Sim-to-Real transfer of Reinforcement Learning Policies in Robotics

Christian Montecchiani
Politecnico di Torino
christian.montecchiani@studenti.polito.it

Marco Sorbi
Politecnico di Torino
marco.sorbi@studenti.polito.it

Gabriele Spina
Politecnico di Torino
gabriele.spina@studenti.polito.it

***Abstract*—Nowadays, one of the main problems of the Reinforcement Learning paradigm is its challenging application to robotics, due to the difficulties of learning in the real world and modeling physics into simulations.**
In this report, we explore one of the more advanced methods of Domain Randomization, Automatic DR, to find out how it deals with former methods' issues and show its strengths and weaknesses.

I. Introduction

As the aim of the project is to analyze different proposed approaches to the Sim-to-Real problem in the field of Reinforcement Learning, we describe in this section the main concepts of this learning paradigm. The code of the implemented methods can be found at the project repository [1].

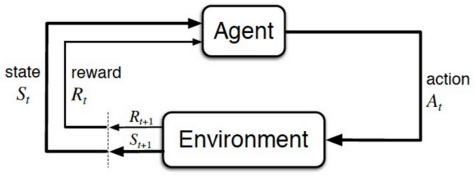

Fig. 1: General Reinforcement Learning schema.

A. Reinforcement Learning

The general learning framework used throughout the project is Reinforcement Learning (RL), in which the knowledge is acquired by *experience*. The goal of RL is to train an *agent* to maximize a numerical *reward*, R_t. The agent collects *observations* of the external *environment*'s current *state*, S_t, which are used to decide the *actions*, A_t, to perform. The actions are taken accordingly to a particular strategy, called *policy* $\pi(\cdot)$. The general schema is summarized in Fig. 1.

As already said, RL deals with learning from experience. The agent learns to *reinforce* those behaviors that will lead to a high reward in the future, while discouraging the poor-rewarding behaviors. In this report, we will analyze some well-known algorithms of RL, in particular, the *policy-gradient* methods. We will talk more in detail about the algorithms in Section II.
We introduce now some common quantities that will be used in the next sections:

- The sequence of $S_0, A_0, \ldots, A_{T-1}, S_T$ is called *episode* and it finishes when the S_T is a terminal state.
- The *return* at a given time t:

$$\begin{aligned} G_t :=& R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots \\ =& \sum_{k=t+1}^{\infty} \gamma^{k-t-1} R_k \end{aligned} \tag{1}$$

  is the sum of rewards starting from time t weighted by powers of the *discount factor* $\gamma \in [0, 1]$.
- 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. It is the expected return starting from the initial state S_0 by following the policy π_θ. It holds that $J(\theta) = V_{\pi_\theta}(S_0)$.

B. Sim-to-Real

Reinforcement Learning can be used to solve very complex tasks and thus it lends itself to being a good learning framework for robots. However, the process of training could be very long and expensive in a real world setup and could also bring dangerous or misleading outcomes. One solution for solving these issues is to speed up the training procedure using a simulator, building an approximate model of the environment dynamics. We call *reality gap* the discrepancy between the real world dynamics and the simulation. This gap leads a model to learn only the approximate dynamics of the simulator and therefore to perform poorly in the real environment. The challenge of correctly transferring the experience from the simulation to the real world is called *Sim-to-Real*, and it deals with training policies that are more likely to adapt to the real environment, while still trained in a different one.

Inside the Sim-to-Real world, we will focus on **Domain Randomization** (DR) [2], which is a way to increase the robustness of the policy by letting the agent address different situations during training. Given the environment, we can make its parameters vary in a random way to create a collection of slightly different environments to be fed to the simulator. In this manner the agent will be trained each time on different dynamics, resulting in a decreased probability of overfitting the simulation. In this report, we will deal with two versions of it: *Uniform DR* in Section III and *Automatic DR* in Section IV.

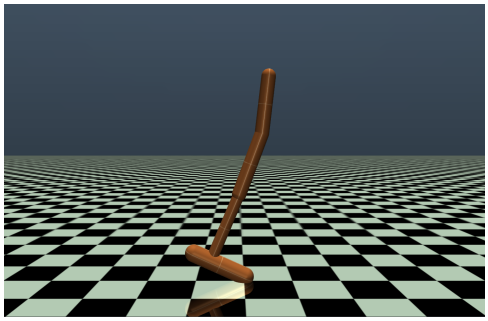

Fig. 2: Hopper environment.

During the project, the randomized parameters were the masses of the body parts. Also, for practical and feasibility reasons, we did not try to transfer the experience to the real world, but instead we tried to mimic this procedure by performing a *Sim-to-Sim* transfer. We call *source environment* the one in which training is performed; *target environment* the one to which we want to transfer the policy. The two environments differ only for the mass of the torso.

C. Simulation setup

The environment taken into consideration is the *Hopper environment* of the Gym API [3], which consists of a one-legged robot composed of torso, thigh, leg and foot (see 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 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*.
In our setting:

- The **environment** is a simulated flat world containing the robot. 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 **agent** controls the robot, based on its current state. It is the one controlling the training process of the **policy** π_θ, represented by a Multi-Layer Perceptron:
  - The input layer has the dimensions of the **State Space**, including the quantities observed by the agent at each state (coordinates, angles of joints, velocities).
  - The output layer has the dimensions of the **Action Space**, that is a set of the three torques that generate the action.
  - $\boldsymbol{\theta}$ is a parameterization of the policy that coincides with the weights and biases of the neural network.
- The **reward** is composed of 3 parts:
  1) *Alive Bonus*: a reward of $+1$ for each time step of life.
  2) *Reward Forward*: at each time step T 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) *Reward Control*: a negative reward for penalizing the hopper if it takes actions that are too large. It acts as a regularization term for learning smooth trajectories without sudden accelerations or jerks.

II. Reinforcement Learning Algorithms

In this section are described the algorithms employed during the project. As aforementioned we have worked with *policy gradient methods*, which are techniques that rely upon optimizing parameterized policies by gradient descent. For each of the algorithms we will present a pseudocode along with the results obtained.

A. REINFORCE

REINFORCE [4] is a vanilla Monte-Carlo policy gradient algorithm (Algorithm 1), which uses an estimation of the episode return to compute the policy gradient, as a result of the general Policy Gradient Theorem [5]:

$$\nabla_\theta J(\theta) = \mathbb{E}\left[G_t \nabla_\theta \ln \pi_\theta(A_t \mid S_t)\right] \tag{2}$$

The policy is trained using a simple Multi-Layer Perceptron with 4 layers and the $\tanh$ activation function.

Algorithm 1 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 $\pi_{\boldsymbol{\theta}}$
  $G_t \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$
  $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \cdot \frac{1}{T} \sum \gamma^t G_t \nabla \ln \pi_{\boldsymbol{\theta}}(A_t|S_t)$
end for

Even if it has good theoretical convergence properties, its high variance can lead to an unstable training process. This motivates the introduction of a *baseline* to be subtracted from G, that keeps the bias unchanged but can reduce the variance. The baseline can be a constant scalar value, such as the expected value of G, or a function of the state.
Three versions of the REINFORCE algorithm were tested:

1) without baseline: $G^* = G$;
2) whitening transformation baseline: $G^* = \frac{G - \bar{G}}{\sigma_G}$ [6];
3) state-value function baseline $G^* = G - \hat{V}(S)$, in which another Multi-Layer Perceptron was employed to estimate $V(S)$.

To compare them, we trained and tested in the source environment 8 different policies per variant, for 20 000 episodes each. We decided to train more policies for the same method to address the high variance of the algorithm. For each variant, we plot the mean return of 50 testing episodes every 250 training episodes, averaged on the 8 policies. From the results

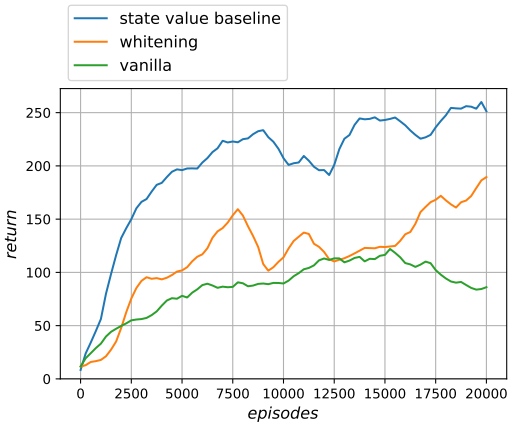

Fig. 3: REINFORCE variants comparison.

TABLE I: Search space for REINFORCE.

Hyper Parameters	Search Space
learning rate agent	{**0.001**, 0.0001, 0.00001}
learning rate baseline	{**0.001**, 0.0001, 0.00001}

TABLE II: Search space for A2C

Hyper Parameters	Search Space
learning rate actor	{0.0003, **0.001**}
learning rate critic	{0.0003, **0.001**}
batch size	{**32**, 64}
use entropy	{True, **False**}

reported in Fig. 3 can be seen that the variant achieving the best performance was the one with the state-value baseline.
To optimize it, we have performed a simple grid search considering different combinations of parameters, shown in Table I. The best one, highlighted in the same table, was able to achieve a mean return of 286 in the source environment, after 20 000 training episodes.

B. Advantage Actor-Critic

Advantage Actor-Critic (A2C) is a policy-gradient method (Algorithm 2) that uses an approximation of the state-value function $\hat{V}$ to perform a bootstrapped estimation of the return G_t. The Bootstrap introduces bias to the model, but has the effect of decreasing the variance. The state-value function is learned using a Multi-Layer Perceptron[1] and constitutes the *critic* part of the method. The policy network assumes here the role of the *actor*, responsible of updating the policy distribution in the direction suggested by the critic.
We tested more variants of the algorithm, differing for:

- Loss function of the critic network:
  - $\ell_{\text{mse}} = (R + \gamma\hat{V}(S_{t+1}) - \hat{V}(S_t))^2$
  - $\ell_V = \delta \cdot \hat{V}(S_t)$
- Whether the updates are batched or not.

[1]With the same architecture described earlier for REINFORCE.

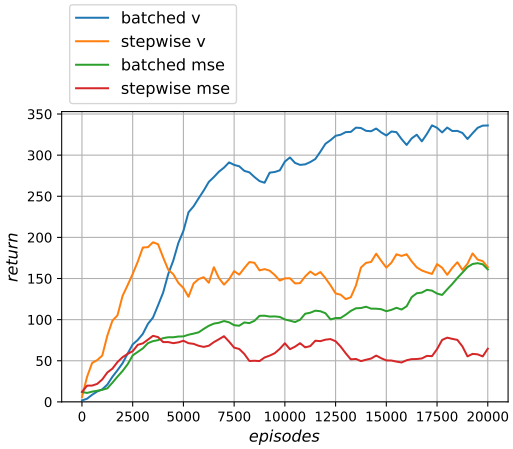

Fig. 4: A2C variants comparison.

Algorithm 2 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

Here, $\delta = R + \gamma\hat{V}(S_{t+1}) - \hat{V}(S_t)$ is called *advantage* term. We performed the test and the tuning using the same procedure described for the REINFORCE algorithm. As we can see from Fig. 4, the batched version with the ℓ_V loss outperforms the other implementations.
The results of the tuning are summarized in Table II. The best combination of parameters, highlighted in the same table, achieved a mean return of 395 in the source environment.

C. TRPO and PPO

Trust Region Policy Optimization (TRPO) [7] and Proximal Policy Optimization (PPO) [8] are two methods for optimizing stochastic control policies using the concept of *Trust Region*. If the policy is performing good, the train procedure should perform the updates in such a way that the new policy has a behavior similar to the old one. This can be expressed in terms of *KL-divergence*, which is a measure of distance between distributions. TRPO is formulated by imposing a constraint

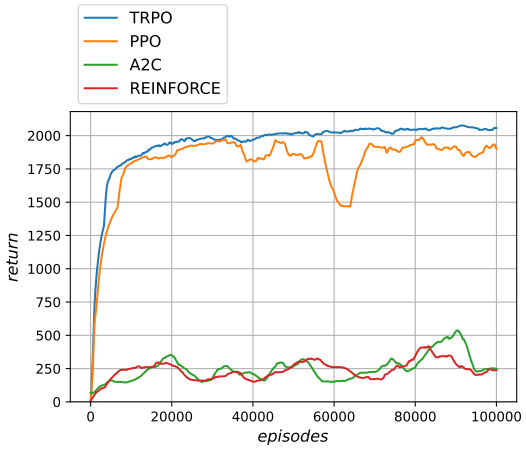

Fig. 5: Comparison between all the algorithms trained in *source* and tested in *source*.

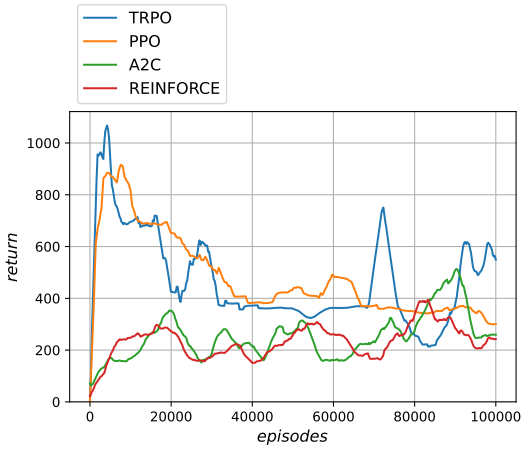

Fig. 6: Comparison between all the algorithms trained in *source* and tested in *target*.

TABLE III: Search space for TRPO

Hyper Parameters	Search Space
learning rate	{3e-4, **2.5e-4**, 1e-3}
batch size	{64, **128**}
n steps	{**1024**, 2048}

TABLE IV: Search space for PPO

Hyper Parameters	Search Space
learning rate actor	{3e-4, **2.5e-4**, 1e-3}
batch size	{64, **128**}
ent coef	{**0**, 0.01}
n steps	{**1024**, 2048}

on the KL-divergence during the optimization of the policy performance measure. PPO on the other hand formulates the concept of trust region in another way:

- *PPO-Penalty* penalizes the KL-divergence in the objective function instead of making it a hard constraint.
- *PPO-Clip* does not rely on KL-divergence, but clips the objective function to obtain a similar behavior.

We used the `stable-baselines3` [9] implementation of both the algorithms, in particular, the *Clip* version for PPO. We tuned them as we did for the other methods. The parameters of TRPO and PPO are reported in Table III and Table IV respectively. The best configurations are highlighted in the same tables.

D. Results

To compare all the methods we trained their best policy for 100 000 episodes in the source environment and tested them in the same environment. The results are visible in Fig. 5. With the policies trained in the source environment, we also compared the performances on the target, obtaining lower results as expected. From Fig. 6 is evident that even PPO and TRPO, which perform extremely well in the source environment, have the same poor results as REINFORCE and A2C. This is a graphical representation of the Sim-to-Real problem.

The following sections are focused on solving these issues by means of Domain Randomization techniques. Even if TRPO seems more stable than PPO, we chose to use the latter as a reference for the next tasks, as it is the one used in the original implementation of Automatic DR [10]. Moreover, the performance of a source-target PPO will be used as a baseline to compare the results of the different methods.

III. Uniform Domain Randomization

A. Description

As introduced in I-B, Domain Randomization is a method that tries to compensate the incapability of the simulator on representing all the complex effects of the real world, by varying the training environment at each episode.

B. Implementation

Since the two environments differ only for the mass of the torso, the parameters to be randomized will be the masses of the other parts. Uniform Domain Randomization associates an uniform distribution $U(a_i, b_i)$ to each parameter, from which their values are sampled at each episode. Algorithm 3 shows a simple pseudocode of the method.

Algorithm 3 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

TABLE V: Search space for UDR

Parameter	Values
mass scaling ratio	[0.717, **1**, 1.40]
distribution width	[**1**, 2, 3]

C. Results

We tried 9 different randomized environments using different widths for the parameters' distributions, and different set of values for their means: the original source masses, the same masses scaled by a factor and then scaled by the inverse of the factor. For the factor we chose the ratio between the masses of the torso in the source and in the target environments. In Table V is reported the search space for these parameters. The best combination, highlighted in the table, was able to achieve a mean return of 1163 in the target environment, after $1e6$ time steps of training.
The results of a policy trained for longer time with UDR are analyzed in detail in Section IV-C, together with those of ADR.

D. Discussion

UDR is a simple algorithm that helps to improve the performance of Sim-to-Real. However, one main problem is that it requires a big effort on the manual tuning. Also, as the environment is randomized a lot since the very beginning, the policy starts to learn how to behave after a lot of time. These are the main justifications for the introduction of *Automatic Domain Randomization*, an improvement of UDR.

IV. Automatic Domain Randomization

A. Description

Automatic Domain Randomization [10] retains the same base idea of UDR of randomizing the environment parameters, but this time without keeping fixed the bounds of the distributions. In fact, the randomized intervals increase or decrease depending on how the agent is performing, in an adversarial fashion. This leads to a DR method that is capable of automatically tune its parameters and that allows for a faster training of the policy, as the difficulty of the task varies to match its performances.

B. Implementation

During the training procedure, the agent is tested at each episode[2] with a given probability p_b and the results of the tests are used to enlarge or restrict a given bound (either lower or upper) of a given parameter. Testing the bound of the i^{th} parameter is performed by setting the parameter value to that bound (e.g. $\lambda_i = a_i$) and collecting performances of m episodes in data buffers (one for each bound) while the other parameters are randomized. When the bound's data buffer is filled, the mean performance is compared to the thresholds to decide the update to perform. Our implementation (Algorithm 4) differs from the original for how the tested bound is selected, mainly for trying to avoid the *collapsing* phenomenon explained in IV-D.

[2]These episodes are used for training as well.

We tried three different strategies for the bound increase Δ:

- *Constant*: $\Delta = \bar{\Delta}$
- *Proportional*: $\Delta = \bar{\Delta}\left(1 + \frac{1}{100}(p - t_H)\right)$
  - where p is the performance on the bound.
- *Random gaussian*: $\Delta = \max\{0.1\bar{\Delta},\ \Delta_{\text{rand}}\}$
  - where $\Delta_{\text{rand}} \sim \mathcal{N}(\bar{\Delta}, \sigma = 0.02)$. Taking the maximum value between the two assures that even a small positive step is performed.

We tried some values between 0.01 kg and 0.2 kg for $\bar{\Delta}$. As we found the collapsing phenomenon (see IV-D) for values greater than 0.05 kg, we chose to keep the base value of 0.02 kg. We also noticed that this is the same order of magnitude of the value used by OpenAI in [10].

Tuning the two performance thresholds t_L, t_H required us to define when the task was considered complete and when not. We found out that high values of t_H made the task too challenging for the policy, while low values brought to a non-ending expansion of the bounds, as the task became too easy. Similar considerations were made for t_L. We tried also a self-tuning approach, with a dynamic update of the thresholds, that unfortunately ended up in a failure. The final constant values 1000 and 1500 were chosen, as they seemed to be a good compromise for our task.

C. Results

To compare the three variants, also with respect to UDR, we tested them every 100 000 time steps on the target environment[3]. We consider also an upper and a lower bound for the task: the former being a vanilla PPO trained in the target environment, and the latter represented by vanilla PPO trained in the source environment (our baseline). Fig. 8 shows the performance evolution of the three variants and the one of UDR, with the references of the upper and lower bound as well. Fig. 7 shows the evolution of the bounds through time of an ADR trained with the *proportional* method.

As we see, both ADR and UDR are above the baseline. However it is evident that ADR outperforms UDR (the constant with $\Delta = 0.05$ does it from the very beginning) and almost reaches the upper bound for the task, obtaining a substantial success on the transfer. Also, ADR seems to have a sudden increase of performance once a certain level of randomization entropy is achieved: finding that level of entropy is exactly what seems to be difficult during the tuning of UDR that, in fact, suffers from our not-so-fine tuning. The only exception for ADR is represented by the gaussian variant, which is comparable with UDR at performance level and which probably needed some more training to reach the other variants.

D. Collapsing phenomenon

In [10] the bound to test is chosen randomly among all the bounds at the beginning of each testing episode. We firstly tried this approach, but we noticed that the buffers

[3]Only for evaluation purposes, we did not use these episodes to train the policy.

Algorithm 4 ADR

Require: $\phi^0 \in \mathbb{R}^d$ {Initial masses values}
Require: $\{D_i^L, D_i^H\}_{i=1}^d$ {Performance data buffers (size m)}
Require: t_L, t_H, where $t_L < t_H$ {Thresholds}
Require: Δ {Update step size}
Require: p_b {Probability of evaluation in this episode}
  $\phi \leftarrow \phi^0$
  $i \leftarrow 1$ {Parameter whose bound is currently tested}
  $b \leftarrow L$ {$b \in \{L, H\}$, lower or upper bound}
  repeat
    $\lambda \sim P_\phi$ {Distribution of parameters}
    $x \sim Bernoulli(p_b)$
    if $x = 1$ **then**
      $\lambda_i \leftarrow \phi_i^b$
      $p \leftarrow$ EvaluatePerformance(λ)
      $D_i^b \leftarrow D_i^b \cup \{p\}$
      if Length(D_i^b) $= m$ **then**
        if Average(D_i^b) $> t_H$ **then**
          Enlarge(ϕ_i^b)
        else if Average(D_i^b) $< t_L$ **then**
          Restrict(ϕ_i^b)
        end if
        Empty(D_i^b)
        if $b = H$ **then**
          $i \leftarrow (i \mod n) + 1$
          $b \leftarrow L$
        else
          $b \leftarrow H$
        end if
      end if
    else
      $p \leftarrow$ EvaluatePerformance(λ)
    end if
  until Training is complete

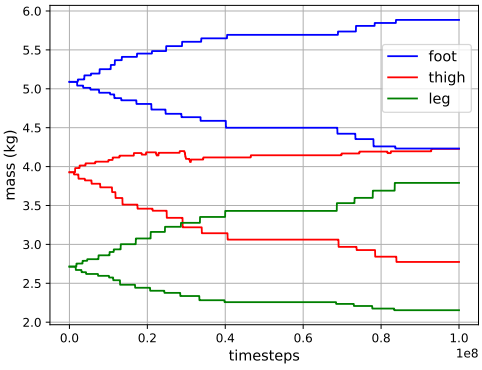

Fig. 7: Bounds evolution for a proportional ADR.

tended to get filled in closed moments: this caused that all the bounds were enlarged at the same time, resulting in a sudden increase of difficulty in only few episodes, becoming too challenging for the policy. In this case, we noticed a

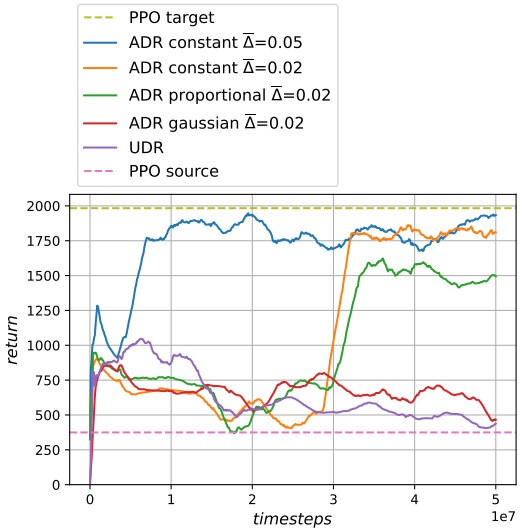

Fig. 8: Comparison between the different variants of ADR together with UDR in the target environment.

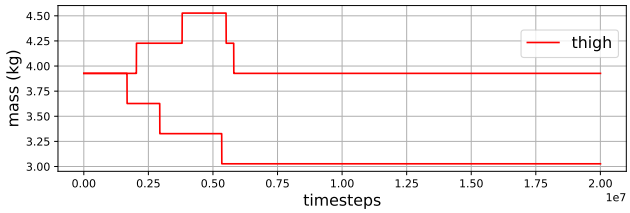

Fig. 9: Collapsing phenomenon for a bound.

collapsing phenomenon that caused the bounds to degenerate again to the starting point, as shown in Fig. 9. To avoid this issue, we decided to test (and update) each bound sequentially, i.e. choosing an order for the bounds and testing the same bound until its data buffer was full. However, we believe that the problem can still be bypassed in other ways, such as decreasing the step parameter Δ, increase the data buffer size m and letting the policy train for much more time.

V. Conclusion

During the project, we dealt with some of the main Reinforcement Learning algorithms and observed their poor robustness in adapting to a slightly different environment. Therefore, we addressed the Sim-to-Real problem with Domain Randomization, testing the vanilla implementation of UDR and the improved method of ADR. They both seemed to improve the performance of the policy transfer, with the latter solving, at least partially, the main problems of the former and achieving much higher results. Our experiments showed that ADR is able to automatically tune the parameters of the randomization process, achieving faster training times and better generalization performance. We also think that other different approaches to ADR are worth to be tried, such as using parameters' distributions other than uniform or finding alternative methods of updating a bound.

REFERENCES

[1] Gabriele Spina, Marco Sorbi, and Christian Montecchiani. *Sim to Real transfer of Reinforcement Learning Policies in Robotics*. https://github.com/smkdGab/Sim-to-Real-transfer-of-Reinforcement-Learning-Policies-in-Robotics. 2022.

[2] Xue Bin Peng et al. "Sim-to-real transfer of robotic control with dynamics randomization". In: *2018 IEEE international conference on robotics and automation (ICRA)*. IEEE. 2018, pp. 3803–3810.

[3] Greg Brockman et al. "Openai gym". In: *arXiv preprint arXiv:1606.01540* (2016).

[4] Ronald J. Williams. "Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning". In: *Mach. Learn.* 8.3–4 (May 1992), pp. 229–256. ISSN: 0885-6125.

[5] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[6] Fork Tree. *Understanding Baseline Techniques for REINFORCE*. Oct. 2019. URL: https://medium.com/@fork.tree.ai/understanding-baseline-techniques-for-reinforce-53a1e2279b57.

[7] John Schulman et al. "Trust region policy optimization". In: *International conference on machine learning*. PMLR. 2015, pp. 1889–1897.

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

[9] Antonin Raffin et al. "Stable-Baselines3: Reliable Reinforcement Learning Implementations". In: *Journal of Machine Learning Research* 22.268 (2021), pp. 1–8. URL: http://jmlr.org/papers/v22/20-1364.html.

[10] OpenAI et al. "Solving Rubik's Cube with a Robot Hand". In: *CoRR* abs/1910.07113 (2019). arXiv: 1910.07113. URL: http://arxiv.org/abs/1910.07113.