Learning from Pixels and Domain Randomization using SAC

Marco Pratticò
Politecnico di Torino
Turin, Italy
s294815@studenti.polito.it

Fabiola Stancati
Politecnico di Torino
Turin, Italy
s291602@studenti.polito.it

Samuele Pino
Politecnico di Torino
Turin, Italy
s303332@studenti.polito.it

Abstract

In this work [1], we treat the Soft Actor-Critic (SAC) algorithm applied to the Hopper environment of Mujoco. We face the sim-to-real transfer problem, trying to illustrate the effectiveness of some techniques, such as Domain Randomization, to reduce the reality gap. We work in two different scenarios: the first in which the state is represented by the physical position of the hopper, and the second in which the state is represented by raw images. We demonstrated that DR is a good technique for reducing the reality gap in the first scenario, while in the second one, we are unable to draw a real conclusion regarding the DR since the computational costs are too high with respect to the available hardware and time. However, we set the course for further studies on vision-based RL.

1. Introduction

Reinforcement learning (RL) is one of the most active research areas in artificial intelligence. This is a computational approach to learning whereby an agent tries to maximize the total amount of reward it receives while interacting with a complex and uncertain environment. In general, RL offers robotics a framework and a set of tools for the design of sophisticated and hard-to-engineer behaviors [1]. Deep learning, together with Reinforcement Learning, can simplify the development of robotic systems. Indeed, rather than understanding the environment, it is better to simply collect a lot of experience and let the algorithm handle the rest. In this work, we put the spotlight on the Reinforcement Learning paradigm and its application in the context of continuous control for robot learning. Here, we face the problem of learning a control policy for a robot in simulation using state-of-the-art reinforcement learning algorithms and methods. In particular, learning the policies in simulation instead of in a real physical system can speed up the training time, avoiding costs that arise from interaction with real configurations. On the other hand, simply transferring the policy trained in simulation to a real physical system can lead to poor performance due to the "reality gap" between the simulator and the physical world. This challenge is also known as the "Sim-to-Real" transfer.

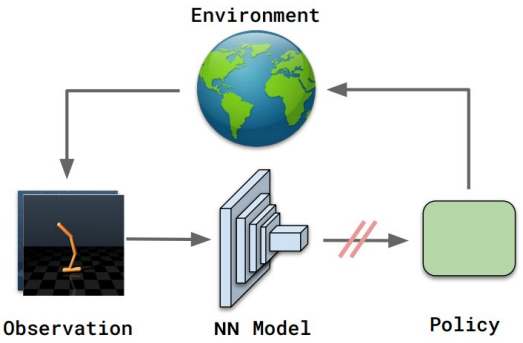

Figure 1. Basic model of deep Reinforcement Learning.

In this work, Sim-to-Real transfer activity was simulated in a sim-to-sim scenario, where a discrepancy between the source (training) and target (test) domains was manually added. So, the gap is triggered by an inconsistency between physical parameters in the two different domains. To close the sim2real gap, we used one of the most used techniques, that is Domain Randomization (DR). Through the DR, we are able to create a variety of simulated environments with randomized properties and train a model that works across all of them (Tobin et al. [2]), improving the robustness of the policy. In our work, this technique has been adapted to choose the hopper parameters according to a uniform distribution.

Finally, as regards the visual input control policies, we tried to train the Agent to no longer use inputs that exploited the position of the Hopper but use images instead. An attempt has been made to bring out the advantages and disadvantages deriving from this choice, subsequently, we set up a comparison of the results obtained. Therefore, it allowed the robot to autonomously interact with the envi-

[1] https://github.com/marcopra/AML-RL_project-main

ronment through raw images such as remarks. We demonstrated how the use of DR can improve the training and how the use of raw images as inputs, makes the training much slower with respect to the first scenario. Extending such methods to work with images is non-trivial. Usually, these methods require significant additional computation and algorithmic improvements to cope with the high-dimensional nature of images as inputs [3].

2. Related Work

2.1. Reinforcement Learning in robotics

According to the numerous applications of Reinforcement Learning, the one dealing with robotics is, without any doubt, particularly important. In fact, referring to [1], robotics as a reinforcement learning domain differs considerably from most well-studied reinforcement learning benchmark problems.

Experience in a real physical system is usually very slow, expensive, and often difficult to obtain and reproduce. Frame, suitable approximations of state, policy, value function, and/or system dynamics need to be introduced in order to learn within a reasonable time. This is a challenge called "Curse of Real-World Samples".

One way to counterbalance the cost of real-world interaction is to use accurate models as simulators but these approaches need to be robust with respect to model errors [4], defined as the "Curse of Under-Modeling and Model Uncertainty".

An often underestimated problem is the "Curse of Goal Specification", which consists in designing a good reward function. Laud et al. [5] remarked how this choice can make the difference between feasibility and an unreasonable amount of exploration.

Moreover, as the number of dimensions grows exponentially, more data and computations are required to cover the entire state-action space. This represents another important challenge for RL and this problem is known as the "Curse of Dimensionality". Finally, it is clear that in the world of Reinforcement Learning, there are still many open challenges that are far from solved.

2.2. Sim2real Transfer and Domain Randomization

Other works referred to the sim2real (or sim-to-real) Transfer problem and Domain Randomization as a possible solution for that. For example, according to [6], debates, posters, and discussions about sim2real are presented. Generally, since training an agent in a real physical system is expensive and unsafe, the use of software simulators is crucial. The problem is that transferring the trained agent from a simulation to a real physical system is not as simple as it seems because of the inaccuracy of simulators.

On the other hand, many efforts are devoted to overcoming the sim2real problem. For example, Domain Randomization has been proposed (Tobin et al [2]) as a simple technique that provides enough simulated variability at training time such that at test time the model is able to generalize to real-world data.

In [7], domain randomization is applied to train a real robot. In this case, the training had been conducted in a simulated system, and all the dynamics parameters were sampled according to a probability distribution, in order to reduce the "reality gap".

Other proposals are related to introducing Meta-Learning for improving Domain Randomization [8]. Indeed, the use of DR creates indirectly a batch of different tasks that the agent should solve. Meta-Learning is a model used for "learning how to learn" and this means that the agent should learn how to solve multiple tasks. Therefore, Meta-learning could generally be applied to empower the policy across different tasks and to perform further work in the real environment.

2.3. Learning from pixels

One of the most challenging approaches of RL systems is the one in which the agent should learn from raw images. The first developed methods have the goal of achieving good performance on Atari games. One of the most promising works [9] mainly consists of a convolutional neural model trained using Q-learning and experience replay, to break correlations between consecutive images. The input images are screens taken from the Atari games and are grouped in a batch of 4 images. This is done to make clear what is the movement in the game. However, other notable improvements had been made. For example, this approach suffers from overestimation bias in the action value function [10] and this is solved by employing double Q-learning [11] - using it together with deep reinforcement learning on the Atari domain fixed the overestimation problem that appeared in some of the games. The previous approaches are promising but they are still not able to reach human-level performance. Additionally, they referred to discrete action spaces and we propose some of these techniques in a continuous action space problem.

Other approaches, such as Image Augmentation [12], has been proposed to improve sample efficiency in some algorithms, such as SAC, and, moreover, this can prevent overfitting problems. This technique consists of padding (by repeating boundary pixels) images and then taking a random crop from that, yielding the original images shifted. They showed that these shifts enable SAC to achieve better performances.

3. Background

3.1. Reinforcement Learning

Reinforcement learning (RL) is learning what to do — how to map situations to actions — so as to maximize a numerical reward signal [13]. The main characters of RL are the agent and the environment. The environment is the world that the agent lives in and interacts with. At every discrete time step t of interaction, the agent sees an observation of the state $s \in \mathcal{S}$ of the world and then decides on an action $a \in \mathcal{A}$ to take accordingly to its policy $\pi : \mathcal{S} \rightarrow \mathcal{A}$ receiving a reward r and the new state of the environment s'. Then, the reward function could be defined as the discounted sum of rewards $R_t = \sum_{i=t}^{T} \gamma^{i-t} r\left(s_i, a_i\right)$ where $\gamma \in [0, 1]$ is a discount factor that determines the priority of short-term rewards, and T is the length of the episode in terms of time steps. So, beyond the agent and the environment, we can identify some sub-elements of an RL system: the state space $\mathcal{S}$, the action space $\mathcal{A}$, a policy $\pi(a_t, s_t)$, and a reward signal r. Different environments allow different kinds of actions and the set of all valid actions in a given environment is often called the action space $\mathcal{A}$. A policy π defines the learning agent's behavior at a given time. The role of the reward r is crucial in reinforcement learning as it defines the goal of a task. We can also use the reward function R_t - as described above - to take into consideration the sum of future (discounted) rewards.

Many Q-learning approaches make use of the recursive relationship known as the Bellman equation:

$$Q^{\pi}(s, a) = r + \gamma \mathbb{E}_{s', a'}\left[Q^{\pi}\left(s', a'\right)\right], \quad a' \sim \pi\left(s'\right) \quad (1)$$

which gives the value of the reward in the state s plus the value of wherever you land next. Here, the optimal policy is the one that takes the action a which maximizes the value of the Q function. A large number of non-linear function approximators for learning value or action-value functions has often been avoided in the past since theoretical performance guarantees are impossible, and practical learning tends to be unstable. For this reason, the use of large deep Q networks has been adopted to approximate these functions [14].

3.2. Markov Decision Process

When the reinforcement learning setup is formulated with well-defined transition probabilities it constitutes the standard mathematical formalism called the Markov decision process (MDP) [15] and it can be described as a 5-tuple $\langle \mathcal{S}, \mathcal{A}, R, P, \rho_0 \rangle$. Here, $\mathcal{S}$ is the set of all valid states, $\mathcal{A}$ the set of all valid actions, $\mathcal{R}$ the reward function, ρ_0 the starting distribution, and P is the probability function $P(s'|s, a)$ indicating the probability of transitioning into state s' if you start in state s and take action a.

3.3. Domain Randomization

Domain Randomization is a class of techniques for adaptation well suited for simulation [2]. With domain randomization, we create a batch of different simulated environments with randomized properties and train a model that works across all of them. The goal of DR is to provide sufficient simulated variability during the training time so that at test time the model can generalize the real-world data [16]. Instead of training a model on a single simulated environment, we randomize the simulator to expose the model to a wide range of environments at training time. We can define the source domain as the environment to that we have full access and the target domain as the environment to that we want to transfer the model.

During policy training, the parameter θ is trained to maximize the expected reward $R(.)$ averaged across a distribution of configurations:

$$\theta^* = \arg\max_{\theta} \mathbb{E}_{\xi \sim \Xi}[\mathbb{E}_{\pi_{\theta}, \tau \sim e_{\xi}}[R(\tau)]] \quad (2)$$

where τ_{ξ} is a trajectory collected in the randomized source domain (e_{ξ}) with ξ.

In addition to randomizing the visual features of a simulation, randomized dynamics have also been used to develop controllers that are robust to uncertainty in the dynamics of the system [7]. With the randomization of dynamics in the simulation environment during the training time, we can develop policies able to adapt to very different dynamics, including ones that differ significantly from the dynamics on which the policies were trained.

The randomization parameter ξ_i can be sampled in different ways. For example, it can be sampled using a certain probability distribution (e.g. uniform) in an interval $\xi_i \in [\xi_i^{low}, \xi_i^{high}]$ and it can be applied as noise or to slightly change the physical parameters.

3.4. SAC overview

For this work, we chose to use a *Sample Efficient* algorithm, that is Soft Actor-Critic (SAC) algorithm [17]. Indeed, this algorithm optimizes a stochastic policy in a *off-policy* way. The *off-policy* methods consist of training on the distribution of episodes produced by a different behavior policy, rather than that produced by the target policy. In this case, the training can reuse any past episodes - stored in an experience replay buffer - for better sample efficiency.

The SAC algorithm deals with entropy regularization. The goal of this algorithm is to maximize a trade-off between expected return and entropy, which measure the randomness in the policy. This is strongly related to the exploration-exploitation trade-off. SAC concurrently learns a policy π_{θ}, two Q-functions Q_{ϕ_1}, Q_{ϕ_2}, to reduce the overestimation bias [10], and a temperature α to find an optimal exploration-exploitation trade-off. At each iteration,

the SAC performs soft policy evaluation and improvement steps. The goal of SAC is to minimize the following Q-loss function:

$$L\left(\phi_i, \mathcal{D}\right) = \underset{(s,a,r,s',d)\sim\mathcal{D}}{\mathrm{E}}\left[\left(Q_{\phi_i}(s,a) - y\left(r,s',d\right)\right)^2\right] \tag{3}$$

where the target is given by

$$y\left(r,s',d\right) = r + \gamma(1-d)\left(\min Q_{\phi_{\mathrm{targ},j}}\left(s',\tilde{a}'\right) - \alpha \log \pi_\theta\left(\tilde{a}' \mid s'\right)\right) \tag{4}$$

where $\tilde{a}' \sim \pi_\theta(\cdot|s')$ (i.e. the next action is retrieved by the policy), and d is a value that indicates if we are dealing with the last step of the episode. Basically, the goal of the Q network is to minimize the discrepancy between the prediction and the target value.

The policy aims to maximize the expected future reward - triggered by the policy-chosen action - and the expected future entropy, according to the following formula:

$$V^\pi(s) = \underset{a\sim\pi}{\mathrm{E}}\left[Q^\pi(s,a) - \alpha \log \pi(a \mid s)\right] \tag{5}$$

So, the goal of SAC is to succeed at the task while acting as randomly as possible, by combining off-policy updates and a stochastic Actor-Critic formulation. The entropy maximization objective results in a more stable and scalable algorithm than deterministic ones, such as DDPG [18]. On the other hand, SAC also suffers from brittleness to the α temperature hyperparameter that controls exploration.

4. Implementation details

4.1. Environment and domain randomization

For our work, we chose the Hopper gym environment to train our RL agent. This environment comes with an easy-to-use python interface that controls the underlying physics engine, MuJoCo, to model the robot. MuJoCo is the first full-featured simulator designed from the ground up for the purpose of model-based optimization, and in particular optimization through contacts.

The hopper is a one-legged figure that consists of four main body parts, which will be the four masses: the torso at the top, the thigh in the middle, the leg at the bottom, and a single foot on which the entire body rests. The task of the hopper consists of learning to jump without falling while achieving the highest possible horizontal speed.

Here, the State Space contains all possible states that an agent can transition to. In this case, the State Space is continuous and it is composed of an 11-element vector, with each element in $(-\infty, \infty)$. The Action Space is a space that contains all possible actions that the agent can perform. In this case, the State Space is continuous and it is composed of a 3-element vector with each element in $[-1, 1]$.

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. To close the sim2real gap, we use the Domain Randomization technique [2]. Again, through DR, we create a variety of simulated environments with randomized properties (i.e. the leg masses) to improve the robustness of training. Remembering that the masses are 4, the first is shifted by 1 between the source - where policy training takes place - and target - which technically represents the real world. When we use DR, we take the masses' values (except for the first) according to a uniform distribution with the default value μ in the center and we chose to use the following distribution: $uniform(\mu - 0.5, \mu + 0.5)$. So, in this work, we adopted the Uniform Domain Randomization technique.

4.2. Physical state as input

In the beginning, we analyze the performance of the agent when we train it with and without Domain Randomization. In this case, the input of the networks is represented by the physical position of the leg, that is - as we previously said - a 3-elements vector.

We made the training on both environments, the source, and the target, and our goal is to see which are the effects of using the DR technique. We expect that this technique is able to reduce the reality gap and, then, increase the performance of the source-trained agent when tested in the target domain. Due to limitations of time and hardware, we couldn't train the agent several times and, then, we report the testing results of just one training. In this first part, we keep the same seed in the target and source training to reduce the probability of considering different stochastic events.

Moreover, We tried to analyze the training using two different sizes of the replay buffer. This buffer is a finite-size set $\mathcal{D}$ in which (s_t, a_t, r_t, s_{t+1}) (previous experiences) are stored. This buffer has two main goals: minimizing the data correlations and optimizing the training procedure by extracting a minibatch from the buffer. In general, all standard algorithms for training a deep neural network to approximate the Q-function make use of an experience replay buffer. Therefore, to have stable behavior in the algorithm, the replay buffer should be large enough to contain a wide range of experiences, but it may not always be good to keep a huge amount of data. If you only use the very-most recent data, you will overfit to that and things will break; if you use too much experience, you may slow down your learning [18], [19].

4.3. Vision-based RL for continuous control

At the end of this work, we use the model-free SAC algorithm to solve the Hopper problem using pixels as input rather than using the real physical position. From the environment, we directly retrieve 500×500 RGB images with a 255 color palette. The use of these raw frames can be

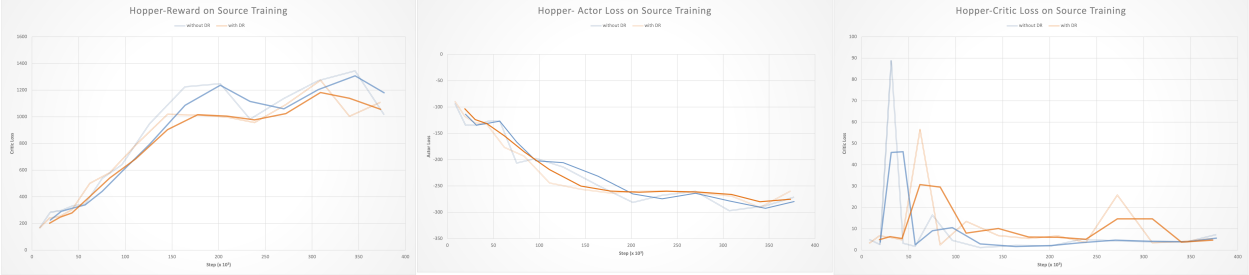

Figure 2. The plots show the reward, the actor, and the critic loss when training the agent with and without using *Domain Randomization*. The darker lines represent the smoothed plot. The blue line represents the training without DR, while the orange one represents the training with DR

computationally demanding, so, we pre-processed them by converting to grayscale and down-sampling them to 84×84 images. To make the Hopper dynamics more understandable, we stack the last 4 frames to produce the input of the networks. We used these parameters according to several previous works ([9], [20], [12]) and due to the time and hardware limitations described below, we were not able to try different configurations.

To better see the potential of this algorithm when using pixel input, we tried different approaches. First, we define two different backbones for the feature extractor part of both networks, the Actor and the Critic. As a backbone (the feature extractor part), we used the AlexNet [21] architecture and the ResNet-18 residual network [22]. In both cases, we used pre-trained weights. Since we slightly changed the network, in order to accept our stack of frames, we expect that the use of pre-trained weights does not give us many benefits. Due to a lack of time, we couldn't compare these configurations with the non-trained ones, so the previous affirmation is just an assumption. Additionally, we also tried to set the feature extractor part to accept each frame (RGB and down-sampled to 224×224) singularly. Then we stack the results of this part in a single flattened array before the fully connected part. We wished to see if this configuration could allow us to take advantage of the pre-trained weights. Due to the unfeasible amount of time for training ($\approx 24\,h$), we do not report the results of this part. Moreover, the use of this latter configuration forces us to reduce remarkably the size of the Replay Memory Buffer, because of GPU memory limitations. So, we would expect non-optimal results if using this configuration.

In these experiments, we opted to use a shared feature extractor part for the Actor and Critic network. Again, due to the lack of time and hardware, we opted to use a shared feature extractor network to maximize the training of this part in the small training sessions we could perform. Indeed, using the described configuration with 4 grayscale images, a single training session of only $300\,k$ steps lasts for at least $\approx 8\,h$, so our analysis is based on a few steps training sessions and this means that it is not optimal. For the sake of simplicity, we report more details on the networks' parameters in A.

After choosing the networks, we introduced image augmentation [12] by padding each side of the 84×84 frame by 4 pixels (by repeating the boundary pixels) and then we select a random 84×84 crop, yielding the original image shifted by ± 4 pixels. We apply this procedure only to frames sampled from the replay buffer. In this part, our goal is to see if image augmentation can improve the agent's performance.

Here, we trained the agent using different seeds, to be sure to avoid special cases. However, the reported results deal with the same fixed seed. We did not find a big gap in results when using different seeds.

Reward				
	100k steps		400k steps	
	w/o DR	w DR	w/o DR	w DR
Train: source - Test: source	972±269	988±26	1576±73	1503±74
Train: source - Test: target	858±191	918±23	1193±171	1428±41
Train: target - Test: target	868±27	-	1512±70	-

Table 1. Results when using the physical state as input. The results refers to training sessions perfomed with and without Domain Randomization.

5. Experiment Results

5.1. Physical State as input

At first, we trained our agent on the source environment and our goal is to see how the replay buffer size can benefit the training. As we can see from the plot 4, the training procedure when using a buffer size of 10^6 performs generally better than when the buffer size is 10^3. So, for further consideration, we will keep the buffer size fixed at 10^6.

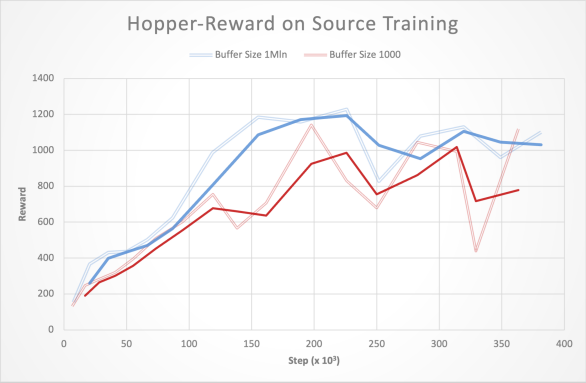

Figure 4. The plot shows the reward when training the agent on the source. The agent has been trained with two different *Buffer Sizes* that are 10^6 and 10^3. The darker lines represent the smoothed plot.

However, we trained the agent using the Domain Randomization technique. As the figure 2 shows, the reward of the agent trained using DR is lower. Indeed, the use of this technique improves the robustness of the training at the expense of optimization. In this case, the agent is trained on a batch of tasks, and the agent training is not completely focused on a single task.

In table 1, we reported the average reward when testing the agent on both environments, the source, and the target. The table clearly shows the benefits of DR, which reduces the reality gap. Indeed, the reward on the target of a source-trained agent is much higher when we used the DR.

In the table, we reported also the reward when we trained and tested the agent on the same target environment. We can consider this reward as the upper optimal bound. In the table, we also report the results when considering a few steps ($100\ k$). In this case, the performance on the target domain is comparable since the agent is not yet completely trained, and the agent reaches low similar performances in both cases, with and without DR.

In the end, we can say that DR is an effective technique to reduce the reality gap.

5.2. Images as input

In this experiment, we tested the performance of the agent when we use images as input. Here, the number of inputs is much bigger than in the previous case, so - as we previously said - we expect a slower train. We remark that in this section, we just considered training sessions of $300\ k$ steps, and considering the huge number of inputs, we assume that the agent cannot reach optimal performance. Additionally, as described above, we should consider that the use of the pre-trained weights, may not be useful for our task.

In figure 3, we see that the agent is able to reach a low reward (≈ 40). So, in this section, we try different approaches to try to maximize the reward in the very few steps of a training session. Generally, we can say the *ResNet-18* has slightly better performances than *AlexNet*. We remark that *ResNet-18*, even if it is deeper than *AlexNet*, has fewer parameters to train. For this reason, our goal is to see if this network is able to perform better in the small training sessions we could perform. Indeed, *ResNet-18* is able to better approximate the Q function, and this is evident from figure 3 where the *ResNet* Q-loss is lower than in the *AlexNet* case. Additionally, we underline the presence of narrow peaks in this latter case. We can address this phenomenon as overfitting, but, in general, is typical of deterministic policy as [10], and the SAC algorithm works with a stochastic policy. On the other hand, we may assume that this is due to the fixed *learning rate* we used, or, in sparse cases, the Critic prediction is highly affected by Actor's loss update (the feature extractor is shared within Actor and Critic). Un-

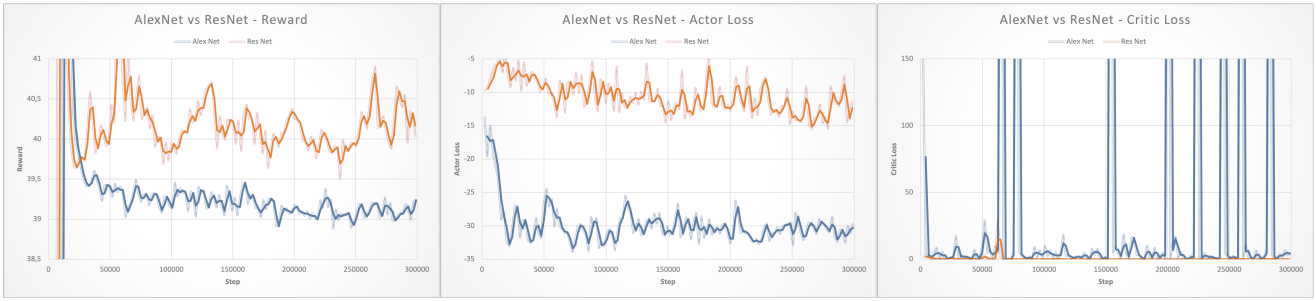

Figure 3. The plots show the reward and the losses of actor and critic when performing a training session using pixel images as input. Here, we compare the performance using AlexNet and ResNet-18 as feature extractors.

Reward				
	AlexNet		ResNet	
	w/o DR	w DR	w/o DR	w DR
Train: source - Test: source	38.5 ± 0.8	40.0 ± 0.9	40.6 ± 0.7	40.2 ± 0.8
Train: source - Test: target	38 ± 1	39.1 ± 0.8	40.1 ± 0.6	39.8 ± 1

Table 2. The results refer to training sessions with and without Domain Randomization using AlexNet or ResNet as Feature Extractor.

fortunately, we weren't able to perform many experiments to understand this phenomenon.

From the plot in figure 3, we see that the Actor's loss of *AlexNet* is lower than *ResNet* one. This could lead to thinking that the performance of AlexNet should be better than *ResNet*, but this is not the case. This discrepancy is due to the fact that the inaccuracies of the Critic network affect the Actor's loss (eq. 5).

We tried to introduce Image Augmentation into the training. We can see from figure 5 that this approach does not benefit the mean reward. The only effect of the Augmentation is that the gap between the *AlexNet* and the *ResNet* concerning the reward is reduced, because, in the *ResNet* case, the performance slightly decreases. The use of Image Augmentation does not make any change in the critic and actor losses. Since this approach does not give remarkable improvements, we will not consider Image Augmentation for further consideration.

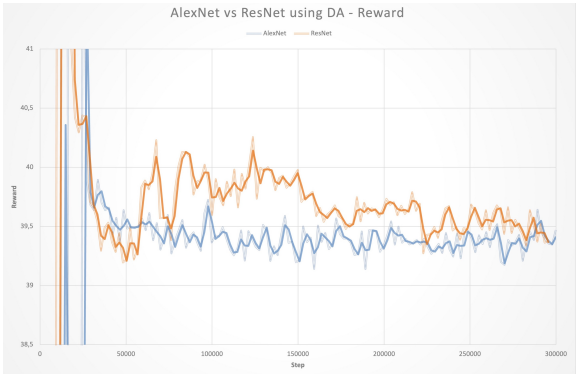

Figure 5. The plot represents the reward using the Image Augmentation when performing the training with AlexNet and ResNet-18

In the end, we test the agent trained with and without Domain Randomization (table 2). Here, we are not able to draw any conclusions about the effectiveness of DR, because these training sessions of the agent are not very effective. Generally, we can see that we can achieve slightly better performances when we use *ResNet 18*.

6. Conclusions

Here, we tested how much is effective the *Domain Randomization* in reducing the reality gap, between source and target domain. In the first case, when we used the physical state as input, we found that this technique is effective and allowed us to achieve better performances on the target domain.

Unfortunately, we couldn't reach the same conclusion when we faced the approach in which we used images as input. Indeed, this second approach has very high computational costs and requires very long training. So, in this case, we couldn't reach any real conclusions regarding *Domain Randomization*, but we set the course for further future studies related to vision-based RL. We remark that in this scenario we expect better results when we take real advantage of pre-trained weights analyzing each frame individually and in an RGB format.

References

[1] Jens Kober, J. Andrew Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *International Journal of Robotics Research*, 32(11):1238–1274, 2013. 1, 2

[2] Josh Tobin, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and Pieter Abbeel. Domain randomization for transferring deep neural networks from simulation to the real world, 2017. 1, 2, 3, 4

[3] Tianxin Tao, Daniele Reda, and Michiel van de Panne. Evaluating vision transformer methods for deep reinforcement learning from pixels, 2022. 2

[4] Björn Lütjens, Michael Everett, and Jonathan P. How. Safe reinforcement learning with model uncertainty estimates. *CoRR*, abs/1810.08700, 2018. 2

[5] Gerald DeJong and Adam Laud. Theory and application of reward shaping in reinforcement learning. 2004. 2

[6] Sebastian Höfer, Kostas Bekris, Ankur Handa, Juan Camilo Gamboa, Florian Golemo, Melissa Mozifian, Chris Atkeson, Dieter Fox, Ken Goldberg, John Leonard, et al. Perspectives on sim2real transfer for robotics: A summary of the r: Ss 2020 workshop. *arXiv preprint arXiv:2012.03806*, 2020. 2

[7] Xue Bin Peng, Marcin Andrychowicz, Wojciech Zaremba, and Pieter Abbeel. Sim-to-real transfer of robotic control with dynamics randomization. In *2018 IEEE international conference on robotics and automation (ICRA)*, pages 3803–3810. IEEE, 2018. 2, 3

[8] Lilian Weng. Meta-learning: Learning to learn fast. *lilianweng.github.io*, 2018. 2

[9] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013. 2, 5

[10] Scott Fujimoto, Herke van Hoof, and David Meger. Addressing function approximation error in actor-critic methods, 2018. 2, 3, 6, 9

[11] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016. 2

[12] Ilya Kostrikov, Denis Yarats, and Rob Fergus. Image augmentation is all you need: Regularizing deep reinforcement learning from pixels, 2020. 2, 5

[13] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018. 3

[14] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015. 3

[15] Richard Bellman. A markovian decision process. *Journal of mathematics and mechanics*, pages 679–684, 1957. 3

[16] Lilian Weng. Domain randomization for sim2real transfer. *lilianweng.github.io*, 2019. 3

[17] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor, 2018. 3

[18] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning, 2015. 4

[19] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin A. Riedmiller. Deterministic policy gradient algorithms. In *ICML*, volume 32 of *JMLR Workshop and Conference Proceedings*, pages 387–395. JMLR.org, 2014. 4

[20] Tianxin Tao, Daniele Reda, and Michiel van de Panne. Evaluating vision transformer methods for deep reinforcement learning from pixels, 2022. 5

[21] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012. 5, 9

[22] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015. 5, 9

A. Networks

In this section, we report the settings and further details for the vision-based approach discussed in the report. The networks we used as the Feature Extractor part are AlexNet and ResNet-18. We used the standard implementation offered by Pytorch. We only changed the first input layer and the last output layer in both our networks, in order to tailor it to the desired shapes. The SAC implementation of SB3 adds a few FC layers after the Feature Extractor. In particular, it adds these layers that take the extracted features as input, and then it tries to predict a future reward as close as possible to the real one. In fact, these two networks usually work with single RGB images $W \times H \times 3$, whilst, in this project, we worked with 4 stacked images each one with only one channel, such as $W \times H \times 1 \times 4$.

A.1. AlexNet

Our first candidate was AlexNet [21], one of the first Convolutional Neural Networks (CCNs) to obtain great results over ImageNet. It's not really deep, with only 5 Convolutional layers and 3 Fully-Connected layers. Despite these few layers, it counts more than 62 million parameters, which makes it quite heavy to train with our resources. A single training session of AlexNet, on just 300k steps, took approximately 9 hours.

A.2. ResNet-18

The second choice was ResNet [22] in particular ResNet-18, one of the few revolutions in the CNNs field, with a striking depth of 18 layers, and a simple but brilliant intuition to avoid Vanishing Gradients: Residual Blocks. Even though the number of layers in this architecture is more than twice that of AlexNet, the number of parameters is about 11 million. These features make ResNet-18 the perfect candidate for our goal, given our constraints in terms of computation and time. A single session training of ResNet-18, on just 300k steps, took about 4 hours. We thought that using a much deeper architecture might have improved the general quality of the extracted features.

A.3. Parameters

In our studies with SAC using images as input state, we tried different setups using Domain Randomization (DR) or Data Augmentation (DA). The tuned hyper-parameters we used for our tests with the SAC algorithm are in Table 3.

Our tests consist in using a combination of DA and DR so that some runs were made with only DA, others with only DR, and others with both techniques.

We choose 300k steps as the standard duration of the training when we deal with images because it was the best trade-off in terms of time.

Domain Randomization is obtained using the masses (except for the first) of the hopper as a random variable in $uniform(\mu_{mi} - 0.5, \mu_{mi} + 0.5)$. The final mass of the Hopper, used for a single episode, will be modified at each reset in that range. Data Augmentation consists of the first padding of 5 pixels (replicating the boundaries) so that the image, which is the state input, will be 94×94 starting from 84×84. Then a random crop is applied to obtain again images 84×84. This brief DA pipeline was implemented online so that every image passed as state input was augmented in the 'forward' step of the network.

Hyper-Parameter	Value
number of stacked frames	4
batch size	2/4
Memory Replay Buffer	10^6
Learning rate	3×10^{-4}
Time Steps	3×10^5

Table 3. Hyper-parameters settings for vision-based section.

B. Learning from pixels: comparison with custom DDPG

For the sake of curiosity, we tried to implement our custom DDPG - a deterministic algorithm - to see if the results we obtained depended on the SAC upgrades that have been made. Since the algorithm has been fully implemented by ourselves, it may be slightly different from the state-of-art. We can affirm that the DDPG is a kind of ancestor of the SAC algorithm and DDPG does not have some useful improvements, such as the twin critic network for reducing the critic overestimation bias.

Before going on, since the use of the DDPG algorithm is out of the scope of this work, we will focus mainly on the most important implementation details and results.

Regarding our implementation, we used two different networks for the actor and the critic with no shared feature extractor part. Both networks are made up of three convolutional layers and two fully connected. The weights are initialized according to the Xavier initialization. We trained the models for 3×10^4 episodes ($\approx 6 \times 10^5$ steps). The input images are taken as described in the main work without considering Image Augmentation. So, we input a stack of four 84×84 grayscale frames. The results we get are comparable to the results we obtained. Indeed, the average reward we obtained is ≈ 50 but considering a bigger amount of steps. We remark that in this case, the models are trained and tested only on the source environment. It's interesting to underline that the critic loss increases with the increase of the training episodes. This is due to an overfitting problem typical of deterministic algorithms [10]. Additionally, we can see that the policy loss decreases very slowly and this causes the slowness of the training.

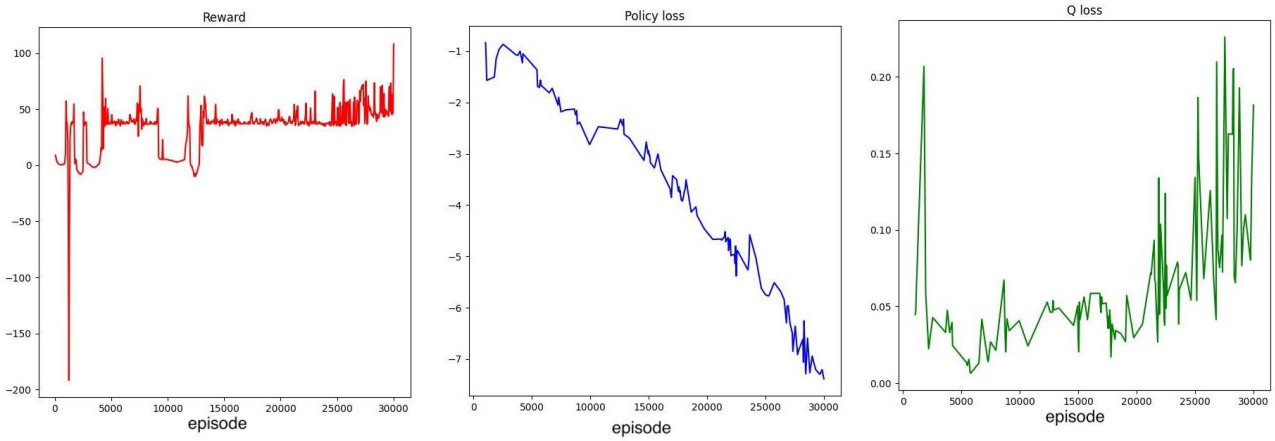

Figure 6. The plots show the result of our custom *DDPG* algorithm. On the left, there is the plot representing the reward over the episodes. In the center, the plot represents the policy loss. On the right, we see the Q loss over the episodes.

We can assume that the ineffectiveness of the (few-steps) training sessions is not due to the stochastic part of the SAC algorithm or to the other improvements, but probably because of the high computational costs due to the use of images. In order to draw accurate conclusions we should analyze longer training sessions.