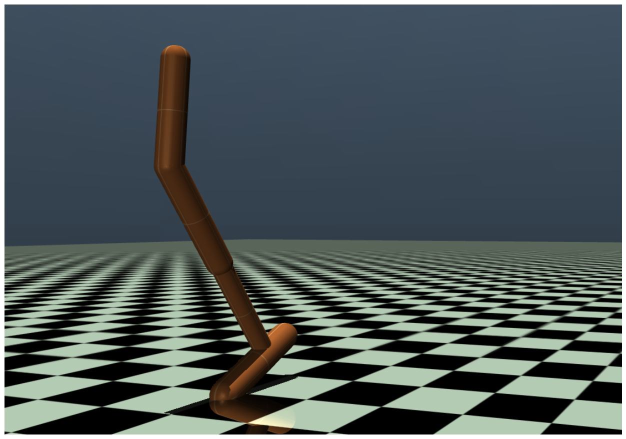
Guided Domain Randomization Through Adversarial Agent

Simone Carena
DAUIN
Politecnico di Torino
Turin, Italy
s309521@studenti.polito.it

Francesco Paolo Carmone
DAUIN
Politecnico di Torino
Turin, Italy
s308126@studenti.polito.it

Ludovica Mazzucco
DAUIN
Politecnico di Torino
Turin, Italy
s315093@studenti.polito.it

***Abstract*—Training agents directly in the real world is a time-consuming and risky-task, as it demands numerous trials in order to familiarize the learner to all the possible scenarios it could face during testing. This approach requires prolonged time requirements due to physical constraints and inherent potential risk to damage the agent structure. Hence, training in simulation begins to raise interest because of its remarkable upsides in terms of parallelizable computations and safety. However, one big issue remains unsolved, that is the repeatability of the different shades of real settings in artificial environments. To address this issue, we present a novel approach that recalls the *Uniform Domain Randomization*, and exploits an additional adversarial agent that is trained to hinder the main one to reach even better scores, so that the latter is driven to be more robust even when tested in reality.**


(a) Mujoco's Hopper environment

I. Introduction

When applied to robotics, Reinforcement Learning offers practical tools to control the behaviour of any given cyber-physical system as a viable alternative way to traditional control strategies based on laborious calculations and exact knowledge of the task environment. It is a middle point between Machine Learning and Control Theory, exploiting abstraction and adaptability capacities of the first, as well as applicability of the latter. In any case, state-of-art RL algorithms allow for optimal outcomes and are able to tie well known classical control techniques, even though they are based on heuristics and thus precise results are not promised and justified by standard theoretical studies.
The power of Reinforcement Learning approaches collide with the need of a non-negligible number of training sessions for the given agent, since not even the most efficient algorithm is able to learn with just a single trial: the more different the situations it faces, the better the result during test time. This fact represents a tedious obstacle for the training process, given that in the real world this would require lot of time and potential damages to the physical structure of the agent (i.e a robot). The solution to this would be to exploit computer simulation, as it makes possible to parallelize a bunch of possible settings in a relatively little time and without any risk. The challenge raised from this approach is that real environments are unpredictable and hard to be reproduced in a computer simulation details. The difference between the simulated environment, and the real environment is commonly referred as the *reality gap*, and the majority of methods exploited to solve it rely on randomizing simulation parameters during training so that it is more likely for the agent to experience something plausible to reality.
With this paper we are going to analyze some of those techniques such as *Uniform Domain Randomization* and *Guided Domain Randomization* in order to study their effect on the performance of the agent, which is in our case the Hopper from MuJoCo Physics Engine [1]. In particular, the aim of this document is to present a possible alternative way to train the learner in making use of an adversarial agent whose task is to hinder the task agent by trying to decrease its rewards; in this way the task agent is induced to be more robust even during the test on the real scenario.

II. Related work

As it was anticipated, studies aiming to close the sim-to-real gap focus on defining a probability distribution over the simulation parameters, so that at each training step the agent experiences a different shape of the environment and its policy is thus able to adapt well to the majority of real settings with the advantage of a training in simulation.

A. Uniform Domain Randomization

The vanilla approach for the objective mentioned above is Uniform Domain Randomization (UDR) presented in detail

in [2]. This is based on defining a closed interval for each simulation parameter, and then sample it uniformly to obtain the parameter used in the training session. In the special case explained in the paper, the task was object localization through RGB images and grasping, consequently the randomization was applied to light conditions, additional blur, camera positions and texture, orientation and color of the items present on the scene as well as different distractors that could tease the policy.
The authors demonstrated the efficiency of this algorithm comparing the results of the application on a pre-trained model and on one trained from scratch with the weights of the policy network randomly initialized, eventually both presented comparable performances. Moreover, the most significant feature of *UDR* is that it does not require any data from real world, since every component of the scene is automatically generated by the randomization process, thus it is possible to present to the agent billions of different scenarios.

B. Guided Domain Randomization

The downside of the method just presented is that it explores *every* possible combination of environment features to increase the likelihood to spot the real one, at the cost of an higher computational effort, given that the algorithm tries to model even unrealistic settings risking to deviate the policy to wrong directions. To face this issue, a variation of *UDR* is represented by Guided Domain Randomization, whose goal is to limit the randomization process in order to match as much as possible the real domain provided that real data is accessible or some measure of performance on the simulated task. Taking into account the work published in [3], from which we derive our intuition for this paper, a sort of Generative Adversarial Network (GAN) is set to allow the agent to become more robust to the change from simulation to reality. Going more in detail, it is made up of two main agents, the Deceptor (hence the name *DeceptionNet*), whose job is to become every time smarter to fool the agent, and the Recognizer, or *task agent*, a classifier in the case of that work. The Deceptor provides fake inputs to the task network and improves its loss function if the other agent makes mistakes on its prediction.

III. Background

In the previous section we have highlighted the theoretical fundamentals of our algorithm, but we have not mentioned yet *how* the policies are trained.

A. Actor-Critic Policy Gradient Algorithms

In this context, we make use of a particular type of policy grandient algorithms falling under the name of Actor-Critic methods. The update rule of policy parameters is formulated as follows:

$$\theta_{t+1} = \theta_t + \alpha(R_{t+1} + \gamma\hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w}))\nabla log\pi(A_t|S_t, \theta_t) \tag{1}$$

The quantity inside brackets looks like the TD-error, it is otherwise referred to as the *advantage* obtained from taking an action out policy and it exploits the approximation of the value function depending on the vector of weights $\mathbf{w}$. This is indeed a way of judging actions along the way the agent experiences the environment, this is where Actor-Critic algorithms derive their name from, since the approximated policy takes the role of actor (driving the behaviour of the agent) while the approximated policy embodies the critic (assessing the actor moves). In this case, two updates are performed at each iteration, the one to policy parameters θ following rule in [1] and the other to value function weights $\mathbf{w}$ as semi-gradient descent for function approximation.

B. PPO and SAC

For the training of our agents we entrust two particular Actor-Critic algorithms, Proximal Policy Optimization and Soft Actor Critic.
The main feature of the first is that it keeps the update step of the policy limited since a big step could move it away from the optimal generating a lot of variance and instability but a tiny one would slow too much the training. This goal is fulfilled substituting to the log-probability the ratio between the updated policy and the old one, representing how the probability of taking a certain action in a certain state has changed with the update:

$$r_t(\theta) = \frac{\pi_{\theta new}(a_t, s_t)}{\pi_{\theta old}(a_t, s_t)}$$

This term is then multiplied by the advantage as any Actor-Critic method. Additionally, a "clipped" version of the loss function is used, to be sure that the ratio above falls in between the interval $[1-\epsilon, 1+\epsilon]$. In a nutshell, PPO tries to maintain the updated policy fairly close to the previous one to guarantee stability.
Conversely, SAC introduces in the loss function a term depending on the entropy of the current policy paired with its entropy regularization parameter, in a way that increasing it will cause a preference to the exploration rather than exploitation.
For our purposes, we exploited SAC for training the deceptor agent, since it is more exploratory and thus generates useful inputs to feed the agent with, while PPO for the latter, leveraging its greater speed of convergence to the optimal policy.

IV. Method

Following the presentation of necessary background information, we disclose the details about our alternative approach for domain randomization through an adversarial agent. In a similar manner to Guided Domain Randomization (refer to II-B), our approach incorporates a Deceptor and a Recognizer, . However, in contrast to employing conventional neural networks, we conceptualize and implement two distinct agents based on Reinforcement Learning (RL): a task agent, and an adversarial agent. The latter engages in alternating training sessions with the first, and its reward system is the inverse of the task agent's.

A. Hopper Gym Environment

The agent we have exploited to develop our method is the Hopper environment from MuJoCo. It is a simple one-legged system with continuous observation and action spaces, the first consists in a vector of 11 elements representing positional values of different body parts of the hopper followed by their velocities, while the second is a vector of 3 entries that correspond to the torque applied to thigh rotor, leg rotor and foot rotor.

B. Adversarial Agent Development

For domain randomization we act on the masses of torso, thigh, leg and foot, in particular in order to simulate the difference between simulation and reality we train the agent in a *source* environment where the torso mass is one unit smaller than the one in the *target* environment. Body masses in the source domain are:

$$\begin{bmatrix}2.53429174 & 3.92699082 & 2.71433605 & 5.0893801\end{bmatrix}$$

while in the target domain are, respectively:

$$\begin{bmatrix}3.53429174 & 3.92699082 & 2.71433605 & 5.0893801\end{bmatrix}$$

Randomization is applied on the last three masses, that are thigh, leg and foot, so we expect that once in test session the agent is able to cope with the difference in the torso mass.
So, the adversarial agent takes in input the current masses of the agent together with the last episode return and its length, its policy generates values for the masses that it has experienced confuse the task policy and provide them to the latter. At this point, the task agent is evaluated with the new masses and its reward is used by the adversarial agent to generate its own reward as the inverse of the first one and update its policy.

C. Alternate Training of Both Agents

The key element for a successful learning process for this method is the alternate training of the adversarial agent and the task agent. In fact, in a first phase we focus on the development of the deceptor following the reasoning presented above, it is thus led to become smart at fooling the current task policy. In a second phase, intead, we move to the specialization of the agent that, after having received as masses parameters the ones generated by the just trained adversarial agent, is trained for 500 simulation steps in order to react every time more efficiently to the challenges of the deceptor.
At the end, we expect that once the model is tested in the target environment where the torso mass is altered, the resulting policy has become enough robust to cope with this discrepancy.

V. Experiments

In this section we discuss the most interesting part of the whole work, that are the outcomes of our approach applied to the Hopper. For the sake of precision, we compare the results obtained in three different settings: in the first, it is highlighted the performance of the agent simply trained in the source environment and tested in the target one, without any help from additional algorithms, then, for a further analysis, Uniform Domain Randomization is adopted in the second set of experiments, finally we test the capacities of our novel method.

A. Sim2Real Transfer Without Domain Randomization

First of all, source and target policies are separately trained on their corresponding environment and to make them excel a preliminary phase of parameter tuning is performed, in particular for both of them more than one learning rate is proved as well as multiple seeds. Research has found as best options a learning rate of $1e^{-3}$ with a seed of 5.
After that, the trained policies are tested in their own environment with the exception for the source one that is tested even in the target setting. Fig. 2 reports the plots of the output for each testing session, in which returns over a total of 100 episodes are shown together with the 50-episodes average return, plus episodes lengths are reported as well. As it can be noticed, agent performance resulted optimal in Fig. 2a, presented a lower-bound in the second setting (Fig. 2b) and enhanced with respect to the latter in Fig. 2c. This proves that without any intervention the task policy performs poorly when the agent is tested in conditions it has never experienced, conversely average episode return sharply increases when the test is carried out in the same setting of the training.

B. Sim2Real Transfer With Uniform Domain Randomization

In order to allow the agent to familiarize with the target environment without explicitly letting it know the exact parameters it will work with, we trained the model making use of Uniform Domain Randomization, which is also useful in order to consider a baseline to compare experiments on our approach with. Hence, during training session at each environment reset the masses of tight, leg and foot are sampled from a uniform probability distribution over the closed interval for each of them centered on their actual value and extended so that all mass magnitudes that could be generated would be feasible and strictly positive. We expect that, unlike the previous case, now the performance of the agent in the task environment results comparable (if not better) than the one where it was trained in and, as Fig. 3 shows, this is exactly what happens. The rationale we used to generate those plots is the same of the case presented in the previous section, in other words, we first tuned learning rate and seed to judge the ones leading to better results and exploited them for the policy evaluation, again the agent was trained with PPO actor-critic on one million of simulation time-steps.

C. Sim2Real Transfer With Adversarial Agent

In what this last section is concerned, further implementation details on our approach are provided. First of all, each training iteration is composed by 10 training steps of the adversarial agent exploiting SAC algorithm, followed by 500 training steps of the task agent with PPO, both with a learning rate of 0.0003. In order to achieve as efficient results as possible, we run a total of one million of iterations like the one just described and then tested the trained task policy on the target environment for 50 episodes, giving the rewards shown in 4. At this point, it is interesting to compare the performances on the target environment of all the studied approaches together, to do so each model has been trained in the same conditions for one million of simulation steps and the corresponding results are reported in Table I.

TABLE I: Summary of test results for the different approaches

Training env-test env	Test rewards on 50 episodes	
	avg	***std***
target-target	1712.438	500.0
source-target	583.996	174.66
source UDR-target	1285.869	357.22
source Adversarial-target	1473.113	403.52

VI. Conclusions

The experiments on the Hopper highlight satisfactory outcomes since our approach that exploits the adversarial agent not only is able to produce better rewards than the plain source-target, but also it outperforms its counterpart with Uniform Domain Randomization, with an average episode return of 1473 versus the UDR one with 1285, even though all the plots present a lot of spikes suggesting high variance.

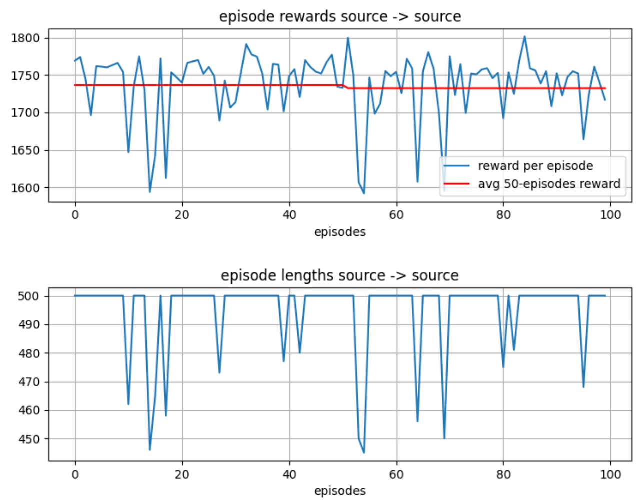
References

[1] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on, pages 5026– 5033. IEEE, 2012.

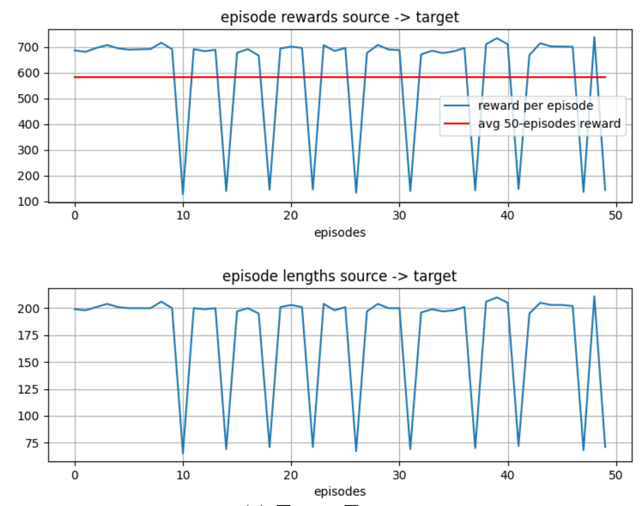
[2] Josh Tobin, et al. "Domain randomization for transferring deep neural networks from simulation to the real world." IROS, 2017.

[3] Sergey Zakharov,et al. "DeceptionNet: Network-Driven Domain Randomization." arXiv:1904.02750 (2019).

(a) Source-Source



(b) Source-Target



(c) Target-Target

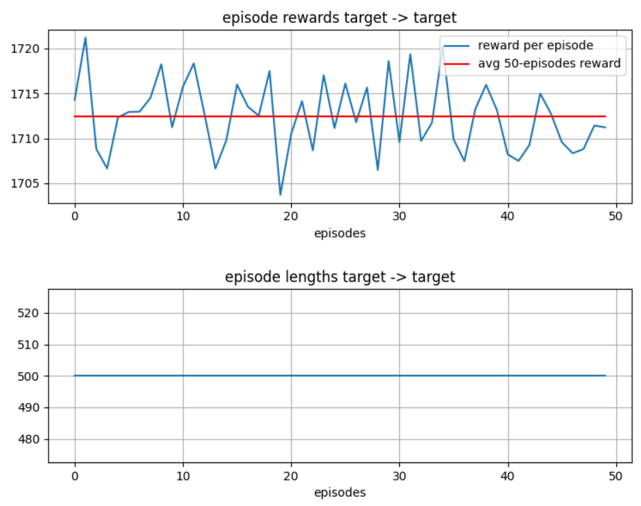


Fig. 2: **Results of source and policy evaluation on the corresponding environment plus test of source policy in target environment.**

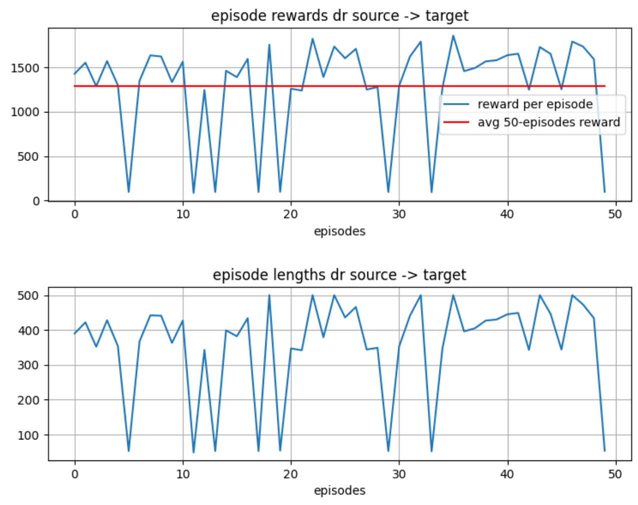


Fig. 3: **Results of source-trained policy evaluation on target environment through Uniform Domain Randomization approach.**

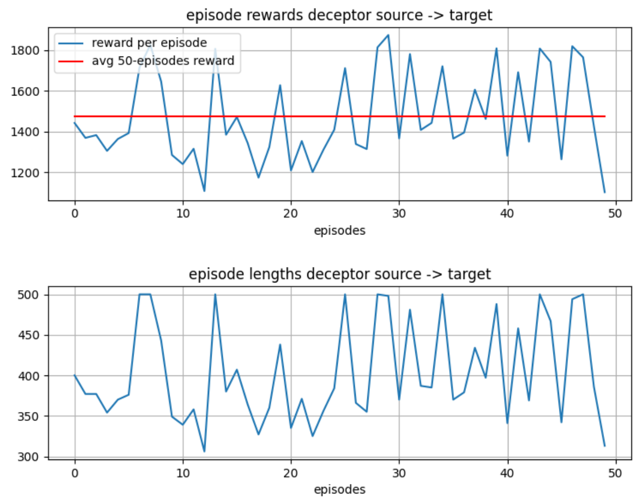


Fig. 4: **Results of task policy trained jointly with the deceptor agent and tested on target environment.**