Sim-to-Real transfer of Reinforcement Learning policies in robotics

Federico Boscolo
294908

Andrea D'Attila
303339

Antonio De Cinque
303503

Link to project's GitHub repository

***Abstract*—The objective of this project is to gain hands-on experience with Reinforcement Learning by applying it in the context of robot learning. We have been introduced to state-of-the-art RL algorithms and methods for learning control policies for a robot in simulation. The focus will be on sim-to-real transfer, a common challenge in the field of robot learning, where policies trained in simulation must be transferred to real-world hardware. To tackle this issue, the project will implement domain randomization, specifically Uniform Domain Randomization (UDR), a technique to make policies robust to variations between the source and target domains. Additionally, we will explore using visual input (images) to train control policies directly through CNNs.**

I. Introduction

Reinforcement Learning is a machine learning paradigm that focuses on enabling an agent to learn how to maximize a numerical reward signal by mapping situations to actions. It is different from supervised and unsupervised learning, as it is not based on labeled examples or hidden structure in data. Reinforcement Learning involves trial-and-error search and delayed reward, making it an interesting and challenging field to study.

One of the unique challenges in Reinforcement Learning is the trade-off between exploration and exploitation, where the agent must try a variety of actions to gain a reliable estimate of their expected reward while also exploiting what it has already experienced to obtain reward. Balancing exploration and exploitation is a long-standing issue in mathematics and remains unresolved.

The four main elements of a Reinforcement Learning system are:

- The *policy*, which defines the agent's way of behaving in a given state;
- The *reward* signal, which sets the goal of the problem;
- The *value function*, which determines the long-term desirability of states;
- The *model* of the environment, which mimics the behavior of the environment, or more generally, allows inferences to be made about how the environment will behave.

Reinforcement Learning can be used to solve various problems in robotics. It enables robots to learn optimal behaviors through interactions with the environment. The designer provides feedback in the form of a scalar objective function, which measures the one-step performance of the robot.

In our project, we will analyze the example of a **hopper**, a one-legged figure that consists of four main body parts - the torso, the thigh, the leg, and a single foot. This model is provided by the OpenAI gym framework, specifically among the MuJoCo environments, which contain different physics simulations for different types of robots. Some of these environments are analogous to real-world robots, and are widely used in machine learning applications to run simulations and train various types of machine learning algorithms.

The goal of the hopper is to make hops that move in the forward (right) direction by applying torques on the three hinges connecting the four body parts. The hopper provides an interesting example of reinforcement learning as it poses a challenging control problem due to the complex dynamics involved in hopping motion.

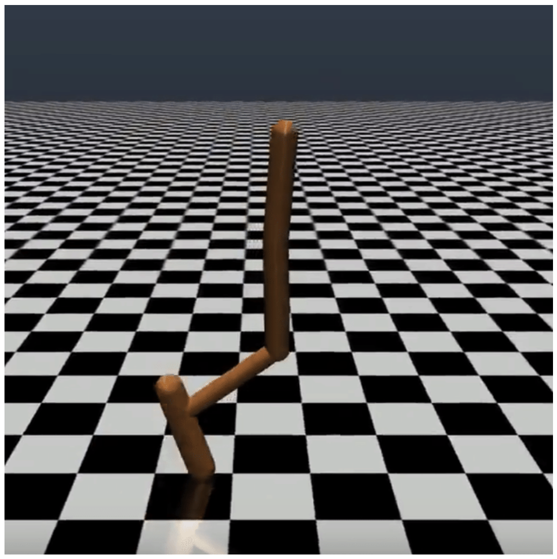
Fig. 1: Hopper representation in MuJoCo

The Hopper environment provides observations via a continuous observation space, which is a vector of eleven values. They describe, in order:

- The z-coordinate of the top of the hopper (m);
- The angle of the top of the hopper (rad);
- The angle of the thigh joint (rad);
- The angle of the leg joint (rad);
- The angle of the foot joint (rad);
- The velocity of the top's z-coordinate (m/s);
- The angular velocity of the top's angle (rad/s);
- The angular velocity of the thigh hinge (rad/s);
- The angular velocity of the leg hinge (rad/s);
- The angular velocity of the foot hinge (rad/s).

Actions over the Hopper can be done in the provided action space, which is also continuous, and represents the possible torques to apply to the different rotors of the hopper (thigh, leg, foot). The action space being continuous allows for smooth control over the robot's movements.

The Hopper environment also provides a reward function to help train agents. A good reward function is fundamental for successful RL pipelines, as it implicitly defines an objective for the algorithm. The reward consists of three parts: a *healthy reward* assigned for every timestep that the hopper is healthy, a *forward reward* which is assigned for hopping forward, a *control cost* reward which penalizes steep movements in the applied torque. The total reward returned is $reward = healthy\_reward + forward\_reward - ctrl\_cost$

In this analysis, we will explore how Reinforcement Learning algorithms can be applied to the hopper problem to find an optimal policy that balances stability and efficiency in hopping motion.

In our experiment, we had two types of hopper environments: **source** and **target**, where the target environment is meant to represent a real-world environment in which the hopper's parameters differ from the simulation. The only modified parameter in our case study was the torso mass. The objective of our research was to adapt the Reinforcement Learning agent to account for the changed mass, using Uniform Domain Randomization (UDR) as well as other techniques. Next, we implement a vision-based agent to simulate a real-world application in which the only input to the agent is a series of images of the environment. We trained various algorithms (such as PPO, TRPO, and SAC) on both environments and compared their performance through various train-test combinations such as source-source, source-target, and target-target. The source-target (agent trained on the source environment and tested on the target environment as-is) performance serves as a lower bound on the source-target adapted performance: the aim of UDR is to improve model performance through generalization. The target-target (agent trained specifically on the target environment and tested on it) serves as an upper bound on our agent: the aim of our model is to reach the performance of a specially-trained one.

II. Related Work

In recent years, there has been a growing interest in Reinforcement Learning applied to Robotics. Many researchers have explored various aspects of this area and proposed different solutions to the problems encountered. [2]

In particular, Sim-to-Real transfer in robotics has been extensively studied and several approaches have been proposed to address this challenge. [3] [8]

Sim2Real, or simulation to reality transfer, is an emerging field in robotics with the goal of improving the performance of robots in real-world scenarios by using simulations during training. To further explore the state of the art and future of Sim2Real, a scientific workshop was held at the Robotics: Science and Systems conference in 2020. [4]

The workshop included three debates with controversial key statements, aimed to consider a wide range of arguments. The topics included the importance of investing in Sim2Real, the definition of Sim2Real, and the application of Sim2Real. The workshop also featured 18 peer-reviewed abstracts and a panel discussion including all debate participants.

This workshop highlighted the growing interest in this field, both academically and industrially. Despite the lack of agreement on the overall effectiveness and future impact of Sim2Real, there was consensus on its current applications in robotics. The growing number of submissions in this field suggests the formation of a community and may even lead to the introduction of Sim2Real as a separate category during submissions to main robotics venues or dedicated meetings. Despite uncertainty about its future, Sim2Real remains an exciting area to explore from various perspectives.

III. Methods

A. Implementing and testing different algorithms for Reinforcement Learning

We have implemented a Reinforcement Learning pipeline to train a control policy for the Hopper environment, making use of a third-party library (`stable_baselines3`) to train an agent with state-of-the-art reinforcement learning algorithms such as TRPO, PPO, and SAC.

TRPO and PPO are **on-policy** algorithms, while SAC is an **off-policy** algorithm.

- In reinforcement learning, "on-policy" algorithms refer to methods that learn from the current policy being executed in the environment. That is, the learning process updates the policy based on the actions taken by the current policy in the environment, and then continues to execute the updated policy. In other words, the learning process is intertwined with the decision making process.

- On the other hand, ”off-policy” algorithms learn from data generated by another policy, called the behavior policy, that is different from the current policy being optimized. In other words, the learning process is decoupled from the decision making process. Off-policy algorithms can learn from past experiences, even if those experiences were generated by suboptimal policies.

- On-policy methods are typically faster to converge and can learn from the current experiences, but they can be limited by the quality of the current policy.

- Off-policy methods are generally more sample-efficient, but they can be slower to converge and are more sensitive to the choice of the behavior policy.

The Trust Region Policy Optimization (TRPO) [5] algorithm is a practical algorithm for optimizing policies in reinforcement learning. It is based on the theoretically-justified procedure for policy optimization and makes several approximations to make it more practical. TRPO is effective for optimizing large nonlinear policies, such as neural networks, and has been shown to be robust and provide monotonic improvement on a wide variety of tasks. This includes controlling simulated robotic swimming, hopping, and walking gaits, as well as playing Atari games using images of the screen as input. Despite its approximations, TRPO has a low sensitivity to hyperparameters, making it a reliable choice for reinforcement learning applications.

Proximal Policy Optimization (PPO) [6] is a family of policy gradient methods for reinforcement learning that optimizes a surrogate objective function using stochastic gradient ascent. The method alternates between sampling data through interaction with the environment and optimizing the objective function using minibatch updates. PPO is simpler to implement compared to TRPO and has better sample complexity. It outperforms other online policy gradient methods in terms of balance between sample complexity, simplicity, and wall-time.

Soft Actor-Critic (SAC) [7] is an off-policy actor-critic deep reinforcement learning (RL) algorithm.

Actor-Critic is a TD (Temporal-Difference) algorithm that combines both the actor and critic concepts.

The agent does not have a model of the environment, meaning it does not know the transition dynamics or the reward function. Instead, it must learn from experience, by observing the states it transitions to and the rewards it receives.

TD methods work by using the temporal difference (TD) error, which is the difference between the predicted reward for a given state and the actual reward received. The algorithm uses this TD error to update the estimated value of each state, with the goal of converging towards the true value function. The value function represents the expected sum of future rewards the agent will receive if it follows a certain policy.

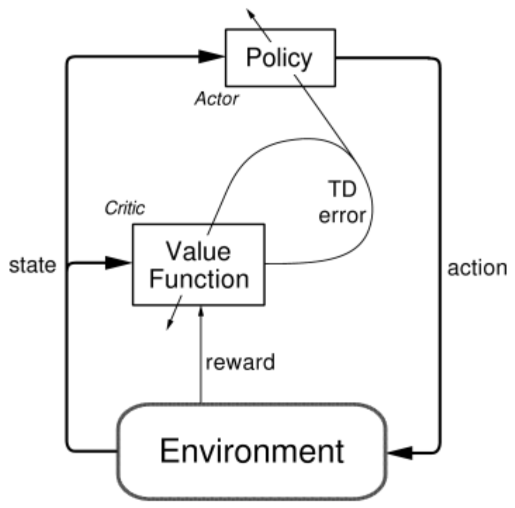

Fig. 2: The actor-critic architecture

- The **actor model** is responsible for selecting actions, and is typically represented by a neural network that outputs a policy, mapping states to actions. The actor model updates its policy based on the gradients of the value function with respect to the actions, using policy gradient methods.

- The **critic model** is responsible for evaluating the value of states and actions, and is also typically represented by a neural network that outputs a value function estimate. The critic model updates its value estimates based on TD error.

The idea is to use the critic to provide a strong, stable estimate of the value function, which the actor can then use to improve its policy.

- Actor-Critic algorithms are considered to be on-policy because they use the current policy to generate data, and then use this data to improve the policy. The current policy is updated after every step, so the data used to train the model is **always the most recent**. This makes the algorithm adapt to changes in the environment more quickly.

- In contrast, Soft Actor-Critic (SAC) is considered to be an off-policy algorithm. This means that it uses data collected **from multiple previous policies**, rather than just the current one, to update the policy. This makes it more stable and sample-efficient, as it can leverage experience from previous policies to make better decisions. However, it also means that SAC is less reactive to changes in the environment.

SAC is based on the maximum entropy reinforcement learning framework. The framework is based on the principle of maximum entropy, which states that the best policy is the one that maximizes the entropy of the action distribution while still satisfying the constraints imposed by the expected reward. The algorithm aims to maximize expected reward and entropy, with the goal of achieving the task while acting as randomly as possible. The method is formulated as a combination of off-policy updates and a stable stochastic actor-critic formulation, which allows for better performance on continuous control benchmark tasks compared to prior on-policy and off-policy methods. The authors also show that SAC is more stable than other off-policy algorithms, as it achieves similar performance across different random seeds.

B. *Implementing Domain Randomization*

Performing robotic learning in a physics simulator can accelerate the impact of machine learning on robotics by allowing faster, more scalable, and lower-cost data collection than is possible with physical robots. However, the discrepancies between physics simulators and the real world create a barrier to using simulated data on real robots, known as the reality gap.

Domain randomization is a technique used to address the ”reality gap” in training machine learning models. The reality gap refers to the difference between the simulated environments used for training and the real-world environments in which the models are deployed. To address this gap, domain randomization randomizes the simulated environment during training to expose the model to a wide range of varying environments. This helps to ensure that the model generalizes better to real-world environments, potentially eliminating the need for additional training on real-world data. The hypothesis being tested is that if the variability in simulation is significant enough, the models trained in simulation will generalize to the real world without additional training.

We have applied Uniform Domain Randomization (UDR) to the masses of each link in the Hopper environment. This involves creating a uniform distribution of probability to randomize the three remaining masses in the source environment, with the torso mass fixed at -1 kg relative to its actual value. During training, values are sampled from the chosen distribution for each episode.

Therefore, we had a custom implementation of the Hopper environment with an added feature of domain randomization optimization. The environment inherits from the MujocoEnv class provided by the OpenAI gym library and implements additional methods to support domain randomization. The masses of the hopper's links can be set randomly within a specified range, and the environment can be reset to a random initial state.

The goal of UDR is to encourage the agent to maximize its reward and solve the task in a range of environments, making its behavior more resilient to small variations in dynamics. However, the success of UDR depends on the choice of distribution, which must be selected carefully as it is a hyperparameter of the method.

The UDR agent was trained in the source environment using the same reinforcement learning (RL) algorithm as before. The obtained policy was then tested in both the source and target environments, and the average reward was recorded in a table.

C. *Vision-Based Reinforcement Learning*

In this last part, we modified the training process so as to only use raw images (i.e. 2D renders) to represent the environment rather than low-dimensional configuration vectors, and we used a CNN to extract relevant features from these images to then feed to the PPO policy. The rendered frames are pre-processed to reduce the workload of the CNN while still retaining relevant information embedded in the images.

First, the rendered images are converted to grayscale and resized to 224x224 pixels. In doing so, the images are downscaled and their RGB channels are reduced to only one. Next, multiple frames are stacked together as channels to give more information to the network about the direction and velocity of the Hopper's bodies. Different amounts of stacked frames have been tested, and we found that 12 frames is the optimal stack size. The entire process is shown in Figure 6.

We also tried training different CNNs: we trained a ResNet18 and a modified VGG architecture. Then, we obtained the best results by using a custom CNN architecture, which extracts a feature set of 128 features from the frame stack, using a custom architecture described in Figure 4.

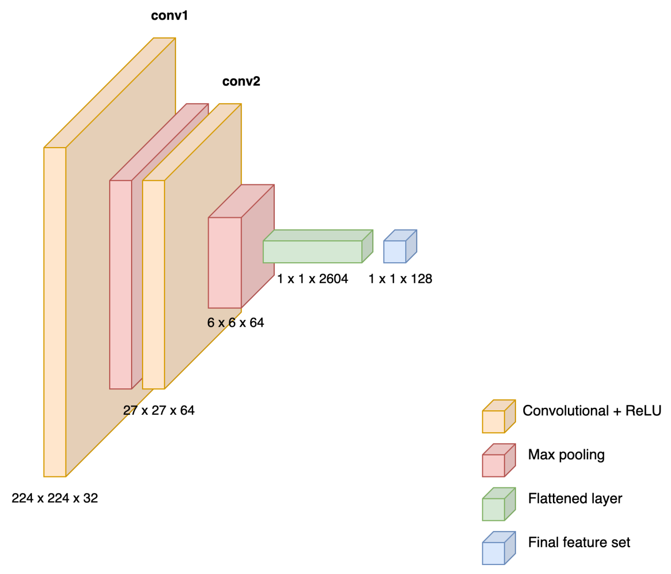

Fig. 3: The CNN architecture used in our experiments: it consists of two 2D convolution layers with ReLU activation and Max-pooling, followed by a flattened layer containing all the features extracted from the images.

IV. Experiments

A. Implementing and testing different algorithms for Reinforcement Learning

In this scenario, two custom variants of a policy have been created for a machine learning model. The policy is trained in both a source environment and a target environment, which is intended to represent the real world. To simulate the difference between the source and target domains, the source domain has been modified by changing the torso mass by 1 kg: the mass values of each link in the source variant of the Hopper environment are $[2.53, 3.93, 2.71, 5.09]$, while the mass values of each link in the target variant are $[3.53, 3.93, 2.71, 5.09]$.

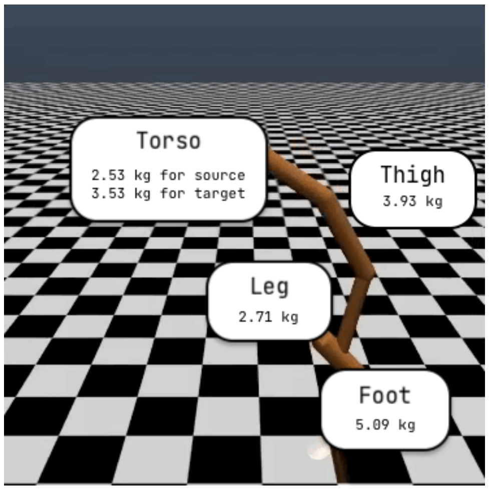

Fig. 4: Mass values of each link in the Hopper, visualized.

After training two agents on the source and the target domain Hoppers, each model has been tested and an evaluation has been conducted by reporting its average return over 50 test episodes.

In the study, three algorithms were trained to evaluate their performance in terms of computation time and average reward. The results showed that PPO performed the best, being faster than TRPO and SAC, and providing good average rewards.

In Table I are reported results for the three analyzed algorithms, using three different ”training-test” configurations. Each model has been trained for 500,000 timesteps, using the default learning rate.

Algorithm	Source-Source	Source-Target	Target-Target
TRPO	1641.26	947.37	1065.94
PPO	1626.11	958.5	1619.94
SAC	1559.79	1221.37	1596.08

TABLE I: All models have been trained for 500,000 timesteps, with $\alpha = 3 \times 10^{-4}$.

A hyperparameter search was then conducted, varying the learning rate and number of timesteps for training. We have trained using different learning rates: 10^{-3}, 3×10^{-4} (default), 10^{-4}, 10^{-5}, and 10^{-6}. The results in Table II showed that the default learning rate of 3×10^{-4} provided the best performance.

Learning Rate	Source-Source	Source-Target	Target-Target
10^{-3}	1263.95	706.13	1257.46
3×10^{-4}	1626.10	958.49	1619.94
10^{-4}	972.62	889.32	1186.15
10^{-5}	138.85	251.49	451.83
10^{-6}	52.44	58.70	57.96

TABLE II: PPO trained using different learning rates.

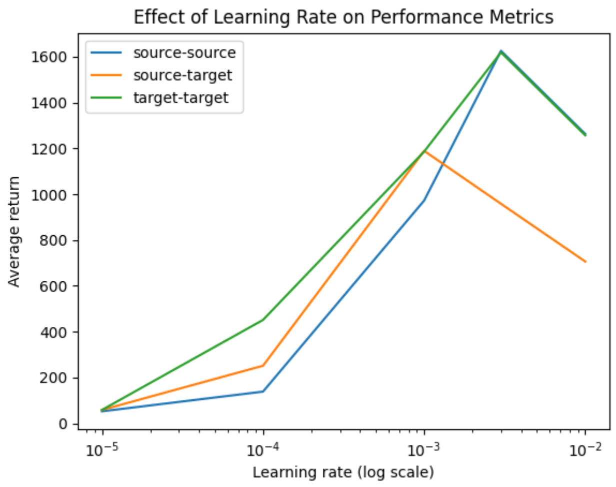

Fig. 5: This graph describes the effect that learning rate has on the average return over 500,000 timesteps. We are particularly interested in the Source-Target return, since that is the main parameter of evaluation.

We also tried to find the optimal number of timesteps, training for 100,000, 500,000 and one million timesteps. The best was found to be one million, but to reduce computation times, most of the analysis was conducted with 500,00 timesteps, which still allows to obtain good performances and also mitigates the risk of overfitting to the source environment.

From now on, the Proximal Policy Optimization (PPO) algorithm with a learning rate of 3×10^{-4} will be used for all the analysis, and the model will be trained for 500,000 timesteps.

The results show the existence of a reality gap, where a model trained on a source environment performs significantly worse on the target environment. In the following section, we will try to reduce the reality gap by using the technique of Domain Randomization.

B. Implementing Domain Randomization

Our analysis was conducted using the PPO algorithm with a learning rate of 3×10^{-4}. To identify the best hyperparameters for the source-target configuration, we performed a grid search. We trained our model with different numbers of timesteps: 100,000, 500,000, and one million. In addition, we randomly sampled 3 of the 4 masses in the Hopper, excluding the torso mass, since it should maintain a fixed value, using a uniform distribution built over the original mass value (Uniform Domain Randomization, UDR), with the lower and upper bounds being the original masses multiplied by a random proportion factor. This factor was used as a hyperparameter and we tested different values (30%, 50%, 70%) to experiment with the maximum divergence from the mass in the source environment. The masses ranges corresponding to each random proportion are reported in Table III

RandP (%)	Masses Range
30	[2.75kg, 5.23kg], [1.91kg, 3.52kg], [3.56kg, 6.62kg]
50	[1.96kg, 5.89kg], [1.35kg, 4.06kg], [2.54kg, 7.63kg]
70	[1.18kg, 6.68kg], [0.81kg, 4.61kg], [1.52kg, 8.65kg]

TABLE III: Maximum masses variations for each different configuration. RandP is the random proportion of masses relative to the source observation space

Finally, we also varied the number of episodes used for training each random configuration, testing values of 1, 5, 10, and 50.

This means that every n episodes the masses of the links are randomized according to an uniform distribution, leading to a different, random set of parameters every time.

Obtained results are shown in Tables IV, V, VI.

Rand. proportion (%)	Episodes	Source-Source	Source-Target
30	1	1113.25	1327.52
30	5	746.26	725.06
30	10	998.81	680.80
30	50	1262.16	1281.82
50	1	1154.29	981.56
50	5	886.73	754.63
50	10	557.95	540.91
50	50	815.10	685.58
70	1	779.33	631.81
70	5	945.90	802.93
70	10	957.13	1036.18
70	50	1048.13	683.83

TABLE IV: PPO using UDR, 100k timesteps. ”Episodes” refers to the number of episodes that use the same randomized environment.

Rand. proportion (%)	Episodes	Source-Source	Source-Target
30	1	1610.74	1640.48
30	5	1367.04	826.38
30	10	1578.28	1622.75
30	50	1139.45	1001.50
50	1	1498.96	1143.03
50	5	1295.77	695.71
50	10	953.38	898.31
50	50	1067.16	1209.69
70	1	1224.76	975.22
70	5	1127.27	795.48
70	10	1189.94	1015.87
70	50	1562.06	997.55

TABLE V: PPO using UDR, 500k timesteps.

Rand. proportion (%)	Episodes	Source-Source	Source-Target
30	1	1454.16	1582.84
30	5	1334.90	725.81
30	10	384.70	317.53
30	50	1657.74	706.22
50	1	1388.10	810.01
50	5	1364.54	999.44
50	10	1487.01	990.08
50	50	1220.54	811.86
70	1	1459.83	1220.87
70	5	1332.43	801.90
70	10	1262.18	1003.75
70	50	1808.22	1115.36

TABLE VI: PPO using UDR, 1M timesteps.

Our analysis showed that the use of UDR does not necessarily lead to improved results in all cases. For example, too few different random environments lead to worse results in both Source and Target environments.

The best results so far have been achieved with the following values: randomization proportion = 30%, number of episodes for each randomized environment = 1, and total number of timesteps = 500,000. Using this configuration, in fact, the average reward obtained over 50 episodes, for the Source-Target environment is 1640.48. These hyperparameters are used for the following step as well.

C. *Vision-Based Reinforcement Learning*

The goal of this step is to implement a Reinforcement Learning pipeline that uses raw images as state observations to the agent rather than low-dimensional configuration vectors, and to use a Convolutional Neural Network (CNN) as the underlying policy structure.

The images rendered by the environment are used as observations after being pre-processed. Three pre-processing steps are implemented before feeding the observation to the policy: first, the output frames are converted from RGB to grayscale, reducing the images' channels from 3 to 1, and they are then resized from 500x500 pixels to 224x224. This step is fundamental to ensure that excess information is stripped from the inputs, leading to faster and more efficient training. However, by feeding only a single rendered frame to the model, it is unable to accurately determine the velocity of the hopper's bodies. Thus, multiple consecutively rendered frames are stacked to give context to the model. The number of stacked frames was experimented with, and we found 12 to be the best performing solution.

This approach is also commonly used to train RL agents to play Atari games [10], and in that case a frame skipping technique is employed to maximize the amount of information contained in the frame stack that is input to the policy. This also keeps into consideration hardware limitations of the Atari 2600, which only updates positions of game objects every two frames. We experimented with frame skipping techniques before adding frames to the stack, however we did not find

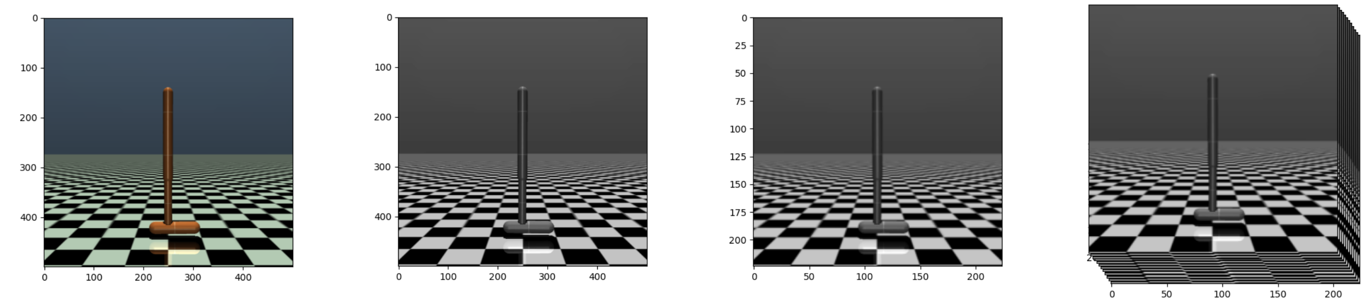
(a) PixelObservationWrapper (b) Grayscale (c) Resize (224 x 224) (d) StackFrame (12 frames)

Fig. 6: Vision-Based Reinforcement Learning - Observation spaces of wrappers pipeline

any significant improvements. Therefore, no frame skipping is employed in our agent.

The CNN then processes the frame stack, and reduces it to a set of 128 features. This number of features is the one that led to the best performance by the agent. CNNs are not perfectly efficient in extracting features from images, therefore it is unrealistic to expect the network to reduce the number of features to only those described in the original state. More efficient results can be obtained by employing deep networks with more complex architectures, however that comes with a great cost in terms of computation time.

The set of features output by the CNN is used by the PPO algorithm to execute actions. We maintained the hyperparameters that gave us the best results, so we used 10^{-3} as learning rate. We implemented UDR, using the best hyperparameters found, randomization proportion = 30% and number of episodes for each randomized environment = 1. The masses of the links are randomized every episode, and the state's render is then processed as described above.

We trained this model for 100k timesteps, and we conducted our evaluations over 50 episodes, computing as always the average reward. The results are described in Table VII.

Algorithm	Steps	RandP (%)	Eps	S-S	S-T
PPO	100,000	30	1	76.09	120.06

TABLE VII: The final results of the PPO algorithm trained for 100,000 timesteps. RandP is the random proportion of masses relative to the source observation space; Eps describes the number of episodes that each randomized environment trained for (one), S-S describes Source-to-Source performance, S-T describes Source-to-Target performance.

The obtained results are significantly lower respect to those obtained in previous steps. This is to be expected, and it is mostly due to the fact that implementing a good working vision-based algorithm is an extremely difficult task. Training a CNN is also a very resource-heavy task, and should be done for very long periods of time on powerful machines to obtain meaningful results.

However, UDR provided great benefits to the vision-based approach, increasing performance on the target environment considerably with respect to the source environment. No improvements could have been hoped for without implementing UDR.

Further optimizations for this step that could be implemented include: using a deeper CNN for the policy, or pre-training a CNN to map stacks of consecutive frames to values in the observation space.

V. Conclusion

In conclusion, our results show that Uniform Domain Randomization (UDR) is a very effective technique to tackle some of the issues with the Sim2Real transfer problem. By using it, we were able to bridge the gap between the source and target environment quite effectively: the average return for Source-Target performance went from 958.5 (PPO without UDR) to 1640.48 (PPO with UDR, with thigh, leg and foot masses in the range $m \pm 30\% \times m$), with an increase in performance of 71%.

As for the vision-based approach, UDR was very effective in this aspect as well, providing acceptable results while there would be none without using it. Further research in this field would have to focus on improving the mapping between image renders of the environment and actual observations, as to obtain better results using a vision-based approach to the problem.

In order to improve our CNNs' performances, different strategies could be implemented: it's possible to use a network that is pre-trained on a large amount of images to reduce computation time, but in our case this was not feasible, and using this approach would have forced us to use that particular image format. Pre-trained networks are often trained on RGB images, so our gray-scaled and resized images are not apt for that kind of strategy.

Another possible approach to obtain better results is using different parallelized environments: images from a single environment are often highly correlated, so, having different environment helps to have uncorrelated images, increases diversity of our samples and improves performances for our network. This is not feasible for us either in this particular case, given the hardware resources available for us.

The vision-based approach is the real focus of the Sim2Real transfer problem, as 2D images of the environment (i.e. pictures from a webcam) are often the only available feedback from real systems. In the case of real images, more techniques should be used to adapt the network to a wide range of possible observations, such as Domain Augmentation.

REFERENCES

[1] Richard S. Sutton and Andrew G. Barto, “Reinforcement Learning: An introduction (Second Edition)”,

[2] Kober, J., Bagnell, J. A., Peters, J. (2013). “Reinforcement learning in robotics: A survey”. The International Journal of Robotics Research,

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

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

[5] Schulman, J., Levine, S., Abbeel, P., Jordan, M., Moritz, P. (2015, June). “Trust region policy optimization”,

[6] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O. (2017). “Proximal policy optimization algorithms”,

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

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

[9] ”Actor-Critic Methods”, Mark Lee 2005, www.incompleteideas.net

[10] V. Mnih et al., “Playing Atari with Deep Reinforcement Learning.” arXiv, Dec. 19, 2013.