© 2022 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

This document contains the accepted version of the paper:
J. Esarte, P. D. Folino and J. C. Gómez, "Moving in a Simulated Environment Through Deep Reinforcement Learning," 2022 IEEE Biennial Congress of Argentina (ARGENCON), San Juan, Argentina, 2022, pp. 1-6, doi: 10.1109/ARGENCON55245.2022.9939868.

Moving in a Simulated Environment Through Deep Reinforcement Learning

Javier Esarte
Grupo de Inteligencia Artificial y Robótica
National Technological University
Ciudad de Buenos Aires, Argentina
esart@frba.utn.edu.ar

Pablo Daniel Folino
Grupo de Inteligencia Artificial y Robótica
National Technological University
Ciudad de Buenos Aires, Argentina
pfolino@gmail.com

Juan Carlos Gómez
Grupo de Inteligencia Artificial
National Institute of Industrial Technology
San Martín, Argentina
juanca@inti.gob.ar

***Abstract*— Reinforcement learning is a field of artificial intelligence that is continuously evolving and has a wide variety of applications. In recent years major progress has been made in the application of deep reinforcement learning to high-dimensional problems with continuous state and action spaces. This paper presents a complete analysis of the application of the soft actor-critic algorithm to teach a four legged robot with three joints on each leg how to move towards the center of a virtually simulated environment. The general formulation of the reinforcement learning problem is first presented, followed by the description of the environment under analysis and the applied algorithm. Afterwards, the obtained results are compared against those of a manually programmed policy, closing with a discussion of some key design choices and common challenges.**

Keywords—deep reinforcement learning, soft actor-critic, tetrapod robot, virtual environment, predictive control, machine learning, robotics, artificial neural networks

I. INTRODUCTION

A. Reinforcement Learning

From the beginning of Artificial Intelligence, the idea of making a machine learn was beguiling. In the earliest '70s at Air Force Cambridge Research Laboratories, A. Harry Klopf proposed a new theory on intelligent adaptive systems [1]. His pioneering work provided a framework to understand neurophysiological, psychological and sociological properties of a living adaptive system. In the end of his book he made an analysis and review of Neural Nets and Heuristic Programming Studies approaches for the modeling and synthesis of learning and adaptive cybernetic systems. Sutton and Barto, in the late '70s, picked up the gauntlet and tackled the issue of machine learning and established its foundation. As they expressed in their book [2] about the simplest concept behind adaptive and learning systems: "This was simply the idea of a learning system that wants something, that adapts its behavior in order to maximize a special signal from its environment". This kind of "hedonistic" point of view was later called Reinforcement Learning.

Many new reinforcement learning algorithms were developed from Sutton and Barto. Those new algorithms faced different problems in a great diversity of disciplines [3]. Despite the success of this approach, reinforcement learning problems with discrete states and actions faced the curse of dimensionality when the number of states and actions increases [4]. For continuous problems, techniques using parametric and nonparametric functions were developed. Some of them used Artificial Neural Networks (ANN) as function estimators to represent states, actions and policies learned [5].

Robotics is a good field for algorithm development in the area of reinforcement learning [6], especially for problems related to the design of complex behaviors, and even more so for those that face dynamic environments. Reinforcement learning algorithms provide a good trade-off solution by telling the robot what to do, through a reward function, and allowing it to learn how to do it by interacting with the environment. Heess et al [7] go further into this concept, saying that complex behaviors emerge from simple reward signals and challenge the agent learning with progressively more and more complex environments.

Another remarkable thing about reinforcement learning is that the "knowledge" gained during the learning process in the simulation stage can be transferred to the real robot. Therefore, it is not necessary to carry out the entire learning process in a physical robot, which can be initially replaced by a simulated model. Neither is it necessary to start from zero, it is possible to transfer knowledge from a human expert (if there exists) to an agent through instrumented command, literally copying the expert actions observed in real interactions. This saves time and money, while only requiring the agent's model.

B. Deep Reinforcement Learning

The first algorithm to successfully apply neural networks to reinforcement learning was the Deep Q-Network (DQN) algorithm [8], which optimized a neural network to estimate the state-action value function by sampling from a replay buffer, while the policy evaluated all possible actions in a discrete action space and selected the one that produced the highest state-action value.

Later, the Double DQN algorithm [9] introduced a change that reduced overestimation by adding a second set of weights for the state-action value function network and alternating on each learning step the weights between the online network (used to determine the optimal action) and the target network (used to estimate the state-action value).

There are also other algorithms like A3C [10] and TRPO [11], where the policy is explicitly represented and is optimized in order to maximize the objective function. These are called policy based methods, while the ones that try to determine the state value function or the state-action value function are called value based methods.

Both approaches have pros and cons, while policy based methods are more stable because they optimize directly the policy, value based methods find the policy indirectly but reuse data gathered more efficiently. Some algorithms combine both approaches like the deep deterministic policy gradient (DDPG) algorithm [12], which extends the structure presented by the DQN algorithm by combining it with

deterministic policy gradients and replacing the discrete action space used by previous algorithms by a parameterized function represented by a neural network allowing the use of continuous action spaces.

The soft actor-critic (SAC) algorithm [13][14], which will be explained in more detail in the following sections, shares a similar structure with the DDPG algorithm but replaces the deterministic policy by a stochastic one in order to regulate exploration according to the policy's entropy.

C. Project Context

The *Grupo de Inteligencia Artificial y Robótica* of the *Universidad Tecnológica Nacional* hosts a number of research and development projects focused on the application of reinforcement learning to simulated and physical robots. One of the group's first projects in this area targeted the planar motion of wheeled robots of different topologies. In recent years research on legged robots has begun with the objective of applying them to a variety of different surfaces and environment. The present paper conveys the results of the first stage of this project, the objective of which is the development of the framework required to carry out the agent's training in a simulated environment.

II. Methods

A. The Reinforcement Learning Problem

The reinforcement learning problem consists in finding the optimum policy, which directs the acting of an agent, in order to maximize the expected cumulative reward obtained from the agent's interactions with its environment.

A complete description of the environment on a given instant defines a state, and the set of all possible states defines the state space $\mathcal{S}$. Similarly, the set of all the possible actions an agent can take in the environment defines the action space $\mathcal{A}$.

A trajectory τ, also called episode, is the sequence of observed states and actions that begin in an initial state and span until a terminal state is reached or a maximum number of steps has been executed.

The reward function r: $\mathcal{S}\times\mathcal{A}\times\mathcal{S}\rightarrow\mathbb{R}$ associated to the environment represents the quality of the state transition produced by the agent's action. Equation (1) represents the infinite-horizon discounted return, the exponentially weighted average of all the obtained rewards in a given trajectory, with discount factor γ.

$$R(\tau) = \lim_{T\to\infty} \sum_{t=0}^{T} \gamma^t \, r(s_t, a_t, s_{t+1}) \tag{1}$$

The state value function is a function V^π: $\mathcal{S}\rightarrow\mathbb{R}$ defined by (2), which represents the expected return of a trajectory obtained by beginning in an initial state s and always acting according to the policy π.

$$V^\pi(s) = \mathbb{E}_{\tau\sim\pi}[R(\tau) \mid s_0 = s] \tag{2}$$

The state-action value function, also known as Q-function, is a function Q^π: $\mathcal{S}\times\mathcal{A}\rightarrow\mathbb{R}$ defined by (3), which represents the expected return of a trajectory obtained by beginning in an initial state s, taking an arbitrary initial action a and afterwards acting by following the policy π.

$$Q^\pi(s,a) = \mathbb{E}_{\tau\sim\pi}[R(\tau) \mid s_0 = s, a_0 = a] \tag{3}$$

The policy is the set of rules followed by the agent in order to choose its next action. Furthermore, it is possible to define the optimal policy as the arguments that maximize the state-action value function.

Finally, the Bellman equations (4) and (5) are a pair of self-consistency equations that recursively connect the state and state-action value functions to their values one step later.

$$V^\pi(s_t) = \mathbb{E}_{s_{t+1}\sim\mathbb{P}(\cdot|s_t)}\{\mathbb{E}_{a_t\sim\pi}[r(s_t,a_t,s_{t+1})] + \gamma\, V^\pi(s_{t+1})\} \tag{4}$$

$$Q^\pi(s_t,a_t) = \mathbb{E}_{s_{t+1}\sim\mathbb{P}(\cdot|s_t,a_t)}\{r(s_t,a_t,s_{t+1}) + \gamma\, \mathbb{E}_{a_{t+1}\sim\pi}[Q^\pi(s_{t+1},a_{t+1})]\} \tag{5}$$

B. The Environment

The environment is formed by an agent and the horizontal plane on which it moves. The agent's objective is to reach within a certain radius of the environment's center.

The agent, presented in Fig. 1, models a tetrapod robot with three degrees of freedom on each leg. The first joint controls the leg's position relative to the main body, while the remaining joints control the angle between adjacent leg segments. Each joint is capable of moving up to 45° in each direction with respect to the rest position.

The state space comprises the agent's position, the agent's orientation and the position of each of the agent's motor controlled joints, while the action space is composed of the target positions for all of the agent's joints.

The reward, defined in (6), is proportional to the change in the distance to the origin, except when the agent reaches the center position, in which case it receives the remaining distance as an additional.

$$r(d_t, d_{t+1}) = \begin{cases} 100(d_t - d_{t+1}) & si\ d_t > 0{,}1 \\ 100 d_t & si\ d_t \leq 0{,}1 \end{cases} \tag{6}$$

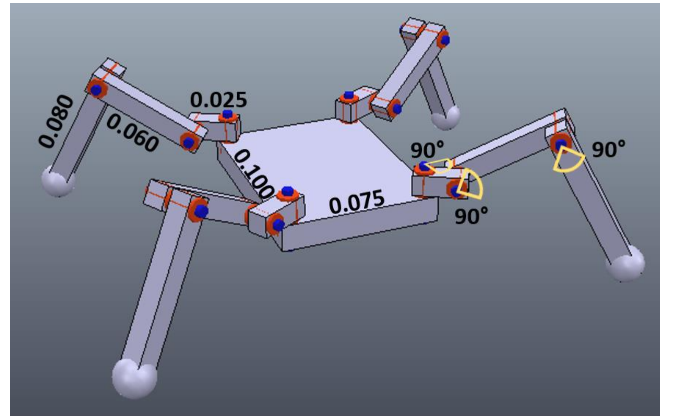

Fig. 1. Agent's model in rest position, dimensions and movement limits.

C. The Reinforcement Learning Algorithm

The agent was trained using the soft actor-critic algorithm [14]. As other actor-critic algorithms, it divides its functionalities in two separate blocks: the actor, which selects the next action to be performed by the agent from the agent's policy, and the critic, which evaluates the expected return of the action from the state-action value function.

Both the policy and the state-action value function are estimated by multilayered artificial neural networks, which provide a flexible way to represent parameterized functions. The stochastic policy network outputs a sample taken from a flattened Gaussian distribution (7), whose mean value μ and standard deviation σ are internally estimated by densely connected layers from the current state of the environment. The state-action value function network is composed by two independently trained, densely connected, subnetworks that estimate the state-action value, the minimum of which is then taken as the network's output, to reduce overestimation and improve the algorithm's convergence [15].

$$\pi(\cdot\,|s_t) = tanh\left(\mathcal{N}\left(\mu(s_t), \sigma(s_t)\right)\right) \quad (7)$$

Learning is achieved through an iterative process of policy application, shown in Fig. 2, and policy improvement, shown in Fig. 3. During policy application the agent acts in the environment, storing all state transitions in the replay buffer. Later the policy is improved by sampling the replay buffer and sequentially training the state-action value function neural network by stochastic gradient descent on (8) and the policy neural network by stochastic gradient descent on (9).

$$L_Q = \left\{Q^\pi(s_t, a_t) - \left[r_t + \gamma\,(1 - end_t)\,V^\pi_{target}(s_{t+1})\right]\right\}^2 \quad (8)$$

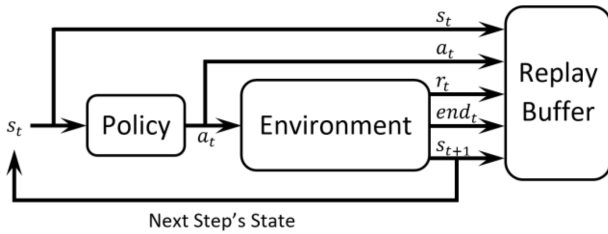

Fig. 2. Data flow during policy application. For each step, the policy selects for the current state s_t the next action a_t, which is then executed in the environment producing the reward r_t, the next state s_{t+1} and the flag end_t that signals if the reached state is terminal. Afterwards the obtained state transition tuple is stored in the replay buffer and the algorithm proceeds to the next step.

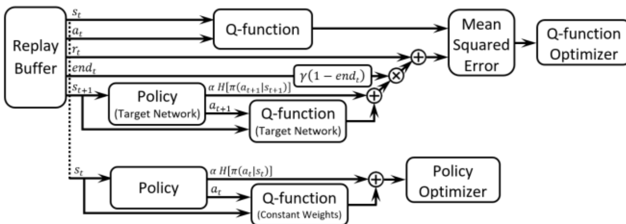

Fig. 3. Data flow during policy improvement. For each training step, a batch of state transition tuples are sampled from the replay buffer and used to train the state-action value function by minimizing the mean squared Bellman error. Afterwards the policy is trained by maximizing the output of the state value function.

$$L_\pi = -V^\pi(s_t) = -\{Q^\pi[s_t, a_t] - \alpha \ln \pi(a_t|s_t)\}\,;\; a_t \sim \pi(\cdot\,|s_t) \quad (9)$$

To regulate the balance between exploration and exploitation during training, the algorithm incorporates an entropy term to the reward. The entropy of a probability distribution is defined by (10) and can be interpreted as a measurement of the randomness of the distribution. When added to the reward, the entropy term preserves the standard deviation needed for the agent to explore, except in those cases in which selecting a specific subset of actions yields a reward greater than the lost entropy.

$$H[\pi(\cdot\,|s_t)] = \mathbb{E}_{a_t\sim\pi(\cdot|s_t)}[-\ln \pi(a_t|s_t)] = \int \pi(a_t|s_t)\,\ln\left[\frac{1}{\pi(a_t|s_t)}\right] da_t \quad (10)$$

Overall, the Soft Actor-Critic algorithm presents a low degree of sensibility to hyperparameter tuning, converging for a wide breadth of values. The first iteration of the algorithm [13] depended mainly on the tuning of the relative weight of the entropy term, controlled by coefficient α. The latest version [14] further simplified this process, dynamically adjusting the value of α during training by stochastic gradient descent on (11), which is approximated from the equations used to solve the dual of the reinforcement learning problem subjected to a minimum entropy constraint.

$$L_\alpha = \mathbb{E}_{a_t\sim\pi(\cdot|s_t)}[-\alpha \ln \pi(a_t|s_t) - \alpha\, H_{min}] \quad (11)$$

D. The Simulation

The simulation framework[1] was divided into a reinforcement learning module and a virtual environment module. This allows replacing one module without the need to modify the other, which simplifies the process of changing environments and reinforcement learning algorithms. Furthermore, to keep the modules independent, the exchanged state and action vectors are normalized to the [-1, +1] range before transmission and are only mapped to the environment's limits within the environment module.

The reinforcement learning module was completely developed in Python, using the NumPy library for vectorized computation of the neural networks. The virtual environment module was divided between the Lua script running in the robot simulator and the environment's interface in Python, the communication between both was implemented using sockets, providing a simple and flexible way to connect multiple parallel instances of the simulator to run simultaneously.

The virtual environment module used the CoppeliaSim robot simulator [16] to build a model of the agent by assembling primitive shapes through joints, the position of which can be controlled by setting a target position for the corresponding proportional–integral–derivative controller. Special care was taken to prevent the accumulation of error in the physics engine by reloading the agent model and restarting the simulation for each episode.

An episode begins in a random initial position and orientation selected by sampling uniform distributions and discarding any combination that has already reached the objective. An episode ends when either the agent reaches the objective, a maximum number of steps is carried out, or the agent gets too far away from the objective.

[1] Authors' code repository: https://github.com/GIAR-SAC/Tetrapod

For comparison's sake, the same environment was executed with a manually programmed policy, in which the legs over the main body's diagonals worked in pairs alternating between being raised and moving in the objective's direction and being lowered and moving in the diametrically opposite direction, resulting in the agent crawling towards the objective.

III. RESULTS

The agent was trained for a duration of 30000 episodes using CoppeliaSim's Newton engine in very accurate precision with a time differential of 50 milliseconds and a single pass per frame. Table I presents the set of hyperparameters used in the agent's training, which were referenced from [14] and tuned individually to improve the expected return and reduce simulation times. Also, the total number of neurons used in both estimators was lowered in order to reduce the computational capacity that will eventually be required for the physical implementation of the robot.

TABLE I. TUNED HYPERPARAMETERS

Hyperparameter	Value
Number of episodes	30000
Maximum number of episode's steps	1000
Replay buffer capacity	10^6
Discount factor	0.95
Update factor	0.005
Replay batch size	1000
Number of steps per update	3
Q-function neural network's layers	128 relu - 64 relu - 32 relu - 1 linear
Q-function optimizer (learning rate)	Adam (0.001)
Q-function regularization (weight)	-
Policy neural network's layers	64 relu - 32 relu - 12 Flattened Gaussian[a]
Policy optimizer (learning rate)	Adam (0.001)
Policy regularization (weight)	-
Initial entropy term weight	0.01
Entropy term weight optimizer (learning rate)	Adam (0.001)
Minimum expected entropy	$\mathrm{Dim}(\mathcal{A}) = 12$

a. Internally computes the mean using a hyperbolic tangent layer and the standard deviation using an exponential layer before using them to sample the flattened Gaussian distribution.

A comparison between the agent's trajectories starting from a variety of initial positions when following the programmed and the learned policies is shown in Fig. 4. Although both policies manage to reach the objective in a small number of steps, the way in which the agent moves is fairly different for each of them. The learned policy exploits the full movement capabilities of the joints, reaching the objective in a lower number of steps. Meanwhile, the programmed policy presents a more regular movement, behaving almost identically when moving in any direction.

The quality of a policy can be determined by the expected return that can be obtained by following it. However, this value may vary significantly depending on the initial state of the episode. For the proposed reward function, the state variables that produce the greatest changes to the expected return are the two components of the position over the plane and the agent's orientation.

Fig. 5 presents the obtainable return from different initial positions when the agent's orientation aligns with the direction of the objective for a number of cases. The rightmost plot presents the return that could be obtained if the agent reached the objective in a single step i.e. the maximum return. The second plot from the right show the return that could be obtained when performing equal distance steps of length 0.1 toward the center, the further away the initial position is from the objective the more the return changes when compared to the maximum due to the exponential increase in the accumulated discount factor. The third and fourth plots correspond to the obtained return when the agent acted by following the programmed policy and learned policy respectively, as the programmed policy needs a larger number of steps to reach the objective the discount factor had a higher impact in the return.

Additionally, Fig. 6 compares the obtained return when following the learned policy against the expected return computed from the learned state-action value function for the same set of states and the action selected by the learned policy for those states. The absolute difference between both was generally small, correctly approximating the expected return, becoming more significant around the edges of the area delimiting the possible random initial positions.

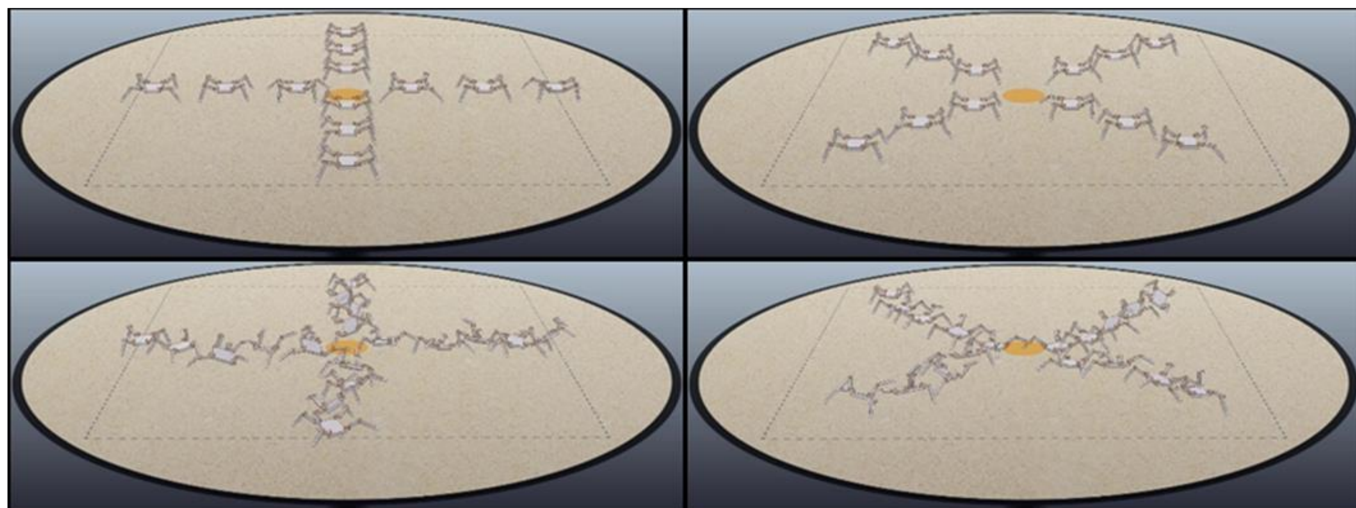

Fig. 4. Trajectories from episodes beginning in different initial positions when following the programmed policy (top) and the learned policy (bottom).

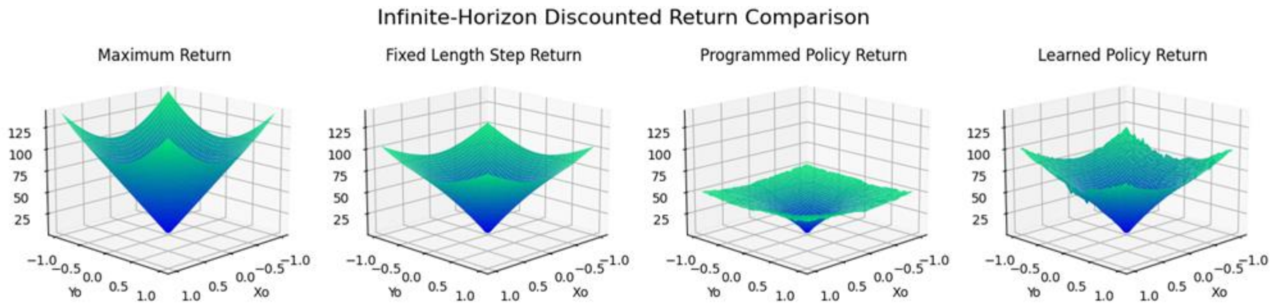

Fig. 5. Obtainable return from different initial positions when the agent's orientation aligns with the direction of objective.

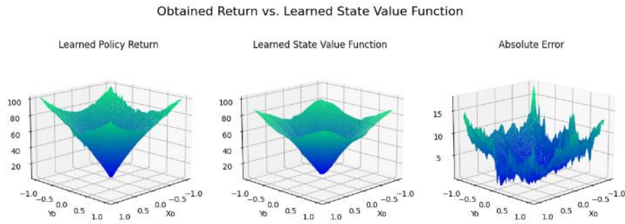

Fig. 6. Obtained return when acting according to the learned policy (left), expected return computed from the learned state value function (center) and the absolute difference between both (right).

IV. Discussion

A. The Reward Function

As in all reinforcement learning algorithms, the way in which the reward is defined becomes a critical point for the learning process. The selected reward function must promote the desired outcome while discouraging undesired behavior.

In this sense a number of alternatives could be proposed to the reward presented in this work. Giving a fixed reward only when the objective is achieved or making the reward inversely proportional to the remaining distance being just a few examples of this.

However, special care should be taken when evaluating these alternatives. A reward inversely proportional to the remaining distance will move the agent closer to the objective but it will learn to avoid reaching it in order to extend the episode and continue obtaining the high reward available in its proximity. On the other hand, rewarding the agent for fulfilling the objective will encourage it to reduce the length of the episodes in order to reduce the discount to be applied to the fixed reward, however it will not be able to differentiate the quality of the steps taken in early episodes as the reward will be zero for most of them and the state-action value function has not yet been stablished.

In comparison, the selected reward provided a way for the agent to evaluate the quality of the taken action since the beginning of the training while also avoiding certain caveats where the agent could exploit the reward function without reaching the intended objective completely.

B. Learned vs. Programmed

As mentioned in the introduction, in reinforcement learning the agent is told what to do, not how to do it. The learned behavior emerges purely from the reward with no additional knowledge of the complexities involved in moving the agent, while the programmed policy requires at least a basic understanding of how each action of the agent will affect its state, e.g. if all legs are kept touching the ground the robot slips on the same position instead of crawling.

Also, as can be observed from the presented results, the learned policy yielded a higher return when compared to the programmed policy, making the later more efficient from an optimality point of view. This stems mainly from the difference in complexity between both policies and is reflected in the agent's behavior, both when it performs larger movements and when it positions itself with its smaller side facing the objective in order to extend the reach of each step.

On the other hand, when comparing the predictability of both approaches, it becomes impossible to assert that the learned policy will not produce any erratic behaviors for certain states due to the dimensionality of the state space. One example of this behavior was observed during training for a small number of states in some iterations of the algorithm, where the agent flipped itself upside down, a state from which it cannot recover due to the limitations in the joints' movements.

C. Virtual Environment

Virtualizing the environment provides a number of benefits at the time of learning. First and foremost, the agent's behavior during the early stages is practically random, which may end up damaging the physical robot, whereas it produces no harm at all to its virtual counterpart.

Another advantage of the virtual environment resides in the possibility of running several instances in parallel. Which not only allows the execution of multiple instances of the algorithm to speed up hyperparameter tuning, but also provides a faster way of interacting with the environment by having multiple agents acting at the same time in separate environments while training a single policy and state-action value function.

Nonetheless, it must be noted that when transferring the learned policy to the physical implementation of the agent, it may be necessary to perform some additional episodes of training to adjust the policy in order to reduce the error due to inaccuracies inherent to the environment's model.

D. Adjusting the Learning Process

Monitoring the evolution of the learning process becomes an especially challenging aspect of high-dimensional environments. As the number of dimensions increases it becomes impossible to represent the policy, the state-action value function and the state value function in their entirety. This is the reason why Fig. 5 showed the return for a limited number of states, prioritizing those variables that produce the biggest changes while adjusting the rest to make the agent start in similar conditions from different positions.

The mean squared Bellman error is another relevant indicator that should be tracked during learning in order to

verify the self-consistency of the learned functions. Although it should be noted that it does not reflect the quality of the current solution but its validity and should always be paired with some measurement of the expected return.

Another challenge arises from the large volume of data that is generated as the training progresses. As the number of episodes increases the visual separation between points in the time axis becomes increasingly small, after a certain number of episodes they start overlapping and eventually result in the minimum and maximum values overshadowing all intermediate points in their surroundings. Reducing the number of plotted points could potentially solve this issue at the cost of losing some information. Instead, dividing the time axis in a fixed number of segments and computing the mean and a few percentiles on each segment produces a similar result without losing information.

Finally, high dimensional environments tend to require prolonged execution times for the algorithms to converge. To avoid having to start over on an eventual premature termination, e.g. power outage, lack of resources, etcetera, it is possible to periodically save the execution context alongside a random number, which is immediately set as the new random seed for the following period. In this way both when executing without interruption or when resuming execution, the random seed at the starting point after a save is always the same, making it possible to exactly replicate an execution from any given save point. This functionality can also be used to test different combinations of hyperparameters for environments that require extremely long training times that make it costly to restart training from zero.

V. Conclusion

In this work the application of a deep reinforcement learning algorithm to an environment with continuous state and action spaces has been analyzed, focusing on the design process of the framework, the transferability of the learned policy to a physical implementation and the repeatability of the presented results.

A brief compilation of the evolution of the field was presented in the introduction, followed by the formulation of the reinforcement learning problem, the description of the environment under analysis, the soft actor-critic algorithm and the simulation environment used to train the agent. Afterwards, the results obtained from applying the reinforcement learning algorithm were shown and compared against those computed from an idealized analysis and the ones obtained when following a manually programmed policy. Finally, some key design choices and possible solutions to commonly encountered difficulties were discussed in the last section.

References

[1] A. Harry Klopf, “Brain functions and adaptive systems - A heterostatic theory,” L. G. Hanscom Field, Bedford, Massachusetts, Special Reports, no. 133, 1972.

[2] R. Sutton, and A. Barto, “Reinforcement learning, an introduction,” Cambridge MA., MIT press, 1998.

[3] L. P. Kaebling, M. L. Littman y A. W. Moore, “Reinforcement Learning: a survey,” Journal of Artificial Intelligence Research, vol. 4, pp. 237 - 285, 1996.

[4] R. Bellman, “Dynamic Programming,” Princeton University Press, Princeton, New Jersey, 1957.

[5] R. Sutton, and A. Barto, “Reinforcement learning, an introduction,” Cambridge MA., MIT press, 2018.

[6] J. Kober, J. A. Bagnell, and J. Peters. "Reinforcement learning in robotics: A survey," The International Journal of Robotics Research, vol. 32, no. 11, pp. 1238-1274, 2013.

[7] N. Heess et al., "Emergence of Locomotion Behaviours in Rich Environment," arXiv, 2017.

[8] V. Mnih et al., "Playing Atari with Deep Reinforcement Learning," arXiv, 2013.

[9] H. Van Hasself, A. Guez, and D. Silver, "Deep Reinforcement Learning with Double Q-learning," Proceedings of the AAAI conference on artificial intelligence, vol. 30, no. 1, 2016.

[10] V. Mnih et al., “Asynchronous methods for deep reinforcement learning,” International Conference on Machine Learning, pp. 1928-1937, 2016.

[11] J. Schulman, S. Levine, P. Abbeel, M. Jordan, P. Moritz, “Trust region policy optimization,” International conference on machine learning, pp. 1889-1897, 2015.

[12] T. P. Lillicrap et al., "Continuous Control with Deep Reinforcement Learning," International Conference on Learning Representations, 2019.

[13] T. Haarnoja, A. Zhou, P. Abbeel, S. Levine, “Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor,” International conference on machine learning, pp. 1861-1870, 2018.

[14] T. Haarnoja, et al., "Soft Actor-Critic Algorithms and Applications," arXiv, 2019.

[15] S. Fujimoto, H. Hoof, D. Meger, “Addressing Function Approximation Error in Actor-Critic Methods,” International conference on machine learning, pp. 1587-1596, 2018.

[16] E. Rohmer, S. P. N. Singh, and M. Freese, “V-REP: A versatile and scalable robot simulation framework,” Proc. of The International Conference on Intelligent Robots and Systems, pp. 1321-1326, 2013.