Adversarial and Dynamics Curricula: Reinforcement Learning Pathways for Sim2Real Transfer in One-Legged Robotics

Maddalena Ghiotti s332834 | Letizia Greco s336195 | Lorenzo Terna s331113 | Chiara Zambon s329148

Abstract

Reinforcement learning has achieved strong performance in solving continuous-control tasks in simulation, training policies that maximise expected reward. However, deploying those policies on real robots remains hindered by the sim-to-real gap caused by differences in dynamics, physical constraints, and sensor noise. This project focuses on training a one-legged robot simulator to walk, using both basic and state-of-the-art Reinforcement Learning algorithms—namely Reinforce, Actor-critic and PPO—and to address the sim-to-real gap by using domain randomization techniques. Additionaly, to boost robustness we adopted two complementary strategies: an adversarial curriculum that progressively exposes the agent's actions to stronger, targeted perturbations, and DORAEMON's dynamics curriculum, which gradually expands the domain-randomization ranges to familiarize the agent with real-world variability. Although these two curricula did not deliver optimal results, they nevertheless led to measurable improvements in the agent's stability and adaptability, validating their value as advanced training techniques.

1. Introduction

The idea that behavior can be learned through interaction with the surrounding environment has been extensively studied, both in anthropology and artificial intelligence. As emphasized by [9], learning from errors is a well-known and effective human strategy. Applied to robotics, this paradigm allows agents to improve performance through experience. Reinforcement Learning (RL) builds on this concept: learning through interaction and environmental feedback. Its goal is to find optimal strategies, known as policies, that an agent must learn in order to reach a specific objective by maximizing a reward. Despite its success in simulation, applying RL to real-world robots is hindered by the *sim-to-real gap*, the mismatch between simulated and physical environments. A common solution is *domain randomization* (DR), which injects variability into the simulation to train more robust policies, but it requires a careful balance: excessive variability can impede learning, while insufficient variability may compromise generalization.

To enable the agent to adapt to passive, structured variations in environment dynamics, specifically in link masses, beyond standard DR, we investigated a *Curriculum Domain Randomization* method. This approach assumes that increased diversity in the sampled dynamics parameters enhances generalization. Actively induced perturbations, such as real-world attacks, hardware malfunctions, or noisy actuators, were addressed through *Curriculum Adversarial Training* techniques. In this framework, an external adversary perturbs the agent's actions to induce suboptimal decisions and promote more robust behavior.

This project explores the aforementioned RL techniques for robotic control in the Hopper environment (MuJoCo suite). The following sections present the methodology, results, and analysis of these strategies' effectiveness in improving transferability and the source code can be found on GitHub.

2. Related Works

Various Sim-to-Real (Sim2Real) transfer strategies have been proposed to address the reality gap, with domain randomization [11] and domain adaptation [5] being among the most widely adopted techniques. However, they often lead to sample inefficiency or suboptimal policies when transferred to real-world tasks.

Curriculum learning has emerged as a promising alternative to improve both the stability and transferability of RL agents. By gradually increasing task complexity, agents can acquire more generalized skills. Approaches such as Automatic Domain Randomization (ADR) [1] automate the curriculum design process, adapting the environment's difficulty based on agent performance. However, these methods are typically environment-agnostic and may not fully exploit the dynamics-specific properties of robotic systems.

Recent research has investigated *Curriculum based domain randomization* [2] [6], and *Curriculum Adversarial learning*, that is used to expose policies to perturbations that challenge their robustness [8] [10]. Our work builds upon these ideas to design reinforcement learning pipelines that encourages generalization and solidness.

3. Methodology

3.1. Environment

The Hopper environment, part of the MuJoCo suite in Gymnasium, is a standard benchmark for testing locomotion and balance in reinforcement learning. It simulates a planar, single-legged robot tasked with hopping forward as fast as possible without falling. The model is made of four body parts with specified masses: the torso, the thigh, the leg and a single foot on which the entire body rests.
The action space is continuous and represented by three-dimensional vectors, where each action corresponds to the torque applied at the robot's hinge joints (hip, knee, and ankle). Each component is bounded in the normalized torque limits $[-1, +1]$. The 11-dimensional state space is also continuous, consisting of the robot's joint angles, velocities, and positional information, ranging between $[-\infty, +\infty]$. To simulate the sim-to-real transfer scenario, we used two distinct domains: the source domain, which is a modified version of the default Hopper environment with the torso mass reduced to the 70% of its original value, and a target domain, corresponding to the default environment.

3.2. Learning Algorithms and Implementation Choices

We implemented and evaluated three policy gradient algorithms: REINFORCE (REI), Actor-Critic (AC), and Proximal Policy Optimization (PPO) [7]. The implementations are inspired by theoretical foundations in [9] and the robotics-oriented surveys [3, 4].

REINFORCE. At each timestep within an episode, the loss is computed as:

$$\text{loss}_{step} = -\log \pi(a_t|s_t) \cdot (G_t - b)$$

where G_t is the return computed as the full discounted sum of rewards from step t onward, using a discount factor of $\gamma = 0.99$. The term b denotes a fixed baseline. At the end of each training episode, the optimization step is performed using the arithmetic mean of the timestep losses across the episode. Multiple values of b were evaluated, including $b = 0$, and performance was averaged over 10 different random seeds for each setting.
As the use of a fixed baseline b has shown suboptimal performance, as further explored in subsection 4.1, we implemented a batch REINFORCE variant, with a dynamically computed baseline. In this setup loss is updated after a batch of 10 episodes, during which actions are sampled and trajectories are collected without modifying the policy parameters. At the end of the batch, the update step is performed using the average or the sum of the losses computed across all episodes in the batch. A key component of our work is the use of batch-dependent baselines: for each timestep t, the baseline is calculated as the average of the corresponding G_t values across all 10 episodes. This results in a baseline that adapts both across time steps and across training progress. This method is consistent with findings from literature, where it has been shown that the optimal baseline in REINFORCE corresponds to the expected return itself [9]. During training, we observed that the use of the following techniques brought a consistent improvement in stability:

- **Return normalization:** Returns G_t are normalized (zero mean, unit variance) before computing the loss;
- **Gradient clipping:** To prevent exploding gradients and enhance convergence, gradients were clipped using a max-norm threshold of 1 or 10 (in batch REI)

Both techniques were consistently applied in all REI experiments, with return normalization used specifically in the non-batch setting.

Actor-Critic. Unlike REINFORCE, which performs policy updates only at the end of each episode using Monte Carlo returns, the Actor-Critic method updates both the policy and the value function at every timestep.
Our implementation follows the standard approach, as we used a learned state-value function $V(s_t)$ to compute the advantage estimate:

$$A_t = G_t + \gamma \cdot V(s_{t+1}) - V(s_t)$$

where G_t denotes the empirical return and $\gamma = 0.99$ is the discount factor. The policy (actor) is optimized by minimizing the following loss:

$$\text{loss}_{\text{actor}} = -\text{I} \cdot \log \pi(a_t|s_t) \cdot A_t$$

where $\text{I} = \gamma^t$, the discount factor to the power of the current timestep. To train the critic we minimized the squared advantage estimate:

$$\text{loss}_{\text{critic}} = \frac{1}{2} A_t^2$$

The overall loss is computed as a weighted sum of actor and critic objectives:

$$\text{loss}_{\text{total}} = \alpha \cdot \text{loss}_{\text{actor}} + (1 - \alpha) \cdot \text{loss}_{\text{critic}}$$

where α is a tunable hyperparameter tested with the values $\{0.25, 0.5, 0.75\}$.

Implementation Notes. For both REI and AC, the policy (and the value function) is modeled as a feedforward neural network with two hidden layers of 64 neurons each and `tanh` activations. The policy output layer returns the mean of a Gaussian distribution for each action dimension (three in total). Actions are sampled from the resulting distribution

during training; at test time, the mean of the distribution is used for deterministic behavior.
The policies were updated using the Adam optimizer with a starting learning rate of 10^{-3}, except for AC which used 10^{-4}, and early stopping was applied to prevent overfitting. Plots are built with values sampled every 75 episodes.

Proximal Policy Optimization (PPO). We also used PPO [7], a state-of-the-art actor-critic algorithm that stabilizes policy updates with the use of clipping. Rather than implementing it from scratch, we used the implementation provided by the `Stable-Baselines3` library[1], adapting it to our environment and training setup. As in the AC implementation, the advantage function is used to guide the policy update. PPO modifies the actor loss by introducing a clipped objective:

$$\text{loss}_{\text{actor}} = -\mathbb{E}_t \left[\min \left(r_t A_t, \text{clip}(r_t, 1-\epsilon, 1+\epsilon) A_t \right) \right]$$

where $r_t = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$ is the probability (likelihood) ratio between the updated policy π_θ and the "old" policy $\pi_{\theta_{\text{old}}}$, A_t is the estimated advantage term, and ϵ is a hyperparameter controlling the clipping range (typically set to 0.2). This method is used to prevent overly large policy updates, that could cause instability or the algorithm to miss the optimum. The critic is trained by regression on mean squared error:

$$\text{loss}_{\text{critic}} = \mathbb{E}_t \left[(V(s_t) - G_t)^2 \right]$$

We trained the models with 200,000 steps and tested them both in the source and target domains. Hyperparameter tuning has been done using grid search, selecting only two candidate values for learning rate, number of training epochs, number of steps and batch size, due to computational limitations. For each hyperparameter configuration we repeated the experiment using 5 different seeds, both with and without domain randomization. The models are then tested with the respective seeds on 50 episodes each, and the average return, standard deviation, and average training time are computed across the episodes. Finally, we computed the average of those metrics across the five runs to evaluate the performance of each configuration.

3.3. Uniform Domain Randomization

According to UDR, during each training episode, a different set of simulation parameters is sampled from predefined distributions, forcing the agent to develop robust policies that do not overfit to a single, fixed environment model.
We implemented UDR in our custom Hopper environment, randomizing the masses of the thigh, leg, and foot by sampling multiplicative factors from a uniform distribution over $[0.5, 1.5]$. We then evaluated a trained PPO model on the target environment, which featured a torso mass configuration not encountered during training. This set up not only improves robustness and adaptability during training, but also enables policy evaluation in a slightly different, more realistic deployment setting.
Furthermore, we explored a more advanced DR technique called Doraemon, details of which are presented in the following sections.

3.4. Robustness-focused advanced techniques

DORAEMON. DOmain RAndomization via Entropy MaximizatiON directly maximizes the entropy of the simulation-parameter distribution during training to promote greater diversity in sampled environments. Crucially, entropy growth is constrained by the current policy's success probability, ensuring stable convergence and preventing performance collapse from excessive randomization. [2]
The algorithm proceeds as follows:

1. Train the RL agent (PPO in our case) by sampling dynamics parameters from the current distribution ν_ϕ (beta distribution in our case), by collecting trajectories $\tau = (s_0, a_0, r_0, s_1, \ldots, s_T)$, and by recording the cumulative return at the end of each episode.
2. Update the parameter distribution by solving:

   $$\max_{\phi_{i+1}} \mathcal{H}\big(\nu_{\phi_{i+1}}\big)$$
   $$\text{s.t.} \quad G(\theta_i, \phi_{i+1}) \geq \alpha, \quad D_{\text{KL}}\big(\nu_{\phi_{i+1}} \| \nu_{\phi_i}\big) \leq \varepsilon,$$

   where θ_i are the parameters of the policy trained under ν_{ϕ_i}; $D_{\text{KL}}(\cdot\|\cdot)$ is the Kullback–Leibler divergence limiting rapid updates of ν_ϕ; $\alpha \in [0, 1]$ is the minimum desired success rate; $G(\theta_i, \phi_{i+1})$ is the success probability estimated via Importance Sampling as

   $$\hat{G}(\theta_i, \phi_i, \phi_{i+1}) = \frac{1}{K} \sum_{k=1}^{K} \frac{\nu_{\phi_{i+1}}(\xi_k)}{\nu_{\phi_i}(\xi_k)} \mathbf{1}_{\{\sigma(\tau_k)=1\}},$$

   using samples $\{(\xi_k, \tau_k)\}_{k=1}^K$ collected under ν_{ϕ_i}. If $\hat{G}$ violates the success constraint, we solve the "backup" problem:

   $$\phi_i^B = \arg\max_{\phi'} \hat{G}(\theta_i, \phi_i, \phi') \quad \text{s.t.} \quad D_{\text{KL}}\big(\nu_{\phi'} \| \nu_{\phi_i}\big) \leq \varepsilon.$$

3. Repeat until the fixed number of iterations is reached.

The algorithm was implemented on the same custom Hopper environment (source) used in our previous experiments, with the code adapted to support sampling multiplicative mass factors from a Beta distribution. We present below the hyperparameters we explored, and the values tested:

- Minimum success probability $\alpha \in \{0.35, 0.5, 0.65\}$;
- Return threshold fixed at 500;

[1] https://github.com/DLR-RM/stable-baselines3

- KL-divergence upper bound ϵ fixed at 2
- The initial Beta distribution parameters were set to $(a, b) = (20, 20)$ for all three link masses, which corresponds to a low-entropy starting distribution to allow the algorithm to gradually increase it;
- Beta transformation: sampled values are linearly remapped to lie in $[0.5,\ 1.5]$.

The tests were conducted with a total budget of 1,500,000 timesteps, where each PPO training run used 150,000 timesteps, resulting in 10 distribution updates. The metrics reported at the end of each iteration were: entropy of the current distribution, KL-divergence between the new and the old distribution, (estimated) performance under the current distribution and evaluation over 50 episodes both in the source and target domains.

Curriculum Adversarial Training. Adversarial learning, and in particular Projected Gradient Descent (PGD), enhances the robustness of RL agents by exposing them to adversarial attacks during training [8] [10]. In the considered setting, for each agent action a_t, an adversarial one is created by performing an $\mathcal{N}$-step gradient descent with step size α on the action distribution $\mathcal{D}$, starting from a_t. The resulting perturbation is the ℓ_2 projection of the difference between the two actions onto a ball $B_{\ell_2}(C_t)$ with a controlled radius, named *budget*. Training is carried out over a total of 50,000 episodes, divided into blocks of 10,000 episodes each. Each block is associated with a different perturbation budget, which gradually increases over time. The set of selected budgets, and thus the curriculum itself, is denoted by

$$\mathcal{B} = [b_0, b_1, b_2, b_3, b_4]$$

and the function C_t selects the appropriate budget from $\mathcal{B}$, based on the training block corresponding to the current episode. A pseudo-code illustrating the procedure is provided in Algorithm 1, and we set $\epsilon_\Delta = 10^{-3}$ and $\epsilon_c = 10^{-6}$. For the implementation of this method, we used a Beta distribution with parameters $\alpha_B, \beta_B > 1$ instead of a Gaussian distribution for sampling actions in the output layer of the policy network. This choice was motivated by the observation that the action space of the hopper is bounded within $[-1, 1]^3$, whereas the normal distribution has an unbounded domain $(-\infty, +\infty)$. As a result, sampling from a Gaussian would frequently yield action components outside the valid range, requiring them to be clipped to -1 or 1. This was leading to a limited set of executed actions. The Beta distribution has a domain of $(0, 1)$, which allowed us to sample within this range and then apply a linear transformation to map the outputs to $(-1, 1)$. This approach enabled a more uniform and continuous exploration of the action space and helped prevent adversarial perturbations from being completely nullified due to clipping.

Algorithmus 1 Curriculum Adversarial Training

Initialize: Budget function C_t, budget set $\mathcal{B}$, step size α, number of optimization steps $\mathcal{K}$, termination tolerance ϵ_Δ, beta ends tolerance ϵ_c
for n in $N_{episodes}$ **do**
  for t in $T_{\text{trajectory}}$ **do**
    Compute budget $C_t = b_i$ if $t \in [t_i, t_{i+1}],\ b_i \in \mathcal{B}$
    Sample action $a_t \sim \pi_\theta(a \mid s)$
    $a_k = a_t$
    for k in $\mathcal{K}$ **do**
      Compute grad of action distribution $\nabla a_k \mathcal{D}$
      Get adversarial action $a_{k+1} = a_k - \alpha \nabla a_k \mathcal{D}$
      Clip adversarial action to interval $[\epsilon_c, 1 - \epsilon_c]$
      Compute adv. action step $\Delta = a_{k+1} - a_k$
      if $\Delta < \epsilon_\Delta$ **then** Interrupt update **end if**
    end for
    Compute perturbation $\delta = a_{k+1} - a_t$
    Compute δ_t projecting δ onto the ball $B_{\ell_2}(C_t)$
    Get perturbed action $a_{\text{pert}} = a_t + \delta_t$
    Clip perturbed action to interval $[\epsilon_c, 1 - \epsilon_c]$
    Scale pert. action $a_{\text{pert}} = 2a_{\text{pert}} - 1 \in [-1, +1]$
    Get next state s_{t+1} by taking a_{pert}
  end for
  Optimize parameter θ using $\mathcal{D}$
end for

Experiments were conducted using both batch and non-batch versions of REI and various curricula with adversarial budgets ranging within $[0, 0.4]$. For batch REI, the following variations were explored:

- Episode losses were either summed or averaged across each batch. When losses were summed, gradient clipping was applied with a norm threshold of $cl = 1$ or $cl = 10$. In contrast, when losses were averaged, gradient clipping was applied only with a norm of $cl = 1$.
- We opted to implement dynamic batch sizes: starting with batches of 10 episodes for the first 30,000 episodes, and increasing to batches of 20 episodes for the following 20,000.
- Different numbers of PGD steps were tested, primarily $\mathcal{N} = 1$ (with $\alpha = 1$) and $\mathcal{N} = 10$ (with $\alpha = 10^{-3}$).

Setting $\mathcal{N} = 1$ and $\alpha = 1$ ensured a single gradient-based perturbation within the allowed budget. Given the shape of the Beta distribution (unimodal for parameters > 1), any clipped step in the descent direction lead to a lower function value, making further iterations unnecessary and reducing computational overhead.
All experiments were run on the source environment using 5 different seeds, and each was tested on the source environment with its corresponding seed across 50 episodes, both with and without adversarial attacks.

4. Results

For reproducibility of results, the following seeds were selected: $\{42, 35, 254, 78, 91, 53, 22, 341, 117, 86\}$. Depending on the experiment, a different number of seeds among these ten was used, also due to computational constraints.

4.1. Basic RL: REINFORCE vs Actor-Critic

In this section, we present the results obtained for REINFORCE (with several baseline values) and Actor-Critic, without exploiting any domain randomization algorithm. Both the training and testing of these models were carried out on Source domain.

To compare the performance of the algorithms, we considered the return obtained for each episode during training and testing, as well as the duration (time) of each episode for training. These values allowed us to cross-check the evolution of the leg's behavior quality throughout training. Another metric we could have considered was the computational time required to train the models. However, a consistent comparison between the times of different algorithms couldn't be made, as different hardware were used.

Regarding REI, among the baseline values used, we explored more thoroughly the cases equal to 0 (no-baseline) and 50, as the latter appears to exhibit slightly more stable behavior compared to the remaining values.

Figure 1 reveals that, contrary to expectations, the algorithm without a baseline performed better and reached convergence earlier. Nevertheless, as expected, the variance observed during the execution of the algorithm with a baseline (light blue oscillations in the first graph) was lower compared to the variance seen without a baseline (light green).

In the second graph of Figure 1 it is possible to see how the episode times follow the trend of the rewards. Especially with baseline 50, early episodes were very short because the leg quickly fell, as confirmed by the low rewards in the first graph. Over time, both rewards and durations increased, reflecting improved stability and behavior.

In general, the presence of a fixed baseline proves to be inefficient, likely due to the fact that it is not a good estimate of the expected reward at each step.

Actor-Critic, as expected, achieves better results than REI in terms of return per episode (Figure 2). It also seems to reach convergence slightly earlier.

This may be due to the fact that the AC does not wait until the end of each episode to compute G_t, and thus fully exploits each intermediate step, learning from data much more frequently. Moreover, the presence of the Critic, acting as a baseline for the Actor (subsection 3.2), leads to slightly more stable policy updates, as shown in the testing Table 1. At the same time, the table shows how the choice of the hyperparameter α, despite having little impact during training, is crucial during testing, highlighting differences in terms of the mean and standard deviation of the return.

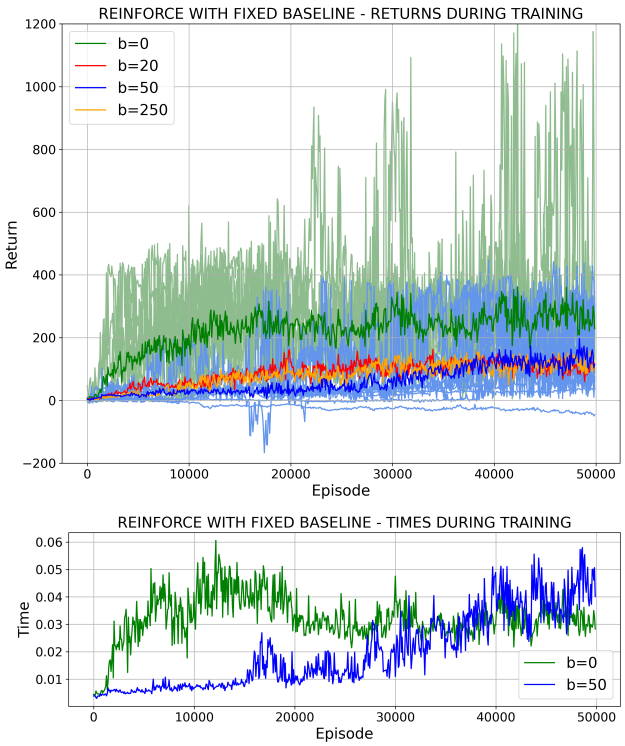

Figure 1. Episode return and duration for REI with fixed baselines, averaged over 10 training runs with different seeds per baseline. In the return plot, light green and light blue lines represent individual seed trajectories for $b=0$ and $b=50$, respectively.

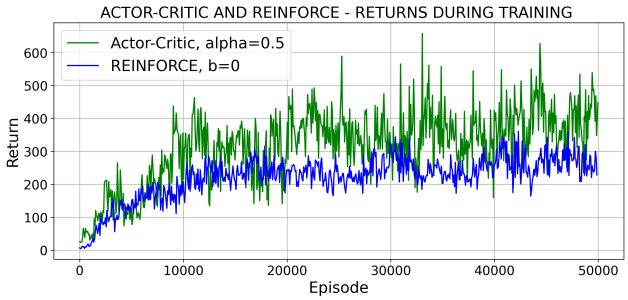

Figure 2. Episode return of AC with $\alpha = 0.5$ and of REI without baseline, averaged over 10 training runs with different seeds.

Table 1. Mean and standard deviation of returns over 50 test episodes on the source environment, averaged by model type (over 10 seeds per REI and 4 per AC algorithms).

Algorithm	**Mean**	**Standard deviation**
Reinforce	288.6	78.0
Reinforce b=50	131.9	41.8
Actor-critic $\alpha = 0.25$	417.7	80.5
Actor-critic $\alpha = 0.5$	429.0	48.3
Actor-critic $\alpha = 0.75$	347.4	41.0

4.2. PPO Results

To analyze the effect of domain randomization, we trained and tested our PPO models in four different settings. The best results have been obtained when both the training and the test have been done on the source domain (S-S), as can be expected considering that there is no distribution shift. The configurations with best results on S-S, reported in Table 2, highlight a significant performance drop when models trained on the source domain are tested on the target domain, an effect of the sim2real gap, which has been then reduced by the use of randomization techniques. Despite the excellent performance analyzed in Table 2, we noticed that only half of the studied hyperparameters configurations reported an improvement in return when UDR was applied in the source-to-target setting. Nevertheless, UDR consistently improved performance stability, reducing the standard deviation range from 9.88–218.61 (without UDR) to 8.13–129.33 in the source-to-target setting, suggesting that the overall performance was more stable. In addition, we monitored the training times for each configuration and noticed that, on average, training a PPO model required significantly less time compared to the other algorithms presented in this study, likely due to built-in library optimizations.

Table 2. Performance of two PPO model configurations on test, across different combinations of training and testing environments (Train–Test) and with or without domain randomization.

Config.	Training	Mean	Std	Time (s)
$lr = 3 \cdot 10^{-4}$	T–T, No DR	1421.43	106.76	246.82
$bz = 64$	S–S, No DR	1507.30	17.38	190.87
$n_{epochs} = 15$	S–T, No DR	1127.89	47.78	186.24
$n_{steps} = 2048$	S–T, DR	1269.80	78.39	190.55
$lr = 3 \cdot 10^{-4}$	T–T, No DR	1412.82	17.26	307.40
$bz = 64$	S–S, No DR	1467.03	35.15	175.27
$n_{epochs} = 15$	S–T, No DR	1055.05	30.21	199.88
$n_{steps} = 4096$	S–T, DR	1107.86	20.56	186.47

4.3. Robustness-focused advanced techniques

DORAEMON. All tests yielded results consistent with those in Table 3. As the number of iterations increases, the entropy, initially negative, rises to nearly zero within the first five iterations, after which the distribution stabilized, as evidenced by the vanishingly small KL-divergence values. In fact, the limiting distribution is a Beta with both shape parameters ≈ 1, i.e. a uniform distribution (Beta(1,1), which maximizes entropy).

Regarding estimated performance, we see the metrics climbing to nearly one as early as the second iteration, thanks to the backup optimization that guarantees its maximization and the fact that the minimum performance threshold was set fairly low.

The source and target test scores at the end of each iteration are quite noisy, especially in the early rounds, where distribution shifts are more pronounced. To speed up the process, these initial tests were run using a single seed (equal to 0). Once all iterations were completed, we selected the models with the highest average target-test scores (as those highlighted in bold in the table) and then evaluated their robustness across the ten different seeds introduced in section 4.

Model	Target Mean	Target Std
$\alpha = 0.5$, iter 1	1833.125	396.939
$\alpha = 0.5$, iter 6	1879.034	446.888
$\alpha = 0.65$, iter 3	1526.798	259.988
$\alpha = 0.35$, iter 5	1055.557	33.146

Table 4. Some of the best models for each value of α in DORAEMON: evaluation performance (averaged across 10 different seeds) on target domain with 50 episodes.

The exact results are reported in Table 4, and it can be observed for the cases $\alpha = 0.5$ and $\alpha = 0.65$ an improvement over those obtained with the UDR previously used (see Table 2), thanks largely to Doraemon's ability to explore

Table 3. DORAEMON iteration statistics. Case $\alpha = 0.5$.

Iteration	Entropy	KL	Perf Est	Source Mean	Source Std	Target Mean	Target Std
0	-3.394×10^{0}	—	—	1892.378	305.442	1092.740	67.912
1	-3.402×10^{0}	0.033×10^{0}	0.284	3964.665	2947.731	**1793.181**	277.745
2	-3.200×10^{0}	0.314×10^{0}	0.986	1209.660	253.777	932.147	17.217
3	-2.023×10^{0}	1.849×10^{0}	0.987	1297.124	216.474	1011.852	9.361
4	-1.162×10^{0}	1.907×10^{0}	0.992	1376.835	86.692	669.874	3.791
5	-2.925×10^{-10}	1.162×10^{0}	0.985	1201.618	189.357	1199.526	130.639
6	-8.113×10^{-7}	7.073×10^{-7}	0.997	6182.636	4498.173	**1891.489**	480.321
7	-2.487×10^{-14}	8.368×10^{-7}	0.985	1042.649	18.252	804.582	7.203
8	-8.049×10^{-9}	1.087×10^{-7}	1.000	1026.338	15.831	873.051	21.032
9	-1.755×10^{-5}	1.698×10^{-5}	0.996	926.808	12.671	847.514	39.380
10	-2.842×10^{-13}	1.754×10^{-5}	1.000	1004.429	10.809	722.283	4.645

Table 5. Mean and standard deviation of returns over 50 test episodes, with or without fixed-budget adversarial attacks. Results are averaged over 5 seeds per model–adversarial setting. Only a subset of the tested models is shown in the table.

	Test adversarial	**No Adv.**		**0.005**		**0.01**	
Model		Mean	Std	Mean	Std	Mean	Std
No adversarial		380.2	59.7	395.9	60.1	390.3	66.5
No adversarial, batch avg		**813.2**	84.9	**798.0**	86.4	**771.8**	82.5
No adversarial, dynamic batch sum		646.7	86.2	609.3	69.2	602.3	62.7
Adv., $\mathcal{N}=10$, $\mathcal{B}$=[0.005, 0.01, 0.01, 0.02, 0.05]		72.6	40.3	127.3	21.1	127.5	21.6
Adv., batch avg, $cl=1$, $\mathcal{N}=10$, $\mathcal{B}$=[0.005, 0.01, 0.01, 0.02, 0.05]		431.3	25.5	422.2	22.9	414.5	24.6
Adv., batch sum, $cl=1$, $\mathcal{N}=1$, $\mathcal{B}$=[0.005, 0.008, 0.008, 0.01, 0.01]		**915.9**	161.2	**880.9**	149.6	**846.1**	146.5

a broader range of distributions. Indeed, some of the best results come from models trained in the earliest iterations, i.e., on distributions markedly different from uniform, suggesting a potential direction for exploring new optimal distributions. However, in the case of $\alpha = 0.35$, even the top-performing models don't match the results of UDR, indicating that being too permissive with the performance threshold prevents the algorithm from exploring in the right direction.

Curriculum Adversarial Training. As we will see, Adversarial training led to performance improvements over REI models. Firstly, using batch-based updates it was possible to significantly improve the learning stability and reduce the variance of the expected function, as already presented by [12]. Moreover, performance was further improved by the use of dynamic baselines, as can be seen in Table 5 (here, models reported with "batch" are trained with both batch updates and dynamic baseline). The experiments with dynamic batch sizes, as shown in Figure 3, decreased the variance but slightly compromised performance.

From in-depth tests, not reported in the table, it emerged that performances with batch variants using summed or averaged episode losses do not differ significantly, nor do those with gradient clipping set to 1 or 10. This suggests that the use of an adaptive optimizer (such as Adam) was effective in compensating differences in gradient magnitude. While also $\mathcal{N}$ had little impact on returns, significant differences in training time were observed as expected, from about 5 hours when $\mathcal{N}=10$, to about 2.5 hours when $\mathcal{N}=1$. The use of Adversarial training with a curriculum of increasing perturbation budgets has shown promising results. However, we noticed a strong dependence of the performance on the parameters used: as shown in Figure 3, the use of curricula reaching a budget of 0.05 in the final block of the training tend to be overly aggressive, causing an evident decline in performance after the first phase of alignment with other models. Instead, using a curriculum defined by $\mathcal{B} = [0.005, 0.008, 0.008, 0.01, 0.01]$ (purple line in the graph), yields to performance similar to the model trained without adversarial perturbations (always using batch updates), achieving higher average returns in testing, both with and without adversarial attacks. The test results with this configuration have a clear higher variability, indicating less consistent performance across tests with same seed. However, the high standard deviation of returns observed in tests with the same seed is offset by low variability across different seeds. As shown in Figure 4, models trained with adversarial perturbations exhibit unimodal distributions centered around 800, indicating consistent performance. In contrast, models trained without adversarial show bimodal distributions, suggesting that different seeds led to significantly divergent levels of policy training.

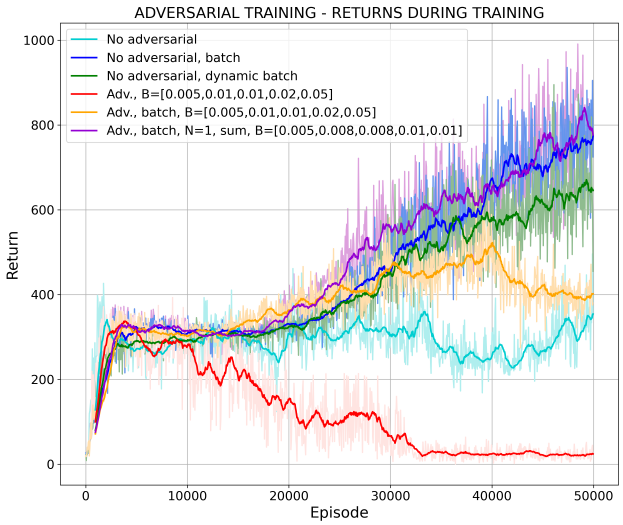

Figure 3. Training return trajectories of the models listed in Table 5, averaged over 5 seeds (in light). Values sampled every 50 episodes. Moving average over 20 values (in dark).

It is also noticeable from the bold models in Table 5 that introducing a fixed adversarial budget of 0.01 during testing led to a performance drop of 5.1% in average returns without adversarial training, and a more pronounced 7.8% decrease when training with the adversarial curriculum, both relative to testing without adversarial attacks. This shows that, despite the good results obtained with adversarial, it

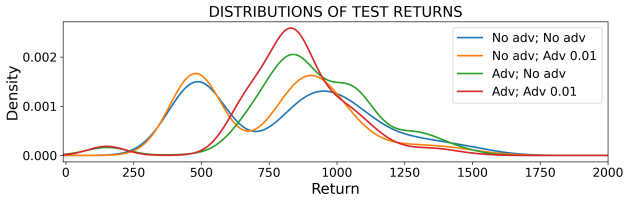

Figure 4. Return distributions on the test set for all 50 evaluations of the models highlighted in bold in Table 5, across all 5 seeds, both with and without perturbation (budget 0.01) during testing.

does not contribute positively to accustom the agent to external attacks during the testing phase, probably due to too small perturbations. Indeed, using more aggressive perturbations like the one previously discussed, the percentage drop in average return only reach 3.9%, and standard deviations are way less pronounced.
These results highlight the delicate balance in using an adversarial curriculum with a dynamic budget: while gradually increasing the budget can improve robustness to attacks at test time, overly aggressive training perturbations may inhibit effective learning. In our work, no model achieved a perfect balance between these two aspects.

5. Conclusion

In this work, we investigated a range of RL strategies to bridge the sim-to-real gap for a one-legged Hopper robot.
Concerning the three standard algorithms, we obtained results fairly in line with theoretical expectations, finding that Actor-Critic outperforms REINFORCE, and that PPO with domain randomization yields even better results with greater robustness. The DORAEMON approach has shown successful expanding domain exploration behavior without sacrificing stability. Models trained under it achieved up to a 25% increase in target-domain performance over UDR, highlighting the power of adaptive distribution shaping even with limited computing resources. In parallel, our Curriculum Adversarial Training framework improved performance, but its effectiveness strongly depended on the perturbation schedule and no good balance has been found.
Despite these promising findings, our experiments were constrained by computational resources. We were unable to apply DORAEMON on REINFORCE, to jointly combine adversarial and entropy-based curricula: an integration that could yield even more robust policies. Moreover, many learning curves were still rising at the end of training, suggesting that longer training horizons would further clarify convergence behavior and optimal hyperparameter settings. More reliable results could also be gained from broader parameter tuning and a larger number of seeds for averaging.
Overall, our comparative study shows that adaptive curricula in reinforcement learning for robotics can lead to improvements. However, it also emphasizes the need to carefully balance exploration, stability, and computational efficiency. These findings can help guide future research toward developing general, reliable policies that can be easily transferred from simulation to real-world applications.

References

[1] Ilge Akkaya, Marcin Andrychowicz, Maciek Chociej, Mateusz Litwin, Bob McGrew, Arthur Petron, Alex Paino, Matthias Plappert, Glenn Powell, Raphael Ribas, et al. Solving rubik's cube with a robot hand. *arXiv preprint arXiv:1910.07113*, 2019. 1

[2] G. Tiboni P. Klink J. Peters T. Tommasi C. D'Eramo and G. Chalvatzaki. Domain randomization via entropy maximization. *ICLR 2024*, 2024. 1, 3

[3] Jens Kober, J Andrew Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32(11):1238–1274, 2013. 2

[4] Petar Kormushev, Sylvain Calinon, and Darwin G Caldwell. Reinforcement learning in robotics: Applications and real-world challenges. *Robotics*, 2(3):122–148, 2013. 2

[5] Xue Bin Peng, Marcin Andrychowicz, Wojciech Zaremba, and Pieter Abbeel. Sim-to-real transfer of robotic control with dynamics randomization. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3803–3810. IEEE, 2018. 1

[6] A. Aflakian J. Hathaway R. Stolkin A. Rastegarpanah. Robust contact-rich task learning with reinforcement learning and curriculum-based domain randomization. *IEE Access*, 2024. 1

[7] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017. 2, 3

[8] Junru Sheng, Peng Zhai, Zhiyan Dong, Xiaoyang Kang, Chixiao Chen, and Lihua Zhang. Curriculum adversarial training for robust reinforcement learning. In *2022 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2022. 1, 4

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

[10] Kai Liang Tan, Yasaman Esfandiari, Xian Yeow Lee, Soumik Sarkar, et al. Robustifying reinforcement learning agents via action space adversarial training. In *2020 American control conference (ACC)*, pages 3959–3964. IEEE, 2020. 1, 4

[11] Josh Tobin, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and Pieter Abbeel. Domain randomization for transferring deep neural networks from simulation to the real world. *arXiv preprint arXiv:1703.06907*, 2017. 1

[12] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8:229–256, 1992. 7