Analysis of Sim-to-Real strategies with Domain Randomization techniques

1st Simone Borella
s317774
Polytechnic University of Turin
Engineering and Artificial Intelligence

2nd Florentin Cristian Udrea
s319029
Polytechnic University of Turin
Data science and Engineering

3rd Matteo Zulian
s310384
Polytechnic University of Turin
Automation and Intelligent Cyber-Physical Systems

***Abstract*—Sim-to-real transfer has been a challenge in robot learning literature till these days. The use of simulation environments for training robots offers advantages such as cost-effectiveness, safety, and scalability. Simulated environments allow for extensive exploration of diverse scenarios and enable the learning of complex tasks. In this paper, we will explore the challenges associated with training a robot in a simulated environment with reinforcement learning techniques and transferring the policy to the real world, filling the reality gap between simulated model and real model. We are focusing on the gym Hopper environment, employing a sim-to-sim technique to emulate sim-to-real applications. We are going to analyze the inherent drawbacks of this approach and show two solutions: Uniform Domain Randomization (UDR) and Bayesian Domain Randomization (BayRn).**

I. Introduction

During the past years, Robots have found a place in an increasing number of fields and applications. They improved in strength, speed, accuracy and their cost grew accordingly. To apply reinforcement learning in robotics, safe exploration becomes a key issue of the learning process. Robots are now valuable and expensive machines, not suitable for training, due to the high maintenance and repairing costs. Learning in a real environment can also be time consuming since a human operator may be required to manually reset an episode.

Furthermore reinforcement learning methods require vast amounts of data to be effectively trained. The more a task is complex, in terms of observation and action dimensionality, or exploration difficulty, the more experience is required. Complex tasks can easily require days or weeks of experience data to be solved. Acquiring such amounts of experience on real robotic systems is impractical. Keeping a complex robotic system running for such lengths of time is unfeasible, additional infrastructure for managing the environment setup are required, and untrained policies can potentially damage the robot or the environment.

For the above reasons, it is common practice to conduct training on an approximation of the real model in a simulated environment. The learned policy is subsequently transferred and applied to the actual real application. This kind of method is called sim-to-real transfer and it's widely used in robotics.

The core issue with simulation training is the *reality gap*, the discrepancy between the characteristics of the simulated environment and those of the real one.

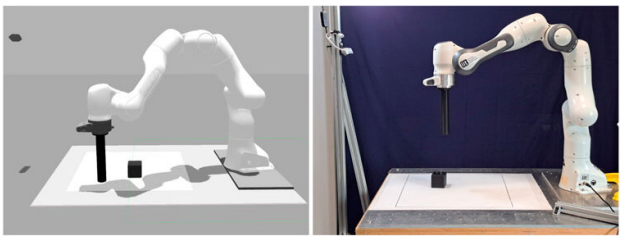

Fig. 1. Simulated and real environments in sim-to-real application

The reality gap is due to some important aspects of simulated environments. First of all simulations involve mathematical models of physical phenomena. While these models can be sophisticated, they might not perfectly represent the complex and dynamic nature of the real world. Deviations in physics modeling can lead to inaccuracies in predicting how objects move, interact, or respond to external forces. Simulated environments often simplify or overlook certain environmental factors, and some factors could be unknown. Finally they often rely on approximations of real-world sensors and actuators.

The problem of transferring control policies from simulation to the real world can be viewed as an instance of Domain Adaptation (DA), where a model trained in a source domain is transferred to a target domain.

In the next sections we present first the problem formulation and notations to introduce two sim-to-real techniques (thus Domain Adaptation techniques) used to mitigate the reality gap between simulation and reality: Uniform Domain Randomization (UDR) and Bayesian Domain Randomization (BayRn).

In the subsequent sections we focus then on analysing the application of these techniques. In our experiments we simulated the sim-to-real transfer task in a sim-to-sim scenario, where a discrepancy between source (training) and target (testing) domains is manually injected. The MuJoCo Hopper environment is employed as case study to introduce evidences of the discrepancies between the real world and simulations.

In these sections we are going to describe the experimental steps and show results obtained enforcing the main characteristics of these strategies.

II. Problem formulation

The following techniques we are going to describe require a slightly different formulation of the Markovian Decision Process (MDP), since parameters of the source domain model are chosen randomly with a certain parameterized stochastic distribution.

The treated MDP system is defined by a tuple $(S, A, P, R, \gamma, \xi, s_0)$, where:

S is the finite set of states

A is the finite set of actions

P is the transition probability function

R is the reward function

γ is the discount factor

ξ are the environment parameters

s_0 is the initial state

The environment is instantiated through its parameters ξ which are assumed to be random variables distributed according to the following probability distribution parameterized by ϕ.

$$\xi \sim \nu(\phi)$$

The initial state s_0 is drawn from an initial state distribution.

$$s_0 \sim \mu_{0,\xi}(s_0 \mid \xi)$$

The transition probability function P_ξ gives the probability of transitioning from one state to another after taking a certain action. It can be written as:

$$P_\xi : S_\xi \times A_\xi \times S_\xi \to \mathbb{R}^+$$

$$P_\xi(s' \mid s, a) = \mathrm{P}\{S_{t+1} = s' \mid S_t = s, A_t = a\}$$

The reward function R defines the immediate reward received after taking an action in a certain state:

$$R : S_\xi \times A_\xi \to \mathbb{R}$$

$$R(s, a) \mapsto r$$

The objective in an MDP is to find a policy π, which is a mapping from states to actions, expressed as a distribution of actions over states. The policy is parameterized with θ parameter.

$$\pi_\xi : S_\xi \to \Delta(A_\xi)$$

$$\pi_\xi(a|s;\theta) = \mathrm{P}(A_t = a|S_t = s)$$

The goal is to find the optimal policy π^*_ξ that maximizes the expected cumulative reward over time (find the optimal parameters θ^*_ξ which characterizes the policy).

$$\hat{J}_\xi(\theta, s_0) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t) \mid \theta, s_0\right]$$

III. Uniform Domain Randomization

Domain Randomization (DR) is a complementary class of techniques for Domain Adaptation (DA) that is particularly well suited for simulations. With domain randomization, discrepancies between the source and target domains are modeled as variability in the source domain. Uniform Domain Randomization (UDR) approach involves the randomization of environmental parameters using uniform distributions. With this strategy we are able to create a variety of simulated environments with randomized properties and train a model that works across all of them, generalizing the policy knowledge.

For each training episode new parameters are sampled from a uniform distribution, in a certain interval $[a, b]$, described as follows:

$$\xi \sim \nu(\phi) \sim U(a, b) = \begin{cases} \frac{1}{b-a} & \text{if } a \le x \le b \\ 0 & \text{otherwise} \end{cases}$$

where $\phi = \{a, b\}$ is the set of the uniform distribution parameters.

An important aspect of this strategy is the choice of parameters intervals. First of all the choice of these parameters is based on an assumption of the uncertainty of the parameter and its real boundaries (e.g. a mass parameter couldn't be a negative value). Furthermore, as the range of a parameter expands, the task of finding an optimal policy becomes increasingly challenging. A broader parameter range introduces a greater diversity of scenarios and potential configurations, requiring the optimization process to navigate a larger and more complex solution space. Therefore, when addressing UDR, there exists a trade-off between robustness to variability in environmental parameters and the maximum expected cumulative reward achieved.

Algorithm 1 describes the training procedure with DR for a model applied to the simulated *source* domain. UDR algorithm can be easily deducted using $\xi \sim \nu(\phi) \sim U(a, b)$.

Algorithm 1: Domain Randomization

Data: Number of episodes: $num\_episodes$,
Parameters distributions: ξ,
$RL\_algorithm\_train\_ep$
Result: Learned policy: $\pi(\theta^*)$

for $num\_episodes$ **do**
  Sample model parameters:
  $\xi \sim \nu(\phi)$
  Rollout the episode with policy ($\pi_\xi(\theta_t)$)
  Train policy π_ξ with new model parameters:
  $\pi_\xi(\theta_{t+1}) \leftarrow RL\_algorithm\_train(\xi, \theta_t)$

IV. Bayesian Domain Randomization

A. Introduction

Drawing a new set of parameters for each training episode with a fixed distribution could be limiting in terms of convergence and robustness of the policy. Furthermore the policy obtained is suboptimal since the reward is evaluated only on the source domain, not considering the behaviour of the policy on the target domain during the training phase. In addition involving fixed distributions means that there is a prior assumption on the parameters domain.

Bayesian Domain Randomization (BayRn) addresses these limitations by dynamically adapting parameter distributions during the learning process, considering the policy performance in the target environment.

BayRn employs Bayesian Optimization to explore the configuration space of source domain distribution parameters. This approach aims to derive a policy that maximizes the cumulative reward (objective function), enabling the adaptation of distributions throughout the policy optimization process.

B. Bayesian Optimization with Gaussian Processes

Bayesian Optimization is a powerful technique for optimizing a complex and expensive objective function.

$$f : X \to \mathbb{R}$$

Bayesian Optimization is widely used in applications like hyperparameter tuning and automated machine learning. In these contexts, it facilitates the iterative exploration of parameter configurations, with each training iteration aimed at discovering potentially better-performing parameter sets.

It exploits a probabilistic model, often based on Gaussian Processes (GP), to model the unknown objective function and guide the search towards promising regions, choosing the next parameters to evaluate.

A Gaussian Process (GP) is a probabilistic model that defines a distribution over functions. In the context of Bayesian Optimization, Gaussian Processes are used to model the underlying objective function.

$$f(x) \sim \mathcal{GP}(m(x), k(x, x'))$$

$$m : X \to \mathbb{R}$$

$$k : X \times X \to \mathbb{R}$$

where m is the mean function and k is the kernel function.

A distinctive feature of Bayesian Optimization is to use the complete history of noisy function evaluations and not only a subset of them.

The key to choose the new set of parameters is the *acquisition function*, which tries to make this choice balancing exploration and exploitation.

$$a : X \to \mathbb{R}$$

Some canonical acquisition functions are *Expected Improvement*, *Probability of Improvement* and *Upper Bound Confidence*.

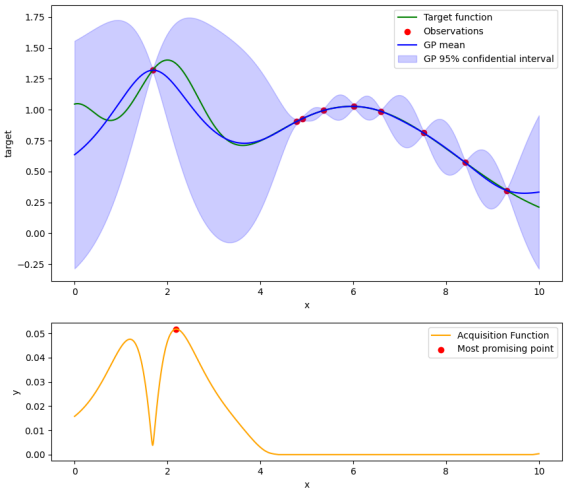

Fig. 2. **Bayesian Optimization Example:** Gaussian Process representation (top) and Acquisition Function (bottom).

The next query point is determined by maximizing the acquisition function, which, in turn, maximizes the probability of selecting parameters that lead to a more promising objective function result.

C. BayRn formulation

In BayRn to improve dynamically the choice of parameters distributions the objective function can be described as follows:

$$f : \Phi \to \mathbb{R}$$

$$f(\phi) \sim \hat{J}_{real}(\theta^*(\phi))$$

In this context, ϕ represents the set of distribution parameters, θ^* denotes the set of parameters defining the optimal policy, and $\hat{J}_{real}$ is the estimated discounted reward achieved by the optimal policy $\pi(\theta^*)$ on the target domain. This policy is discovered during model training, where randomized parameters are sampled from distributions described by $\xi \sim \nu(\phi)$.

The Bayesian Optimization is employed here to evaluate the expected discounted reward on the real domain trained with parameters drawn from distributions parameterized with parameters ϕ.

The problem of source domain adaptation based on returns from the target domain can be expressed in the following two formulations:

$$\phi^* = \arg\max_{\phi}(J_{real}(\theta^*(\phi))) \tag{1}$$

$$\theta^*(\phi) = \arg\max_{\theta}(J_{\xi\sim\nu(\phi)}(\theta, \xi)) \tag{2}$$

In this way this algorithm is able to find the next set of domain distribution parameters ϕ^* that maximizes the discounted reward on the real-world target domain.

Algorithm 2: Bayesan Domain Randomization

Data: Parameters distributions: ξ, Parameter space: $\Phi \sim P^n, \quad P = [\phi^-_{bound}, \phi^+_{bound}]$, $RL\_algorithm$, Gaussian Process: GP, Acquisition function a, Hyperparameters: J_{succ}, n_{max_iter}, n_{init}, n_t

Result: Learned policy: $\pi(\theta^*)$, Domain distribution parameters: ϕ^*

Initialization phase
$D = \{\}$
for n_{init} **do**
  Random parameters sample:
  $\phi \leftarrow U(\phi^-_{bound}, \phi^+_{bound})$
  Get best policy parameters with a training process with domain randomization (Algorithm 1):
  $\theta^* \leftarrow DR(\nu(\phi^*), RL\_algorithm)$
  Evaluation of the policy on target domain:
  $\hat{J}_{real}(\theta^*) \leftarrow \frac{1}{n_t} \cdot \sum_{i=1}^{n_t} J_{real\_i}$
  Extend the data set:
  $D = D \cup \{\phi^*, \hat{J}_{real}(\theta^*)\}$
while $J_{real}(\theta^*) < J_{succ}$ *and* $n_{iter} < n_{max\_iter}$ **do**
  Get next source domain distribution parameters:
  $\phi^* \leftarrow \arg_\phi \max(a(\phi, D))$
  Get best policy parameters with a training process with domain randomization (Algorithm 1):
  $\theta^* \leftarrow DR(\nu(\phi^*), RL\_algorithm)$
  Evaluation of the policy on target domain:
  $\hat{J}_{real}(\theta^*) \leftarrow \frac{1}{n_t} \cdot \sum_{i=1}^{n_t} J_{real\_i}$
  Extend the data set and update the GP posterior distribution:
  $D = D \cup \{\phi^*, \hat{J}_{real}(\theta^*)\}$
  $GP(m, k) \leftarrow GP(m, k|D)$

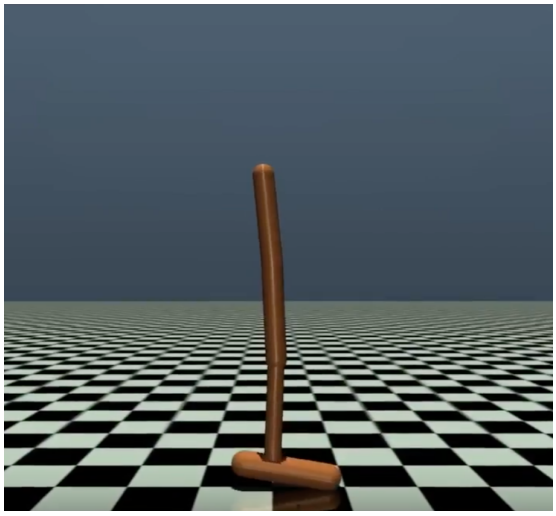

Fig. 3. Hopper Environment

V. HOPPER ENVIRONMENT

The hopper is a two-dimensional one-legged figure that consist of four main body parts - the torso at the top, the thigh in the middle, the leg in the bottom, and a single foot on which the entire body rests. The body masses values are expressed in Table II. The goal is to make hops that move in the forward (right) direction by applying torques on the three hinges connecting the four body parts. This environment is a part of the OpenAI Gym toolkit, which provides a variety of environments for developing and testing RL algorithms, it has been used as a standard control benchmark in a variety of papers.

Observations consist of positional values of different body parts of the hopper, followed by the velocities of those individual parts (their derivatives) with all the positions ordered before all the velocities.

The *observation space* is 11-dimensional, continuous and unbounded.

The *action space* is also continuous, but bounded between [-1, 1]. An action represents the torques applied at the hinge joints. Since there are four masses and three joints the Action Space is 3-dimensional.

	LB	**UB**	**DIM**	**type**
Action Space	-1.0	1.0	3-D	float32
Observation Space	-inf	inf	11-D	float64

TABLE I
HOPPER ENVIRONMENT SPACES

The reward consists of three parts:

- *healthy_reward:* Every timestep that the hopper is healthy it gets a reward of fixed value *healthy_reward*.

$$hr = healthy\_reward$$

- *forward_reward:* A reward for hopping forward.

$$fr = fr\_weight \cdot (x_{\text{before}} - x_{\text{after}})/dt$$

  where dt is the time between actions. This reward is positive if the hopper hops forward in the positive x-direction.
- *ctrl_cost:* A cost for penalizing the hopper if it takes actions that are too large.

$$cc = cc\_weight \cdot \sum(action^2)$$

The total reward returned is given by:

$$reward = hr + fr - cc$$

VI. Experiments

A. Approach

In order to replicate the sim-to-real scenario within the Hopper environment, we adopt a sim-to-sim approach. Specifically, we intentionally decrease the torso mass of the hopper by 1 kg in the simulated environment, strategically emulating the reality gap. In this context, the simulated environment is referred to as the source domain, while the real environment is denoted as the target domain.

	Source (Kg)	**Target (Kg)**
Torso mass	2.534	3.534
Thigh mass	3.926	3.926
Leg mass	2.714	2.714
Foot mass	5.089	5.089

TABLE II
HOPPER MASSES

The challenge is to perform Domain Adaptation with Domain Randomization techniques focusing only on retaining unchanged mass parameters. Importantly, we abstain from changing the torso mass in order to emulate non-modelable real-world complexity. In sum the objective is to discover an optimal policy that enhances the adaptability of simulation to real-world scenarios through effective domain randomization.

The *PPO* reinforcement learning algorithm provided by the *stable-baselines3* python library is employed in the following procedures, with 1 million total time steps .

B. Hyperparameters tuning

To get the best performance from the algorithms outlined in these papers, an analysis to identify the most effective pair of PPO hyperparameters (γ, α) to use for our problem was needed, where γ is the *discount factor*, while α is the *learning rate*.

We trained and tested the hopper in the source environment with four different values for γ and six for α. More precisely we examined four constant gamma values, three constants learning rates and three dynamic learning rates, which decrease over time, in order to have more precision during the last episodes. In particular we used an exponential schedule (that decreases from the highest constant value used, to the lowest exponentially during training), a linear schedule (that also decreases from highest constant to lowest during training, but linearly) and a step schedule (which halves its learning rate every 20% of the learning procedure). The various learning rates can be seen in Figure 4.

The results of the tests are shown in a heatmap where it's easy to see that the best pairs (γ, α) were (0.99, 0.0003) and (0.99, exp). Further investigations where made to choose the best between the two. We trained five more runs with both models. Finally we chose the pair (0.99, exp) because it yielded better results with less variance during training (as it can be seen in Figure 6). For all the following testing and training it'll be implicit that we used these pair of hyperparameters.

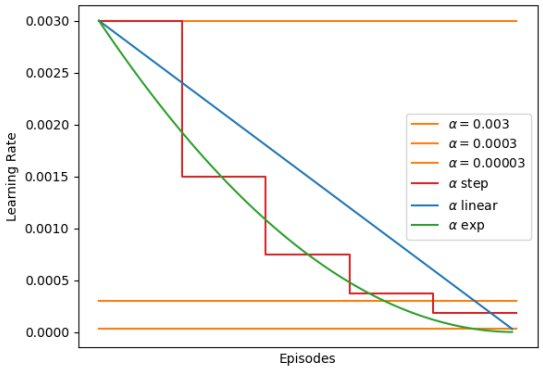

Fig. 4. Learning rate schedules tested (from 0% to 100% of training progress). 'step' is a dynamic learning rate that halves every few episodes starting from 0.003. 'linear' and 'exp' are two learning rate curves that decrease linearly and exponentially.

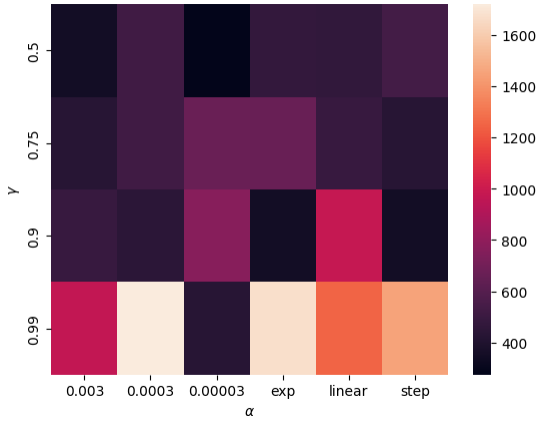

Fig. 5. Heatmap of tests gathering the results of the hyperparameter tuning, where on the x-axis are reported the learning rates while on the y-axis the gammas

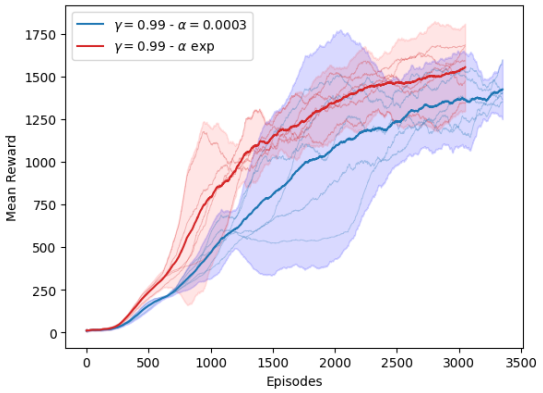

Fig. 6. Learning curves (running average on 200 episodes) for five runs of (0.99,0.0003) in blue; and five for (0.99, exp) in red. The ticker lines are the mean over the five curves, while the semi-transparent are the $2 \cdot \sigma$ (standard deviation), representing the 95% probability of finding other curves in that space. Generally the pair (0.99, exp) performed better.

C. Lower and Upper bounds

To evaluate the effectiveness of the knowledge transferring of subsequent strategies, a lower bound and an upper bound were evaluated. In order to evaluate the lower bound a policy is learned on the source domain and then transferred and tested on the target domain. To ensure completeness, we also present the testing results of this policy on the source domain. The upper bound is then defined learning a policy from the target domain and testing the policy in the same domain.

D. UDR

For this experiment we added Uniform Domain Randomization to the Hopper environment introduced in the previous sections.

As the source domain is randomized, mass parameters (Thigh, Leg and Foot) are represented as parametric uniform distributions.

$$\xi \sim U(a, b)$$

Different strategies were used to define a and b distribution boundaries. Let's call the target masses m_t, which are assumed to be the mean μ of our uniform distributions, ensuring that $a > 0$ and $b > 0$, since they are masses.

As a first strategy, defined a Δm (equal for all parameters), we describe the uniform randomization interval by considering the neighborhood of m_t.

$$\xi \sim U(m_t - \Delta m, m_t + \Delta m)$$

As a second strategy the boundaries distance from the mean are defined by a percentage p (equal for all parameters).

$$\xi \sim U(m_t(1-p), m_t(1+p)), \quad p > 0$$

Various configurations of uniform distribution boundaries were tested, and for each configuration a policy is computed on the randomized source domain. Subsequently policies are tested on both randomized source and target domain. In particular the parameters used were:

$$\Delta m \in \{1, 5, 10\}$$

$$p \in \{0.10, 0.5, 0.95\}$$

E. BayRn

Finding the optimal set of distribution parameters ϕ in domain randomization can be a challenging and time-consuming task, especially when performed manually.

To streamline this process, Bayesian Optimization can be employed. This approach automates the tuning of distribution parameters by iteratively training new policies, each time using the most promising set of parameters ϕ^* identified through historical trials and their relative testing results on the target domain (modeled by a Gaussian Process, denoted as GP).

The Gaussian Process GP is modeled as seen before to approximate the discounted reward $\hat{J}_{real}(\theta^*(\phi))$ for each set of distribution parameters ϕ already tested.

$$f(\phi) \sim \mathcal{GP}(m(\phi), k(\phi, \phi')) \sim \hat{J}_{real}(\theta^*(\phi))$$

To implement Bayesian Optimization, the *BoTorch* library is employed, which incorporates the *GPyTorch* library internally. Botorch deploys techniques for optimizing expensive black-box functions, while GPyTorch plays a crucial role in shaping the underlying Gaussian Process model. A crucial point of the implementation is the choice of the acquisition function. Our choice is the *ExpectedImprovement (EI)*.

$$EI(\mathbf{x}) = \mathbb{E}\left[\max(f(\mathbf{x}) - f(\mathbf{x}^*), 0)\right]$$

where $\mathbf{x} \equiv \phi$ is the candidate point in the parameter space, and $\mathbf{x}^* \equiv \phi^*$ is the current best point known so far.

Uniform Domain Randomization

The initial strategy involved enhancing the Uniform Domain Randomization (UDR) approach, as previously described, by automating the tuning of distribution parameters ϕ.

Given the presence of three mass parameters in this scenario, the complete set of distribution parameters includes Δm for each mass. Δm represents the distance of the range extremes from the original value in the source domain.

$$\xi \sim \nu(\phi) \sim U(a, b) \sim U(m_t - \Delta m, m_t + \Delta m)$$

$$\phi = \{\Delta m\}$$

$$\Phi = \{\phi_0, \phi_1, \phi_2\} = \{\Delta m_0, \Delta m_1, \Delta m_2\}, \quad \Delta m_i \in [0.1, 5.0]$$

Gaussian Domain Randomization

As a second approach, we aimed to conduct Domain Randomization shaping the model parameters as Gaussian distributions. In this iteration, we provided increased flexibility to Bayesian Optimization by allowing greater freedom in selecting the mean parameter μ.

$$\xi \sim \nu(\phi) \sim \mathcal{N}(\mu, \sigma^2)$$

$$\phi = \{\mu, \sigma\}$$

$$\Phi = \{\phi_0, \phi_1, \phi_2\} = \{\mu_0, \mu_1, \mu_2, \sigma_0, \sigma_1, \sigma_2\}$$

$$\mu_i \in [0.1, 10.0], \quad \sigma_i \in [0.1, 6.5]$$

VII. Results and discussion

A. Lower and Upper bounds

The outcomes of the evaluations described in the above section are summarized in Table III, providing mean rewards.

Training domain	Testing domain	Mean Reward
Source	Source	1619.7
Source	Target	1019.2
Target	Target	1645.0

TABLE III
Lower and Upper bounds testing results

Optimal performance is achieved when training and testing the agent within the same domain. While this is often unfeasible in reality due to practical constraints outlined earlier in

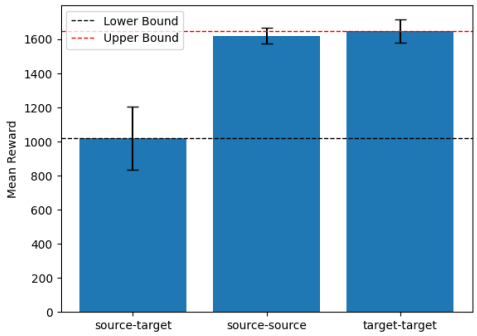

Fig. 7. The mean reward of the three combinations of training-testing in different domains, saving for later plots the lower (black) and the upper (red) bounds.

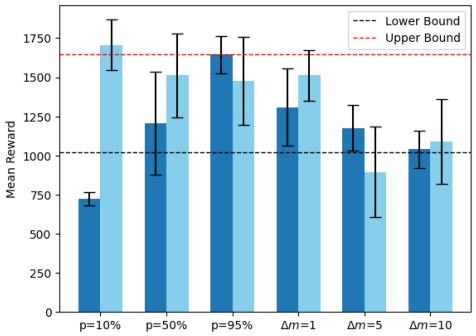

Fig. 8. UDR tests with different p and Δm, and for each of them the tests were made in the *target* domain (left, darker blue) and in their own *training* domain (right, lighter blue)

this paper, simulating a sim-to-real scenario with sim-to-sim allows us to conduct both training and testing in the target domain, thereby establishing an upper bound.

On the other hand, training on the source domain and testing on the target domain yields inferior results, serving as a lower bound for the problem. In this case, the agent's performance suffers because the torso mass used during testing is 1 kg heavier than the one it was trained with, thereby increasing the difficulty of the task.

Figure 7 and Table III displays the upper and lower bounds of this problem with two lines, which will serve as a benchmark for comparing the subsequent results in this paper.

B. UDR

UDR aims to enhance robustness against parameter variability, but it does so at the expense of performance. By analysing the bar-graph representation (Figure 8) where, for each randomization parameter, the test results on the target domain are shown on the left in dark blue, while on the right in light blue is shown the results of the training on the source domain.

In general the larger the sampling interval, the more the algorithm needs to adapt to accommodate a wider range of possible parameters. This behavior can be seen in the three rightmost dark blue bars: the tests results decrease proportionally as Δm increases.

However when the algorithm runs with the proportional interval, so using p, something interesting happens. Since the masses are quite low (Table II) the interval with $p = 10\%$ is not large enough to overcome the differences due to the domain transfer, furthermore some randomization is made during training, so its tests yields results that are even worst than the lower bound previously found. On the other hand, when p increases, the interval increases accordingly, leading to a better randomization of the domain, subsequently increasing the performances in the target domain; even matching the upper bound with $p = 95\%$.

The difference between Δm and p analysis shows how specifying a personalised interval for each mass noticeably increases the domain adaptation performance.

C. BayRn

Uniform Domain Randomization

Figure 9 illustrates the testing results on the target domain obtained during the Bayesian Optimization process. In this experiment, we conducted 5 initial training processes with randomly generated sets of distribution parameters, in order to initialize the Bayesian Optimization model followed by 50 training iterations.

Analyzing the outcomes it is evident that both exploitation and exploration phases were involved. Table IV presents the top 5 results achieved during this experiment. An overall increase in performance is observed, punctuated by exploration phases. Notably, the diverse set of parameters representing the best distribution parameters indicates that they were discovered through effective exploration of the search space.

Surprisingly some of the best results perform better than the upper bound.

Several training processes results to perform worse than the set lower bound reward. This behaviour could be addressed to the attempt to achieve more generalization on the derived policy by setting randomization levels either too high or too low. When randomization levels are set too high, the policy tends to exhibit overall random behavior, making it challenging to learn appropriate responses across diverse scenarios. Conversely, when randomization is too low, the policy not only doesn't generalise, but adds also noise in the training process.

By carefully navigating through both exploitation and exploration phases, the Bayesian Optimization process demonstrates

Δm_0	Δm_1	Δm_2	**Mean Reward**	**Reward Variance**
2.51	2.367	0.708	1714.3	50.3
3.427	0.541	2.327	1700.5	144.9
2.998	4.931	0.986	1646.2	85.7
1.13	1.713	2.643	1627.5	172.1
4.708	1.851	0.64	1625.2	50.8

TABLE IV
FIVE BEST RESULTS FOR THE HYPERPARAMETERS Δm_0, Δm_1 AND Δm_2 (IN KG). MEAN AND VARIANCE GIVEN BY POLICY EVALUATIONS IN THE TARGET DOMAIN

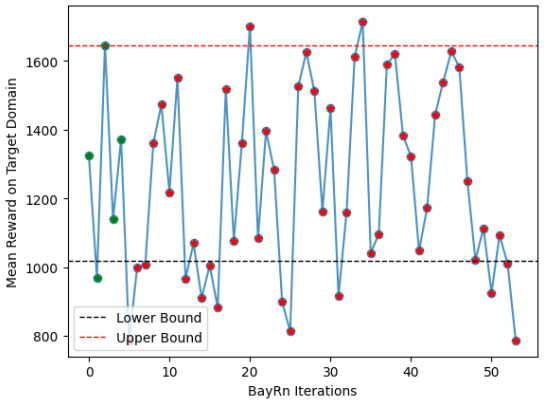

Fig. 9. BayRn Uniform Distribution parameters optimization process and testing results on target domain. Green points represent randomly generated initial parameters, red points represent Bayesian Optimization iterations.

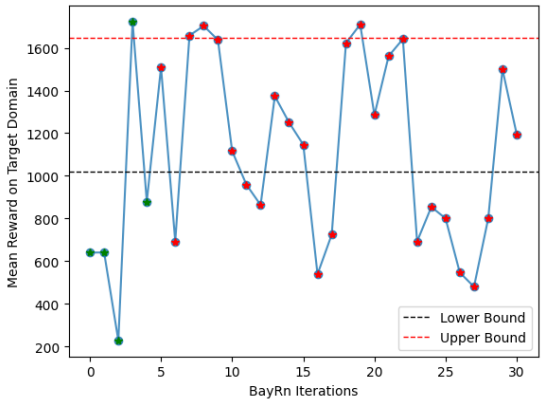

Fig. 10. BayRn Gaussian Distribution parameters optimization process and tasting results on target domain. Green points represent randomly generated initial parameters, red points represent Bayesian Optimization iterations.

its capability to progressively enhance performance, leading to improved outcomes in the target domain.

Gaussian Domain Randomization

Figure 10 illustrates the testing results on the target domain obtained during the Bayesian Optimization process. In this experiment, we conducted 5 initial training processes with randomly generated sets of distribution parameters, in order to initialize the Bayesian Optimization model followed by 30 training iterations.

Table V shows the top 5 results obtained during this experiment. Unlike the previous experiment, the resulting optimal parameters exhibit remarkable similarity. This suggests that despite some exploration phases, the process may have converged to a local optimum without discovering a significantly better solution. One possible explanation for this behavior could be attributed to the huge size of the search space. Bayesian Optimization, as per theory, tends to perform well when the domain of the target function is relatively small. However, in this scenario, the domain is $X = \mathbb{R}^6$, where each dimension is a subset of the real numbers. Such a large search space and dimensionality could pose challenges for the optimization process, potentially leading to difficulty in effectively exploring and exploiting the space to discover the global optimum.

μ_0	μ_1	μ_2	σ_0	σ_1	σ_2	**Mean R**	**R Var**
5.123	2.113	4.416	1.471	3.838	4.753	1723.0	137.7
5.264	1.828	4.563	1.604	3.62	4.574	1709.9	69.1
5.347	1.77	4.402	1.954	3.594	4.411	1703.0	33.2
5.351	1.926	4.408	1.746	3.717	4.608	1658.0	63.9
5.227	1.802	4.422	1.775	3.543	4.673	1641.3	9.1

TABLE V
FIVE BEST RESULTS FOR THE HYPERPARAMETERS μ_0, μ_1, μ_2, σ_0 , σ_1 AND σ_2 (IN KG) OF THE THREE MASSES. MEAN AND VARIANCE OF THE REWARD GIVEN BY POLICY EVALUATIONS IN THE TARGET DOMAIN

VIII. CONCLUSION

We have introduced two methods to overcome the reality gap inherent in sim-to-real transfer: **UDR** and **BayRn** with *Uniform Distribution* and *Gaussian Distribution*.

Firstly we tried to apply UDR, which proved to be a practical way to make the transferred policy more robust against poor parameters tuning and noisy measures used in simulations.

The effectiveness of general DR strategies is strongly affected by the choice of probabilistic distributions to randomize the source domain model.

BayRn propose a probabilistic way to find the best distribution parameters iterating over training processes performed with promising parameters, appropriately obtained thanks to the internal Bayesian Optimization process.

Its strength lies in its ability to refine its best predictions relying on some informations from the real world, thereby yielding more precise results than manually tuned DR.

This method though is much more computationally demanding since a full training phase is required at each iteration. Furthermore, every iteration of BayRn requires testing in the target domain, a task that may prove impractical in real-world applications due to constraints such as time or economic limitations. Additionally the real world usually do not present a metric testing, thus doesn't provide a reward feedback.

Nevertheless both UDR and BayRn techniques are valid solutions for Sim-to-Real problems. While UDR needs manual hyperparameter tuning, BayRn automatically discovers the optimal ones, given an easy interaction and analysis of the real environment.

REFERENCES

[1] Fabio Muratore, Christian Eilers, Michael Gienger, Jan Peters, "Data-efficient Domain Randomization with Bayesian Optimization"