Reinforcement Learning for Sim-to-Real transfer

Abdessamed Qchohi
Politecnico di Torino
s328834@studenti.polito.it

Alessio Giuffrida
Politecnico di Torino
s328964@studenti.polito.it

Francesco Giannuzzo
Politecnico di Torino
s328830@studenti.polito.it

Abstract

The main objective of this report is to present the paradigm of Reinforcement Learning in the context of robotic systems. The main focus is the Sim-to-Real transfer problem for the Hopper environment: we illustrate several policy-gradient algorithms and compare their performances, concluding with the implementation of Automatic Domain Randomization (ADR), an advanced approach that allows, using a training curriculum, to gradually expand a distribution over environments for which the model can perform well.

1. Introduction

In this section we describe the main theoretical concepts of the Reinforcement Learning paradigm, along with an introduction of the Sim-to-Real transfer problem in the context of robotics. The code of the implementation of the various algorithms can be found at the project repository [3].

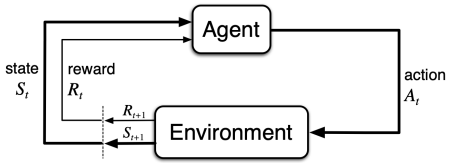

Figure 1. The agent–environment interaction

1.1. Reinforcement Learning paradigm

The Reinforcement Learning framework is based on the idea that the learner, called *agent*, interacts continuously with the external world, called *environment*, and through these interactions it learns from experience. The agent collects an observation of the current *state* of the environment and performs an *action* on it; the environment, in turn, presents to the agent a new state and a reward corresponding to the action. The main goal of the agent is to maximize the rewards obtained from the environment: these rewards are usually scalar values and they guide the agent's choice of actions. The strategy used to decide the actions is called *policy* and it depends on the current state. Fig. 1 illustrates the schema of this paradigm. The elements of the Reinforcement Learning problem can be formalized using the Markov Decision Process (MDP) framework: we define an action space A, a state space S and a reward function R, from which follows the probability distribution over state distribution, called MDP dynamics:

$$p(s', r|s, a) = Pr\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\} \tag{1}$$

This probability distribution is not known by the agent, which only observes realizations of it. We then introduce the concept of *trajectory* τ, that is a sequence of states, actions and rewards with finite length T: a trajectory that reaches a terminal state is called *episode*. For a trajectory we are interested in computing the *discounted return*, which is the cumulative reward obtained by following the trajectory τ:

$$G_t = \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k \tag{2}$$

The parameter $\gamma \in [0, 1]$ is called *discount rate* and determines the present value of future rewards, so how much the agent cares about rewards in the distant future relative to those in the immediate future. Actions are chosen according to a policy $\pi(a|s)$, which is a mapping from states to probabilities of selecting each possible action. The state-value function $v_\pi(s)$ of a state s under a policy π is the expected return obtained by starting in s and following π. The policy performance measure $J(\theta)$ with respect to the parameters θ of the policy π_θ is the expected return starting from the initial state s_0 and following the policy π_θ. It can be verified that $J(\theta) = v_\pi(s_0)$. In this report we analyse several policy-gradient methods, whose goal is to maximize the expected return $J(\theta)$: the core idea is to compute the gradient of the objective function with respect to the policy parameters and adjust the parameters in the direction of this gradient, approximating gradient ascent:

$$\theta_{t+1} = \theta_t + \alpha \widehat{\nabla J(\theta)} \tag{3}$$

Estimating the performance gradient with respect to the policy parameters proves to be challenging, considering that the gradient depends on the unknown effect of policy changes on the state distribution. However, this problem can be resolved thanks to the the *Policy Gradient Theorem*:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) G_t] \tag{4}$$

which provides an analytic expression for the gradient of the performance with respect to the policy parameters that does not involve the derivative of the state distribution.

1.2. RL for robotics and Sim-to-Real transfer

Reinforcement learning offers to robotics a framework for the design of complex behaviors: RL enables a robot to autonomously discover an optimal behavior through trial-and-error interactions with the environment. The main challenge of this approach is that the training process could be very long and expensive in a real world setup and could also lead to dangerous outcomes. A possible solution is to carry out the training using a simulator, in which we build an approximate model of the environment dynamics. The discrepancy between the real world dynamics and the simulation is called *reality gap*. The downside of this approach is that the model learns only the approximate dynamics of the simulator and therefore performs poorly in the real environment. The challenge of correctly transferring the experience from the simulation to the real world is called *Sim-to-Real transfer* and its goal is to train policies that are able to adapt to the real environment even if they were not trained on it. In this context, we will focus on *Domain Randomization* (DR), a technique that allows to train models that can generalize well to real-world environments by exposing them to a wide variety of simulated conditions, by randomizing the environment's parameters. In the following we will illustrate two different versions of this approach: Uniform DR and Automatic DR. For the sake of practicality, we did not transfer to the real world, but used a different environment, performing a Sim-to-Sim transfer: the *source environment* is the one in which training is performed, while the *target environment* is the one to which we want to transfer the policy.

1.3. Simulation setup

The environment on which simulation is performed is the *Hopper* environment of the Gym API [2], which consists of a one-legged robot composed of torso, thigh, leg and foot, as shown in Fig. 2. The goal of the Hopper is to learn how to make hops in the forward direction as fast as possible, without falling down. The movement is obtained by applying

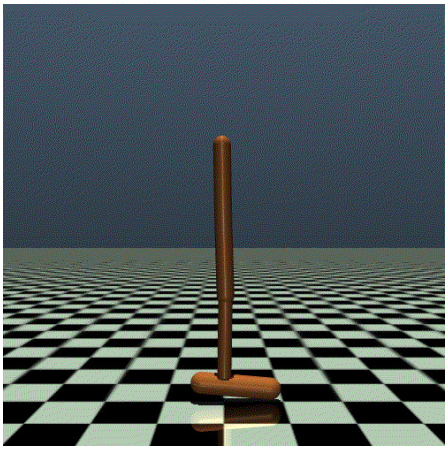

Figure 2. The Hopper environment

torques on the three hinges connecting the body parts. The hopper is considered *dead* when it falls down or its configuration does not allow it to remain in balance; otherwise, it is considered *alive*. Some setup considerations:

- For the source environment, the body parts of the robot have the following masses: 2.53 kg for torso, 3.93 kg for thigh, 2.71 kg for leg and 5.09 kg for foot. The target environment, instead, differs only for the mass of the torso, which is 3.53 kg;
- The state space is composed by the different characteristics observed by the agent at each state (coordinates, angles of joints, velocities), whereas the action space is represented by the set of torques applied between the links;
- The reward is composed of 3 parts:
  1. Alive Bonus: a reward of +1 for each time step of life;
  2. Reward Forward: a reward of hopping forward measured as $(x_{t+1} - x_t)/\Delta t$, where Δt is the duration of the action in time steps;
  3. Control cost: a cost for penalising the hopper if it takes actions that are too large;

2. Reinforcement Learning Algorithms

2.1. REINFORCE

REINFORCE is a policy gradient algorithm which exploits an estimation of the episode return to compute the policy gradient, as a result of the general Policy Gradient Theorem Eq. (4). Rewards are received at different points in time, so the discounted rewards formulation Eq. (2) allows us to take into account the variation of rewards over time. The policy is parameterized by a simple Multi-Layer

Perceptron, composed by three layers and using *tanh* as activation function. REINFORCE algorithm follows in Fig. 3.

Algorithm REINFORCE

Input A differentiable policy parameterization $\pi_{\boldsymbol{\theta}}(A|S)$
Algorithm parameter: Step size $\alpha > 0$
Initialize The policy parameters $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ at random.
for Loop forever (for each episode): **do**
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi_{\boldsymbol{\theta}}$
  $G_t \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$
  $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \cdot \sum \gamma^t G_t \nabla \ln \pi_{\boldsymbol{\theta}}(A_t|S_t)$
end for

Figure 3. REINFORCE Algorithm

Subtracting a *baseline* from the return G_t leads to a reduction in the variance of the policy gradient estimates without changing their expected value: the baseline may be equal to a constant value or to the value function (which is the best possible baseline as it minimizes the variance of the gradient estimates). A high variance in gradient estimates can cause large, erratic updates to the policy parameters, potentially leading to unstable training dynamics. Alternatively, it is possible to scale the returns using the mean and standard deviation: this is a *whitening transformation*, where $G^* = \frac{G - G_{\text{mean}}}{G_{\text{std}}}$. We tried three different approaches: no baseline, baseline equal to 20 and whitening transformation. We then analyzed the performances in terms of reward and runtime on the source environment for each policy for 10000 episodes and evaluated them on the target environment for 500 episodes, with *learning rate* = 1e−3 and $\gamma = 0.99$. The loss function that we want to minimize is $-J(\theta)$, which is equivalent to maximizing the expected return. The training curves can be observed are in Fig. 4.

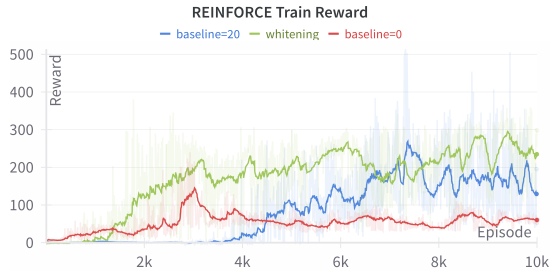

Figure 4. Train Reward for REINFORCE with different *baseline* values and whitening approach.

From Fig. 4, we can observe that during the training the presence of a baseline leads to a better policy. Moreover, for REINFORCE with baseline and with whitening the training time is higher, as it is shown in Tab. 1 correlated to test results in Tab. 2.

Approach	Training Time
No baseline	918.943 s ∼ 15 minutes
Baseline = 20	1795.759 s ∼ 29 minutes
Whitening	2637.899 s ∼ 43 minutes

Table 1. Training time for different REINFORCE approaches

Approach	Mean Reward
No baseline	48.994
Baseline = 20	147.949
Whitening	242.786

Table 2. Test results of REINFORCE

2.2. Actor Critic

Actor Critic methods combine the advantages of value-based and policy gradient methods. They use the state-value function of π:

$$v : S \rightarrow R$$

$$(s_t) \rightarrow \mathbb{E}_{p,\pi}[G_t]$$

Algorithm ACTOR-CRITIC

Input A differentiable policy parameterization $\pi_{\boldsymbol{\theta}}(A|S)$
Input A differentiable state-value function $\hat{V}_{\boldsymbol{w}}(S)$
Parameters: Step sizes $\alpha^{\boldsymbol{\theta}} > 0, \alpha^{\boldsymbol{w}} > 0$
Initialize The policy parameters $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\boldsymbol{w} \in \mathbb{R}^{d}$
for Loop forever (for each episode): **do**
  Initialize S_0 (first state of episode)
  $I \leftarrow 1$
  while S_t is not terminal (for each time step t) **do**
    $A \sim \pi_{\boldsymbol{\theta}}(\cdot|S)$
    Take action A, observe S_{t+1}, R
    $\delta \leftarrow R + \gamma \hat{V}_{\boldsymbol{w}}(S_{t+1}) - \hat{V}_{\boldsymbol{w}}(S_t)$
    $\boldsymbol{w} \leftarrow \boldsymbol{w} \alpha^{\boldsymbol{w}} \delta \nabla \hat{V}_{\boldsymbol{w}}(S_t)$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} I \delta \nabla \ln \pi_{\boldsymbol{\theta}}(A|S_t)$
    $I \leftarrow \gamma I$
    $S_t \leftarrow S_{t+1}$
  end while
end for

Figure 5. Actor Critic Algorithm

The approximation of the state-value function is used to compute a *bootstrapped estimation* of the one-step return $G_{t:t+1}$:

$$G_{t:t+1} \rightarrow r_t + \gamma v(s_{t+1})$$

Despite the presence of some bias in $\nabla_\theta J(\theta)$, using this algorithm we have less variance during training. Moreover, following the gradient allows to bootstrap and not to wait for the full trajectory τ. We use a separate neural network (whose structure is identical to the one used for the policy) in order to estimate the state-value function v: this is the critic part of the method. The policy network assumes the

role of the actor, responsible of updating the policy distribution in the direction suggested by the critic. The loss function used for the actor part is the same as REINFORCE, so $-J(\theta)$, while the loss function of the critic part is the MSE between the value function computed on the current state and the bootstrapped estimation of the one-step return. We trained on the source environment for 10000 episodes using the same learning rate and γ values of REINFORCE (Fig. 6).

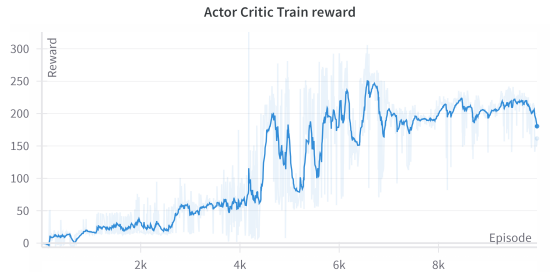

Figure 6. Training Reward for Advantage Actor critic

The time consumption of the Actor Critic training is 2387.095 s ($\sim$ 40 minutes). We obtained a mean reward of 185.611 over 500 test episodes on the target environment. Comparing the training curves of actor critic and REINFORCE it is noticeable that the actor critic convergence speed and learning stability are slightly better than REINFORCE. The rewards obtained by actor critic are higher than REINFORCE with a baseline of 20, but the whitening transformation helps REINFORCE achieve better results.

2.3. Proximal Policy Optimization

2.3.1 Description

Proximal Policy Optimization (PPO) is a state-of-the-art reinforcement learning algorithm that balances efficiency and performance through clipped probability ratios to ensure stable and reliable policy updates. PPO trains a stochastic policy in an on-policy way: this means that it explores by sampling actions according to the latest version of its stochastic policy. There are two primary variants of PPO: *PPO-Penalty* and *PPO-Clip*:

- PPO-Penalty includes a Kullback-Leibler divergence (which is a measure of distance between distributions) penalty term in the objective function and automatically adjusts the penalty coefficient during training so that it is scaled appropriately.

- PPO-Clip does not have a KL-divergence term in the objective and relies on clipping in the objective function.

To implement the algorithm we used the *Stable Baselines3* (SB3) library [4]. The variant of the algorithm implemented by the library is PPO-Clip, which is based on the following objective function:

$$L(s,a,\theta_k,\theta) = \min(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s,a), \\ \text{clip}\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, 1-\epsilon, 1+\epsilon\right) A^{\pi_{\theta_k}}(s,a)) \tag{5}$$

The objective function takes the minimum of two terms:

- unclipped term: the original advantage multiplied by the probability ratio. This term is used when the policy change is within acceptable limits.

- clipped term: the advantage multiplied by the clipped probability ratio. This term ensures that if the policy changes too much (either increasing or decreasing too drastically), the update is clipped to prevent instability.

2.3.2 Tuning and Training

We selected some hyperparameters that we considered the most relevant for the algorithm and tuned them. Tab. 3 shows the hyperparameters chosen and the intervals from which their values was sampled according to a uniform distribution during the tuning. We trained ten iterations each on the source and target environments for 5 million time steps, obtaining the best configuration of hyperparameters showed in Tab. 4.

Algorithm	Parameter	Values
PPO	learning_rate	{min: 0.0005, max: 0.001}
	clip_range	{min: 0.25, max: 0.35}
	entropy_coefficient	{min: 0.005, max: 0.02}

Table 3. Hyperparameters of Proximal Policy Optimization

Environment	Parameter	Values
Source	learning_rate	0.0005956
	clip_range	0.3342
	entropy_coefficient	0.01653
Target	learning_rate	0.0005013
	clip_range	0.303
	entropy_coefficient	0.00669

Table 4. Best hyperparameters of Proximal Policy Optimization in source and target environments

Fig. 7 shows the training of the two best models obtained for the source and target environments. We can make some considerations about the behaviour of the two curves:

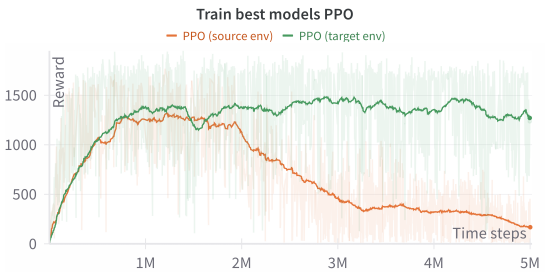

Figure 7. Training Reward of PPO best models for source and target environment

- for the source environment, there is a significant decline of the curve after 1.5 million time steps, possibly due to overfitting.
- for the target environment, the performance is more stable, maintaining higher rewards consistently beyond 1 million time steps.

2.3.3 Results

We evaluated each model and reported their average reward and standard deviation over 50 test episodes for three different configurations, results are in Fig. 8.

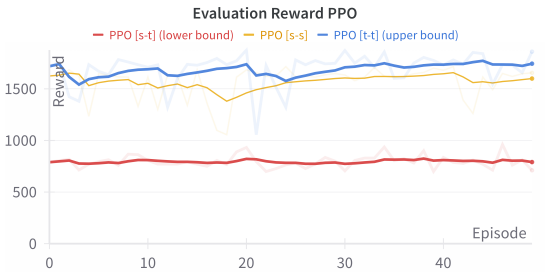

Figure 8. Evaluation of the two best models in different configurations

- source → source : The agent achieves an average reward of 1570.461 ± 155.195.
- source → target (**lower bound**) : The agent achieves an average reward of 795.475 ± 58.711.
- target → target (**upper bound**) : The agent achieves an average reward of 1695.263 ± 156.839.

Comparing source → source and source → target shows that the change of environment for evaluation causes a significant drop in rewards that can be explained by the reality gap, highlighting the challenge of domain transfer in RL. This suggests that policies trained in one environment may not generalize well to another, underlining the importance of advanced domain randomization techniques. The lower bound represents a naive approach to the Sim-to-Real problem, whereas the upper bound is the maximum performance achievable by PPO in the target environment. These two values are then used as a benchmark to evaluate the results of the next two DR approaches.

3. Uniform Domain Randomization

3.1. Description

The ability to generalize is crucial to ensure that the agent can successfully cope with real-world scenarios that may vary in unpredictable ways. One of the main challenges is the discrepancy between the training data (source environment) and the test data (target environment), which can result from variations in environmental conditions, certain parameters and other factors. To address this challenge, it is effective to use DR during the training, where random variations in the parameters of the simulated environment are introduced to increase the diversity of the training data. However, a proper implementation of DR requires a critical selection of the parameters to be varied and an even distribution of these variations to ensure adequate performance.

3.2. Implementation

We exploit *Uniform Domain Randomization* (UDR), based on a uniform distribution that is used to vary the parameters during training. This approach aims to achieve a better coverage of the state space, allowing trained agents to generalise more effectively to new scenarios. Specifically, to approach correctly the source → target scenario, we have taken the optimal hyperparameters given by the tuning of PPO on the source environment in Tab. 4 and searched for the optimal intervals for randomizing the masses of some elements of the agent; after that, we tested the policy on the target environment. Since the two environments differ only for the mass of the *torso*, the parameters to be randomized will be the masses of the remaining parts: *thigh*, *leg* and *foot*. Uniform Domain Randomization associates each body part mass to a uniform distribution $U(a_i, b_i)$, from which their values are sampled at each episode, as show in the algorithm in Fig. 9.

3.3. Tuning and Training

In order to tune the ideal distributions of masses, we define various values of Δ that will be subtracted and added to the default value of each selected body part to compose the bounds (a_i, b_i) of the uniform distributions. We trained for 5 million time steps on the source environment and tested for 50 episodes on the target environment each policy with

Algorithm UDR

Require: $\{m_i\}_{i=1}^{d}$ {Parameters to be randomized}
Require: $\{(a_i, b_i)\}_{i=1}^{d}$ {Distribution bounds}
  repeat
    for $i \in d$ **do**
      $m_i \sim U(a_i, b_i)$
    end for
    Generate Data (m_i)
    Update policy
  until Training is complete

Figure 9. Uniform Domain Randomization Algorithm

different mass distributions, correlated to a specific Δ in Tab. 5.

Algorithm	Parameter	Values
Uniform DR	Δ	$\{0.25, 0.50, 0.75\}$

Table 5. Hyperparameters of UDR

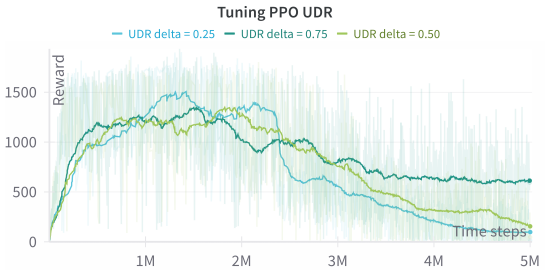

Figure 10. UDR Tuning on Source Environment

From Fig. 10 we expect the policy with $\Delta = 0.75$ to have the best performance in the target environment. An important effect that stands out is that the evolution of the training curves of UDR shows a decrease in the performance as the number of time steps increases. A possible explanation to this behaviour is that sampling the masses from unchanged distributions may lead to dealing always with the same configurations which, as the time steps progress, do not give sufficient stimulus to the agent. The agent will therefore find itself in situations it has already seen and that do not help the learning process. Another possible problem is that after many iterations, the agent might overfit to this broad distribution of environments, leading to a loss of generalization to the specific target environment.

3.4. Results

We tested on the target environment the obtained policies, in order to be able to compare it with the upper bound and lower bound previously found in Sec. 2.3.3, showed in Fig. 11. As expected the policy with the best results in the tuning phase actually performed really close to the optimal target $\rightarrow$ target policy of the PPO and all UDR policies fall within the bounds. This means that UDR is able to overcome the shift in the target environment represented by the different value of the mass of the torso. In Tab. 6 the means and standard deviations of the rewards over of 50 episodes for each UDR policies on the target environment are displayed.

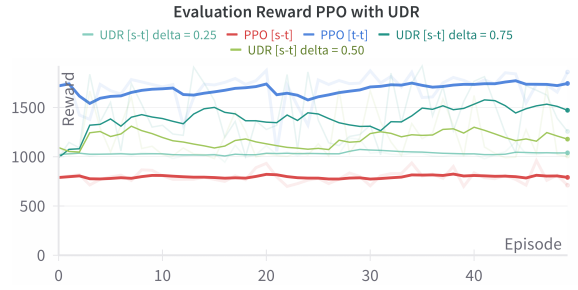

Figure 11. UDR policies tested and compared with PPO evaluation bounds

Δ	Mean Reward	Standard Deviation
0.25	1035.25	43.66
0.50	1183.11	220.88
0.75	1423.09	276.50

Table 6. Evaluation source-to-target of UDR policies

The results are already quite satisfactory, but it is still possible to improve the performance and to achieve a result closer to the optimal target $\rightarrow$ target training curve of PPO. This led us to try a different approach with *Automatic Domain Randomization*.

4. Automatic Domain Randomization

4.1. Description

Automatic Domain Randomization [1] is based one the same concept of Uniform Domain Randomization of randomizing environment parameters. However, unlike UDR, ADR does not maintain fixed bounds for these distributions. Instead, the randomized intervals dynamically increase or decrease based on the agent's performance, functioning in an adversarial manner. This results in a domain randomization technique that can automatically adjust its parameters, thereby enabling faster policy training. The difficulty of the task is modified in response to the agent's performance, facilitating more efficient learning. This is the so-called *curriculum effect*, a systematic approach to progressively expose the agent to increasingly challenging environments.

Algorithm ADR

Require: ϕ^0
Require: $\{D_i^L, D_i^H\}_{i=1}^d$
Require: m, t_L, t_H, where $t_L < t_H$
Require: Δ
  $\phi \leftarrow \phi^0$
  repeat
    $\lambda \sim P_\phi$
    $i \sim U\{1, \ldots, d\}, x \sim U(0, 1)$
    if $x < 0.5$ **then**
      $D_i \leftarrow D_i^L, \lambda_i \leftarrow \phi_i^L$
    else
      $D_i \leftarrow D_i^H, \lambda_i \leftarrow \phi_i^H$
    end if
    $p \leftarrow$ EVALUATEPERFORMANCE(λ)
    $D_i \leftarrow D_i \cup \{p\}$
    if LENGTH(D_i) $\geq m$ **then**
      $\bar{p} \leftarrow$ AVERAGE(D_i)
      CLEAR(D_i)
      if $\bar{p} \geq t_H$ **then**
        $\phi_i \leftarrow \phi_i + \Delta$
      else if $\bar{p} \leq t_L$ **then**
        $\phi_i \leftarrow \phi_i - \Delta$
      end if
    end if
  until training is complete

Figure 12. Automatic Domain Randomization algorithm

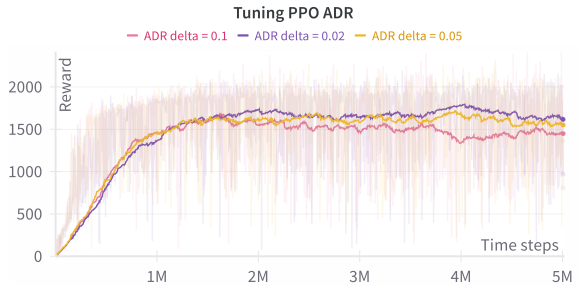

Figure 13. Training of ADR for 3 different values of Δ

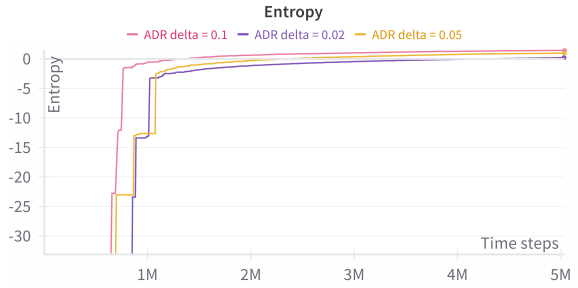

Figure 14. Evolution of ADR entropy

4.2. Implementation

During the training process, the agent is evaluated in each episode with a certain probability p_b. The test results are then used to adjust a specific bound (either lower or upper, Φ_i^L or Φ_i^H) of a given parameter. To test the bound of the i-th parameter, the parameter value is set to that bound (e.g., $\lambda_i = a_i$) and performance data from m episodes are collected in data buffers (one for each bound) while other parameters remain randomized. Once the data buffer for the bound is full, the average performance is compared against predefined thresholds (in our case $t_L = 1000$ and $t_H = 1600$) to determine the necessary update of the parameters' bounds by a fixed value Δ. In our case the parameters to take into consideration are the masses of the thigh, the leg and the foot. To quantify the amount of ADR expansion, we define the concept of *ADR Entropy* as:

$$H(P_\Phi) = \frac{1}{d} \sum_{i=1}^{d} log(\Phi_i^H - \Phi_i^L) \tag{6}$$

The higher is the value of ADR entropy, the broader the randomization sampling distribution.Initially the values of the lower bound and the upper bound are equal to the default value of the mass of each body part, so the difference would be 0 and the theoretical value of the entropy would be $-\infty$: to avoid computational problems we initialize them so they differ by a value equal to 2ϵ, where $\epsilon = 2 \cdot 10^{-15}$.

4.3. Tuning and Training

For the value of the bound increase Δ we tried three different constant values showed in Tab. 7. We trained three different policies for 5 million time steps on the source environment (Fig. 13) and then we tested them on the target environment for 50 episodes. The behaviour of the algorithm can be described by considering the relation between the reward obtained by the policy and the values of the predefined thresholds:

- Initially the rewards obtained are below the low threshold, so the algorithm dictates that the interval defined by the bounds should shrink, but this is not possible since the lower and upper bound cannot overlap, so in this case the bounds are not modified. As a consequence, in the first iterations it is possible to observe that the value of the entropy does not increase.
- After a while, the reward surpasses the low threshold, but not the high threshold and so the bounds are not modified: the entropy does not increase.
- Only when the algorithm rewards start to be higher than the high threshold the interval defined by the bounds starts to expand: this happens around 1 million time steps. After this we can observe a considerable increase of the entropy and the rewards follow the same behaviour, until we reach a plateau in both curves.

Algorithm	Parameter	Values
Automatic DR	Δ	$\{0.02, 0.05, 0.1\}$

Table 7. Hyperparameters of ADR

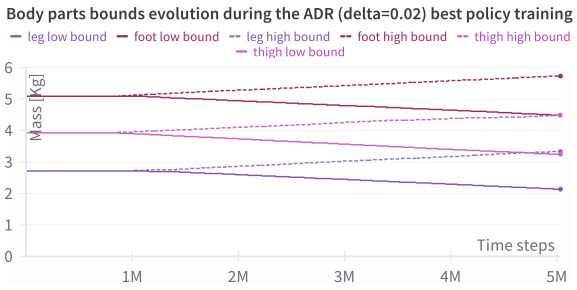

Figure 15. Evolution of the bounds of the masses

The behaviour of the entropy in Fig. 14 is confirmed by the evolution of the bounds of the masses, as it is illustrated in Fig. 15.

4.4. Results

As we can see in Fig. 16, the value of $\Delta = 0.02$ gives the best results. We can observe that the all ADR policies fall within the bounds provided by PPO in Sec. 2.3.3.

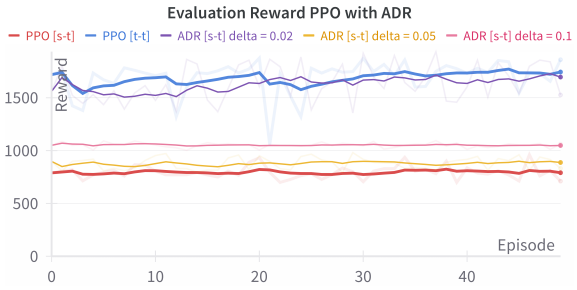

Figure 16. Test rewards of ADR for different values of Δ

Considering that the UDR and ADR implementations were based on the PPO algorithm and that we always used the best configuration of hyperparameters found for PPO, we can make some comparisons between the obtained results: as it is possible to observe from Fig. 17 the performances of both ADR and UDR are below the rewards of PPO in the target-target scenario, but ADR is considerably better than UDR and is closer to the upper bound for the source-to-target scenario. Comparing the training curves of ADR and UDR, we can conclude that ADR solves the limitations of UDR because it allows the policy to gradually adapt to increasingly more complex environments thanks to the curriculum effect.

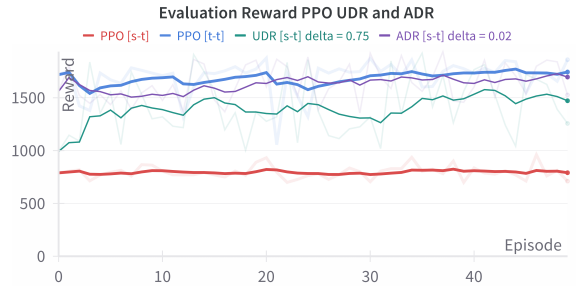

Figure 17. Comparison of test rewards of PPO, UDR and ADR

Δ	Mean Reward	Standard Deviation
0.02	1641.35	170.95
0.05	879.73	55.59
0.1	1051.94	22.26

Table 8. Evaluation source-to-target of ADR policies

5. Conclusion

In this report, we explored several Reinforcement Learning algorithms and observed their limited robustness when adapting to slightly different environments. To tackle the Sim-to-Real problem, we employed Domain Randomization techniques, specifically implementing Uniform Domain Randomization (UDR) and Automatic Domain Randomization (ADR). Both methods showed improved performance in policy transfer, with ADR addressing and solving the main issues of UDR, ultimately achieving significantly better results. Our experiments demonstrated that ADR can automatically adjust the parameters of the randomization process, resulting in faster training times and enhanced generalization performance.

References

[1] OpenAI et al. Solving rubik's cube with a robot hand. *CoRR*, abs/1910.07113, 2019. arXiv:1910.07113. http://arxiv.org/abs/1910.07113. 6

[2] OpenAI. Hopper - gym documentation. https://www.gymlibrary.dev/environments/mujoco/hopper/#hopper. 2

[3] Abdessamed Qchohi, Alessio Giuffrida, and Francesco Giannuzzo. Reinforcement Learning for Sim-to-Real transfer, https://github.com/franceth/RL_CustomHopperSim2Real, 2024. 1

[4] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021. http://jmlr.org/papers/v22/20-1364.html. 4