An Example of Active Domain Randomization

Luca Faieta[1], **Stefano Barcio**[2]

[1] *Polytechnic University of Turin, Artificial Intelligence and Data Analytics*
E–mail: *s320174@studenti.polito.it*

[2] *Polytechnic University of Turin, Artificial Intelligence and Data Analytics*
E–mail: *s320174@studenti.polito.it*

Abstract

One of the most significant issues of Reinforcement Learning methods applied to robotics is the so called Reality Gap. Training an agent in a simulated environment brings a number of advantages, ranging from decreased cost to increased levels of security and scalability. However, policies trained in simulated environments, as good as they may be, usually show degradation in performance when tested in real-world applications. To bridge this gap, in this paper we analyze the main drawbacks of training on simulations and show two possible solutions to better transfer learned policies to real-world applications: Uniform Domain Randomization (UDR) and an instance of Active Domain Randomization (ADR). Finally, we test the two proposed solutions on the Mujoco Hopper environment to show their results.

Key words. Robots, Learning, UDR, ADR, Hopper, Reality Gap

1. Introduction

Reinforcement Learning has been one of the most popular way to train robotic agents for a while, because of its model-free approach to policy training. An agent trained with RL it's able to find nearly optimal solutions for a wide range of problems, without the need for complete knowledge of the environment that it works in, instead performing non-informed actions and collecting rewards from the environment. This proves to be particularly useful in situations where it's impossible to parametrize all the aspects of the environment, as it's the case in most robotics applications. However it's impossible, or rather too expensive and too risky, to train agents (i.e. robots) directly in real life. Due to the initially random nature of RL training pipeline, the risk of potentially catastrophic failures, especially in the first stages of training, is significant. To overcome this difficulty, recent literature has focused much on finding ways to accurately train an agent using mainly simulated environments. Again, the problem is that reality is too complex, and creating a simulator that is able to perfectly reproduce a real life scenario it's either too costly or just impossible. This creates a Domain Adaptation (DA) problem, where the shift from the source environment (the simulation) to the target environment for the agent (the real world) must be managed and reduced to obtain better performances. A possible solution, which is the one that we investigate in this paper, it's to focus on parameters distributions in simulation, making sure to train the agent on different enough scenarios, in order to make it able to generalize the environment and make it robust to the Reality Gap that exists between simulation and real application. This strategy takes the name of Domain Randomization. The first algorithm that we show is Uniform Domain Randomization, and we

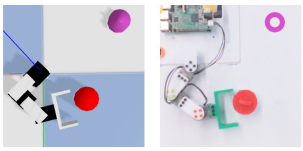

Fig. 1: An example of sim2real application in grasping objects.

then slightly modify it to obtain an instance of Active Domain Randomization. In the last sections of this paper we then test these algorithms on the Mujoco Hopper environment in a set of sim-to-sim experiments where the target environment has different parameters from the source, in order to simulate the Domain Adaptation problem. In the following sections we are going to further examine our methodology and describe the nature of our solutions in greater detail.

2. Related Works

In literature there have been many attempts to bridge the Reality Gap. In 2019, OpenAI proposed (3) the first example of Automatic Domain Randomization, generating a distribution over randomized environments of ever-increasing difficulty. Prior to that (2) in Mehta et al. proposed a more direct way to sample set of parameters, chasing the most difficult ones for the agent to learn instead of sampling randomly as UDR typically does. This was of loose inspiration for our implementation. Finally, (5) and (4) show two different ways to perform ADR, one adjusting its distribution by continuously testing the agent against the true environment in an online fashion, and the other working offline before the test, analyzing a pre-acquired set of real-environment distributions to gain knowledge about the target environment.

3. Methodology

3.1. Problem Formulation

Training a robot to perform a certain task basically equals to solve a decision-making problem involving sequential interactions with an external environment. It is common practice to modelize this kind of situation is using a Markovian Decision Process.

The primary objective is to determine an optimal policy $\pi^*(a|s)$ that maximizes the expected cumulative reward $J^{\hat{\xi}}(\theta, s_0)$ over time. This entails selecting actions based on the current state of the environment, with the goal of achieving the highest possible return.

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

Our MDP system is described by a tuple consisting of the following elements:

- the finite set of possible states S
- the finite set of possible A actions the agent can take
- a function, called transition matrix P, that defines the probability of transitioning from one state to another state given an action
- a reward R, that is a function that specifies the immediate reward or penalty received by the agent for taking a particular action in a particular state.
- a discount factor γ to reduce the weight (in terms of cumulative reward) of actions that are far in the future (or in the past).
- a set of random environment parameters ξ, drawn from an arbitrary distribution ν parametrized by θ

Notice that the cumulative reward is parameterized by the state, obviously, but also by the parameters of the θ distribution, meaning that we are trying to model the environment choosing distributions of parameters in such a way that maximizes the cumulative reward of the agent, in order to address the Domain Adaptation problem.

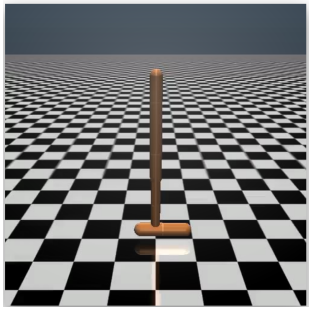

Fig. 2: Image of the Mujoco Hopper.

3.2. The Mujoco Hopper Environment

The Mujoco Hopper environment offers a compelling space for exploring locomotion dynamics and control strategies within reinforcement learning (RL). Comprised of four key anatomical segments—the torso, thigh, leg, and a singular foot acting as its fulcrum—the hopper's primary objective is to propel forward motion through precise torque application at the interconnected joints.

Observations within the environment consists of positional coordinates and velocities of the hopper's elements, presented as an 11-dimensional continuous dataset. The action space, on the other hand, is also continuous, with each scalar action representing torque applied at specific hinge joints, thereby capturing the range of interactions within the system.

The reward mechanism is designed to incentivize actions that keep the hopper standing in a safe state, while simultaneously trying to jump forward at the maximum horizontal speed. The reward is composed by three distinct factors, each one of them evaluated at the end of an episode:

- *healthy_reward*: a reward for being in a *safe* state: it means the hopper is not in a position that would make it fall
- *forward_reward*: a reward for the horizontal space that it traversed during the time between

actions.

$$fr = weight \cdot (x\_after - x\_before)/dt \quad (1)$$

- *control_cost*: a parameter to penalize actions that are too large

Overall, the reward of each actions is composed by the sum of the three components mentioned before:

$$reward = healthy\_reward + fr + cc \quad (2)$$

3.3. Uniform Domain Randomization

The first solution that we propose in this paper is Uniform Domain Randomization. UDR is a widely used technique to solve Domain Adaptation problems. Instead of relying solely on training in a single fixed environment, uniform domain randomization involves training agents in a wide range of simulated environments with randomized parameters. By systematically varying these parameters during training, agents are exposed to a diverse set of environments that simulate different possible scenarios they might encounter in the real world. This randomness is typically applied uniformly across the parameter space, ensuring that the agent encounters a balanced distribution of environmental conditions. This helps prevent overfitting to specific environmental characteristics and encourages the learned policy to generalize well to unseen situations. The paramaters distribution used in the Markov Decision Process can be modified as follows:

$$\xi \sim \nu(\phi) \sim U(a,b) = \begin{cases} 1/(b-a) & \text{if } a \leq x \leq b \\ 0 & \text{otherwise} \end{cases} \quad (3)$$

The most significant issue while using this kind of method is the range of the uniform distributions from which we sample each parameter. Ranges too large could generally be robust in terms of generalization but usually show difficulties in convergence, while a too narrow range would not provide the necessary generalization to the environment. We further discuss the choice of this parameters for our application in the Experiments section of this paper. The steps of this strategy are shown in Algorithm 1

3.4. Active Domain Randomization

The concept of Active Domain Randomization (ADR) has been widely discussed in the literature (2) (3), representing a strategy that involves guiding the process of parameter randomization to effectively train models on a more diverse and meaningful set of parameters. While the specifics of how this algorithm guides the sampling process and defines a "significant" set of parameters can vary depending on the application, the final goal remains consistent: to enhance the robustness and adaptability of machine learning models, particularly in the context of domain adaptation problems.

Algorithm 1 Uniform Domain Randomization (UDR)

1: **Input:** Number of training episodes N, Environment parameters P
2: Initialize policy π with random parameters
3: **for** $i = 1$ to N **do**
4: Randomize environment parameters: $\xi \leftarrow$ sample_parameters($\nu(\theta)$)
5: Initialize environment with parameters ξ
6: Initialize episode
7: **while** episode not finished **do**
8: Select action a_t using policy π
9: Execute action a_t, observe next state s_{t+1} and reward r_t
10: Update policy parameters using reinforcement learning
11: **end while**
12: **end for**
13: **Output:** Trained policy π

In our proposed instance of Active Domain Randomization tailored for our Domain Adaptation problem, we draw inspiration from the principles of Uniform Domain Randomization (UDR). However, instead of randomizing the parameters using a uniform distribution, we introduce a Gaussian distribution to govern the parameter sampling process.

Mathematically, we express this as follows:

$$\xi \sim \nu(\phi) \sim \mathcal{N}(\mu_\xi, \sigma_\xi^2) \quad (4)$$

where $\nu(\phi)$ represents the distribution governing the parameters ξ, and μ_ξ and σ_ξ^2 denote the mean and variance of the Gaussian distribution, respectively.

The Gaussian distribution is defined by its mean μ_m and variance σ_m^2, which are periodically updated, specifically after every N timesteps during training. This update process entails sampling new values for both the mean and standard deviation following a test of the agent against the target environment. This strategy allows the agent to receive indirect feedback from the environment in which it will ultimately be tested. Consequently, the agent can adjust its parameter distributions to regions within the search space that are closer to the target environment. These regions, which yield better rewards when tested on the target environment, can be deemed more significant for the training process.

To implement this sampling process, we construct a Gaussian distribution centered around a mean μ_m and with a fixed standard deviation σ_m^2:

$$\mu_\xi \sim \mathcal{N}(\mu_m, \sigma_m^2)$$

Similarly, we update the standard deviation σ_ξ^2 by sampling from another Gaussian distribution with mean μ_s and variance σ_s^2:

$$\sigma_\xi^2 \sim \mathcal{N}(\mu_s, \sigma_s^2)$$

By incorporating these adaptive sampling mechanisms into our Active Domain Randomization framework, we aim to enhance the model's ability to adapt to the target environmental conditions without ever directly accessing them, and improve its overall performance in real-world scenarios. This approach facilitates more effective domain adaptation by enabling the model to learn from a more significant range of parameter distributions, ultimately leading to better generalization and robustness.

Algorithm 2 Train Model

Output: Trained Model
Initialize M and S to initial values
while $n_{\text{timesteps}} > 0$ **do**
    Sample μ and σ from normal distributions
    for each batch **do**
        Train $model(\mu, \sigma)$ using SAC
        Obtain reward from testing on ϕ_t
        while reward $<$ threshold **do**
            Retrain $model(\mu, \sigma)$ using SAC
            Obtain reward from testing on ϕ_t
            Decrease $n_{\text{timesteps}}$
        end while
        if reward $>$ best_reward **then**
            Update M, N
            Decrease $n_{\text{timesteps}}$
        end if
    end for
end while
Return Trained Model

4. Experiments

4.1. Experimental Setup

We conducted our experiments using two Mujoco Hopper environments, denoted as Environment Source and Environment Target. Both environments are identical in structure and dynamics, simulating a two-dimensional bipedal robot with four body segments: torso, thigh, leg, and foot. The objective of the hopper is to achieve forward locomotion by applying torques to the joints connecting the body segments.

In Environment Source, we introduced a mass perturbation by decreasing the mass of the torso by 1 kg. Environment Target served as the baseline, retaining the default mass configuration.

We trained in the finals stages our agent for one million timesteps testing on 50 episodes tests on the target enviroment

Additionally, we employed Uniform Domain Randomization (UDR) to introduce variations in the environment dynamics. Specifically, we randomized the masses of the thigh, leg, and foot segments sampling values from a uniform distribution. We also randomized the angle of the floor sampling the degrees from another uniform distribution to simulate variations in terrain inclination.

Table 1: Table with the masses in our Hopper enviroments.

Enviroment	Torso	Thight	Leg	Foot
Source:	2.53	3.92	2.71	5.09
Target:	3.53	3.92	2.71	5.09

Our study incorporates an innovative Active Domain Randomization technique, which refines the selection of randomization parameters. Through an iterative process, we search for the most fitting normal distribution observed during training, guided by the average reward obtained from a fifty-episode test. At each step of this iterative process, we adjust the mean of the normal distribution to mirror the best-performing values encountered. This refinement ensures that the randomization parameters evolve in tandem with the enhancements in the learning algorithm's performance. By dynamically tailoring the distribution of randomized parameters, our approach significantly improve the adaptability and resilience of the learned policies against fluctuations in environment dynamics.

4.2. Training Procedure

We employed the Soft Actor-Critic (SAC) (1) algorithm for training policies in both Mujoco Hopper environments. The SAC algorithm is well-suited for continuous control tasks, offering stability and sample efficiency. We trained the policies using a distributed training setup with parallel actor-learner instances.

The training process consisted of iteratively collecting trajectories by executing policies in the Mujoco Hopper environments. We utilized an experience replay buffer to store and sample past experiences for policy updates. The SAC algorithm optimized policy parameters to maximize the expected cumulative reward, incorporating entropy regularization to encourage exploration.

To evaluate policy transferability, we trained policies separately in Environment Source and Environment Target for a fixed number of training iterations. We then evaluated the performance of these policies in both environments using the average reward obtained through 50 episodes of test.

4.3. Evaluation Metrics

We assessed the performance of trained policies using the mean reward obtained from a 50 episode test as an evaluation metric to quantify the effectiveness in achieving forward locomotion and maintaining stability. Key components of the reward in the Hopper enviroment included:

- Average Forward Velocity: The mean velocity of the hopper in the forward direction over a fixed time horizon, indicative of its locomotion efficiency.
- Energy Consumption: The cumulative energy expended by the hopper during locomotion, calculated as the integral of torque over time.
- Stability Metrics: Various stability metrics, such as the standard deviation of joint angles and the frequency of falls, were used to evaluate the robustness of policies to disturbances.

Another metrics used to evaluate the performance was setting the upper and the lower bounds for our tests using the result of an agent trained for one million timesteps on the Source enviroment as a lower bound and the result of an agent trained on a million timesteps on the Target eviroment.

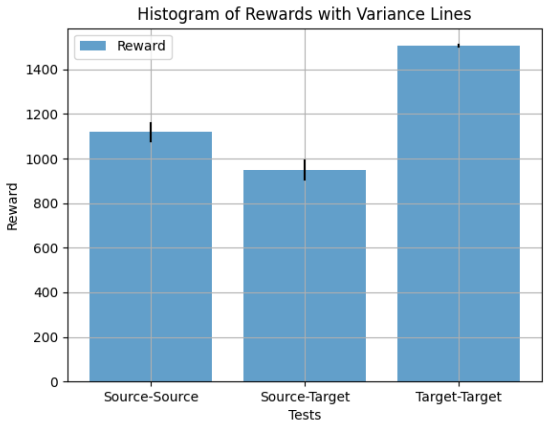

Fig. 3: The baseline of our tests.

4.4. Hyperparameters Tuning

We conducted hyperparameters tuning to optimize the performance of the Soft Actor-Critic (SAC) algorithm for our specific problem, we searched for the best combination of learning rate and discounted factor gamma. We searched our best configuration simulating some combination directly in our problem framework on 500000 timesteps training each and measure them up on the average reward obtained on a 50 episodes test. Results of this experiment can be seen in Fig.4

The best two results are:learning rate of 0.001 along a gamma of 0.95 and learning rate equal to 0.003 with a gamma of 0.99; we then tested the training performance with one million timesteps of these two configuration head to head in order to help our choice between them, showing that using the combination learning rate of 0.003 and gamma of 0.99 is clearly the most suitable for our problem achieving better results than the other (Fig. 5).

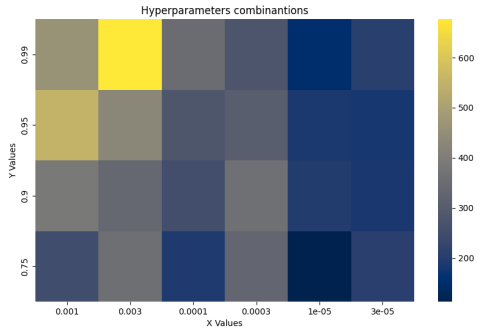

Fig. 4: Combinations of different values of learning rate and discount factor, associated with their reward.

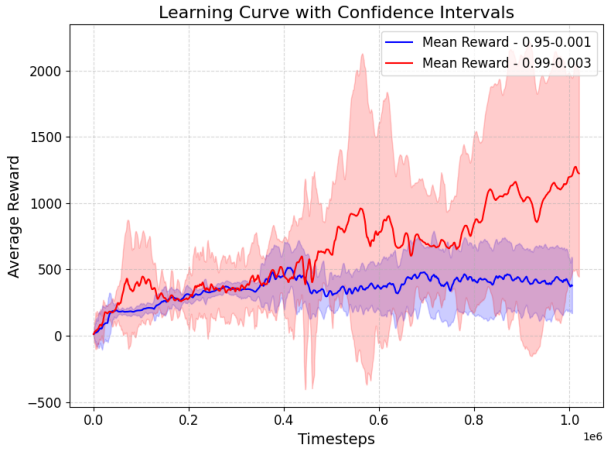

Fig. 5: The two learning curve compared.

5. Results

We present the results of our experiments, comparing the performance of policies trained with and without mass perturbations, UDR randomization, and our ADR implementation. We analyze the effects of these factors on policy transferability, robustness, and generalization across Mujoco Hopper environments. We experimented both of our randomization on a range of value to asses the best fit for our problem also combining them together.

5.1. UDR

We tried some combinations of randomization changing the intervals of sampling of our parameters,masses and angles. The best result obtained was obtained by a wide only inclinaton floor randomization that enabled the Hopper to obtain the best results, this is due the nature of the inclination floor that changes the masses interaction with the enviroment and between themselves but keeping the intrinsec nature of the body without any major imbalances

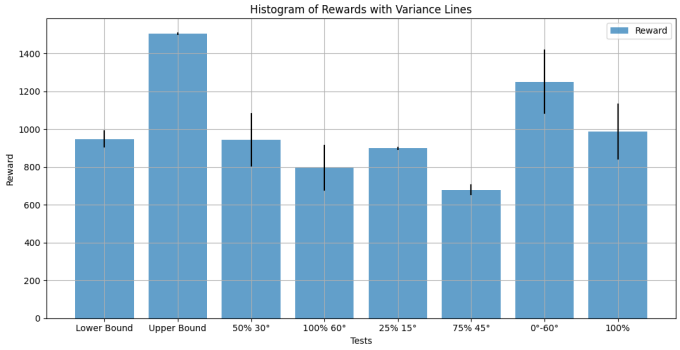

Fig. 6: UDR experiments rewards.

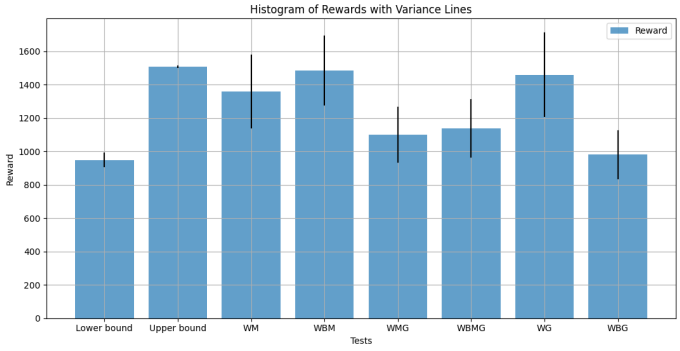

Fig. 7: ADR experiments rewards.

genereted by the randomization.

The results also highlight that models with a lower degree of randomization (e.g. only randomizing the masses and not the angle) show better performances, while more random models exhibit low convergence and generically worse results in the task at hand.

5.2. ADR

Here we present the results of our iterative search style algorithm guiding the UDR process to more profitable zone of our solution space, we tried different setups of implementations combining masses and angle of the floor randomization and the plain iterative search. The results show that if guided the mass randomization is more efficient the inclination floor ones, with results in the opposite direction with the UDR ones, also here we confirm that randomizing both parameters at the same is not an efficient way to solve the problem maintaining the convergence problems encountered before. These results also show an enhancement in performances improving in the worst parameters configurations.

5.3. Conclusions

We propose two algorithms to solve a Domain Adaptation problem on the Mujoco Hopper environment. The first one, Uniform Domain Randomization, makes use of uniform parameters sampling to explore the solution space. The second one is an instance of Active Domain Randomization: an informed algorithm that moves in the search space in an active way, rather than just randomly, and that is able to be robust on a vast range of parameters. This is just a quite rough implementation of two concepts that, in our opinion, could deserve to be subject of future studies and could yield interesting results also for other robotics applications.

6. Acknowledgements

We would like to thank the teachers of the Robot Learning course at Politecnico di Torino, a.y. 2023/24, and all their teacher assistants. The topics of the course have been really interesting to study and try first-hand, and that's also thanks to their dedication and their availability in assisting us during this work .

REFERENCES

[1] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. 2017.

[2] Bhairav Mehta, Manfred Diaz, Florian Golemo, Christopher Pal, and Liam Paull. Active domain randomization. 2019.

[3] Marcin Andrychowicz Maciek Chociej Mateusz Litwin Bob McGrew Arthur Petron Alex Paino Matthias Plappert Glenn Powell Raphael Ribas Jonas Schneider Nikolas Tezak Jerry Tworek Peter Welinder Lilian Weng Qiming Yuan Wojciech Zaremba Lei Zhang OpenAI, Ilge Akkaya. Solving rubik's cube with a robot hand. *OpenAI blog*, 2019.

[4] Gabriele Tiboni, Karol Arndt, and Ville Kyrki. Dropo: Sim-to-real transfer with offline domain randomization. *Robotics and Autonomous Systems*, page 104432, 2023.

[5] Viktor Makoviychuk Miles Macklin Jan Issac Nathan Ratliff Dieter Fox Yevgen Chebotar, Ankur Handa. Closing the sim-to-real loop: Adapting simulation randomization with real world experience. 2018.