G(s_t, a_t) - V(s_t)$ is the advantage function indicating the desirability of taking action a_t in state s_t where $G(s_t, a_t)$ is the estimated return when taking action a_t in state s_t.
Critic Update: The critic (value function) updates its parameters ϕ by minimizing the temporal difference (TD) error:

$$\phi_{t+1} = \phi_t + \alpha_\phi \cdot \delta_t \cdot \nabla_\phi V(s_t) \quad (6)$$

where $\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$ is the TD error, r_{t+1} is the reward received after action a_t in state s_t, γ is the discount factor, and $V(s_t)$ is the critic's estimate of the value function for state s_t.
By jointly training the Actor and the Critic, the Actor-Critic algorithm can learn to improve the policy more efficiently than pure policy gradient methods like REINFORCE. The Critic provides valuable feedback to the actor, guiding it towards actions that are likely to lead to higher rewards.

4.2.2 Results

The synergistic relationship between the Actor and Critic enables more stable and faster convergence during training. The algorithm was tested under the same experimental conditions used to evaluate the REINFORCE algorithm, with the cumulative training rewards depicted in Fig. 3. Compared to REINFORCE, the Actor-Critic approach demonstrates notable advantages. It achieves faster convergence to high reward levels and exhibits improved sample efficiency.

During training, distinct phases of reward dynamics were observed. Initially, the rewards were modest as the Actor and Critic initialized their learning processes. However, there was a marked improvement in performance as the actor leveraged feedback from the Critic to optimize its policy. This phase showcased the Actor-Critic's capability to swiftly adapt and exploit learned insights.

Despite these strengths, the training process exhibited variability. While performance often peaked, there were also periods of declining rewards. These fluctuations underscore the complex interplay between the actor's policy updates and the critic's value estimations. Managing these dynamics effectively is crucial for maximizing the Actor-Critic algorithm's potential in reinforcement learning tasks.

The Actor-Critic algorithm achieved peak rewards of 600, which are significantly higher than the peaks reached by the REINFORCE algorithm. However, these peaks were not sufficient to enable the Hopper to perform the desired jumping tasks consistently. This highlights the challenges in training the agent to achieve stable and reliable performance in complex environments.

Looking forward, advancements in algorithms and approaches are essential to further enhance the capabilities of Hopper jumping tasks. Later discussions will delve into these advancements and their implications for future research and applications in reinforcement learning.

5. Proximal Policy Optimization

Following our discussion on the Actor-Critic algorithm, we turn our attention to Proximal Policy Optimization (PPO) [5], a state-of-the-art reinforcement learning algorithm that addresses limitations of traditional policy gradient methods like REINFORCE and Actor-Critic, such as high variance, slow convergence, and instability. PPO mitigates these issues through innovative techniques. PPO aims to answer a critical question: how can we take the largest possible improvement step on a policy using the current data without causing performance collapse? PPO addresses this with two techniques: *PPO-Clip* and *PPO-Penalty*. PPO-Clip uses a clipped surrogate objective function to limit policy updates, ensuring new policies do not deviate excessively from old ones. PPO-Penalty adds a penalty term to the objective function that directly penal-

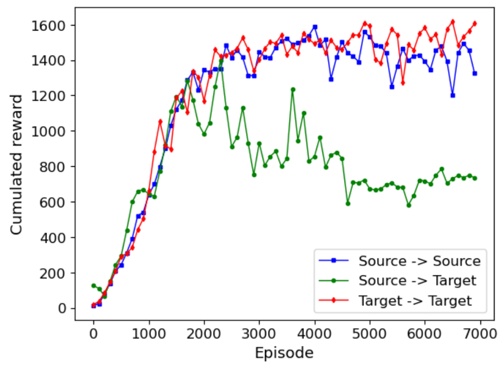

Figure 4. PPO training and testing

izes large deviations between new and old policies. Both variants improve stability and efficiency in policy optimization. The choice between them depends on the task requirements and the desired balance between simplicity and control. Unlike the REINFORCE and Actor-Critic algorithms implemented with PyTorch, we use the PPO agent from the stable-baselines3 library [4]. This library simplifies the process: once the agent is created and linked to the training environment, it can be trained for a specified number of timesteps, with options to add callbacks for saving outputs or testing during training.

5.1. Hyper-parameters tuning

The PPO has a lot of different parameters that need to be tuned. Including: The *learning rate*, the *clip range*, the *batch size*, the *discount factor* γ and the *number of steps*. These five parameters have been tuned using optuna [1], a Python library used to do hyperparameter tuning. The main advantage of this library, compared to five nested 'for' loops, is the ability to parallelize the computations and explore continuous domains. 50 trials with different combinations of hyper-parameters have been tested on a training of 100,000 timesteps. Finally, the best model has the following hyper-parameters: $\gamma = 0.993$, a batch size of 64, a clip range of 0.27, and a learning rate of 8e-4.

5.2. Performance

Once we found the best hyperparameters, we trained the PPO agent for 70,000 episodes on both the target and source environments. We converted the timesteps of the PPO into episodes using the stable-baselines3 library, which increments the current episode when a termination condition is met. This conversion allowed us to compare PPO's performance with the previous algorithms effectively.

The testing rewards are shown in Fig. 4, in which the arrows in the legend indicate the transition from the training environment to the testing environment. The robot's performance depends significantly on that; when the training environment matches the testing environment, as depicted by the blue and red curves, the robot performs well. This is because, during training, it learns optimal actions based on the current state and how those actions lead to future states that maximize rewards.

However, when the robot is trained in one environment but then tested in a different environment (green curve), its performance declines. This discrepancy occurs because the dynamics of the environments differ. Specifically, the transition function $P(s'|s,a)$, varies between environments. Factors like different masses in the environment can alter this transition function. The green curve represents the PPO agent trained in the Source environment and tested in the Target environment every 100 episodes. This process was used to simulate, with an high level of approximation, transferring the policy from a simulated environment to a real-world setting, where direct transfer wasn't possible. The ascending portion of the green curve is explained by the fact that the agent was able to perform discreetly in the Target environment because it had not yet overlearned from the starting one, preserving its ability to explore and adapt to new conditions. The performance gap between the curves highlights the error a robot trained in one environment would make in another environment. To address this issue, the next section will explore the technique of domain randomization.

6. Domain Randomization

6.1. Motivation

Domain Randomization [11] aims to improve the robustness of the learned policies by exposing the agent to a variety of environments during training, thereby reducing the discrepancy in performance when transitioning to new and unobserved environments. The environment used for training is the Source environment, while the environment of test, on which we want our robot to perform is the Target. To model the sim-2-sim transfer, we don't want to train our hopper on the Target environment, as it is used to model the real life, that we don't know. Instead, we train our robot in different environments, none of them being the target one, and all of them derived from the Source one.

In these experiments, the domains differ from each other by the masses of the thigh, leg, and foot. We don't modify the torso mass because we want our robot to improve its robustness, and to make it able to perform in an environment that it has never seen during its training. Uniform domain randomization (UDR) and Gaussian domain randomization (GDR) are tested first, before improving the complexity and hopefully the performance with active domain randomization (ADR). We will skip previous hyperparameter tuning for PPO since the optimal settings identified may not be universally applicable.

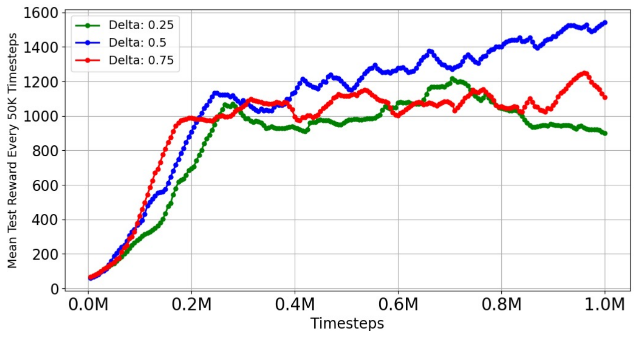

Figure 5. Delta tuning for UDR

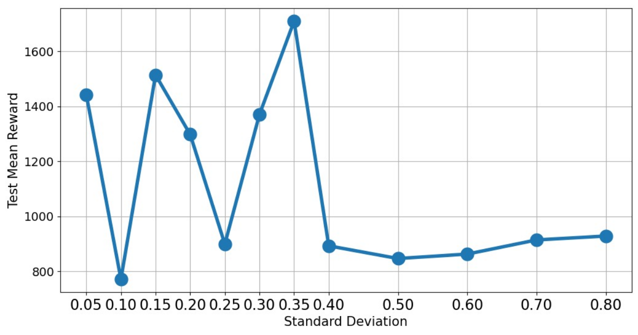

Figure 6. Standard deviation tuning for GDR

6.2. Uniform Domain Randomization

One way to modify the masses is to set them as a random value at the beginning of each episode. Here, we draw the mass from a uniform random variable.

As we know, a uniform distribution is defined by a range within which values are randomly selected with the same probability. We set the two ends of this range based on the original mass, adjusted by adding and subtracting a delta multiplied by the mass itself to have proportional bounds for each mass. As mentioned earlier, we only randomized the masses of the parts of the hopper other than the torso.

Algorithm 1 Uniform domain randomization

```
Require: $D$ the source domain, $T$ the number of episode,
  $\pi$ the policy, $D'$ the target domain
  $m_0 \leftarrow D$.masses
  for $t = 1..T$ do
    $M \sim Uniform(m_0 - \delta m_0, m_0 + \delta m_0)$
    $D$.masses $\leftarrow M$
    train 1 episode on $D$
  end for
  test on $D'$
```

We experimented with three different deltas, training the algorithm in these randomized environments and periodically testing the model in the target environment. This helped us see which delta worked best for this task.

The results of the deltas tuning are shown in the Fig. 5. We observed that the delta value of 0.5 consistently outperforms the other two deltas across the testing episodes in the target environment. This indicates that moderate randomization of limb masses, neither too small nor too large, strikes a balance that enhances the agent's adaptability without significantly compromising its learned behaviors.

6.3. Gaussian Domain Randomization

Another way to generate a domain that differs from the source and the target is to use Gaussian randomization. This approach samples each mass from different normal distributions. The means are set to the original masses (which is the most obvious and functional approach) and a set of different standard deviations is used to conduct a fine-tuning process to determine the optimal one. The algorithm used to perform training with normal domain randomization is explained in Algorithm 2.

Algorithm 2 Normal domain randomization

```
Require: $D$ the source domain, $T$ the number of episode,
  $\pi$ the policy, $D'$ the target domain, $\sigma^2 > 0$
  $m_0 \leftarrow D$.masses
  for $t = 1..T$ do
    $M \sim Normal(m_0, m_0^2\sigma^2)$
    $D$.masses $\leftarrow M$
    train 1 episode on $D$
  end for
  test on $D'$
```

We tested twelve different standard deviations using 500,000 timesteps for each configuration and evaluated their impact on model performance when tested in the target environment. The results, shown in Fig. 6, illustrate the average rewards obtained in these tests.

Interestingly, we observed that lower standard deviations tend to facilitate better performance in terms of robustness. This suggests that excessively different environments between episodes may hinder the model's learning effectiveness. Despite the absence of a clear pattern, it is evident that randomization of the Gaussian domain effectively improves the model's fit.

In conclusion, Gaussian domain randomization proves to be highly effective, significantly improving algorithmic robustness. These results highlight its usefulness in mitigating the gap between simulated training and real-world application scenarios.

7. Active Domain Randomization

7.1. Presentation

While Uniform Domain Randomization (UDR) and Gaussian Domain Randomization (GDR) provide effective means for enhancing the robustness of models by exposing them to a wide range of environments, they rely on predetermined, static distributions of parameters. These methods randomly sample domain parameters uniformly or according to a Gaussian distribution, without considering the model's current performance or the complexity of the task at hand. This can lead to inefficient training, as some sampled environments might either be too easy, providing little benefit, or too difficult, hindering learning progress.

ADR addresses these limitations by incorporating a more adaptive approach. Instead of sampling domain parameters from a fixed distribution, ADR dynamically adjusts the sampling strategy based on the model's performance in different environments. The core idea of ADR is to identify and focus on environments that are particularly informative for the training process. This is achieved through an iterative loop, where the model's performance is evaluated, and domain parameters are adjusted accordingly to maximize learning efficiency.

As we have seen in previous domain randomization strategies, the environment is built based on the masses of the Hopper components. With ADR the masses are selected by Actor-Critic agents, called particles. As explained by [7] the particles propose a diverse set of environments, trying to find the environment parameters that are currently causing the agent the most difficulty.

To train our particles, we need actions, rewards, and states. Here actions are vectors of differences. This difference is added to the current mass to obtain the next mass, at each iteration. The states are the masses themselves. Finally, the reward is a number indicating if the last mass produced a trajectory similar to a trajectory generated on the test domain.

To generate this reward, we define a discriminator, which is a neural network. The discriminator takes as input a triplet (state, action, next-state) generated in a trajectory and returns a value between 0 and 1. The discriminator is trained at the end of each episode, by mapping (state, action, next-state) vectors to 1 if generated from a training domain, and 0 if generated from the target domain. Then, the discriminator return is averaged, and its logarithm is given to the particle. Algorithm 3 explains better the process.

7.2. Experiments

ADR algorithm has been applied to PPO agents. In our experiments, we aim to evaluate two different strategies to understand their impact on the model's performance in the Target environment. We call the ADR-randomized

Algorithm 3 Active Domain Randomization

```
Require: $N \in \mathbb{N}$, $\mu_i$ particles, $D$ source domain, $D'$ target
    domain, $T$ number of episodes, $d$ discriminator, $\pi policy$,
    $D_i$ copies of $D$
  for $t$ = 1...$T/N$ do
    for $i$ = 1...$N$ do
      $\Delta m_i \leftarrow \mu_i(D_i.\text{masses})$
      $D_i.\text{masses} \leftarrow D_i.\text{masses} + \Delta m_i$
      train $\pi$ on $D_i$
      extract $s, a, s'$ at each step
      $r_i \leftarrow d(s, a, s')$
      train $\mu_i$ with the reward $-log(r_i)$
      train $d$ with $s, a, s'$ mapping to 1
    end for
    test $\pi$ on $D'$
    extract $s, a, s'$ at each step
    train $d$ with $s, a, s'$ mapping to 0
  end for
```

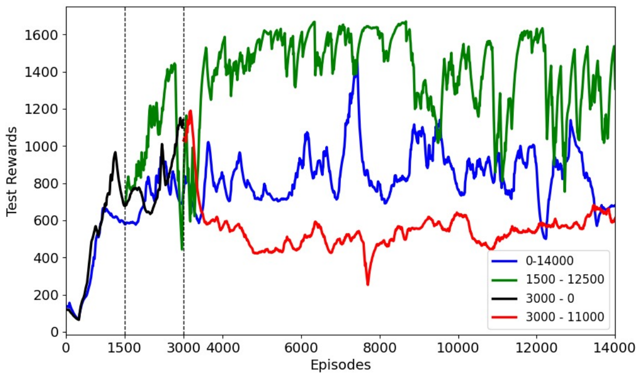

Figure 7. ADR performance with different pre-train episodes

Source environment *ADR-Source* for simplicity. Specifically, we compared the performance of models trained solely in ADR-Source against models that underwent pre-training in the static Source followed by training in ADR-Source.

7.3. Results and comparison with other algorithms

The results, presented in Fig. 7, compare the performance of the two strategies. The blue line represents a model trained without pre-training, undergoing 14,000 episodes of ADR-Source training from the start. The green and red lines depict models that underwent pre-training in the static Source environment before continuing training with in the ADR-Source.

The results suggest that pre-training in a static environment before transitioning to a dynamic one can significantly improve the model's performance. Pre-training in a static environment allows the model to learn fundamental policies without the added complexity of a dynamic environment, leading to quicker convergence to a stable policy. Once this foundation is established, the model is better equipped to handle the variations introduced by the ADR environment, making the adaptation process faster and more effective than learning from scratch in a complex environment. This approach essentially employs a form of curriculum learning, where the training process starts with simpler tasks and gradually increases in complexity, effectively balancing stability and generalization.

However, pre-training for too long in a static environment can hugely slow down the learning process due to overfitting in the static environment, as can be seen in the result provided by the model pre-trained for 3000 episodes in Source (red line in Fig. 7). Therefore, it's crucial to find the right balance in pre-training duration to maximize the benefits without hindering overall performance.

Overall, this strategy enhances the performance and robustness of PPO models in complex environments by leveraging the stability of static environments and the generalization capabilities of ADR environments.

Moreover, this approach can significantly bridge the sim-to-real gap, enabling models trained in simulated environments to perform more effectively in real-world applications.

8. Conclusion

While algorithms such as REINFORCE and Actor-Critic are foundational, they often lag behind the more efficient PPO algorithm discussed in this report. Domain randomization techniques have demonstrated their effectiveness in enhancing a robot's efficiency within simulated environments. Active Domain Randomization (ADR) stands out for its complexity and capacity to improve generalization.

However, training robots using these algorithms involves inherent stochasticity, resulting in performance variability. This necessitates iterative training processes where algorithms are tested and refined multiple times to mitigate stochastic effects and achieve optimal performance. Despite being time-consuming and requiring meticulous adjustment of hyperparameters, this iterative approach is crucial for developing sophisticated robotic behaviors.

In conclusion, while the optimal performance of these techniques in real-world scenarios remains uncertain, their ability to simulate diverse and complex environments shows promise in bridging the sim2real gap. Techniques such as Generalized Domain Randomization (GDR), Active Domain Randomization (ADR), and Uniform Domain Randomization (UDR) are pivotal considerations. They demonstrate significant potential in preparing robots trained in simulation to effectively adapt to the complexities and variations encountered in real-world applications

References

[1] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2623–2631, 2019. 5

[2] Guillaume Desjardins Hubert Soyer James Kirkpatrick Koray Kavukcuoglu Razvan Pascanu Raia Hadsell Andrei A. Rusu, Neil C. Rabinowitz. Progressive neural networks, 2022. 2

[3] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016. 2

[4] Ashley Hill, Antonin Raffin, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, and Matias Traore. Stable baselines. `https://github.com/hill-a/stable-baselines`, 2018. 5

[5] Prafulla Dhariwal Alec Radford Oleg Klimov John Schulman, Filip Wolski. Proximal policy optimization algorithms, 2017. 4

[6] Alex Ray Jonas Schneider Rachel Fong Peter Welinder Bob McGrew Josh Tobin Pieter Abbeel Wojciech Zaremba Marcin Andrychowicz, Filip Wolski. Hindsight experience replay, 2017. 2

[7] Bhairav Mehta, Manfred Diaz, Florian Golemo, Christopher J. Pal, and Liam Paull. Active domain randomization. In Leslie Pack Kaelbling, Danica Kragic, and Komei Sugiura, editors, *Proceedings of the Conference on Robot Learning*, volume 100 of *Proceedings of Machine Learning Research*, pages 1162–1176. PMLR, 30 Oct–01 Nov 2020. 7

[8] Abiola Obamuyide and Andreas Vlachos. Model-agnostic meta-learning for relation classification with limited supervision. pages 5873–5879, 01 2019. 2

[9] Marcin Andrychowicz Maciek Chociej Mateusz Litwin Bob McGrew Arthur Petron Alex Paino Matthias Plappert Glenn Powell Raphael Ribas Jonas Schneider Nikolas Tezak Jerry Tworek Peter Welinder Lilian Weng Qiming Yuan Wojciech Zaremba Lei Zhangs OpenAI, Ilge Akkaya. Solving rubik's cube with a robot hand, 2019. 2

[10] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, second edition, 2018. 1, 2, 3

[11] Joshua Tobin, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and P. Abbeel. Domain randomization for transferring deep neural networks from simulation to the real world. *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 23–30, 2017. 5

[12] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control, 2012. 2

[13] Chi Jin Lihong Li Liwei Wang Xiaoyu Chen, Jiachen Hu. Understanding domain randomization for sim-to-real transfer, 2022. 2

[14] Yunchu Zhang, Liyiming Ke, Abhay Deshpande, Abhishek Gupta, and Siddhartha Srinivasa. Cherry-picking with reinforcement learning : Robust dynamic grasping in unstable conditions, 2023. 1