Learning how to walk on a Mars Yard with Reinforcement Learning

Robot Learning Final Project

1st Alessio Giorgetti
Politecnico di Torino
Matricola: 345637
S345637@studenti.polito.it

2nd Anna Roma
Politecnico di Torino
Matricola: 345819
anna.roma@studenti.polito.it

3rd Stefano Giulianelli
Politecnico di Torino
Matricola: 347911
stefano.giulianellli@studenti.polito.it

***Abstract*—This project focuses on learning control policies for simulated robots using advanced Reinforcement Learning (RL) algorithms, with particular attention to policy transfer from simulation to reality. Training was conducted on a monopod robot within the Gym Hopper environment, specifically adapted to tackle complex terrains inspired by the Mars Yard. The Domain Randomization technique was employed to enhance the Hopper's generalization ability by introducing variations in environmental parameters during the training process. The results indicate that using Domain Randomization significantly increases the effectiveness of policy transfer, helping to mitigate the impact of the gap between the training and testing domains. The project extension involves simulating complex Martian terrains to further evaluate the Hopper's adaptability, demonstrating that the adopted techniques improve robustness under dynamic and unpredictable conditions.**

***Index Terms*—Reinforcement Learning, Hopper, Sim To Real**

I. Introduction

This project explores the learning of control policies for simulated robots using advanced Reinforcement Learning (RL) algorithms and analyzes the challenges of applying these policies in the real world. The training environment chosen for the robot is Gym Hopper (see the documentation at [3]), a simulated model of a mono-pod robot designed to learn how to jump without falling while maximizing horizontal speed. The Hopper has a single articulated limb, including an ankle, knee, and hip, each controlled by actuators. The complexity of the control lies in the Hopper's need to maintain dynamic balance while generating effective movement, addressing a combination of stability and propulsion challenges.

Our project focuses specifically on the issue of *sim-to-real transfer*, which involves adapting policies learned in simulation for deployment on real hardware, thereby reducing costs and risks. In this project, the transfer is simulated using a *sim-to-sim* approach, introducing a controlled discrepancy between the training domain (*source*) and the test domain (*target*). When transitioning from a *source* to a *target* environment, a discrepancy between the two domains inevitably arises, which can compromise the effectiveness of the learned policies. To mitigate this gap, the project investigates *domain randomization*, a technique that enhances the robustness of control policies by exposing the model to a wide variety of simulated conditions during training. The goal is to determine whether sufficient variability in simulation enables the model to generalize to real-world scenarios without additional adaptations. During the project, the initial environment for the Hopper will be a flat and smooth terrain (Fig. [1]), where the Hopper will be trained to move stably and effectively. Subsequently, the environment will be made more complex, incorporating uneven surfaces with obstacles and slopes inspired by the Mars Yard—a synthetic representation of Martian terrains designed for testing exploratory robots (Fig. [2]). The objective will remain the same: to enable the Hopper to walk in a straight line even in the presence of these challenges.

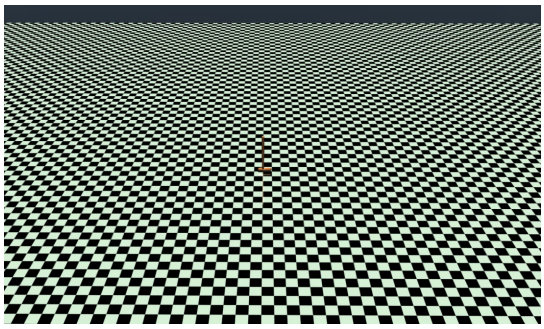


Fig. 1: Original flat terrain

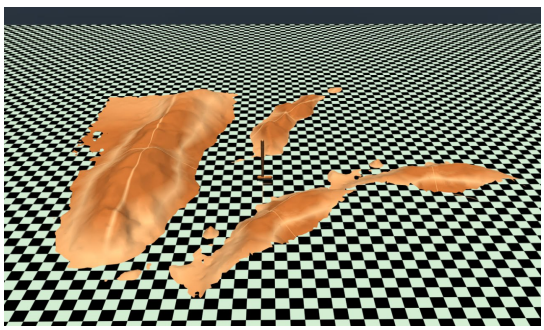


Fig. 2: Martian terrain

To further enhance the Hopper's ability to generalize, we will apply Uniform Domain Randomization to the environment. This involves rotating the map by a certain number of degrees at each step the Hopper takes. As a result, the Hopper will encounter different obstacles and terrain configurations as it moves, forcing it to adapt and generalize the task as much as possible. By constantly varying the environment through random rotations, the Hopper will learn to navigate effectively despite unpredictable terrain variations, improving its robustness in more complex and dynamic conditions. Finally, the project will analyse the training techniques used in both environments and evaluate the impact of terrain characteristics on the Hopper's ability to adapt and transfer the learned policies from one domain to another.

II. Related works

Reinforcement Learning (RL) is a widely discussed topic, particularly in [1], where it is presented as a method for learning to perform actions through interaction with the environment. Similar to how a child learns to walk by trial and error, a robot can use RL to develop adaptive behaviors by receiving feedback from its surroundings. This feedback-driven approach is a fundamental concept underlying most theories of learning and intelligence, as it enables agents to improve their performance iteratively. RL's principles have broad applications, ranging from robotics to decision-making systems, showcasing its versatility in solving complex tasks.

A. Definitions and main concepts

In Reinforcement Learning (RL) the two central components are the agent and the environment. As said in [2], the agent is the one that performs actions, while the environment is the world that the agent interacts with. At every step, the agent observes the state of the environment, which can be either complete or partial, and then chooses an action. This action affects the environment, leading to changes, although the environment can also change on its own, independent of the agent's actions.

The agent's behaviour is determined by a policy, a rule that specifies which action to take in a given situation. Policies can be *deterministic*, often denoted as μ, or *stochastic*, denoted as π. A policy $\pi(a|s)$ defines a distribution over the action space A given a particular state S, where each query to the policy samples an action a from the conditional distribution. In deep RL, policies are parametrized functions, such as neural networks, whose parameters can be adjusted via optimization algorithms to modify the agent's behaviour.

It is called reward function $r : S$ x $A \rightarrow R$ a scalar signal that reflects the quality of the action that the agent took and the desirability of performing an action in a given state. Following the documentation provided for the Hopper environment [3], we decided to adopt the definition 1 for the reward function for our problem:

$$reward = healthy\_reward + forward\_reward - control\_cost \quad (1)$$

where:

- *healthy_reward* = for each timestep that the agent is heathy (which means that it respects some conditions), it gets a reward
- *forward_reward* = $forward\_reward\_weight * (pos\_before\_action - pos\_after\_action)/dt$, where dt is the time between two actions (default $dt = 0.008$). This equation measures the horizontal progress of the Hopper.
- *control_cost* = $ctrl\_cost\_weight * sum(action^2)$, where $ctrl\_cost\_weight$ has a default value of $1e-3$. This equation penalizes the Hopper if it takes too large actions.

The state of the environment at timestep t is denoted by $s_t \in S$ and it provides a complete depiction of the world, encompassing all pertinent information. On the other hand, an observation O offers only a limited perspective of the state, which may overlook certain details. The set of actions that the agent can take is determined by the environment and is known as the action space. In environments with discrete action spaces there are a limited number of possible moves. In contrast, environments that involve physical systems, like robots, usually feature continuous action spaces, where actions are represented as real-valued vectors. In our case, both the state and action spaces are represented by Boxes. A Box is a continuous interval defined by a lower and upper bound. The dimensions of the box represent the different variables that make up the state or action.

TABLE I: State and Action Spaces

Type	Space	Bounds	Shape	Data Type
State space	Box	$[-\infty, -\infty, -\infty, \ldots, -\infty]$ to $[\infty, \infty, \infty, \ldots, \infty]$	(11,)	float64
Action space	Box	$[-1, -1, -1]$ to $[1, 1, 1]$	(3,)	float32

The table [I] shows that both the state and action spaces of the Hopper environment are continuous spaces. In fact, the state space is a continuous Box with 11 dimensions ($\mathbb{R}^{11}$), representing properties like position and velocity of the Hopper's torso, thigh, leg, and foot. On the other hand, the action space is a Box with 3 dimensions ($\mathbb{R}^3$), where each value ranges from -1 to 1, corresponding to the control of the Hopper's joints.

The article [2] also provides a detailed overview of policy optimization, a critical aspect of RL. It explains how optimization methods are used to improve policies by adjusting parameters to maximize the cumulative reward. OpenAI's implementation demonstrates the effectiveness of such techniques in training agents to achieve high performance across diverse tasks.

B. Sim-To-Sim scenario

Simulations serve as essential tools for training agents, providing a wealth of data while minimizing safety risks throughout the development process. Nevertheless, the behaviours acquired in these controlled settings frequently rely on the unique features of the simulator. As a result

of modelling inaccuracies, strategies that prove effective in simulation may not necessarily be applicable in real-world situations. Implementing RL algorithms in real-world situations raises significant safety concerns, as exploration, that can lead agents to make choices that may endanger themselves or their environment. Although simulations can help mitigate some of these risks, applying learned policies to real-world scenarios remains difficult due to the "reality gap," which underscores the discrepancies between simulated and actual conditions.

Unlike the methods outlined in [4] and [5], which emphasize using simulations to bridge knowledge to real-world applications, our focus is on a sim-to-sim transfer scenario. With no physical robot available for testing, we simulate the transfer process by establishing two separate simulated environments: a source and a target. We impose a controlled "reality gap" between these environments, enabling us to thoroughly investigate the transfer dynamics and assess the resilience of the learned policies within a simulated transfer framework.

C. Domain Randomization

Domain randomization is a key technique used to address the reality gap. Its primary goal is to introduce sufficient variability during training in simulation so that, at test time, the model can generalize effectively to real-world data. The approach described by [7] involves altering simulation parameters, such as environmental conditions, or rendering characteristics, to create a diverse training dataset. In this way, models trained on simulated images with randomized rendering settings can successfully transfer their learned skills to real images.
On the other hand, with [5], they proved that dynamics randomization involves introducing random variations in the simulator's physical parameters, such as mass, friction, and sensor noise, during training. Their approach was illustrated with a Fetch Robotics arm that was assigned the task of pushing an object to a designated target. The policies were developed solely in a simulated environment with varied dynamics and were then directly applied to the physical robot. Even with variations in friction or robot setup, the trained policies adjusted effortlessly, removing the necessity for additional real-world training or exact simulator tuning (those results are shown in the video [6]). This method showcases the ability of simulators to produce a wide range of training data, allowing the implementation of sophisticated reinforcement learning techniques that would typically be unfeasible for direct use in real-world scenarios.

III. Methods

Our goal is to train the Hopper to learn how to walk, first on a smooth surface and then on a terrain filled with obstacles such as craters and rocks. The Hopper must learn to recognize obstacles and adjust its steps accordingly. The remainder of this section describes the specific methods we use, such as Actor-Critic methods, particularly PPO, combined with the technique of Uniform Domain Randomization.
Actor-Critic methods represent an advanced class of Reinforcement Learning (RL) algorithms that integrate the strengths of Policy-Based and Value-Based methods. This synergy aims to enhance stability and efficiency, primarily by reducing the variance associated with gradient estimates while leveraging bootstrapping techniques to improve learning speed. Actor-Critic methods combines policy gradient (*Actor*) with a value function approximation (*Critic*) and they include popular approaches such as Proximal Policy Optimization (PPO) and Soft Actor-Critic (SAC).

A. Proximal Policy Optimization

PPO is an on-policy algorithm designed to enhance the stability of agent training. Its key innovation lies in preventing excessively large updates to the policy, which contributes to more stable training. Specifically, PPO is defined by the ratio between the new policy $\pi_{theta}(a|s)$ and the new one $\pi_{\theta_{old}}(a|s)$ and new policies, expressed as:

$$r(\theta) = \frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)}. \quad (2)$$

The results from the article [8] show that this method improves the stability by reducing the likelihood of excessive updates that could destabilize the training. As stated in [8], "the new policy does not benefit by going far away from the old policy." The PPO algorithm (see the pseudocode [1]) gathers trajectories by running the current policy and then computes cumulative rewards and advantage estimates. The policy is refined by maximizing a clipped objective function, which restricts significant changes from the previous policy. Meanwhile, the value function is updated by minimizing the squared error in relation to the estimated rewards-to-go. This method helps stabilize learning by avoiding drastic shifts in the policy.

B. Soft Actor-Critic

SAC is an off-policy algorithm that employs a replay buffer and entropy regularization to balance exploration and exploitation. The objective of the article [9] is using SAC to maximize the expected return while increasing entropy to avoid premature convergence to suboptimal policies. The optimization objective is:

$$J(\theta) = \mathbb{E}_{s_t \sim D}[\mathbb{E}_{a_t \sim \pi_\theta}[Q_{\theta'}(s_t, a_t) - \alpha log(\pi_\theta(a_t|s_t)]], \quad (3)$$

where $Q_{\theta'}(s_t, a_t)$ is the action-value function, $\pi_\theta(a_t|s_t)$ is the policy, and α is the entropy regularization coefficient that controls the exploration-exploitation trade-off. At test time, the policy's performance is assessed by removing stochasticity and using the mean action instead of a sampled action, which improves overall performance. In conclusion, the SAC algorithm (shown in the pseudocode [2]) leverages a stochastic policy to interact with the environment, storing transitions in a replay buffer. During the update phase, it computes targets

Algorithm 1 Proximal Policy Optimization

Require: Initial policy parameters θ_0, initial value function parameters ϕ_0

1: **for** $k = 0, 1, 2, \ldots$ **do**

2: Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi_{\theta_k}$ in the environment

3: Compute rewards-to-go $\hat{R}_t$

4: Compute advantage estimates $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}

5: Update the policy by maximizing the PPO-Clip objective:

$$\theta_{k+1} = \arg\max_{\theta} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min \Big(r(\theta) \hat{A}^{\pi_{\theta_k}}(s_t, a_t), \; \mathrm{g}(\epsilon, \hat{A}^{\pi_{\theta_k}}(s_t, a_t)),$$

typically via stochastic gradient ascent with Adam.

6: Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \Big(V_\phi(s_t) - \hat{R}_t \Big)^2,$$

typically via some gradient descent algorithm.

7: **end for**

for the Q-functions by considering the minimum of two Q-function estimates and incorporating the policy entropy. The Q-functions are updated by minimizing the squared error between the Q-values and their targets, while the policy is optimized to maximize the expected return adjusted by the entropy term. To enhance stability, the target Q-function parameters are updated gradually.

C. Uniform Domain Randomisation

UDR aims to evaluate how a RL agent trained in a *source* domain behaves when transferred to a *target* domain, where variations or discrepancies are introduced. The goal is to assess the agent's ability to generalize across different environmental conditions, by randomizing various variables in the environment, while mostly working on sim-to-sim scenarios:

- *Source* is the environment where the RL agent is trained and it has a fixed configurations.
- *Targhet* is the real-world or modified domain that introduces changes (e.g., increasing the torso mass by 1 kg compared to the *source*). This domain simulates the same discrepancies that we could encounter in sim-to-real scenarios.

Initially, in the original environment provided by the library [3], the following mass values are given "to simulate the reality gap, the source domain Hopper has been generated by shifting the torso mass by 1kg with respect to the target domain" (from [10]), and he following table (II) shows this discrepancy between the two environments, source and target.

Algorithm 2 Soft Actor-Critic

Require: Initial policy parameters θ, Q-function parameters ϕ_1, ϕ_2, empty replay buffer $\mathcal{D}$

1: Set target parameters equal to main parameters: $\phi_{\text{targ},1} \leftarrow \phi_1, \phi_{\text{targ},2} \leftarrow \phi_2$

2: **repeat**

3: Observe state s and select action $a \sim \pi_\theta(\cdot|s)$

4: Execute a in the environment

5: Observe next state s', reward r, and done signal d to indicate whether s' is terminal

6: Store (s, a, r, s', d) in replay buffer $\mathcal{D}$

7: **if** s' is terminal **then**

8: Reset environment state

9: **end if**

10: **if** it's time to update **then**

11: **for** j in range (however many updates) **do**

12: Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$

13: Compute targets for the Q-functions:

$$y(r, s', d) = r + \gamma(1 - d) *$$

$$* \left(\min_{i=1,2} Q_{\phi_{\text{targ},i}}(s', \tilde{a}') - \alpha \log \pi_\theta(\tilde{a}'|s') \right), \quad \tilde{a}' \sim \pi_\theta(\cdot|s')$$

14: Update Q-functions by one step of gradient descent using:

$$\nabla_{\phi_i} \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_{\phi_i}(s, a) - y(r, s', d) \right)^2, \quad \text{for } i = 1, 2$$

15: Update policy by one step of gradient ascent using:

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} \left(\min_{i=1,2} Q_{\phi_i}(s, \tilde{a}_\theta(s)) - \alpha \log \pi_\theta(\tilde{a}_\theta(s)|s) \right),$$

where $\tilde{a}_\theta(s)$ is a sample from $\pi_\theta(\cdot|s)$ which is differentiable w.r.t. θ via the reparametrization trick.

16: Update target networks with:

$$\phi_{\text{targ},i} \leftarrow \rho \phi_{\text{targ},i} + (1 - \rho)\phi_i \quad \text{for } i = 1, 2$$

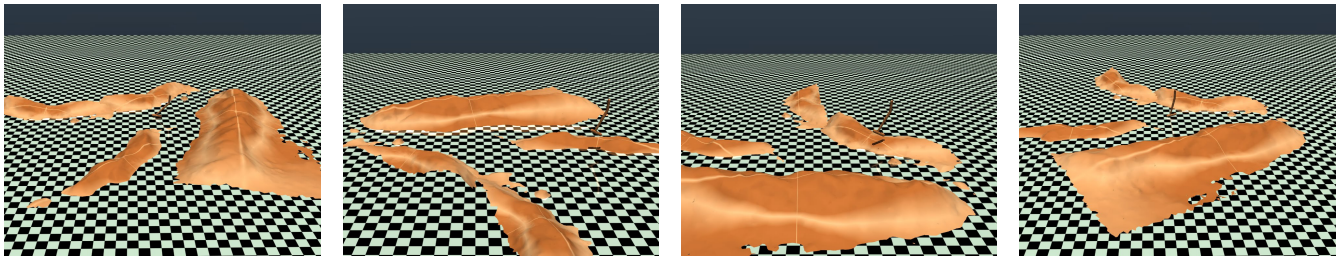
17: **end for**

18: **end if**

19: **until** convergence

On the other hand, for the second environment the randomization process goes beyond just the mass values. It also includes the map on which the hopper needs to walk. Specifically, the randomization now extends to the angle variable, which determines the orientation of the map. The Figure [3] shows this environment randomisation technique, where the map is rotated at each step the Hopper takes.

This expanded randomization helps the agent learn to adapt to more varied conditions, improving its robustness and ability to generalize across different environments.



(a) (b) (c) (d)

Fig. 3: Snapshots of the rotated map at each iteration.

TABLE II: Body Masses for Source and Target Domains

Body Name	Source Mass (kg)	Target Mass (kg)	Difference (kg)
World	0.0	0.0	0.0
Torso	2.534	3.534	1.0
Thigh	3.927	3.927	0.0
Leg	2.714	2.714	0.0
Foot	5.089	5.089	0.0

IV. Experiments

All simulations are performed using the MuJoCo documentation [11] that helped us to visualize the results of our work. The first step was to determine which algorithm, between SAC and PPO, would be the most suitable for training the Hopper. To evaluate this, we plotted the learning curves for both algorithms (Fig. [4] and [5]). PPO outperforms SAC by reaching convergence earlier and achieving higher rewards, with a score of approximately 1650, instead of SAC, that reaches only a maximum of 1400 as reward.

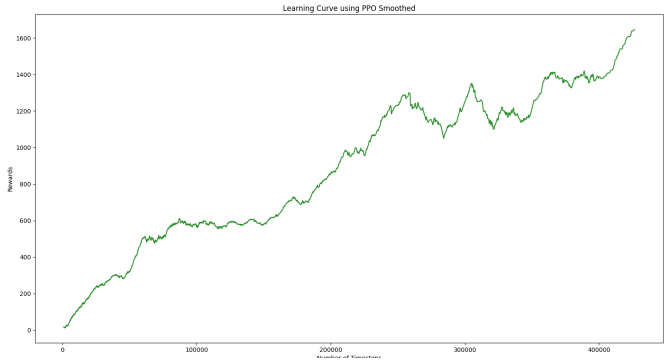


Fig. 4: PPO Learning Curve

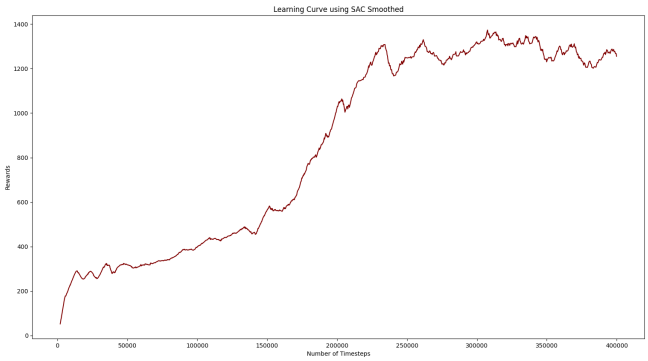


Fig. 5: SAC Learning Curve

This difference can be attributed to the fact that PPO is better suited for environments with high-dimensional continuous action spaces, as it balances exploration and exploitation more effectively. SAC, on the other hand, may struggle in such scenarios due to its reliance on soft Q-learning, which can lead to slower or unstable convergence in some cases. Based on these observations, we decided to use the PPO algorithm for this project moving forward

Building on our initial efforts, we next explored the ”reality-gap” between the two environments, source and target. We evaluated our algorithm by training two agents separately in the source and target domains. We tested across 50 episodes in three model configurations:

- source $\rightarrow$ source;
- source $\rightarrow$ target;
- target $\rightarrow$ target.

Therefore, we tested each model and report its average return over 50 test episodes with and without the application of Domain Randomization (on the original environment) and the results are presented in Table [1]. Since we are trying to answer to the question ”Is it worth it to use Domain Randomisation or not?”, these outcomes reinforce the effectiveness of incorporating UDR into our approach.

The most significant result is in the “*Source to Target*” scenario, where the average test reward increases substantially when UDR is applied, rising from 1080.5 ± 137.16 to 1471.2 ± 211.2. This shows that UDR can effectively apply what it learned in the source environment to a new, unfamiliar target environment, supporting our belief that UDR plays a vital role

TABLE III: Test reward (avg +/- std) without and with UDR

Scenario	without UDR	with UDR
Source to Source	1401.28 +/- 221.4	1429.2 +/- 216.5
Source to Target	1080.5 +/- 137.16	1471.2 +/- 211.2
Target to Target	1581.36 +/- 2.39	

in enhancing transferability across different domains.
Although in the *Source to Source* scenario there is an increase in the average test reward from 1401.28 +/- 221.4 to 1429.2 +/- 216.5, its practical relevance is limited. In fact in real-world situations, *Source to Source* scenarios are not used because the source environment is specifically designed to be used exclusively in simulation, as defined by its purpose. Consequently, applying the source environment to real-world scenarios leads to results that are inconsistent and inapplicable. This is why many studies avoid *Source to Source* scenarios altogether, focusing instead on *Source to Target* scenarios (see article [4], [5] and YouTube video [6]).
As further evidence, *Target to Target* testing was conducted only without UDR because applying UDR in this context is not feasible. This is due to the fact that Domain Randomization is intended to generalize the source environment, whereas the target environment, by definition, cannot be generalized. This reasoning leads us to conclude that using UDR is an excellent choice for teaching the Hopper to walk. However, we also ask ourselves whether this conclusion remains valid when making bigger changes on the environment, like changing the map.

V. Extension

Based on the results achieved so far, we have decided to extend the Hopper project by implementing the modification previously outlined. Specifically, we will replace the original map, in which the Hopper was trained to walk, with a new map simulating Martian terrain. This updated environment will incorporate obstacles such as hills, craters, and rocks. Our goal is to demonstrate that, if we change the environment to a Martian terrain, where the Hopper must learn to walk on a different and more uneven surface, is it more effective to use Domain Randomization. In order to do that, we conducted four separate experiments.

In the first experiment, we trained a model in a source environment where randomization was applied both to the masses of the Hopper (as used in the previous section) and to the orientation of the map. Specifically, the orientation range was set to $180°$. Subsequently, we tested the trained model on a target environment, which consisted on a map with a fixed orientation of 180 degrees (as shown in Fig. [6]).

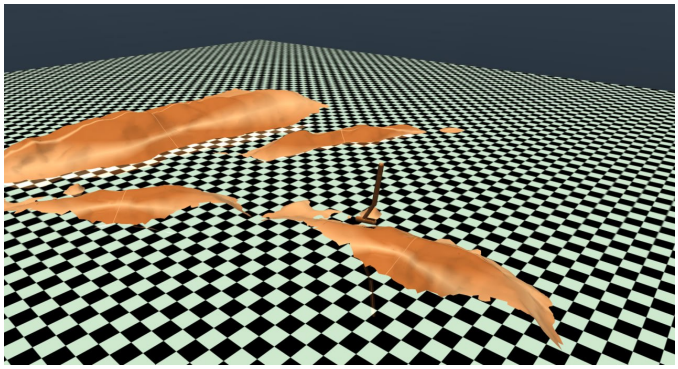


Fig. 6: Test Map with fixed orientation of $180°$

As a result, we obtained a test reward that was too low even after 40000 timesteps of training, specifically :

> *Test reward (avg ± std): (704.037 ± 24.522) – Num episodes: 50.*

This result was far from satisfactory, leading us to question whether randomization was indeed the right choice for our context.
In the second experiment, we trained the Hopper in a source environment with the map fixed at an angle of $183°$. We then tested the trained model on the same map as the first experiment (Fig. [6]), introducing a reality gap, yet without applying any randomization during training.

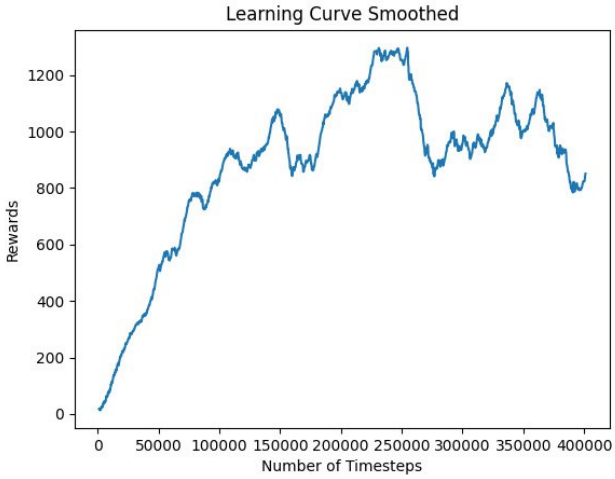


Fig. 7: Second Experiment Learning Curve

The Graph [7] shows the Learning Curve after 400000 timesteps. It is clear how the curve reaches convergence, but still keeping the reward value too low. In fact, the total reward obtained after testing is still very low:

> *Test reward (avg +/- std): (903.81570468 +/- 17.45211491439606) - Num episodes: 50.*

This leads us to question whether the reason the Hopper is failing to learn is that we are not teaching it effectively enough. Therefore, we decided to trust the results from the previous experiments and apply UDR.

The third experiment involves training in the same source environment but applying randomization exclusively to the masses, leaving the map fixed with an orientation of $183°$. The test was conducted on the same target map as in the previous experiments (Fig. [6]).

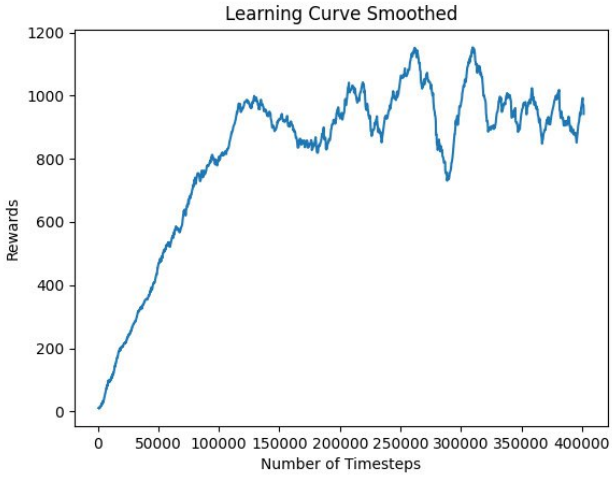


Fig. 8: Third Experiment Learning Curve

The graph [8] shows the Learning Curve obtained from this experiment's training, indicating that significant improvements cannot be expected from the test: even if the curve reaches convergence fairly quickly, the quality of learning is still not optimal.
Indeed, as expected, the test reward obtained from the tests was:

Test reward (avg +/- std): (1152.87832128 +/- 267.9774099091476) - Num episodes: 50.

Therefore, there is clearly an improvement compared to the previous cases, but we knew that better results could be achieved (we were on the right path!).

In the fourth experiment, we attempted to randomize as much as we could, including both the masses and the map. However, to address the issues encountered in the first experiment, we decided to limit the randomization range for the map's orientation, narrowing it to a range of $[175°, 185°]$. The test was conducted on the same map of the previous experiments, the one oriented of $183°$.

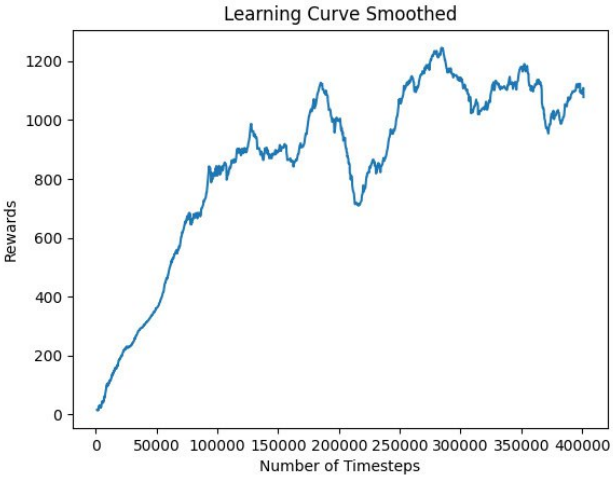


Fig. 9: Fourth Experiment Learning Curve

The graph [9] illustrates the Learning Curve from this experiment, showing a clear improvement compared to the previous cases. To gain a deeper understanding of these results, we tested the model on the map with a fixed orientation of $180°$ and obtained remarkable outcomes.

Test reward (avg +/- std): (1454.37570358 +/- 11.419838179484348) - Num episodes: 50 .

Compared to where we started, we can now happily observe, by reviewing the rendered video from MuJoCo, that the Hopper has finally learned to walk on a Mars Yard and overcome obstacles (Fig. [10a], [10b], [10c]).

As further evidence, we tested the model on maps with various orientations to determine whether the Hopper could also tackle other hills present in the map, which were not part of the training environment. Those are some of the Test Rewards obtained:

- Test reward (avg +/- std) with a map orientation of $210°$: (1196.17161644 +/- 289.984910741124) - Num episodes: 50 .
- Test reward (avg +/- std) with a map orientation of $270°$: (1487.7438754 +/- 22.712098025408846) - Num episodes: 50 .

Finally, from the rendered videos and the snapshots [10], we observe that not only is the Hopper capable of successfully navigating craters of the same height as the one used during training, but it also manages to overcome a hill twice as tall with equal success. Additionally, it is able to traverse the "valleys" between the various mountains, where the terrain is smooth, demonstrating further adaptability.

VI. Conclusion

The experiment is started by using the domain randomization in a simple hopper environment. The starting experiment was conducted on a flat floor with two types of environment, the source and the target, which differ only by a shifted

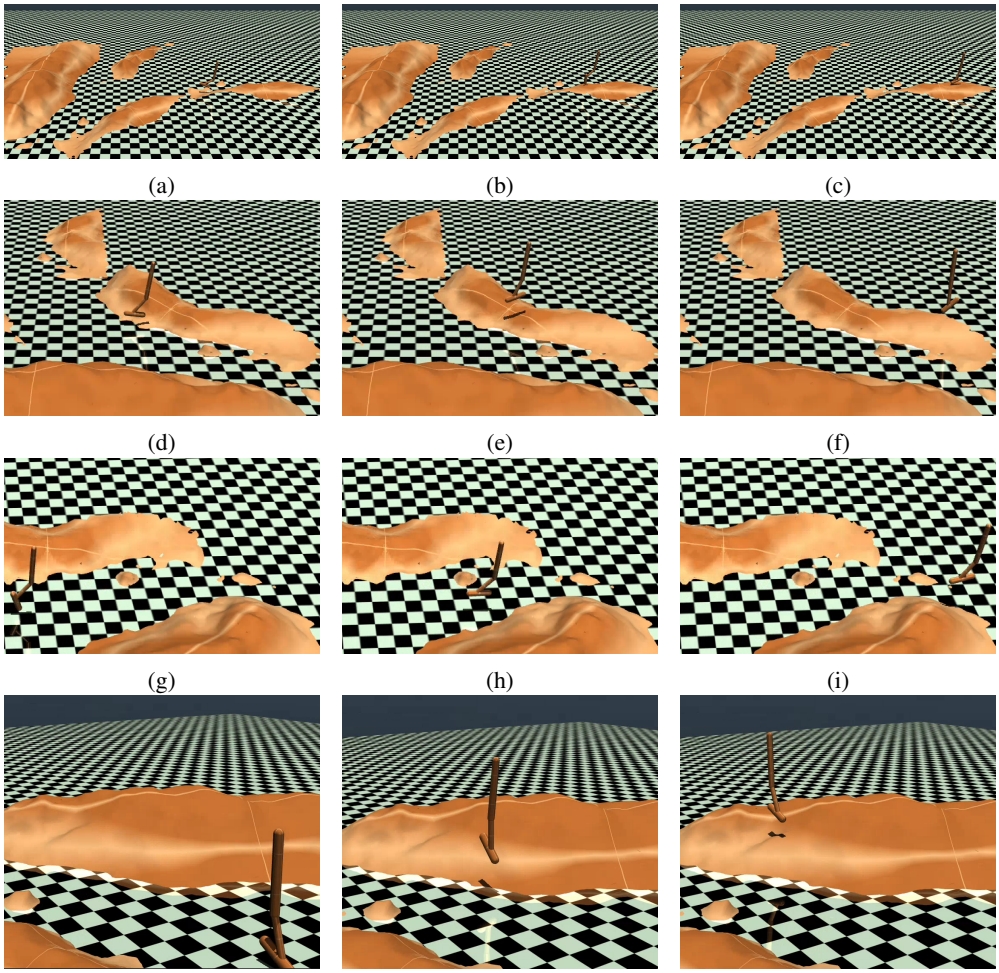


Fig. 10: Screenshots of the Hopper walking across different maps

torso mass. The first experiment of domain randomization in this case was on the mass of the other parts of the hopper: thigh, leg and foot. We took an interval of 1 between the real value of the mass of the leg and used a random distribution to train the agent with different values every episode. The experiment progressed by introducing a new type of terrain, a floor featuring small hills designed to challenge the trained hopper. To enhance domain randomization the map was rotated at every episode, forcing the hopper to adapt and learn to navigate height differences during the training phase. Initially, we attempted to fully randomize the map's rotation, but this introduced excessive variance between training episodes, preventing the agent from effectively learning how to overcome obstacles. To address this, we restricted the rotation values to a smaller range to focus the agent on a limited portion of the map. This adjustment reduced the variance and led to a significantly improved behaviour. The results shows that training on a randomized path across various terrains leads to improved performance during the test phase. Furthermore, we observed that testing the agent trained in this manner on different types of hill, beyond the one encountered during training, enables the hopper to adapt and overcome these novel obstacles effectively. Our conclusions are that domain randomization can achieve optimal results in real-world applications, especially when controlling the movement of a robot system. This approach proves to be a reliable method as it enhances the generalization of parameters, enabling the system to adapt effectively to diverse and unpredictable environments.

REFERENCES

[1] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. Cambridge, MA: The MIT Press, 2018.

[2] OpenAI Spinning Up, "Introduction to Reinforcement Learning" available online: https://spinningup.openai.com/en/latest/spinningup/rl_intro.html.

[3] Gym Library Documentation, "Hopper Environment" available online: https://www.gymlibrary.dev/environments/mujoco/hopper/.

[4] S. Höfer, K. Bekris, A. Handa, J. C. Gamboa, F. Golemo, M. Mozifian, C. Atkeson, D. Fox, K. Goldberg, J. Leonard, C. K. Liu, J. Peters, S. Song, P. Welinder, and M. White, "Perspectives on Sim2Real Transfer for Robotics: A Summary of the R:SS 2020 Workshop" available online: https://arxiv.org/pdf/2012.03806.

[5] X. B. Peng, M. Andrychowicz1, W. Zaremba1 and P. Abbeel1 "Sim-to-Real Transfer of Robotic Control with Dynamics Randomization" available online: https://arxiv.org/pdf/1710.06537

[6] YouTube, “Sim-to-Real Transfer of Robotic Control with Dynamics Randomization Results Video“ available online: https://www.youtube.com/watch?v=XUW0cnvqbwM.
[7] 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“ available online: https://arxiv.org/pdf/1710.06537.
[8] OpenAI Spinning Up, “Proximal Policy Optimization (PPO)“ available online: https://spinningup.openai.com/en/latest/algorithms/ppo.html.
[9] OpenAI Spinning Up, “Soft Actor-Critic (SAC),” available online: https://spinningup.openai.com/en/latest/algorithms/sac.html.
[10] Robot Learning Project Exercise, https://file.didattica.polito.it/dl/MATDID/33847698.
[11] MuJoCo documentation available online: https://mujoco.org/

Acknowledgments: The repository available on GitHub includes a folder containing recordings of the final rendered simulations.