Project Robot Learning

1st Rialti Jacopo
LM Ing. Informatica
Politecnico di Torino
Torino, Italy
s346357@studenti.polito.it

2nd Giunti Alberto
LM Ing. Informatica
Politecnico di Torino
Torino, Italy
s336374@studenti.polito.it

3rd Gjinaj Stiven
LM Ing. Informatica
Politecnico di Torino
Torino, Italy
s333805@studenti.polito.it

***Abstract*—This project explores sim-to-real transfer in robotics, focusing on training control policies in simulation and applying them to real-world systems. Using reinforcement learning algorithms like SAC alongside Uniform Domain Randomization, the study aims to improve policy robustness and minimize performance loss in the target domain. The results highlight challenges and opportunities in transferring robust policies.**

***Index Terms*—sim-to-real transfer, reinforcement learning, domain randomization, robotics**

I. Introduction

Sim-to-real transfer is a critical problem in robotics, where policies trained in simulation must perform effectively in real-world environments despite differences in dynamics. Domain randomization helps address these discrepancies by training policies over a variety of simulated conditions to improve robustness. This project implements RL algorithms and UDR to evaluate their effectiveness in a simulated sim-to-sim transfer scenario, introducing controlled discrepancies between training and testing domains.

II. Hopper Environment and Analysis

The Hopper environment, based on MuJoCo, is a single-legged robot tasked with learning to jump and achieve the highest possible horizontal speed without falling. Training policies on this environment involves navigating a continuous state and action space, which is well-suited for reinforcement learning algorithms.

In the simulation, the environment models the dynamics of the robot with specific parameters:

A. State Space

The state space of the Hopper environment is continuous, consisting of 11 variables that represent the robot's joint positions, velocities, and potentially other sensory data. These variables describe the system's current configuration and dynamics, which are crucial for decision-making during training.

Identify applicable funding agency here. If none, delete this.

TABLE I
State Space Variables of the Hopper Environment

Obs.	Name	Joint	Unit	Range
0	height of hopper	rootz	position (m)	$(-\infty, \infty)$
1	angle of the top	rooty	angle (rad)	$(-\infty, \infty)$
2	thigh joint angle	thigh_joint	angle (rad)	$(-\infty, \infty)$
3	leg joint angle	leg_joint	angle (rad)	$(-\infty, \infty)$
4	foot joint angle	foot_joint	angle (rad)	$(-\infty, \infty)$
5	top velocity (x)	rootx	velocity (m/s)	$(-\infty, \infty)$
6	top velocity (z)	rootz	velocity (m/s)	$(-\infty, \infty)$
7	top angular velocity	rooty	ang. vel. (rad/s)	$(-\infty, \infty)$
8	thigh ang. velocity	thigh_joint	ang. vel. (rad/s)	$(-\infty, \infty)$
9	leg ang. velocity	leg_joint	ang. vel. (rad/s)	$(-\infty, \infty)$
10	foot ang. velocity	foot_joint	ang. vel. (rad/s)	$(-\infty, \infty)$

[a]Values describe system state continuously.

B. Action Space

The action space is also continuous and consists of three variables, each corresponding to the torques applied to the robot's actuators. These torques directly control the movement of the Hopper's joints, enabling it to jump and maintain balance.

TABLE II
Action Space Variables of the Hopper Environment

Num	Action	Control Range	Name	Joint	Unit
0	Torque on thigh	$[-1, 1]$	thigh_joint	hinge	torque (N m)
1	Torque on leg	$[-1, 1]$	leg_joint	hinge	torque (N m)
2	Torque on foot	$[-1, 1]$	foot_joint	hinge	torque (N m)

C. Mass Values and Dynamics

In the source variant of the environment, the masses of the robot's components are as follows:

TABLE III
Mass Values of the Hopper Environment Components

Component	Source Mass (kg)	Target Mass (kg)
Torso	2.5343	3.5343
Thigh	3.9270	3.9270
Leg	2.7143	2.7143
Foot	5.0894	5.0894

The target variant introduces a 1 kg shift in the torso mass to simulate the reality gap. This controlled discrepancy, tests

the robustness of policies when transferring from the source to the target domain.

D. Environment Dynamics and Behavior

The Hopper environment defines its components as *world*, *torso*, *thigh*, *leg*, and *foot*, with 6 degrees of freedom distributed across the robot's joints. The simulation terminates episodes based on specific criteria such as falling or achieving the maximum allowable timesteps, and resetting the environment ensures consistent training conditions.

This analysis highlights the challenges of developing robust policies that can handle variations in dynamics parameters, making it a valuable benchmark for testing reinforcement learning approaches in sim-to-real transfer scenarios.

III. Implementation of Lower/Upper Bound Baselines

A. Setup and Environment

We used the Stable-Baselines3 library to train the agent with the Soft Actor-Critic algorithm. SAC is an off-policy reinforcement learning algorithm designed for continuous action spaces. It optimizes a stochastic policy using a maximum entropy framework, encouraging exploration by adding an entropy term to the reward. This makes it particularly suitable for environments like the Hopper, where robust exploration is critical for learning effective policies.

The custom Hopper environment was implemented with the following setup:

- A source domain `CustomHopper-source-v0`, where the torso mass is set to 2.5343 kg.
- A target domain `CustomHopper-target-v0`, where the torso mass is shifted to 3.5343 kg to simulate the reality gap.

Using the `args` configuration, we dynamically selected whether to train on the source or target environment, and set the desired number of training timesteps.

B. Training Procedure

The training was conducted as follows:

1) Define hyperparameters for the SAC model, including learning rate, batch size, and discount factor.
2) Train the model in the specified domain (source or target) for a total number of timesteps.
3) Evaluate the trained model using the Stable-Baselines3 `EvalCallback`, which logs performance metrics and saves the best-performing model.
4) Save evaluation results, including rewards and timesteps, for further analysis.

C. Visualization and Analysis

To analyze the performance, we plotted episodic rewards over training timesteps. The rewards were processed using a moving average with a window size of 50, to smooth out fluctuations and provide a clearer view of overall trends. The generated reward curve highlights:

- Reward trends over training timesteps.
- The smoothed mean reward trajectory to evaluate overall policy improvement.

The resulting plots were saved in the specified directory for later review, providing a visual representation of how the reward improved as training progressed.

D. Code Implementation

The implementation details are summarized in the following steps:

- Environment creation using `gym.make()`, dynamically selecting the domain (source or target).
- Training with the SAC algorithm, leveraging Stable-Baselines3 for efficient policy optimization.
- Logging and saving model checkpoints for reproducibility.
- Visualization of reward trends to assess training stability and performance.

E. Results with Default Hyperparameters

To further evaluate the model's performance, we trained two models for 500,000 timesteps using default hyperparameters:

- **Hyperparameters:**

```
learning_rate: 3e-4;
batch_size: 256;
buffer_size: 1000000;
tau: 0.005;
gamma: 0.99;
train_freq: 1;
gradient_steps: 1;
learning_starts: 10000;
ent_coef: 'auto'.
```

- **Training Setup:**
  - Model 1: Trained on the `source` environment.
  - Model 2: Trained on the `target` environment.

The following plots showcase the reward trends for the two models over the training timesteps. Each plot represents the episodic rewards and their smoothed mean using a moving average with a window size of 50.

Fig. 1. Reward trends for `CustomHopper-source-v0` environment.

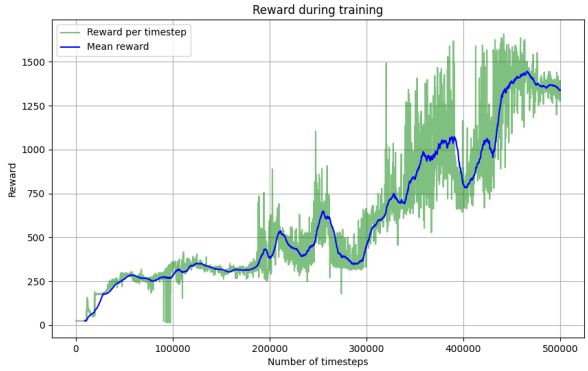

Fig. 2. Reward trends for `CustomHopper-target-v0` environment.

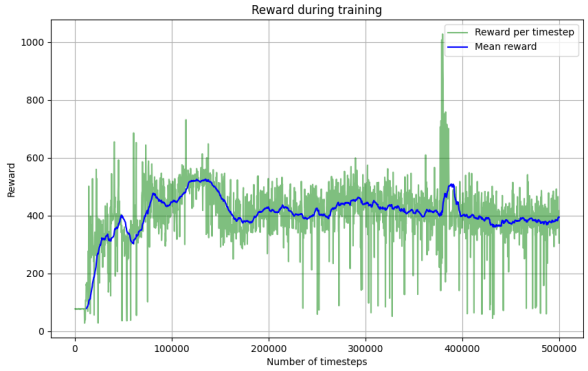

Fig. 3. Reward progression for the SAC model trained with Optuna-suggested hyperparameters.

These plots demonstrate how the reward improves over time, indicating successful policy training in both environments. The difference in performance between the source and target domains highlights the impact of the simulated reality gap.

F. Results with Hyperparameters Hyperparameters Suggested by Optuna

Optuna is an automatic hyperparameter optimization framework that allows users to efficiently find optimal configurations for their machine learning models. It uses a technique called Bayesian optimization, which iteratively evaluates the performance of different hyperparameter sets to maximize or minimize a defined objective function.

*1) **Optimization Procedure**:* For this experiment, we utilized Optuna to optimize the hyperparameters for the SAC algorithm. The optimization process consisted of:

- Running 20 independent trials, each with a training duration of 30,000 timesteps.
- Beginning with default hyperparameters and iteratively modifying them based on observed results.
- Using the performance metrics from each trial to guide subsequent parameter adjustments, with the goal of maximizing the reward.

Once the optimal hyperparameters were identified, they were applied to train a new SAC model for 500,000 timesteps to maintain consistency with the previously trained models.

*2) **Results and Analysis**:* The resulting reward plot for the model trained with Optuna-suggested hyperparameters did not meet the expectations. Unlike the models trained with default hyperparameters, the reward curve exhibited a significantly lower performance, with an average reward stabilizing around 400. This outcome suggests that the Optuna-suggested parameters might not generalize well to this specific environment or task, highlighting the complexity of hyperparameter tuning in reinforcement learning.

The following plot illustrates the training progression for the model trained with Optuna-suggested hyperparameters:

The results indicate that while Optuna provides a robust framework for hyperparameter optimization, the optimal settings it identifies, may not always translate into better performance for all scenarios. The lower reward values and slower learning observed in this case, underscore the importance of validating suggested hyperparameters within the context of the specific environment and task.

IV. Evaluation of Source and Target Models in Different Domains

To evaluate the trained models, we tested their performance under four configurations:

- **Source → Source**: The model trained in the source environment is tested in the same source domain.
- **Source → Target (Lower Bound)**: The source-trained model is tested in the target environment to simulate the sim-to-real transfer scenario.
- **Target → Target (Upper Bound)**: The model trained in the target environment is tested in the same target domain.
- **Target → Source**: The target-trained model is tested in the source environment.

The evaluation was conducted using the Stable-Baselines3 `evaluate_policy` function, which calculates the average return over 50 test episodes. Additionally, we recorded videos of the agent's performance to visualize the results.

*1) **Evaluation Results**:* The following table summarizes the results for the different configurations:

Configuration	Mean Reward	Standard Deviation
Source → Source	1604.10	2.15
Source → Target (Lower Bound)	1210.71	296.44
Target → Target (Upper Bound)	1326.08	1.82
Target → Source	1171.93	4.69

TABLE IV
Mean rewards and standard deviations.

A. Analysis of Results

*1) **Source → Source**:* The source model performs best when tested in its own domain, achieving a high mean reward of 1604. This is expected, as the model is optimized for this environment.

*2) **Source → Target (Lower Bound)**:* The mean reward drops significantly to 1210 when the source model is tested in the target domain. This decrease highlights the impact of the reality gap, as the model is not optimized for the increased torso mass in the target environment. The changes in dynamics affect state-action correlations learned in the source domain, reducing the model's effectiveness. Furthermore, the high standard deviation (296.44) indicates inconsistent performance across episodes, likely due to the challenges introduced by the domain discrepancy.

*3) **Target → Target (Upper Bound)**:* The target model achieves a mean reward of 1326 in its own domain, which is lower than the source model's performance in the source domain but more consistent (standard deviation of 1.82). This reflects the target environment's increased difficulty due to the heavier torso mass.

*4) **Target → Source**:* When the target-trained model is tested in the source environment, the performance is slightly lower than in the Target → Target configuration (1171 vs. 1326). This suggests that the target model has adapted to the higher mass, potentially overfitting to the target domain's dynamics, which may reduce its effectiveness in the source domain.

B. Problems of sim-to-real

In a sim-to-real setting, training directly on the target environment (real world) is often infeasible due to several constraints:

- **Safety Concerns:** The agent may take unsafe actions that could damage hardware or pose risks in a real-world setting.
- **Cost of Training:** Training in the real world requires significant time and resources, which may be impractical for iterative learning.
- **Lack of Feedback:** The real-world environment may not provide as detailed feedback as simulated environments, making optimization more challenging.

C. Discussion on Similar Results

Interestingly, the results for some configurations, such as Target → Source and Source → Target, are relatively close. This could be due to:

- **Partial Overlap in Dynamics:** Both environments share similar underlying dynamics, so the models retain some generalization capability.
- **Robustness of SAC:** The Soft Actor-Critic algorithm's ability to handle continuous action spaces and stochastic policies may contribute to better-than-expected performance across domains.

V. Implementation of Uniform Domain Randomization (UDR)

A. Overview of UDR

Uniform Domain Randomization (UDR) is a method used to improve the robustness of reinforcement learning policies by introducing variations in the dynamics of the training environment. The primary objective of UDR is to ensure that the learned policy can generalize across a wide range of environments, effectively mitigating the effects of domain shift between the training (source) and testing (target) environments.

In our implementation, we applied UDR to the Hopper robot environment by varying the masses of its components, excluding the torso mass. This approach forces the agent to learn a policy that is robust to these variations, improving its adaptability when deployed in the target environment.

B. Implementation Details

To implement UDR, we modified the `CustomHopper` environment to allow randomization of the masses of the thigh, leg, and foot components at the beginning of each episode. The torso mass was kept fixed to preserve the domain shift characteristic between the source and target environments.

The randomization process was implemented as follows:

- For each episode, the masses of the thigh, leg, and foot were sampled from a uniform distribution centered around their original values. The sampling ranges were defined as $\pm 50\%$ of the original masses.
- The random masses were applied to the Hopper robot's simulation model at the start of each episode using the `set_parameters` method.
- This randomization was activated by setting the `udr` flag in the environment configuration.

The modified environment was registered as `CustomHopper-dr-v0`, and the randomization logic was encapsulated in the `set_random_parameters` and `sample_parameters` methods.

C. Training Process

We trained a Soft Actor-Critic model on the `CustomHopper-dr-v0` environment for a total of 500,000 timesteps. The training was conducted with the following hyperparameters:

- **Learning Rate:** 3×10^{-4}
- **Batch Size:** 256
- **Buffer Size:** 1,000,000
- **Discount Factor (γ):** 0.99
- **Target Smoothing Coefficient (τ):** 0.005

During training, evaluation was performed at regular intervals to monitor the model's performance.

D. Results and Observations

After training, the UDR-trained policy was tested on both the source and target environments. The mean episodic rewards obtained were compared with the results from the naive `source→target` configuration to assess the effectiveness of UDR. The results are summarized in Table V.

TABLE V
MEAN AND STANDARD DEVIATION OF EPISODIC REWARDS FOR DIFFERENT CONFIGURATIONS

Configuration	Mean Reward	Standard Deviation
Naive Source → Source	1604.10	2.15
Naive Source → Target	1210.71	296.44
UDR-Source → Source	1007.90	107.43
UDR-Source → Target	1488.71	13.47

*1) **Comparison with Naive Configuration**:* The UDR-trained policy demonstrated superior robustness when tested on the target environment. Compared to the naive configuration, where the source-trained policy achieved a mean reward of 1210.71 on the target environment with high variance (296.44), the UDR-trained policy achieved a much higher mean reward of 1488.71 with significantly reduced variance (13.47). These results highlight the ability of UDR to adapt effectively to variations in the target environment.

On the source environment, the naive configuration outperformed the UDR-trained policy in terms of mean reward (1604.10 vs. 1007.90). However, the primary goal of UDR is to enhance generalization to the target environment, where its advantages are evident.

*2) **Training Reward Progression**:* Figure 4 shows the reward progression during training for the UDR-trained policy. The rewards increased steadily, reaching approximately 1500, with a low variance throughout training. This stability contrasts with the higher variability observed in the naive configuration.

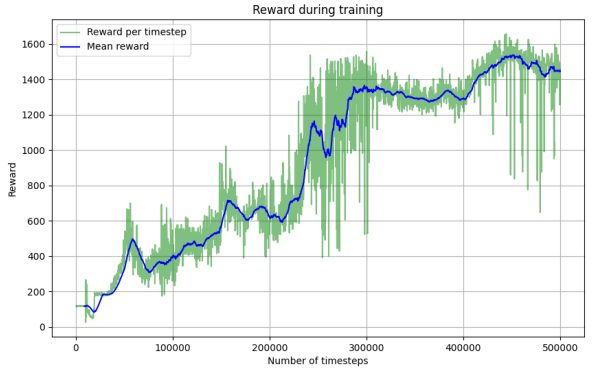

Fig. 4. Reward progression during training for the UDR-trained policy. The plot shows a steady increase in reward with low variance.

E. Limitations of UDR

While UDR proved effective in mitigating the effects of unmodeled dynamics shifts, it has certain limitations:

- **Computational Overhead:** The process of sampling and applying randomized parameters increased the computational complexity of training.
- **Distribution Design:** The choice and tuning of randomization ranges play a critical role in determining performance and require manual effort.
- **Incomplete Generalization:** UDR might not account for all variations in the target environment, especially if the randomization ranges fail to encompass critical dynamics.

VI. INTRODUCTION

Our research on the Hopper environment provides insights into sim-to-real transfer challenges in robotic systems and the effectiveness of domain randomization techniques. The experimental results demonstrate several key findings. The impact of domain shift is significant, as evidenced by the performance drop when transferring policies from source to target environments. The source model's performance decreased from when tested in the target domain, highlighting the challenges of the reality gap. UDR proved highly effective in improving policy robustness. The UDR-trained policy significantly outperformed the naive transfer approach while also showing much lower variance. This highlights UDR's ability to generate more stable and generalizable policies. While UDR-trained policies showed lower performance in the source environment compared to specialized policies ,they demonstrated superior generalization to the target domain, which is crucial for real-world applications. These experiments contribute to the broader understanding of sim-to-real transfer in robotics and provide practical insights for developing more robust control policies for real-world robotic systems.

REFERENCES

[1] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction (Second Edition)*.

[2] J. Kober, J. A. Bagnell, and J. Peters, "Reinforcement learning in robotics: A survey," *The International Journal of Robotics Research*, 2013.

[3] P. Kormushev, S. Calinon, and D. G. Caldwell, "Reinforcement learning in robotics: Applications and real-world challenges," 2013.

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

[5] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel, "Domain Randomization for Transferring Deep Neural Networks from Simulation to the Real World," arXiv, Mar. 20, 2017.

[6] X. B. Peng, M. Andrychowicz, W. Zaremba, and P. Abbeel, "Sim-to-real transfer of robotic control with dynamics randomization," 2018.

[7] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor."

[8] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," 2017.