Reinforcement Learning Project

Federico Grasso 319332
Irene Michelotti 319721
Filippo Scaramozzino 312856

Abstract

This project focuses on reinforcement learning (RL) in the context of robotic systems, with a particular emphasis on transferring control policies from simulated to real environments (sim-to-real transfer) and it provides also a theoretical introduction to reinforcement learning. Classical RL algorithms such as REINFORCE and Actor-Critic Policy Gradient methods are implemented, followed by advanced techniques like Proximal Policy Optimization (PPO) and Soft Actor-Critic (SAC). To address the discrepancies between training and real environments, Uniform Domain Randomization (UDR) is implemented. Finally, an adaptive extension is proposed to further enhance the sim-to-real transfer process.

1. Theoretical Introduction

1.1. Basic Concepts

Reinforcement Learning (RL) is a branch of artificial intelligence where an agent learns to make decisions by interacting with an environment. The agent aims to achieve a goal by taking actions that maximize cumulative rewards over time. The agent learns by receiving rewards and punishments, adjusting its actions to maximize cumulative reward. RL is distinct from supervised learning, where the correct actions are provided by a supervisor, RL requires the agent to discover these actions through trial and error. To carry out the analysis in this project we must familiarize with the basic concepts of RL:

- Environment: is the system with which the agent interacts, it provides feedback to the agent's actions in the form of rewards and transitions to next states
- Agent: takes actions based on what he did learn on the environment
- State: a representation of the agent within the environment, it can be fully observable (where the agent has complete information about the state) or partially observable (where the agent has incomplete information about the state)
- Action: the set of decisions the agent can make in a given state
- Reward: a scalar feedback signal given to the agent after taking an action in a particular state
- Policy: is used by the agent to determine the next action based on the current state. It can be deterministic (a specific action is chosen) or stochastic (actions are chosen according to a probability distribution)
- Value Function: the value function estimates the expected cumulative reward of states or state-action pairs, helping the agent evaluate the long-term benefit of its actions.
- Trajectory: A sequence of states, actions, and rewards that the agent experiences during an episode
- Discount Factor: A factor between 0 and 1 that balances the importance between immediate and future rewards.

1.2. Algorithms

Keeping in mind the basic concepts of reinforcement learning, we can look more in detail at the algorithms. We can divide the main algorithms into value-based methods, policy-based methods, and actor-critic methods.

- Value-Based Methods
  1. Q-Learning: off-policy algorithm that learns the value of actions in states (Q-values) and uses them to make decisions.
  2. SARSA (State-Action-Reward-State-Action): on-policy algorithm similar to Q-learning but updates the Q-value based on the action actually taken by the current policy
- Policy-Based Methods

1. REINFORCE (Monte Carlo Policy Gradient): policy gradient method that directly optimizes the policy by adjusting its parameters to maximize the expected cumulative reward G_t. The policy is updated using gradient ascent

$$\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(a \mid s) G_t$$

2. Proximal Policy Optimization (PPO): policy gradient method that balances exploration and exploitation it ensures that the policy updates are not too large, improving stability.

- Actor-Critic Methods: Combine value-based and policy-based methods. The actor updates the policy, while the critic evaluates the policy by learning value functions

1. Soft Actor-Critic (SAC): off-policy actor-critic method that incorporates an entropy term in the objective function to encourage exploration by maximizing both expected reward and entropy

2. Related Works

Reinforcement Learning (RL) has been the subject of a wide variety of research for many years. Before the 1980s the research was divided in three main branches: learning by trial and error, optimal control and temporal difference, which were investigated with different goals. These threads were linked when it was developed Q-learning in 1989 by Watkins, and in the following years researchers used the knowledge gained to try and master different types of games, such as Backgammon and chess. In Sutton et al. ([1]) we have an introduction to reinforcement learning, in Kormushev Calinon & Caldwell et al.([4]) we see the application of RL in robotics. In more recent time new methods were developed in order to obtain more robust results, namely PPO and SAC introduced respectively by Schulman et al. ([2]) and Haarnoja et al. ([3]). Moreover with the aim of reducing the reality gap Domain Randomization (DR) was initially introduced in the context of randomization for visual properties of simulators by Tobin et al. ([5]) and then widely investigated. In Muratore et al. ([6]) randomized simulations for robot learning and techniques like domain randomization are discussed.

3. First RL Agent

In the first part of our project we will focus on training an agent on the gym Hopper environment. The hopper is a robot with the Mujoco physics engine which is a physics engine used for robotic tasks. The hopper environment is a single-leg jumping robot, the goal of the agent is to control this robot to perform horizontal jumps. The state representation for the hopper indicates information about the current state of the robot which can include for example joint angles and angular velocities. The action space is the set of actions that the agent can choose from to control the robot, such as torques applied to the joints. The rewards for the system are feedbacks to the agent based on its actions and the state of the robot, designed to encourage to achieve a specific goal(in our case moving forward without falling). In this project, the hopper we will use to perform the training has the torso mass that has been modified decreasing the mass of the first body link (index 1) by 1.0 unit (kg shift), this means that the mass value of the torso link is 1.0 unit less than its default value in the original Hopper environment.

3.1. REINFORCE

In the previous section we introduced REINFORCE, we now try to go more in deep to understand how it works. We have to investigate how policy gradient methods works: unlike traditional value-based methods in RL that estimate the value function (expected cumulative reward) the policy gradient methods optimize the policy directly. The policy is typically parameterized by a function (for example a neural network) that outputs action probabilities given a state. Policy gradients aim to increase the probabilities of actions that lead to higher rewards and decrease the probabilities of actions that lead to lower rewards. We will use different versions of REINFORCE, one without a baseline, one with a constant baseline $b = 20$, and another with constant baseline $b = 100$.

3.2. Baseline

In the context of the REINFORCE algorithm with a constant baseline, choosing a good value for the baseline is very important for improving the training efficiency, a common approach is to use the empirical average of the rewards observed so far during training. The baseline in the REINFORCE algorithm is used to reduce the variance of the gradient estimates without introducing bias. If we subtract the baseline from the rewards, we can stabilize the formula that we use to estimates the gradients. Lower variance in the gradient estimates generally leads to more stable and reliable updates to the policy parameters, which can result in faster convergence and better performance. In order to better understand this, let's look at the formula :

$$\nabla J(\theta) = \mathbb{E}\left[\sum_{t=0}^{T} \nabla \log \pi_\theta(a_t|s_t)(G_t - b)\right]$$

This is the formula for the gradient of the expected return with respect to the policy parameters, G_t is the return (cumulative reward) and b is the baseline. By subtracting b, we are effectively centering the reward, which reduces the variance of the returns.

3.3. Results REINFORCE

For the REINFORCE algorithm, we now report some results obtained by training the agent on 20000 time steps, we decided to perform a shorter training with respect to the standard 100000 time steps used as standard for the next sections because the training in some cases lasted about two hours and didn't show a great improvement going forward. We also report some plots to show examples of scenarios of training rewards during the time steps, while we report the full results in the table below.

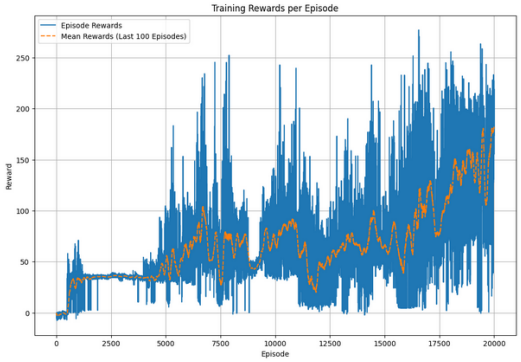

Figure 1. REINFORCE with learning rate 0.001

Figure 1 shows the rewards obtained with REINFORCE with learning rate 0.001, we may note a high variance during training and a little improvement during the last 5000 steps. The mean reward of the last 1000 steps is around 150, let's see if with the REINFORCE baseline we can improve this situation, specifically the variance in the rewards.

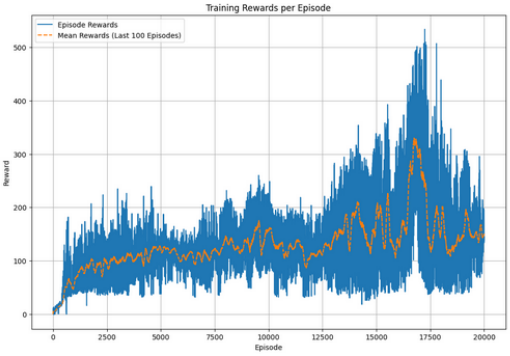

Figure 2. REINFORCE with learning rate 0.001 and baseline 20

We may note in Figure 2 that the situation has improved, both for the variance and the value of the reward, the variance now has decreased (even if it is still high) and the reward now reaches higher values faster. To better evaluate the methods and their performances, we tried two different learning rates and we analyzed the mean and standard deviation of the last one thousand rewards.

Results of REINFORCE				
Algorithm	Time	Learning rate	Mean Reward	Std.Dev.
NoBaseline	1h	0.001	148.38	46.81
NoBaseline	2h	0.0005	132.80	67.59
Baseline 20	45 min	0.001	156.05	44.85
Baseline 20	10 min	0.0005	9.46	2.62
Baseline 100	1h	0.001	335.61	43.15
Baseline 100	45 min	0.0005	293.53	95.01

Analyzing the result we may note how we tried two different baselines, one with $b = 20$ the other one with $b = 100$, the choice of the second value of the baseline came directly from our knowledge on the theory, in fact a good choice for the baseline can be the average of the rewards: the average for the REINFORCE without baseline was around 150; but since the variance was very high we decided to put b a bit below the average. We may also note from the results that the learning rate $lr = 0.001$ proved to be better than $lr = 0.0005$ A great improvement that we may note also analyzing the plot in Figure 3, that is for the method with baseline $b = 100$ and $lr = 0.001$ where it seems like convergence to an average reward is reached and the values of the rewards are greater than the methods tried before, even if the variance is still high.

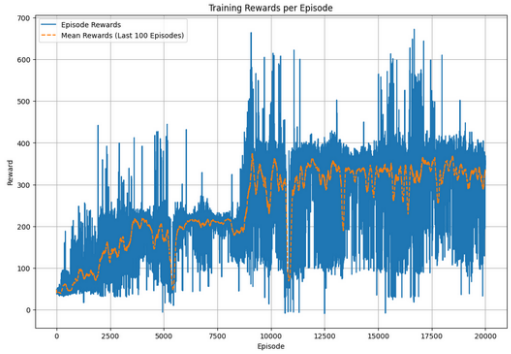

Figure 3. REINFORCE with learning rate 0.001 and baseline 100

3.4. ACTOR CRITIC

The actor-critic methods combine the advantages of policy-based and value-based approaches. These methods are designed to address some of the weaknesses of policy gradient methods, such as high variance and instability in updates. The actor is responsible for selecting actions according to a policy π while the critic evaluates the actions taken by the actor, the critic's role is to provide feedback to the actor on how good the selected actions are, which helps in reducing the variance of the policy gradient updates. The actor updates the policy parameters θ in the direction sug-

gested by the critic, while the critic updates the value function parameters accordingly. Let's look a bit more in detail at the equations that we are working with:
The policy is updated using the policy gradient theorem

$$\nabla_\theta J(\theta) = \mathbb{E}_{s_t, a_t \sim \pi_\theta} \left[\nabla_\theta \log \pi_\theta(a_t|s_t) Q^{\pi_\theta}(s_t, a_t) \right]$$

where $Q^{\pi_\theta}(s_t, a_t)$ is the action-value function which estimates the expected return. We will start by using A2C (Advantage Actor-Critic), where the advantage function

$$A(s, a) = Q(s, a) - V(s)$$

is used instead of the action-value function to reduce variance. So the policy gradient now is

$$\nabla_\theta J(\theta) = \mathbb{E}_{s_t, a_t \sim \pi_\theta} \left[\nabla_\theta \log \pi_\theta(a_t|s_t) A(s_t, a_t) \right]$$

For the next analysis, we will keep the learning rate fixed to 0.001 which has been proved to be better, and we will train using A2C. Since this algorithm should be more stable, we will train for 100000 time steps, also to have a better view of the learning process.

3.5. Result A2C

With A2C we can see that there is a reduction in variance but unfortunately the longer time of training and the more advanced implementation of the algorithm didn't show an increase in the average of rewards but we may note that the variance is decreasing during training especially in the last time steps. The result obtained in the last 1000 time steps was a mean reward equal to about 120 and a standard deviation equal to 15.

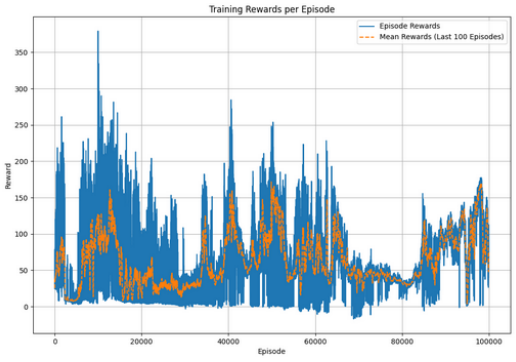

Figure 4. A2C training rewards per timesteps

4. PPO and SAC

We will now focus on the sim-to-real problem, in particular since we won't work with a real robot we will simulate it in a sim-to-sim scenario. This setting for simulation was manually created by introducing a difference between the source and the target environments, that is the torso mass of the hopper in the source domain is 1 kg lighter than the one in the target domain. In order to carry through the simulation of the reality gap we trained an agent using Stable-Baselines 3, a state-of-the-art RL library which provides efficient and stable implementations of various algorithms, among which PPO and SAC. These two methods are the ones we trained on both the source and target environments and then tested the agents obtained with different configurations. In this part, after a more thorough theoretical overview regarding the algorithms used, we will present the experimental results we derived, which will highlight a faster training and an overall better outcome. The most important result that we will present in this section is a visual representation of the hopper that is able to complete the task of jumping forward, we will insert four pictures that we took from the video of the rendering of the hopper tested with our model using PPO even if with SAC we had basically the same result.

4.1. PPO

As previously introduced Proximal Policy Optimization (PPO) is an on-policy algorithm meaning it directly optimizes the policy that dictates the agent's actions, based on the idea that after an update, the new policy shouldn't be too far from the old one. This goal is reached through the optimization of a clipped objective function using stochastic gradient descent, in fact this new objective function limits the size of the policy change at each step. PPO updates policies by solving the following maximization problem :

$$\theta_{k+1} = \arg\max_\theta \mathbb{E}_{s,a \sim \pi_{\theta_k}} \left[L(s, a, \theta_k, \theta) \right]$$

where L is defined as the minimum between two terms: the product of the ratio function and the advantage function; and the product between the advantage function and the ratio function clipped between $1-\epsilon$ and $1+\epsilon$, whose formula is:

$$L(s, a, \theta_k, \theta) = \min \left(rt(\theta) A(s, a),\ \text{clip}\left(rt(\theta), 1-\epsilon, 1+\epsilon \right) A(s, a) \right)$$

In particular, the ratio function

$$rt(\theta) = \frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}$$

represents the ratio between the probability of taking action a at state s in the current policy and the same probability computed with respect to the old policy, which well estimates the difference between the two policies. By clipping the ratio function the algorithm ensures a more conservative approach avoiding performance collapse, in other words it regularizes the policy update by keeping the new one in the range defined by the parameter ϵ, which usually takes value equal to $\epsilon = 0.20$.

4.2. SAC

On the other hand, Soft Actor Critic (SAC) is an off-policy algorithm, where the objective becomes maximizing the trade-off between expected return and entropy. In order to better understand SAC we first introduce entropy and consequently entropy-regularized reinforcement learning. Given a random variable x and its probability mass or density function P, the entropy of x is defined as follows

$$H(P) = \mathbb{E}_{x\sim P}\left[-\log P(x)\right]$$

hence, in our case entropy is a measure of randomness in the policy.

Taking into account this new measure, the entropy-regularized RL problem aims at finding a policy that maximizes a combination of the expected discounted cumulative reward and the entropy of the policy, which can be written as

$$\pi^* = \arg\max_{\pi} \mathbb{E}_{\tau\sim\pi}\left[\sum_{t=0}^{\infty}\gamma^t\Big(R(s_t,a_t,s_{t+1}) + \alpha H(\pi(\cdot|s_t))\Big)\right]$$

where $\alpha > 0$ is the trade-off coefficient. Consequently, the entropy is included also in the definitions of the value function V^π and of the Q-function Q^π. This new formulation ensures a balance between exploration and exploitation.

SAC belongs to the family of actor-critic algorithms, it combines off-policy updates with a stable stochastic actor-critic formulation. In this setting SAC learns a policy π_θ and two Q-functions Q_{ϕ_1}, Q_{ϕ_2} using a fixed coefficient α throughout the training. The Q-functions are learned via MSBE (mean squared Bellman error) minimization, while the policy is obtained through the maximization of the expected future return and the expected entropy, that is by maximizing

$$V^\pi(s) = \mathbb{E}_{a\sim\pi}\left[Q^\pi(s,a) - \alpha\log\pi(a|s)\right]$$

By combining the learned functions, and thanks to some reparameterization tricks, the policy is obtained solving the following optimization

$$\max_\theta \mathbb{E}\left[\min_{j=1,2} Q_{\phi_j}(s,\tilde{a}_\theta(s,\epsilon)) - \alpha\log\pi_\theta(\tilde{a}_\theta(s,\epsilon)|s)\right]$$

4.3. Results of PPO and SAC

In the following table we will report the results obtained with PPO and SAC that show a great improvement with respect to the previous algorithms both in terms of mean reward and also standard deviation. To evaluate the performance of these algorithms we used the *evaluate_policy* method using 10 episodes of test.

Results of PPO and SAC				
Algorithm	Time	Learning rate	Mean Reward	Std.Dev.
PPO	10 min	0.0001	168.63	3.51
PPO	5 min	0.0003	1147.85	4.55
PPO	5 min	0.0005	1308.68	96.41
PPO	5 min	0.001	795.69	16.38
SAC	1h	0.0001	423.98	10.34
SAC	1h	0.0003	687.10	6.26
SAC	1h	0.0005	1519.83	27.80
SAC	1h	0.001	1623.06	14.35

For what concerns PPO, we may note that the best learning rate is 0.0005, it is interesting to look at the overall plot for the rewards of each episodic task to see how the method has learnt during training (see Figure 5). For what concerns SAC, we may note that the best learning rate is instead 0.001, we report also the plot for this configuration (see Figure 6). Our decision for the next section to carry out the analysis on different configurations of source and target environment was to use the trained model of PPO because it was the best trade-off for what concerns both the mean reward and the time required to train.

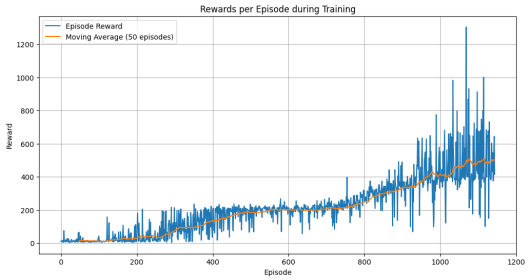

Figure 5. Rewards for PPO with learning rate 0.0005

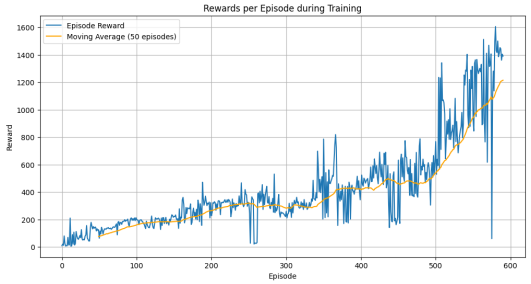

Figure 6. Rewards for SAC with learning rate 0.001

4.4. Visual Representation using PPO

We decided to insert four pictures of the jumping hopper, we inserted the most important steps in the jumping phase.

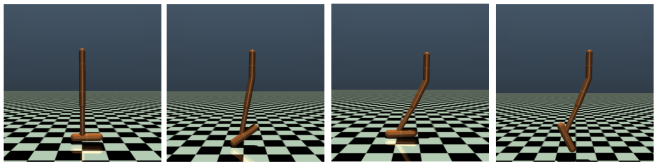

Figure 7. Jumping movement

4.5. Lower/Upper Bounds

In this section we will focus on training two agents with the model PPO, on the source and target domains respectively. We will also report the average return over 100 test episodes. In particular, we will report results for the following "training → test" configurations:

- source→source
- source→target (lower bound)
- target→target (upper bound)

To carry out this analysis we decided to maintain the learning rate fixed to its original value of the model of PPO which is 0.0003 because we didn't see any great improvement changing the learning rate, we report in the following table the results we obtained for the different configurations:

Results		
Configuration	Mean Reward	Std.Dev.
source→source	989.49	215.44
source→target (lower bound)	931.81	188.96
target→target (upper bound)	1257.29	7.59

We also report the plot of the rewards for the lower bound and upper bound configurations, that show interesting features. We may note that the configuration source→target has lower performances than target→target, this is indeed what we expected because in the lower bound configuration the source and target environments may have some differences and the agent may also perform well in the source environment because it is overfitting, and when it is transferred to the target environment, the policy learned may not generalize well. Training directly on the target environment of course will result in higher performance because the agent learns and optimizes its policy directly in the environment where it will be employed, but several reasons prevents us from doing so. One of them is safety, working with robotics can be troublesome in real-life simulations; others include velocity of training in real-life environments and the difficulties of non-randomization of simulations in the environment. To mitigate the sim-to-real gap, we may use Domain Randomization and that is what we are going to do in the next section.

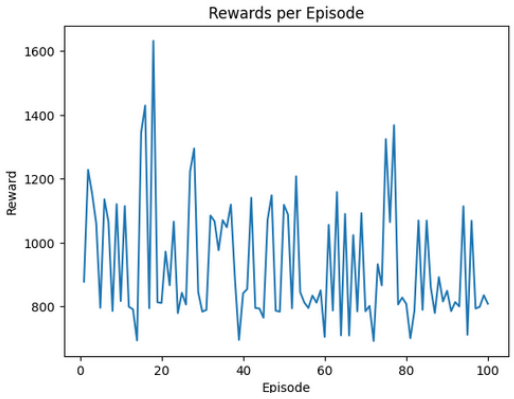

Figure 8. Lower Bound test rewards

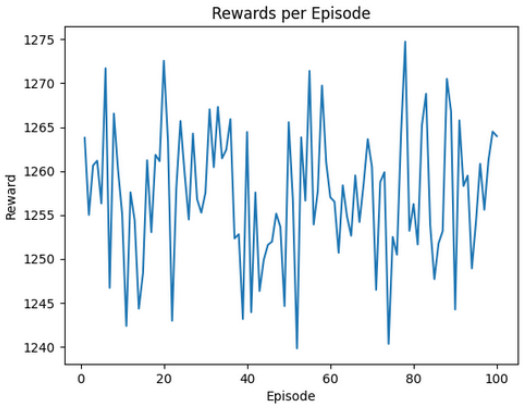

Figure 9. Upper Bound test rewards

5. Domain Randomization

In the previous sections we introduced different methods able to build the best agent with respect to the environment it was trained in. The results obtained by applying PPO in different configurations show how the reality gap wasn't still addressed, since the real world might differ from the domain in which the agent learned the policy. In order to tackle this issue we implemented domain randomization (DR), whose main idea is to train a model that works well in different random environments. In particular, DR consists of training the model in an environment we can control, called *source domain*, and then test the policy obtained on the target domain, which can be the real world or another simulated environment. In the source domain we can control a set of randomization parameters e_ξ with a configuration ξ sampled from a set distribution. In our project

we controlled three out of the four masses of the hopper (we didn't randomize the torso mass). Thanks to this randomization the agent becomes more robust to changes in the target domain and learns to generalize, hence it is able to reduce the reality gap. Thus DR aims at finding a policy such that it maximizes the expected reward across a distribution of configurations, in formula it can be expressed as

$$\theta^* = \arg\max_{\theta} \mathbb{E}_{\xi\sim\Xi}\left[\mathbb{E}_{\pi_\theta,\tau\sim e_\xi}\left[R(\tau)\right]\right]$$

where τ is a trajectory collected in the source domain.

5.1. UDR

To perform our analysis on uniform domain randomization we will use PPO, we will first train the model on the source environment and then test it on the both the source and target environment. Two distributions to sample random parameters for the body masses were used: Normal distribution and uniform distribution; in both cases we ensured that the new random value didn't differ of more than 10% of the original value to do not have performance collapsing and also values that were too far from our goal of domain randomization. In practice, to sample new random parameters for the body masses we put the mean of the distribution equal to the original value of the mass, and then we apply sampling from the density of this distribution.

5.2. Results UDR with normal distribution

We report in the following table the results obtained with the respective train to test configuration, we provide also the plot for the source→target configuration, with domain randomization according to normal distribution.

Results UDR with normal dist.		
Configuration	Mean Reward	Std.Dev.
train on source	781.80	3.37
source→source	780.45	2.74
source→target	805.32	3.1

To analyze the result, we may note how the standard deviation has greatly improved in decreasing, even if the mean rewards have decreased we may note how the reduction in standard deviation can be explained as a more robust policy when the agent is put in the target environment and this is thanks to domain randomization.

5.3. Results UDR with uniform distribution

Following the same reasoning we applied UDR with a uniform distribution, and we report in the table the results:

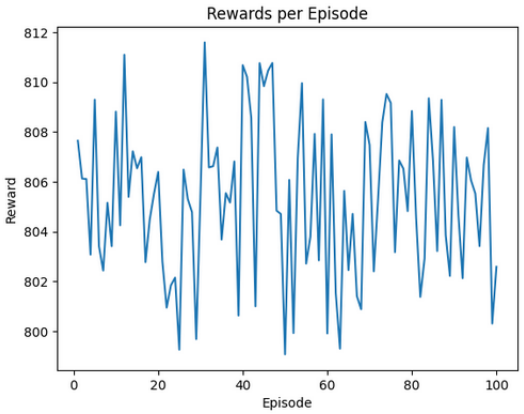

Figure 10. Rewards in test for source→target using UDR with normal distribution to sample parameters

Results UDR with uniform dist.		
Configuration	Mean Reward	Std.Dev.
train on source	1282.96	62.39
source→source	1266.50	55.88
source→target	1096.75	81.33

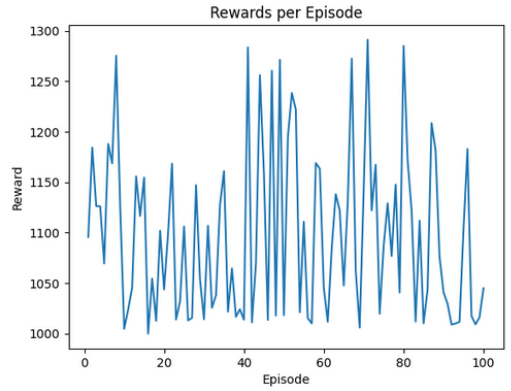

Figure 11. Rewards in test for source→target using UDR with uniform distribution to sample parameters

We may note how the uniform distribution used to sample random parameters results in better overall performances, specifically in the source→target configuration it performs better than the naive source→target without UDR where the rewards mean was around 931 while now is around 1100, this improvement is thanks to UDR.

6. Project Extension

In the last part of the project we focused on trying to implement our strategy of domain randomization that exploits an adaptive approach based on the performances of the agent during training. Naive domain randomization in-

volves varying certain parameters of the environment during training, for example the masses in our case. Adaptive domain randomization makes a step forward and its goal is that of adjusting the randomization based on the performances of the agent. In our implementation we used dynamic parameter adjustments with adaptive thresholds such as exponential moving average (EMA) to track performances trend and gaussian distributions to sample new parameters. For this last part of the project we decided to increase the number of timesteps to 800000 to effectively evaluate the performances of the agent, that was trained using PPO that was the method that showed the best trade-off in performances and time of training with standard learning rate. The EMA was used to keep track of the most recent rewards using a smoothing coefficient that makes the last observations of the rewards more important, in this way starting from a low threshold reward (50) we can randomize if the average of the last rewards is above this threshold and update the scaling coefficient of randomization and the performance threshold accordingly, setting it equal to the average of the last rewards; this ensures an approach of randomization that gradually adapts to the performances of the agent. The results showed a great improvement with respect to the previous implementations. As anticipated before, the new masses at each step of randomization were sampled using a normal distribution whose mean μ was set equal to the original masses (in order to have a sample of the same order of magnitude of the original masses) and the sample obtained was then scaled by a scaling factor based on the performances and added to the original value to obtain randomization.

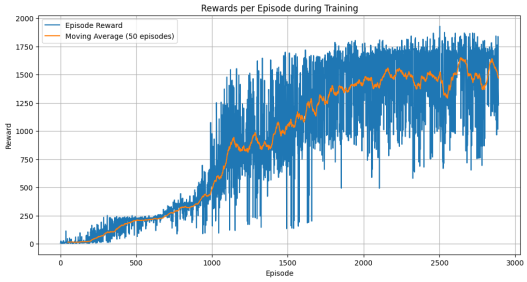

Figure 12. Rewards in training

In the test phase we obtained a mean reward of about 1738 and standard deviation of 11 which were the best results obtained so far.

7. Conclusion

In this project we presented different methods with the ultimate goal of training an agent able to produce a policy to make the hopper jump horizontally. The method REINFORCE didn't show great results but the addition of the baseline improved both the rewards and the variance, the same observation can be made for the actor critic A2C in which we didn't see a real improvement in training. On the other hand, much better results were obtained by implementing PPO and SAC, indeed these methods improved significantly the performances of the hopper that was able to achieve his goal. Moreover domain randomization provided robustness and stability to our agent, as it can be seen in the higher rewards and reduced variance obtained in the different configurations. Applying domain randomization was a fundamental step to mitigate the gap between simulation and reality, in fact the best performances were obtained thanks to domain randomization, specifically by implementing it with an adaptive approach that scaled according to the performances of the agent, increasing randomization when the agent was performing well.

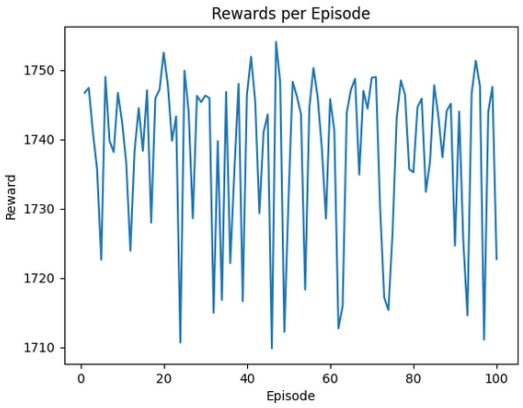

Figure 13. Rewards in test

References

[1] R. S. Sutton & A. G. Barto, "Reinforcement Learning: An introduction " 2

[2] J. Schulman, F. Wolski, P. Dhariwal, A. Radford & O. Klimov, "Proximal policy optimization algorithms" 2

[3] T. Haarnoja, A. Zhou, P. Abbeel & S. Levine, "Soft Actor-Critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor" 2

[4] P. Kormushev, S. Calinon, & D.G. Caldwell, "Reinforcement learning in robotics: Applications and real-world challenges" 2

[5] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, & P. Abbeel, "Domain Randomization for Transferring Deep Neural Networks from Simulation to the Real World." 2

[6] F. Muratore, F. Ramos "Robot learning from randomized simulations: A review." 2