Sim-to-Real Transfer Problem in Robot Learning

Arash Daneshvar
dept. control and computer engineering (DAUIN)
Politecnico di Torino
s314415@studenti.polito.it

***Abstract*—The current challenge to use reinforcement learning in robotics is bridging the reality gap, which involves applying simulation models to real-world robotics problems. In this report, I discuss how to address this challenge using sim2real techniques, specifically by employing domain randomization algorithms.**

***Index Terms*—Reinforcement Learning, Robotics, PPO, Domain Randomization (DR), Uniform DR, Automatic DR**

I. Introduction

The main goal of this project is to learn how to develop a control policy for a robot in simulation using advanced Reinforcement Learning (RL) algorithms and address the challenges of applying that policy in the real world. During this project I explored the sim-to-real transfer problem, focusing on training policies in simulation that can be effectively transferred to real-world hardware. The project involves simulating the sim-to-real transfer task in a controlled environment by introducing discrepancies between training and testing domains and implementing domain randomization of dynamics parameters to create robust policies.

A. Reinforcement Learning Overview

Reinforcement learning is a computational approach where an agent learns to make decisions by interacting with an environment to maximize cumulative rewards. It differs from supervised and unsupervised learning by focusing on learning optimal decision-making policies through trial and error. RL involves an agent, environment, states, actions, rewards, and policy, with the agent aiming to learn a strategy that maximizes long-term rewards. Despite its computational challenges, RL has seen success in various domains like robotics and gaming, making it a key area of research in academic and industry. Fig 1 shows the main component of RL

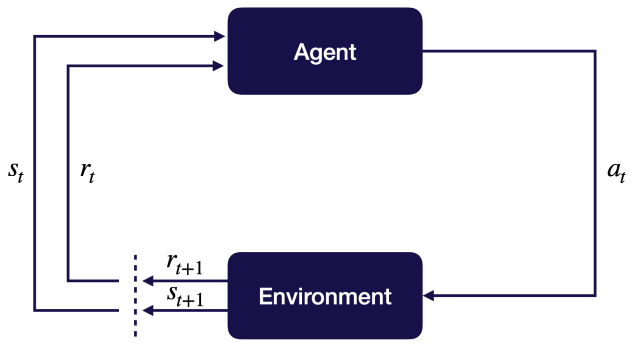


Fig. 1: Main components of RL

This report focuses on the analysis of *Proximal Policy Optimization (PPO)* [1] algorithm in reinforcement learning. A detailed exploration of this algorithm can be found in Section II.

To facilitate understanding in subsequent sections, I introduce key concepts as follows:

Episode: An episode is a sequential series of states and actions, denoted as $S_0, A_0, \ldots, A_{T-1}, S_T$, terminating when S_T reaches a terminal state.

Return at time t: The return G_t at time t is defined as the sum of discounted rewards:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots = \sum_{k=t+1}^{\infty} \gamma^{k-t-1} R_k,$$

where γ (with $0 \leq \gamma \leq 1$) is the discount factor.

State-value function $V^\pi(S)$: This function represents the expected return starting from state S under policy π. It quantifies the long-term value of being in state S and following policy π.

Policy performance measure $J(\theta)$: This measure evaluates the performance of policy π_θ with respect to its parameters θ. It calculates the expected return starting from the initial state S_0 and following policy π_θ. Mathematically, $J(\theta) = V^{\pi_\theta}(S_0)$.

B. The Role of Sim2Real

There is a real danger (in fact, a near certainty) that programs which work well on simulated robots will completely fail on real robots because of the differences in real world sensing and actuation (it is very hard to simulate the actual dynamics of the real world [2]).

Sim2Real (Simulation to Reality) involves transferring models or policies trained in simulated environments to real-world applications. [3] In RL, this means an agent learns tasks in a simulation setting and then applies its knowledge to real-world scenarios. Technique like *Domain Randomization (DR)* [4] help bridge differences between simulation and reality, making the agent's performance robust and effective in real-world conditions. This approach is crucial for safe, cost-effective training in fields like robotics and autonomous driving.

DR is a technique used in RL to improve the transfer of models from simulation to real-world environments. By varying simulation parameters (e.g., lighting, textures, object properties) during training, the technique makes learned policies more robust to real-world variations, reducing the *reality gap* and enhancing performance in diverse, real-world conditions. In this report, I will deal with two versions of it: *Uniform DR* in Section III and *Automatic DR* in Section IV.

Throughout the project, the randomized parameters included the masses of the body parts. For practical and feasibility reasons, I did not transfer the experience to the real world. Instead, I simulated this process by conducting a Sim-to-Sim transfer. The environment used for training is referred to as the *source environment*, while the environment to which I aim to transfer the policy is called the *target environment*. The primary difference between these two environments is the *mass of the torso*.

C. Simulation Environment

The RL agent utilized in this project operates within the *Gym Hopper environment* [5]. This environment provides a user-friendly Python interface that manages the MuJoCo physics engine, facilitating the modeling of the robot. Within the Hopper environment of the Gym API, there exists a one-legged robot comprised of torso, thigh, leg, and foot components (Fig. 2). The objective of the Hopper is to learn the optimal strategy for executing forward hops as swiftly as possible while maintaining balance. This movement is achieved by applying torques to the three hinges connecting the various body parts. The Hopper is deemed unsuccessful if it falls or loses balance, otherwise, it is considered successful.

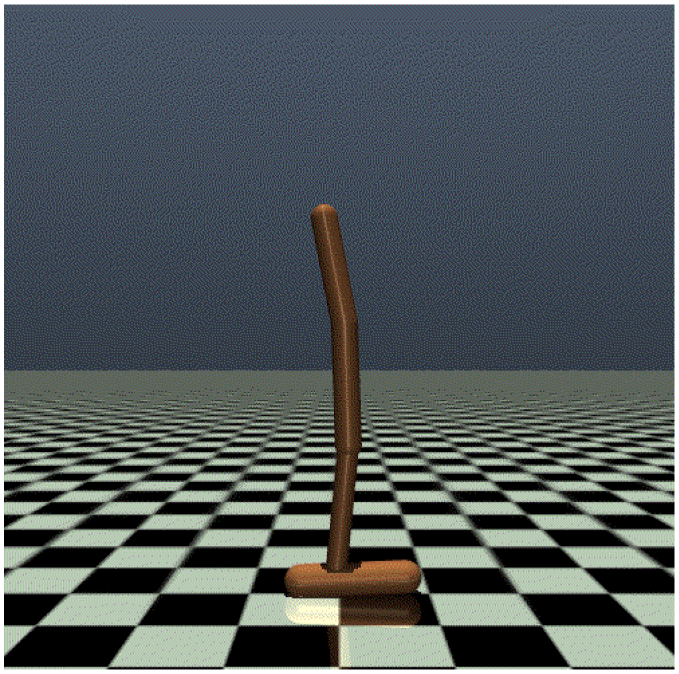


Fig. 2: Hopper Enviroment in OpenAI Gym

Defining the core RL component within the Gym Hopper environment for this scenario:

- *Environment*: The environment is a virtual flat landscape where the robot operates. In the source environment, the robot's body parts have specific masses: 2.53 kg for the torso, 3.93 kg for the thigh, 2.71 kg for the leg, and 5.09 kg for the foot. The target environment, instead, differs only for the mass of the torso, which is 3.53 kg.
- *Agent*: The agent controls the robot based on its current state. It controls the training process of the policy $\pi\theta$.
- *Reward*: The reward is composed of three parts: *Alive Bonus* (a reward of +1 for each time step of life), *Reward Forward* (reward of hopping forward), and *Reward Control* (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.

D. Playing Around with the Underlying Hopper Environment

First and foremost, it is essential to become acquainted with the Gym Hopper environment using a Python script. To familiarize oneself, the following questions will be addressed:

Question 1.1: What is the state space in the Hopper environment, and is it discrete or continuous?

Answer The Box class in OpenAI Gym represents a continuous space. It defines the shape and bounds of the space. Here, the space is a 3-dimensional box with each dimension ranging from -1.0 to 1.0, and the data type is float32.

Here is the Python code snippet:

```
>>> env = gym.make(
        'CustomHopper-source-v0')
>>> print(env.action_space)
Box(-1.0, 1.0, (3,), float32)
>>> print(env.action_space.sample())
[ 0.95167387 -0.5549509   0.48059082]
```

Question 1.2: What is the action space in the Hopper environment, and is it discrete or continuous?

Answer The output indicates that the observation space is a continuous space represented by a Box with 11 dimensions, ranging from -inf to inf, with float64 data type. Here is the Python code snippet:

```
>>> env = gym.make(
        'CustomHopper-source-v0')
>>> print(env.observation_space)
Box(-inf, inf, (11,), float64)
>>> print(env.observation_space.sample())
[ 0.16027109  0.54143471  0.40487662
3.26660005  0.56097582  0.58330911
-1.17927852 -2.14839704  0.41121517
-1.44608237  0.56991584]
```

Question 1.3: What are the mass values of each link in the Hopper environment, both in the source and target variants, respectively?

Answer The mass values of each link in the Hopper environment for both the source and target environments are detailed in the code below Here is the Python code snippet: source enviroment

```
$source enviroment$

>>> env = gym.make(
        'CustomHopper-source-v0')
```

```
>>> print("name:",
          env.sim.model.body_names)
>>> print("value:",
          env.sim.model.body_mass)
name: ('world', 'torso',
       'thigh', 'leg',
       'foot')
value: [0. 2.53429174
        3.92699082 2.71433605
        5.0893801 ]
$target enviromet$

>>> env = gym.make(
        'CustomHopper-target-v0')
>>> print("name:",
          env.sim.model.body_names)
>>> print("value:",
          env.sim.model.body_mass)
name: ('world', 'torso',
       'thigh', 'leg',
       'foot')
value: [0. 3.53429174
        3.92699082 2.71433605
        5.0893801 ]
```

II. Implement RL Pipeline

In this section, I developed a RL pipeline to train a basic control policy for the Hopper environment. Utilizing a third-party library, I employ state-of-the-art RL algorithm PPO. For thorough comprehension, I evaluate the performance of this algorithm when trained and tested within the source environment, as well as when trained in the source environment and tested in the target environment and also trained and tested in the target environment.

A. Proximal Policy Optimization (PPO) Algorithm

PPO is a RL algorithm that enhances training stability and efficiency by leveraging a *trust region* approach. This approach ensures that updates to the policy parameters are constrained to a manageable range around the previous policy, preventing large deviations. PPO introduces two main variants:

- *PPO-Penalty*: Integrates KL-divergence as a penalty in the objective function, discouraging significant policy changes.
- *PPO-Clip*: Constrains policy updates by clipping the objective function, limiting changes to a predefined threshold.

For this project implementation, I utilized the Clip version of PPO from the stable-baselines3 library [6]. The specific parameters for PPO are detailed in Table I.

B. Result

In this project, I employed the PPO algorithm to train and test in three distinct scenarios, each involving training the best policy for 10,000,000 timesteps. All the plots depict episodes per reward mean. The x-axis represents timesteps, and the y-axis represents the mean reward.

```
Algorithm 1 Proximal Policy Optimization
Require: Initial policy parameters $\theta_0$
Require: Learning rate $\alpha$, batch size $B$, number of epochs $K$
Require: Discount factor $\gamma$, Generalized Advantage Estimation (GAE) parameter $\lambda$
  Initialize actor-critic network with parameters $\theta$ and $\theta'$
  Initialize experience buffer $D$
  for iteration $t = 1, 2, \dots, T$ do                ▷ Collect experience
      Sample trajectories $\{\tau_i\}$ using policy $\pi_\theta$
      Compute rewards-to-go $R_t$ and advantages $A_t$ using GAE
      Store $\{(s, a, R_t, A_t)\}$ in $D$
  for epoch $k = 1, 2, \dots, K$ do                    ▷ Optimize surrogate objective
      for mini-batch $b$ in $D$ do
          Compute policy gradient $\nabla_\theta \mathcal{L}_b(\theta)$ using current policy $\pi_\theta$
          Compute clipped surrogate objective $L_b^{clip}(\theta)$
          Update policy parameters $\theta$ using gradient ascent on $L_b^{clip}(\theta)$
```

TABLE I: 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}

- **Source→Source:** The policy was trained and tested within the source environment, and the results are presented in Fig. 3.

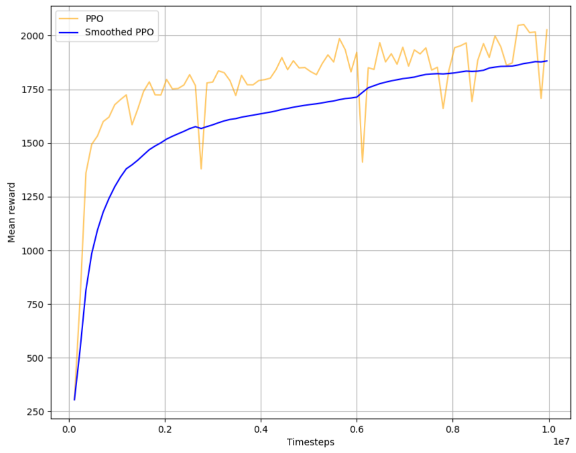


Fig. 3: PPO algorithm trained and tested in source.

- **Source→Target (lower bound):** The policy trained in the source environment was tested in the target environment. The results can be seen in Fig. 4.

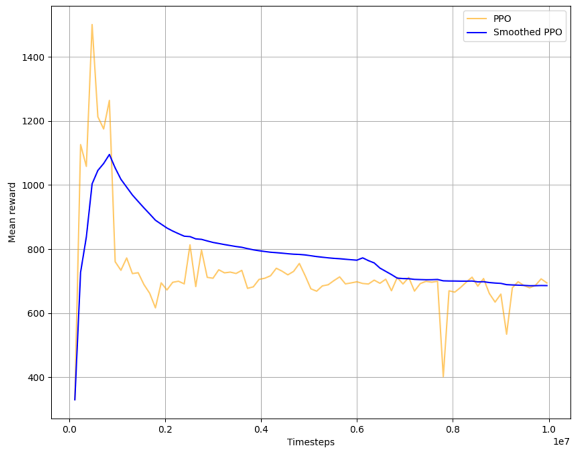


Fig. 4: PPO algorithm trained in source and tested in target.

- **Target→Target (upper bound):** The policy was trained and tested exclusively within the target environment, with the results shown in Fig. 5.

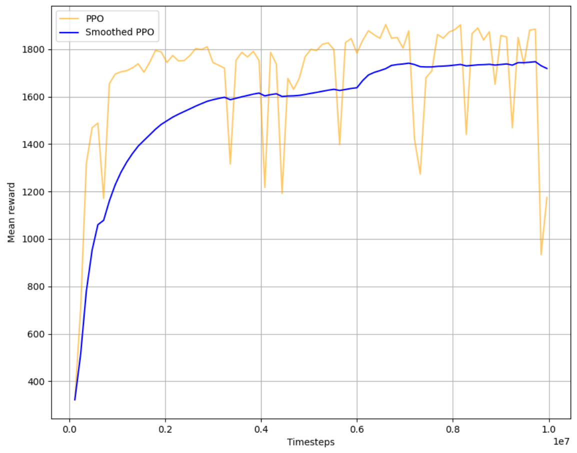


Fig. 5: PPO algorithm trained in source and tested in target.

To compare the results, I address these two questions:

Question 2.1: Why do we expect lower performance from the 'Source→Target' configuration compared to the 'Target→Target'?

Answer:I expect lower performance in the "source→target" configuration because the policy learned in the source environment is tailored to its specific dynamics. When transferred to the target environment, which has different dynamics, the policy may not perform optimally. In contrast, training directly in the target environment aligns the policy with the test conditions, leading to better performance.

Question 2.2: If higher performance can be achieved by training directly on the target environment, what prevents us from doing so in a sim-to-real setting?

Answer: Training directly on the real environment is often impractical due to high costs, safety risks, time constraints, and the need for controlled, reproducible conditions. Simulations allow for faster, safer, and more cost-effective training, making them preferable despite the reality gap they introduce.

I observed reduced performance when training and testing in different environments, highlighting the Sim-to-Real challenge. The following sections focus on addressing these issues using DR techniques.

III. Uniform Domain Randomization (UDR)

As mentioned in Section I-B, DR addresses the limitations of simulators in accurately representing the complexities of the real world by introducing variability into the training environment across episodes.

A. Methodology

Considering that the main distinction between the two environments is the torso mass, the approach emphasizes randomizing parameters related to other components. In Uniform Domain Randomization (UDR), each parameter is given a uniform distribution $U(a_i, b_i)$, where a_i and b_i represent the lower and upper limits. At the start of each episode, values for these parameters are independently sampled from their respective distributions.

Algorithm 2 Uniform Domain Randomization (UDR)

```
Require: $\{m_i\}_{i=1}^d$                ▷ Parameters to be randomized
Require: $\{(a_i, b_i)\}_{i=1}^d$          ▷ Distribution bounds
1: repeat
2:     for $i \in \{1, \dots, d\}$ do
3:         $m_i \sim U(a_i, b_i)$          ▷ Sample from uniform
   distribution
4:     Generate Data ($\{m_i\}_{i=1}^d$)
5:     Update policy
6: until Training is complete
```

TABLE II: Search space for UDR

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

B. Results

I investigated nine different randomized environments by adjusting the widths of parameter distributions and using various sets of mean values: the original source masses, these masses scaled by a factor, and the inverse of that factor. The scaling factor was calculated based on the ratio between the torso masses in the source and target environments. Table II presents the parameter search space. The optimal combination, highlighted in the table, achieved a mean reward of 1034 in the target environment following 10 million training steps.

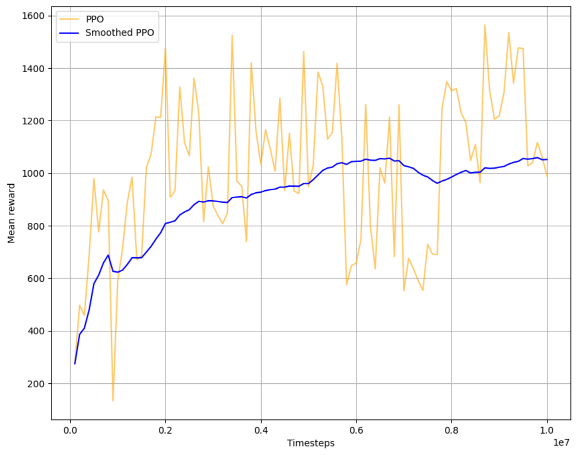


Fig. 6: UDR algorithm.

The results of the UDR are shown in Figure 6.

At the end, I answered these two questions related to the UDR technique.

Question 3.1: Is UDR able to overcome the unmodeled effect (shift of torso mass) and lead to more robust policies w.r.t. the naive “source→target” configuration in task 3?

Answer: Yes, UDR helps in training more robust policies by exposing the agent to diverse simulated conditions, which mitigates the negative impact of the torso mass shift between source and target environments. This improves the policy's ability to generalize and perform well in varied real-world scenarios.

Question 3.2: Can you think of limitations or downsides of UDR?

Answer: UDR may increase computational costs and training time due to the need for multiple simulations with randomized parameters. It also poses challenges in finding an optimal policy that performs well across all randomized conditions. Additionally, there's a risk of overfitting to the simulated distribution, and UDR might not fully cover all variations encountered in real-world environments.

IV. Automatic Domain Randomization (ADR)

ADR expands on the basic principle of UDR by incorporating dynamic adjustments to the ranges of randomized environment parameters. Unlike UDR, which uses fixed distribution bounds, ADR modifies these bounds based on the agent's performance, adjusting them in an adversarial manner. This results in a domain randomization method that can automatically tune its parameters, leading to faster policy training. By aligning task difficulty with the agent's performance levels, ADR ensures that challenges are appropriate for the agent's capabilities.

Algorithm 3 ADR

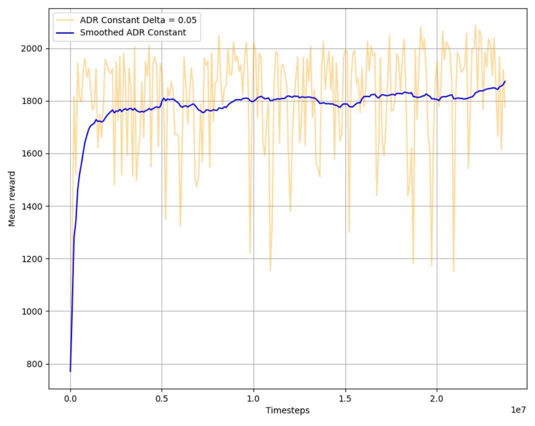
```
Require: $\phi_0 \in \mathbb{R}^d$ Initial masses values
Require: $\{D_L, D_H\}^d$ Performance data buffers (size $m$)
Require: $t_L, t_H$, where $t_L < t_H$ Thresholds
Require: $\Delta$ Update step size
Require: $p_b$ Probability of evaluation in this episode
    $\phi \leftarrow \phi_0$
 1: $i \leftarrow 1$ Parameter whose bound is currently tested
 2: $b \leftarrow L$  $b \in \{L, H\}$, lower or upper bound
 3:
 4: repeat
 5: $\lambda \sim P_\phi$ Distribution of parameters
 6: $x \sim$ Bernoulli$(p_b)$
 7:
 8:     if $x = 1$ then
 9: $\lambda_i \leftarrow \phi_b$
10: $p \leftarrow$ EvaluatePerformance$(\lambda)$
11: $D_{ib} \leftarrow D_{ib} \cup \{p\}$
12:
13:         if Length$(D_{ib}) = m$ then
14:             if Average$(D_{ib}) > t_H$ then
15: Enlarge$(\phi_b)$
16:             else
17:                 if Average$(D_{ib}) < t_L$ then
18: Restrict$(\phi_b)$
19:             Empty$(D_{ib})$
20:             if $b = H$ then
21:                 $i \leftarrow (i \mod d) + 1$
22:                 $b \leftarrow L$
23:             else
24:                 $b \leftarrow H$
25:     else
26: $p \leftarrow$ EvaluatePerformance$(\lambda)$
27: until Training is complete
```

A. Implementation

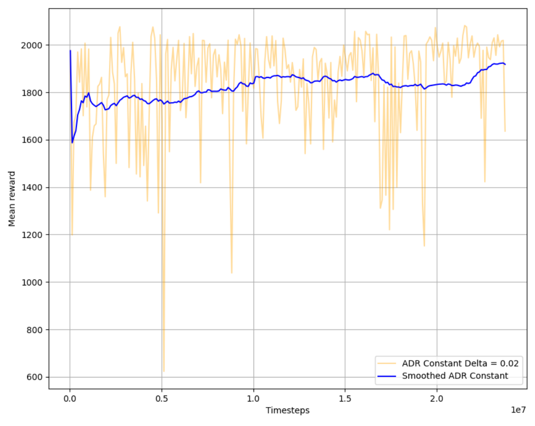
During the training process, the agent is periodically tested with a probability p_b per episode. The results of these tests are used to dynamically adjust the bounds (either lower or upper) of specific parameters. Testing involves setting the parameter to the bound value (e.g., $\lambda_i = a_i$), randomizing other parameters, and collecting performance data over m episodes in dedicated buffers for each bound. Once the buffer is filled, the mean performance is compared against predefined thresholds to determine the necessary adjustment.

I explored three distinct strategies for increasing the bounds (Δ):

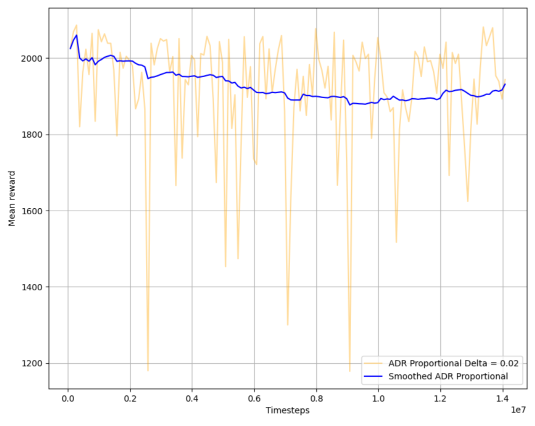
- Constant: $\Delta = \bar{\Delta}$
- Proportional: $\Delta = \frac{\bar{\Delta}}{100} \times \left(1 + \frac{p - t_H}{100}\right)$, where p represents the performance on the bound.
- Random Gaussian: $\Delta = \max\{0.1\bar{\Delta}, \Delta_{\text{rand}}\}$, with $\Delta_{\text{rand}} \sim \mathcal{N}(\bar{\Delta}, \sigma = 0.02)$. This approach ensures a positive adjustment even with a small step.



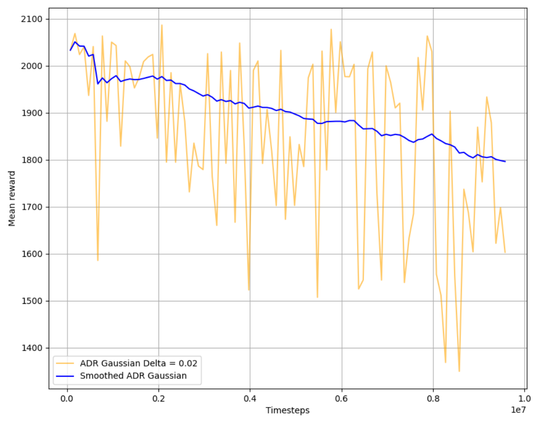
(a) ADR algorithm constant $\Delta = 0.05$



(b) ADR algorithm constant $\Delta = 0.02$



(c) ADR algorithm proportional $\Delta = 0.02$



(d) ADR algorithm gaussian $\Delta = 0.02$

Fig. 7: ADR algorithm with 4 variants

I experimented with $\bar{\Delta}$ values ranging from 0.01 kg to 0.2 kg. Upon observing significant changes occurring for values exceeding 0.05 kg, I decided on a base value of 0.02 kg, which aligns with values used by OpenAI [7].

Setting the performance thresholds t_L and t_H required defining task completion criteria. Higher t_H values rendered tasks overly challenging for the policy, while lower values led to perpetual boundary expansions, indicative of tasks becoming too simplistic. Similar considerations guided the selection of t_L. Attempts at a self-tuning approach for dynamically adjusting thresholds proved unsuccessful. Ultimately, constant values of 1000 and 1500 were chosen as they struck a balanced compromise suitable for this task.

B. Results

To evaluate the performance of the four variants, assessments were run 10,000,000 timesteps in the target environment. Figure 7 illustrates the performance trajectory of the four variants.

ADR consistently outperforms UDR, particularly evident from the beginning with the constant variant using $\Delta = 0.05$. ADR nearly approaches the upper task bound, demonstrating significant efficacy in transfer learning. Notably, ADR exhibits a marked performance enhancement once it reaches a certain level of randomization entropy. Determining this optimal entropy level presents a challenge in UDR tuning

V. CONCLUSION

This project involved an exploration of RL algorithm and its performance when confronted with environmental variations. To address the Sim2Real challenge, DR techniques were employed, evaluating both UDR and the more advanced ADR approach. Both methods demonstrated improvements in policy transfer capabilities, with ADR effectively overcoming UDR's limitations and achieving notably superior results. My experiments highlighted ADR's ability to autonomously adjust randomization parameters, thereby speeding up training cycles and enhancing generalization capabilities. Moreover, further exploration of alternative strategies within the domain of ADR, such as the adoption of non-uniform parameter distributions or novel methodologies for updating bounds, holds potential for advancing this field.

* The code of the implemented methods can be found at the project repository [8].

REFERENCES

[1] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). "Proximal policy optimization algorithms"

[2] Brooks, Rodney. (1998). "Artificial Life and Real Robots"

[3] Höfer, S., Bekris, K., Handa, A., Gamboa, J. C., Golemo, F., Mozifian, M., ... & White, M. (2020). "Perspectives on sim2real transfer for robotics: A summary of the R: SS 2020 workshop"

[4] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel, (2017)"Domain Randomization for Transferring Deep Neural Networks from Simulation to the Real World."

[5] Greg Brockman et al. (2016). "Openai gym".

[6] Antonin Raffin et al. (2021) "Stable-Baselines3: Reliable Re- inforcement Learning Implementations".

[7] OpenAI et al. (2019) "Solving Rubik's Cube with a Robot Hand"

[8] project-sim2real-arash-daneshvar