Learning to Play Precision Ball Sports from scratch: a Deep Reinforcement Learning Approach

Liliana Antão*†, Armando Sousa†§, Luís Paulo Reis†¶, Gil Gonçalves*†
*SYSTEC - Research Center for Systems and Technologies
†FEUP - Faculty of Engineering, University of Porto
§INESC TEC - INESC Technology and Science
¶LIACC - Artificial Intelligence and Computer Science Laboratory
Email: {lpsantao, asousa, lpreis, gil}@fe.up.pt

***Abstract*—Over the last years, robotics has increased its interest in learning human-like behaviors and activities. One of the most common actions searched, as well as one of the most fun to replicate, is the ability to play sports. This has been made possible with the steady increase of automated learning, encouraged by the tremendous developments in computational power and improved reinforcement learning (RL) algorithms.**

This paper implements a beginner Robot player for precision ball sports like *boccia* and *bocce*. A new simulated environment (*PrecisionBall*) is created, and a seven degree-of-freedom (DoF) robotic arm, is able to learn from scratch how to win the game and throw different types of balls towards the goal (the jack), using deep reinforcement learning. The environment is compliant with OpenAI Gym, using the MuJoCo realistic physics engine for a realistic simulation. A brief comparison of the convergence of different RL algorithms is performed. Several ball weights and various types of materials correspondent to *bocce* and *boccia* are tested, as well as different friction coefficients. Results show that the robot achieves a maximum success rate of 92.7% and mean of 75.7% for the best case. While learning to play these sports with the DDPG+HER algorithm, the robotic agent acquired some relevant skills that allowed it to win.

I. Introduction

Over the last decades, the field of robotics has been evolving substantially. The scientific and technical challenges that were once only focused on industrial assembly lines now are also shifting towards interaction with humans, pursuing human-like behaviors and activities. In order to get robots to obtain these types of skills, typically, many hours of hard coding are involved. The coding becomes even more complex in regards to dynamic and elaborated environments that require reprogramming the robotic system every time a goal changes.

Due to this complexity, methods for learning behaviors automatically and generalize knowledge are more and more used in robotics. This is the case of RL methods that revealed an immense potential in a wide range of different robotics tasks. RL allows robots to autonomously determine the best behavior through experimental interactions with their environment and getting rewards from those interactions, rather than giving the solution to the problem by explicitly programming it [1]. Given that RL enables robots to learn from their own experience, it is a highly utilized method in non-static environments identical to pursuing human-like activities.

One of the examples of dynamic and complex environments, when it comes to human activities, is the world of sports and physical games. There has been an increasing interest in robotics competitions and performance in these types of challenges. Not only robot-against-robot team competitions, but also teams with both robots and humans, or even robot single players, are gradually more used in all types of sports. To give robots the ability to play a specific game, it involves several captivating research fields: from the obvious robotics area to computer vision, mechanical and electrical engineering, and, obviously, artificial intelligence. This makes robot sports appealing not only for researchers in these fields but also for the players and even people in search of entertainment [2]. In sum, robotic sports promote intelligent robotics by engaging several different types of communities.

(a) *Bocce* (b) *Boccia*

Fig. 1. Precision Ball Sports

In this work, we show that deep reinforcement learning allows a realistic simulated robotic arm to play a single-player version of Precision Ball Sports like *bocce* or its adapted version for players with severe physical disabilities, *boccia* (Fig. 1 (a) and (b), respectively). The robot has a predefined goal: throwing balls as close to the jack (target ball) as possible. Typically each player has four balls of a given color to play, the player with the smaller sum of distances to the jack, wins. In these sports, balls have different weights and materials, and the playing surfaces have unique physical characteristics. Our insight is that with a deep reinforcement learning algorithm, the robot will be able to effectively play a precision ball sport, with ball and surface variants.

For this, we propose a strategy divided into three different goals: 1) reaching the balls to throw; 2) being able to throw them close to the jack, with the target within the robots' reach; and 3) to throw the balls to wherever position the jack is. For these goals, different models are created using three different

978-1-7281-6926-2/20/$31.00 ©2020 IEEE

policy gradient (PG) RL algorithms, suitable for continuous spaces. OpenAI Baselines [3] is used, and the algorithms tested to see which has the best performance. All models are trained in a modified simulated robotics environment from OpenAI Gym [4]. The best algorithm is then used to analyze the effect of different ball/court characteristics. Our work presents a new RL environment with a simulated robotic arm for a single-player version of *bocce* or *boccia*. In this, we use PG RL algorithms to teach the robot essential plays of precision ball sports. These types of methods are used for their immense success in the aforementioned robotic sports applications. In resume, our contributions are the following:

1) State-of-art contribution with Precision ball sports environment compatible with OpenAI Gym, providing a more complex and realistic scenario for RL benchmarks.
2) The physical characteristics of the environment's objects (friction and weight) are tested. Their impact on the model's convergence and the success rate is evaluated.
3) Empirical demonstration that a robot is able to learn essential plays of precision ball sports correctly.

The paper is structured as follows: first in Section II, a brief explanation of RL, specifically PG methods tested in our implementation, is delivered. In Section III, the related work with robotic sports and games is presented. In Section IV, the proposed approach is explained. Section V describes the results collected from the simulated experiments, also providing a discussion of those results. Finally, in Section VI, the conclusions, and future work are provided.

II. Background

A. Reinforcement Learning

Reinforcement Learning (RL) is a machine learning (ML) technique that allows an agent to learn optimal actions under different conditions, to maximize a reward signal. The agent performs this by interacting with its environment. When performing the chosen actions, the environment responds by bringing the agent to new states, where it tries to map different states into corresponding optimal actions by a trial and error learning scheme. RL is typically modeled by a Markov Decision Process (MDP), defined by the tuple $< S, A, T, R, \gamma >$. S is the state-space (set of continuous states), A the action space (set of continuous actions), T is the transition function, R the reward function, and γ the discount rate ($\gamma \in [0; 1]$) [5].

At each timestep, the environment's state $s \in S$ is received by the agent, where he chooses the action a to be performed based on $\pi(s_t)$, the applied policy : $a = \pi(s_t) = \mu(s|\theta_\mu) : S \times A$. The policy is a mapping of states to actions, a guideline on what is the optimal action to take in a certain state, and θ_μ are its parameters (for instance the weights in a neural network, or coefficients of a complex polynomial). After executing the action a reward $r = R(s, a) : S \times A \to R$ is received, and the new state of the environment $s' \in S$ is sampled, in agreement with the transition function $T : s' = T(s, a) : S \times A \to S$. The final objective of RL is to discover a policy $\pi(s_t)$ that maximizes the discounted sum of future rewards (expected return) presented in Eq.1, where (s_t, a_t) are the pair state and action, at timestep t.

$$J(\theta_\mu) = \mathbb{E}_{(s_t, a_t)} \sum_t \gamma_t R(s_t, a_t) \tag{1}$$

B. Policy Gradient Methods

Given the objective described above, by finding the policy parameters θ_μ that maximize $J(\theta_\mu)$, the problem is solved. To help find these parameters, ML literature often uses gradient ascent/descent. This is where policy gradient (PG) methods come in. PG methods aim to directly learn the policy and its parameters by using gradients to find the best θ_μ. So, by updating θ_μ in the direction of $\nabla_{\theta_\mu} J(\theta_\mu)$ these methods maximize the expected return $J(\theta_\mu)$ (Eq 1). In robotic applications, the space is typically continuous, and therefore an infinite number of states exists. Given this, value-based approaches become unbearable in computational costs, justifying the extensive use of PG methods instead [6]. PG methods have been applied to a wide range of robotic tasks, and have many benefits for robotics. These benefits can be: incorporating previous domain knowledge, choosing a policy representation significant for a specific job, as well as often needing fewer parameters in the learning process when compared to value-based methods.

Furthermore, there is no need for extensive and in-depth knowledge of the problem or robot kinematics since PG algorithms can be applied without any predefined models (model-free). As in any other RL algorithm, PG methods have two classes of approaches used for optimization: on-policy or off-policy. Off-policy methods, during training, use a behavior policy (i.e., the one used for action selection) different from the optimal policy it tries to estimate (the one being optimized). On the other hand, on-policy chooses actions using a policy derived from the current estimate of the optimal one, optimizing the same policy that is used to make decisions during exploration [7]. This classification will be utilized to define the algorithms used in our tests, described in the following subsections.

C. Proximal Policy Optimization

Proximal Policy Optimization (PPO) is one of the most recent and promising methods in RL, proposed by Schulman *et al.* [8]. It is a simplification and improvement from Trust-Region Policy Optimization (TRPO), being an on-policy method. As other PG methods, it uses stochastic gradient ascent over several epochs to perform each policy update. It is as stable and reliable as trust-region methods (like TRPO) but is much simpler to implement, requiring only a few lines of code. It can be used for environments with either discrete or continuous action spaces, and it is straightforward to implement. It is considered one of the best performers in several complex robotic scenarios, like in humanoid applications. It is known for the objective it optimizes, which can be translated by the equation presented in Eq. 2:

$$L(\theta) = \hat{\mathbb{E}}_t[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t] \quad \text{where} \quad r_t(\theta) = [\frac{\pi_\theta(a_t, s_t)}{\pi_{\theta old}(a_t, s_t)}] \tag{2}$$

$\hat{A}_t$ is the advantage function estimator at timestep t. The clip function clips the probability ratio $r_t(\theta)$ in the interval given by its parallel implementation alternates between sampling data from multiple identical sources and performing several epochs of stochastic gradient ascent on the sampled data, optimizing the objective function.

D. Deep Deterministic Policy Gradient Algorithm

One example of a widely used PG algorithm is Deep Deterministic Policy Gradient (DDPG) [9]. DDPG is an actor-critic off-policy algorithm, being an extension of Deep Q-Networks (DQN) [10] to the continuous action space. It can learn deterministic policies that are configured as deep neural networks. DDPG has two neural networks implemented: a target policy network (actor) $a = \mu(s|\theta_\mu)$, and an action-value approximator network (critic) $Q^\mu(s, a|\theta_Q)$. θ_Q are the parameters of the critic's network. The critic has the task of evaluating the current policy by estimating the expected return of state s when the actor takes action a. The actor maps states and deterministic actions.

First, experiences are originated using a noisy version of the current policy, $a = \mu(s|\theta_\mu) + N$. The algorithm uses *experience replay* to save the previous transitions in the form of a tuple $< s, a, r, s' >$, where s is the current state, a the corresponding action, r the reward, and s' the following state. The tuples are then added to a replay buffer D. After this, the algorithm learns in parallel the actor and critic networks, using samples of transitions from D for training. The critic network minimizes the loss function presented in Eq. 3.

$$L(\theta_{Q^\mu}) = \mathbb{E}_{s,a,r,s' \sim D}[(Q^\mu(s, a|\theta_{Q^\mu}) - y)^2] \quad (3)$$

In Eq. 3 y is the target value, equal to $y = r + \gamma Q^\mu(s', \mu(s')|\theta_{Q^\mu})$. Conversely, the actor network is updated according to the policy gradient $\nabla_{\theta_\mu} J(\theta_\mu)$ (Eq 4) as a means to perform actions that maximize the expected return.

$$\nabla_{\theta_\mu} J(\theta_\mu) = \mathbb{E}_{s \sim D}[\nabla_a Q^\mu(s, a|\theta_{Q^\mu})|_{a=\mu(s)} \nabla_{\theta_\mu} \mu(a|\theta_\mu)] \quad (4)$$

E. Hindsight Experience Replay

Hindsight Experience Replay (HER) [11] is one of the most recently introduced off-policy, PG methods. The main concept is to imitate the human skill of learning from failures. So, even if the goal was not achieved, it uses the knowledge from all the episodes and learns from what he accomplished. Essentially, HER views the state achieved by the agent as a pseudo-goal, facilitating the learning process. With HER, learning policies with off-policy methods and sparse rewards is possible [12].

In this method, *experience replay* is also used. Nevertheless, additionally to storing the transition tuples of an episode where the original goal g was achieved ($< s_g, a, r, s_g >$,), the transitions where a pseudo-goal g' was reached are also stored ($< s_{g'}, a, r, s'_{g'} >$). Without HER, any sampling from the experience replay buffer D that did not accomplish the predefined goal, g, is not rewarded. When using HER, the goal g is substituted by the achieved goal g' in that episode, where g' is sampled randomly from the attained states. Any sampling from D can be then explored for off-policy RL algorithms like DDPG. This algorithm ultimately allows to simplify learning by making the agent first learn from more straightforward goals, and then achieve the predefined one when $g' \rightarrow g$.

III. Related Work

Most of the fundamental tasks in sports and games are easily carried out by humans due to their perspicacity and agility, but can become very challenging when to be carried out by robots. For robots, even the most simple of movements like grasping a ball can be very complex to reproduce. Recently, RL has been utilized to solve these kinds of tasks in the robotics field, showing incredible results in a lot of those applications. For instance, in [6], a PG method to effectively play T-Ball (baseball-like sport), using a robotic arm, is proposed. In their approach, the state is given by the robot's joint velocities and angles, and the actions are the joint accelerations. First, they used supervised learning to teach the robot by imitation, not achieving the goal of hitting the ball. Nevertheless, when applying a Natural Actor-Critic RL method, the performance is improved, accurately hitting the ball after 200 to 300 trials.

Another sport is approached in [13], where a humanoid robot is successful at throwing darts, both in the physical and simulated worlds. Kober *et al.* introduce a new RL algorithm based on reward-weighted regression, but with kernels: Cost-regularized Kernel Regression (CrKR). The algorithm can converge after 1000 throws. Tetherball is also solved with RL. In [14], Daniel *et al.* propose a hierarchical policy search method based on relative entropy. With this method, several solutions were obtained at once, selecting the best primitive's policies that specify the robot's actions. In [15], a simulated 2D basketball task is solved. In this task, a hand needs to pick up the ball and throw it to the hoop. The training is performed using DDPG with HER, and their novel technique: Filtered-HER with Instructional-Based Strategy (IBS). Although the results are promising, the environment is 2D.

In [16], Kober *et al.* present an approach for a simulated robot to perform a Ball-in-a-Cup. Imitation learning is used for the initialization of the motor primitives and a novel RL algorithm - Policy learning by Weighting Exploration with the Returns (PoWER) - for their improvement. The authors utilize a simulated SARCOS arm that successfully performs the game. The PoWER algorithm is also used as a comparative method in [17]. In this work, Kormushev *et al.* propose an approach that allows a 53-DOF humanoid robot (iCub) to learn to shoot an arrow to a target's center. Their method is based on a local regression algorithm with chained vector regression named ARCHER. To effectively converge, ARCHER only needed ten rollouts, showing significant improvements in terms of convergence speed when compared to PoWER.

One of the most popular games solved with RL in robotics is table tennis. In [13] Kober *et al.* use their CrKR algorithm to successfully learn how to hit the ball in the air, outperforming other PG algorithms, in a simulated and real robot. Mulling *et al.* provide other solutions, where a robot can learn basic table tennis, playing against a human in simulated and real environments. They propose an algorithm named Mixture of Motor Primitives (MoMP), allowing them to obtain a task

policy composed of weighted movement primitives [18]. Also, in [19], Peters *et al.* introduce a new RL algorithm named Relative Entropy Policy Search (REPS). In this algorithm, the highest expected reward is obtained while limiting information loss. REPS is validated in a simulation environment of a robotic table tennis game, where it is proven effective in selecting the fitted motor primitives.

The sport with more developments and interest in robotics is soccer. RoboCup is considered the greatest robotic competition worldwide [20]. With competitions as large as RoboCup, it is even more challenging to use RL methods to teach robots how to play. For instance, in [21] RL with Decision Trees is used to make a NAO humanoid robot learn how to score goals in penalty kicks (in simulation and real-world). This method allowed to limit the number of trials needed for learning, being the first example of RL successfully applied in a NAO robot. In [22], PPO is effectively used to solve natural running and dribbling, also by a simulated NAO.

IV. Implementation

To get a robot to learn how to play a precision ball sport it is essential to have a well-defined environment and robotic agent. For the learning process, a new RL environment was created from the adaptation of OpenAI Gym robotics environments. OpenAI Gym [4] is a toolkit developed in 2016 for creating, evaluating, and comparing RL algorithms. The choice of using OpenAI Gym derives from its establishment as a provider of RL benchmarks. Where for supervised learning methods, the benchmarks are large datasets (like CIFAR-10 or ImageNet), for RL, the corresponding would be an immense variety and vast collection of environments. Given this, by adapting and creating a new environment for *boccia* and *bocce*, we are contributing to those benchmarks. Besides this, OpenAI Gym allows for a widely used environment standardization and formalization, simplifying their use, set-up, and testing.

The Fetch environments were the ones adapted for our solution. These environments utilize the physics engine MuJoCo [23] for a realistic and swift simulation. Their leading agent is a 7-DoF Fetch robotic arm[1] with a two-fingered gripper, and allows four different types of tasks. The first task is reaching (*FetchReach*): where the goal is to achieve a particular position. This task is the easiest. The second one is pushing (*FetchPush*), where the mission is to move a block to a target location by pushing it, not allowing grasping. Then we have sliding (*FetchSlide*), where the goal is to get a puck, located in a table, to slide into a target position on that same table. Finally, we have the pick-and-place task (*FetchPickAndPlace*), where the objective is to grasp a block, then move it to a goal location [24].

So, for our implementation, a general environment called *PrecisionBall* was created by modifying the *FetchSlide* OpenAI example. The *PrecisionBall* environment (Fig. 2) contains red balls for the robot to learn to throw, a white ball - jack or palino - as the target, blue balls to simulate the opponent,

[1]https://fetchrobotics.com/robotics-platforms/

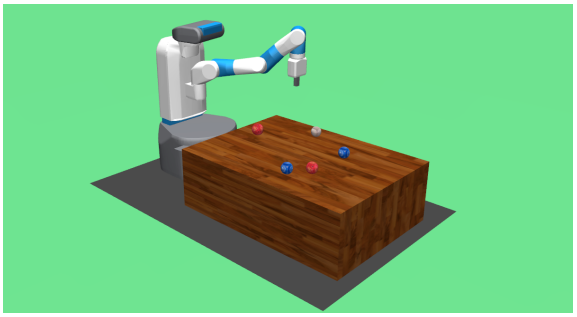


Fig. 2. *PrecisionBall* Simulation Environment

and a flat table as the court. Depending on the precision ball sport being played, balls can have slightly different shapes and weights. The red/blue and white balls can be of two different types. In the game of *bocce*, hard plastic composite with 920 ± 10 grams and 0.11 meters of diameter are used for the red/blue balls. For the palino, no weight is defined, rules only state a 0.05 meters diameter. On the other hand, in *boccia*, both the red/blue and white balls are made of leather with 275 ± 12 grams and 0.135 meters of diameter.

The court can also be made of different materials. For instance, *boccia* fields can be polished concrete, wood, or natural/synthetic rubber, while for *bocce*, compacted clay-like materials are used (sand, or topping clay). This implicates different levels of ease in rolling over the surface, entailing distinct levels of friction. These court characteristics were translated into our environment by different friction coefficients and different material types. All these properties, both for the court and balls, were edited to the defined standards using the MuJoCo physics engines. For the general environment, the following features were set as default: red/blue balls with 920 grams and 0.11 meters of diameter; jack with 0.11 meters and 275 grams; wood field with a friction coefficient of 0.1.

OpenAI's Fetch tasks were also chosen for their goal-oriented environments. In our case, the objective mission is to place the player's (robot) red balls as close to the white one as possible. So, our desired goal is a distance of approximately zero from the red balls to the white one. This desired zero distance would lead to very sparse rewards if fixed, given this, a fixed tolerance of 0.05 meters was defined. With this type of environment, a goal-aware observation space is needed, i.e., a space not only with current environment observations but also with the desired goal, as well as the achieved goal (the distance actually produced by the red ball). In the ideal case, the achieved goal would converge fast to the desired goal.

Therefore our observation space is formed by a dictionary composed by three different keys: $achieved\_goal$, $desired\_goal$ and $observations$. The $achieved\_goal$ is given by the euclidean distance between the current gripper position and the goal position (the white ball). The $desired\_goal$ is an optimal euclidean distance of 0 ± 0.05 meters, as mentioned above. Finally, the $observations$ key is composed of thirteen variables: the absolute position, positional (Cartesian) velocity in the world frame and the rotational velocity of the gripper, the absolute and relative red and blue ball position, rotation in

Euler angles, positional and rotational velocities. The relative positions are in relation to the current position of the gripper, and all retrieved in x, y, z form from the MuJoCo simulator.

With the observation space defined, another essential aspect to specify is the action space. Since our goal is only to hit the red ball, the gripper movement is not considered for our application, and its position is fixed. The action space is composed of three dimensions, where each specifies the x, y, and z in the next timestep of the gripper's desired relative position. To move the gripper to the desired location, MuJoCo constraints are used [11]. Actions are applied for 0.05 seconds (equivalent to 25 succeeding simulator steps). The robot is only able to perform one action to each red ball.

Another aspect of most importance is reward modeling. In this environment, rewards are sparse and binary and compensate the agent for attaining the desired distance between the initial and target positions. In our approach, the reward is dependent on d, where d is the distance between the red ball and jack. If $|d|$ is smaller than the defined threshold of 0.05 meters, then the reward is zero. If not, the reward is -1. The module is applied so the agent gets rewarded when d is between -0.05 and 0.05. A penalty of -50 is also given in the reward if the robot applies more than one action to the same red ball. Given this, the agent will try to maximize its reward. With these aspects specified, the initial and termination conditions of the simulation were defined. At the beginning of each episode, the following conditions are assured:

- The location of jack and opponent's ball is generated randomly on the sports field. This random generation was used to simulate the initial white ball throwing, as well as replicating the effect of other player's moves that can lead to different goal positions.
- Red balls initial positions are random but in the robot's reach, so the robot can learn to reach and throw it.
- The robot is always initiated in the same position, with the same joint angles at each episode.

This randomization changes all ball's positions, every episode, so the robotic agent will be able to generalize the learned abilities. The Fetch robot will learn where to hit the red ball correctly, and the amount of force that it needs to apply to win the game. The strategy learned to apply a particular force to a ball is affected by the physical parameters of the environment mentioned before, like ball weight and field friction. At each episode, the termination conditions are evaluated to test if the simulator needs to reset the agent and initialize a new episode. The conditions are the following:

- Any ball outside bounds: all ball's x and y positions are tested against court's limits to assure they are still in play.
- Timeout: the predefined maximum duration of an episode. By testing, four seconds were defined (2000 timesteps).
- Goal achieved: distance from a red ball to the jack is 0 ± 0.05 meters for more than 50 timesteps.

V. Experiments and Results

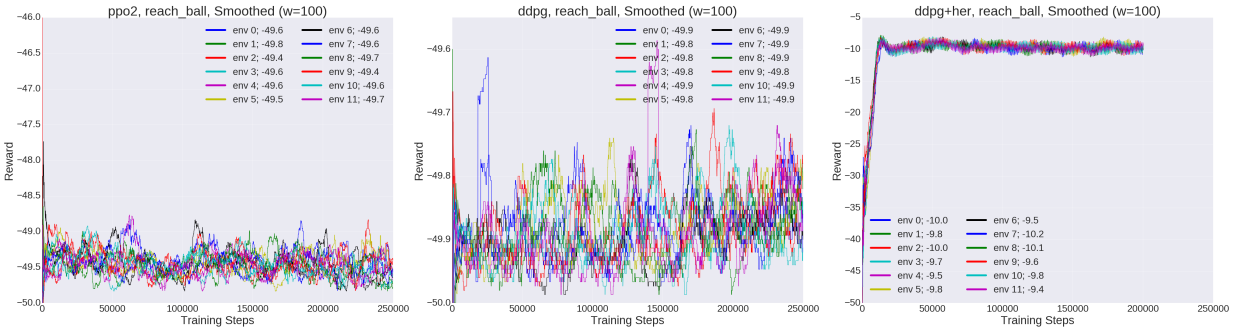
As previously introduced, the game was divided in three stages that entail different goals. In Stage 1, the goal is only to reach the ball with the color allocated to the robot (red). Stage 2 has the objective of throwing the red ball as close as possible to a white target ball (jack) that is within the robot's reach. And finally, Stage 3 is where the final version of the game is played, having as goal to throw a red ball close to the jack to obtain a near-zero distance between the two. The environment created (*PrecisionBall* presented in Fig 2) was used with its default settings defined in Section IV.

Since our environment was created using OpenAI Gym, the OpenAI Baselines tool [3] was utilized to simplify testing with some of the most used model-free PG RL algorithms (referenced in Section II): PPO, DDPG, and DDPG+HER. The algorithms' hyperparameters used were the ones provided by OpenAI as the best for training a Fetch 7-DoF robotic arm [11], [25], [8]. To better evaluate the suitability of each algorithm at each stage, all methods were tested in the three phases. For all stages, the training process was performed on a single machine with 12 CPU cores, for 80 epochs, equivalent to 250k training steps. Each core uses two parallel rollouts (train and test) and MPI for synchronization [24]. The use of MPI not only speeds up the training process but also provides better results by increasing the original defined batch size of 256 (it increases linearly with the number of cores: $batch\_size = n\_cores \times 256$), providing more experience.

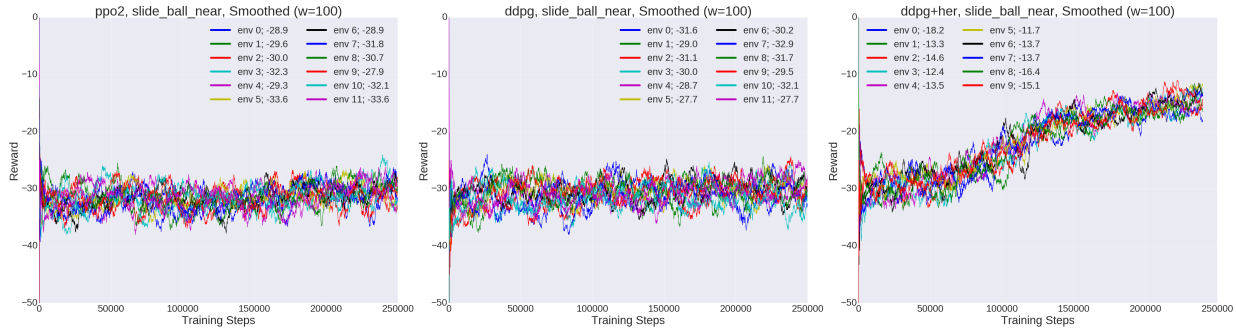
Three different random seeds were used for training every algorithm, and their mean reward computed. The evolution of these mean rewards in each MPI worker, over the training steps, for each algorithm and stage, are presented in Fig. 3. The colored lines shown represent MPI workers from a total of 12 (number of CPU cores). As one can see, only DDPG+HER was able to converge in all three stages successfully. PPO is not capable of learning at any stage. This is possibly due to this method not relying on a replay buffer, and since the goal ball position is randomly generated every episode, it rarely achieves the goal state. Also, with our environment having binary and sparse rewards it hampers the learning process, being stuck at nearly -50 for stages 1 and 3 (left Fig. 3(a) and Fig. 3(c), respectively), and -30 for stage 2 (Fig. 3(b) left).

On the other hand, DDPG starts to increase reward after 250k eps for the first stage, but does not converge. In stages 2 and 3, it performs poorly, getting stuck at minimum rewards (-30 and -50), in the same manner of PPO. DDPG, as every algorithm in RL, is not very prone to learn from sparse rewards in complex environments, so without HER, it does not achieve convergence [25]. Finally, with the DDPG+HER algorithm, convergence is attained in all stages of the game, producing mean stable rewards of -10 for stage 1, -14 for stage 2, and -44 for the final stage. With this algorithm it is very clear that reaching the red ball is by far the most straightforward task to learn. When the goal is to achieve a zero distance from the red ball to the jack (second and third stages), the reward signal is not as stable or as high. After this evaluation was done, DDPG+HER was the algorithm used for our final model.

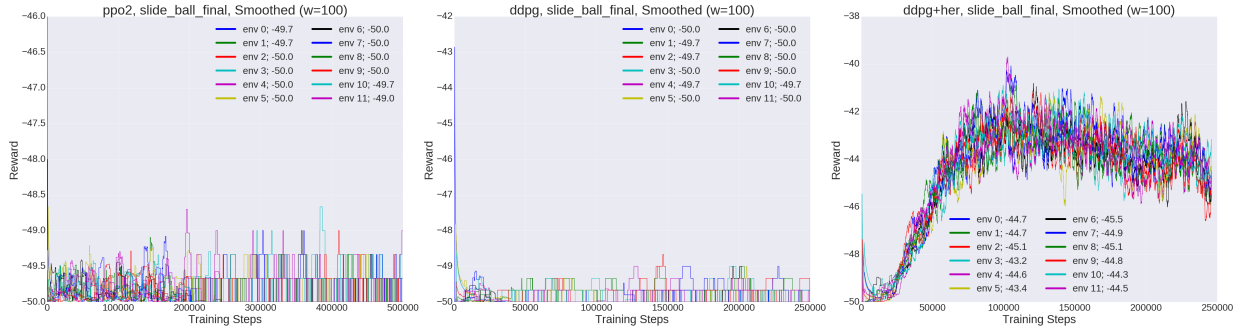
With the final model chosen, the train and test success rates were calculated for all stages, only using DDPG+HER. In Figures 4 to 6, the training success rates for stages 1, 2, and



((a)) Stage 1 (reach red ball)



((b)) Stage 2 (throw red ball with jack within reach)



((c)) Stage 3 (throw red ball with jack anywhere)

Fig. 3. Reward signal Smoothed (w=100) per training step for PPO (left), DDPG (center) and DDPG+HER(right)

3 are shown. In Figures 7 to 9, the test success rates are shown for the same stages. The train results display worse performance since, in the training algorithm, random actions are selected with a 30% probability, and noise is added to those actions, aiding the agent to explore, but preventing it from reaching better success rates. On the other hand, in the test rollout worker, both the probability and noise are set to zero, allowing the agent to achieve much higher success rates since random actions do not impair it.

For stage 1 (Fig. 4 and 5 left), it is once again clear that the task is rather simple since a mean train and test success rates of 0.7 and 1, respectively, are constant and obtained right from the beginning. For stage 2 (Fig. 4 and 5 center), the success rate evolves along with the episodes, achieving 0.8 for training and 1 for testing. Finally, for stage 3, the rates once again prove the complexity of this final task. Values no higher than 0.2 are reached for the train success rate (Fig. 4 right) and 0.84 for the test (Fig. 5 right). This final stage is also more unstable, showing variances of 0.2 max and 0.02 min for the test success rates, even after converging. All the above-mentioned results were provided with the *PrecisonBall* environment with default physical properties explained in Section IV.

TABLE I
EFFECTS OF DIFFERENT PHYSICAL ENVIRONMENT CHARACTERISTICS ON THE RL ROBOTIC AGENT

Ball weight (grams)	Friction Coefficient	Max Success Rate	Mean Success Rate	Convergence (n^o of eps)
275	0.1	89.2%	69.7%	833
275	0.4	92.7%	75.7%	1100
275	0.7	92.5%	72.4%	1300
275	1	66.0%	53.4%	1460
920	0.1	84.2%	69.4%	1200
920	0.4	78.3%	66.6%	1553
920	0.7	70.8%	54.3%	1670
920	1	64.2%	52.6%	1580

As previously referred, *boccia* and *bocce* have various

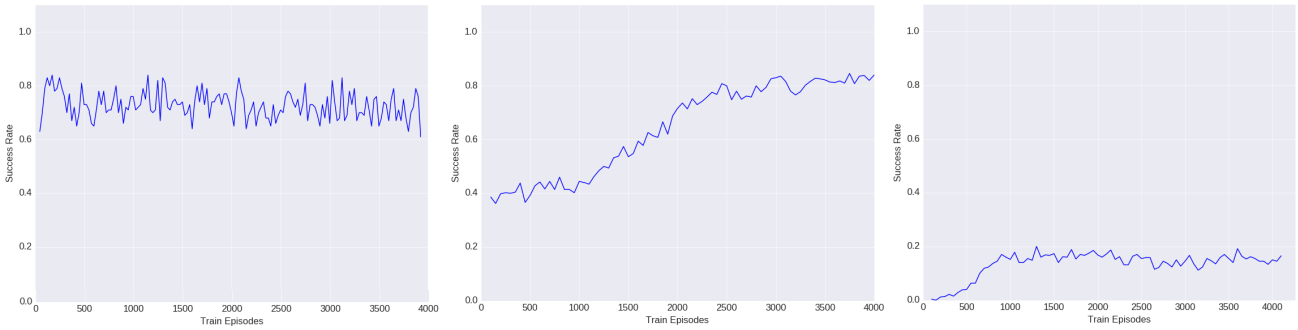


Fig. 4. Train Success Rate for Stage 1 (left), Stage 2 (center) and Final Stage (right)

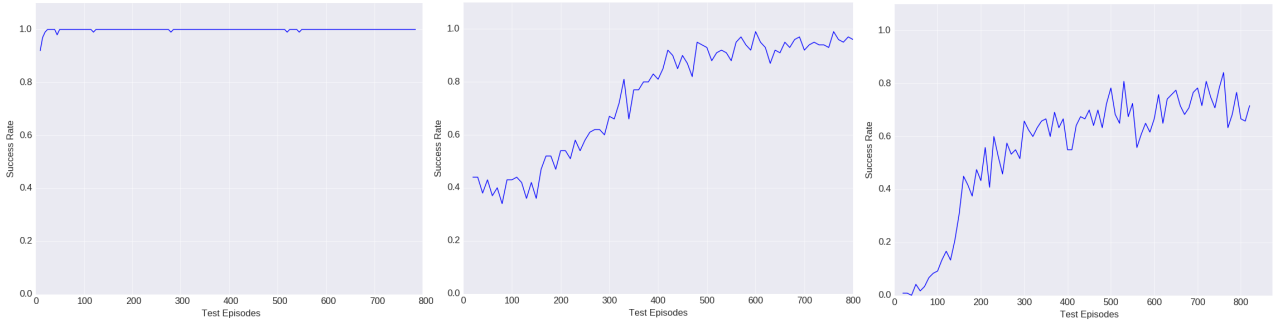


Fig. 5. Test Success Rate for Stage 1 (left), Stage 2 (center) and Final Stage (right)

ball and court weights and materials, translated into different weights and frictions. Given this, the effects on the DDPG+HER model of different possible ball weights and tangential friction coefficients were evaluated. All the possible combinations between two ball weights (275g as in *boccia* and 920g as in *bocce*) and four different friction field coefficients (0.1, 0.4, 0.7, and 1) were tested. The results are presented in Table I, with the maximum and mean success rates, and the number of episodes needed for converging, for each combination. The mean and max success rates were calculated only for timesteps after the model's convergence. The last coefficient of 1 was used simply to test the effects of high friction on the agent since no precision ball court is made of a material with stated resistance higher than 0.7.

As intuition would tell us, the heavier the red ball, the harder for the robot to learn in the same amount of episodes, since it needs to apply more force in order to throw it to the jack. For the coefficients of 0.1, 0.4, and 0.7, the environment with the 920g ball took more time to converge than for the lighter ball, needing a mean of 400 more episodes to converge. For the friction coefficient of 1, the number for convergence is only 120 episodes bigger. This is explained by the magnitude of the coefficient that holds back the movement of the sliding ball, no matter its weight. Also, the success rates for the lighter ball are, in general, better than for the heavier one, except for the largest coefficient, where the difference is almost insignificant (less than 2%). For heavier balls and higher frictions, the mean success rate is smaller due to a higher instability, besides not achieving as high max rate values.

When comparing the max and mean success rates, their values decrease with the increase of the friction coefficients due to difficulty in moving heavier balls, as expected (max success decreases from 88% to 64.2%, and mean from 69.2% to 52.6%). Interestingly, for the 250g balls, mean and max rates increase slightly with friction's rise. For 0.1, 0.4, and 0.7 coefficients, max success rates go from 90% to almost 93%, and the mean from 70% to nearly 75%. This is possibly due to higher friction coefficients slowing down the lighter ball's movement, and therefore allowing it to stop closer to the jack. Otherwise, any force the robot applied to the 250g ball would lead it to surpass the jack, due to small field friction. With this analysis, it is clear that our robotic agent achieves good results overall, but is a better player of *boccia* than of *bocce* (mean and max success rates higher around 75% and 90% for *boccia* and around 65% and 80% for *bocce*).

The behaviour of the already trained agent was analyzed for the final stage with a modified reward. R=0 if the robot achieves a distance d not only smaller than the threshold, but also equal to or smaller than d from the opponent's balls. R=-1 otherwise. In this analysis, relevant behaviours were shown. Not only it was able to reach the jack most of the times, as it choose to reach it from certain sides, according to the observation of the opponent's balls position. For instance, with the jack with both blue balls by its side, the agent choose to throw the red ball in between the two blue ones, so it could achieve a smaller d when compared to the one reached by the

opponent. Besides this, the robot also learned to hit the jack in order to increase its distance to the opponent, but due to limitations in the movement dynamics (sometimes when hit it did not cause substantial movement), it was not a converged behaviour from our agent.

VI. Conclusions

In this work, a deep reinforcement learning algorithm is used to solve a robotic task in a simulated environment of precision ball sports like *boccia* or *bocce*. A new setting for this task was created, adapting an existent environment from OpenAI Gym. The approach was divided into three stages: reaching the red ball; throw it to the jack when within the robots reach; and finally throw it to hit the jack anywhere.

The performance of three different RL algorithms (PPO, DDPG and DDPG+HER) over 250k timesteps was compared for each stage, where DDPG+HER revealed the best results. The success rate for DDPG+HER in its training and testing phases was then evaluated, also for each stage. The first stage achieved a success rate of 100% in less than 20 episodes, while the second stage took 500 episodes to reach the same rate. The final stage only achieved a max success rate of 84.2% in 600 test episodes. However, the approach is not very stable in this last stage where the sport is played, which is not directly disabling for the robot player but can result in more failures and defeats than other opponents.

The effects on the approach's convergence and success rate for different physical characteristics inherent to *bocce* and *boccia*, were evaluated for the final stage. For this evaluation, the materials, ball weights, and tangential friction coefficient were altered for each kind of field/ball. The smaller the friction and weight, the easier it was for the agent to learn, converging faster and having higher success rates. Maximum success rates of 92.7% were achieved for *boccia*, and 84.2% for *bocce*. Our approach shows promising results, where the robot was able to learn relevant skills and develop strategies that allow it to win. We feel that our empirical demonstration illustrates the possibility for one more robotic competition, or even for a simulated training partner.

As future work, we intend to refine this work by allowing increasing the realism of the presented simulator. First, the movement dynamics from the balls and jack will be improved. Then, to simulate real measures from vision systems or lasers in the robot, an error will be added to the distance values retrieved from the simulator. Also, the final result of the four plays can be set as the goal. This approach could bring impressive results, like allow the robotic agent to learn and converge to more complex strategies, such as how to win by damaging the plays of the adversary. Other improvements encompass testing RL algorithms' performance with dense rewards, a more thorough search for algorithms parameters, and more training episodes.

References

[1] 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.

[2] J. M. Calderon, E. R. Rojas, S. Rodriguez, H. R. Baez, and J. A. Lopez, "A robot soccer team as a strategy to develop educational iniciatives," in *Latin American and Caribbean Conference for Engineering and Technology, Panama City, Panama*, 2012.

[3] P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, Y. Wu, and P. Zhokhov, "Openai baselines," *GitHub, GitHub repository*, 2017.

[4] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *arXiv preprint arXiv:1606.01540*, 2016.

[5] O. Kilinc, Y. Hu, and G. Montana, "Reinforcement learning for robotic manipulation using simulated locomotion demonstrations," *arXiv preprint arXiv:1910.07294*, 2019.

[6] J. Peters and S. Schaal, "Policy gradient methods for robotics," in *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2006, pp. 2219–2225.

[7] D. Precup, R. S. Sutton, and S. Dasgupta, "Off-policy temporal-difference learning with function approximation," in *ICML*, 2001.

[8] J. Schulman, F. Wolski, P. Dhariwal, and A. Radford, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[9] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.

[10] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, p. 529, 2015.

[11] M. Andrychowicz, B. Baker, M. Chociej, R. Jozefowicz, B. McGrew, J. Pachocki, A. Petron, M. Plappert, G. Powell *et al.*, "Learning dexterous in-hand manipulation," *arXiv preprint arXiv:1808.00177*, 2018.

[12] T. Chen, A. Murali, and A. Gupta, "Hardware conditioned policies for multi-robot transfer learning," in *Advances in Neural Information Processing Systems*, 2018, pp. 9333–9344.

[13] J. Kober, E. Oztop, and J. Peters, "Reinforcement learning to adjust robot movements to new situations," in *Twenty-Second International Joint Conference on Artificial Intelligence*, 2011.

[14] C. Daniel, G. Neumann, and J. Peters, "Learning concurrent motor skills in versatile solution spaces," in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2012, pp. 3591–3597.

[15] B. Manela and A. Biess, "Bias-reduced hindsight experience replay with virtual goal prioritization," *arXiv preprint arXiv:1905.05498*, 2019.

[16] J. Kober, B. Mohler, and J. Peters, "Learning perceptual coupling for motor primitives," in *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2008, pp. 834–839.

[17] P. Kormushev, S. Calinon, R. Saegusa, and G. Metta, "Learning the skill of archery by a humanoid robot icub," in *2010 10th IEEE-RAS International Conference on Humanoid Robots*. IEEE, pp. 417–423.

[18] K. Mülling, J. Kober, O. Kroemer, and J. Peters, "Learning to select and generalize striking movements in robot table tennis," *The International Journal of Robotics Research*, vol. 32, no. 3, pp. 263–279, 2013.

[19] J. Peters, K. Mulling, and Y. Altun, "Relative entropy policy search," in *Twenty-Fourth AAAI Conference on Artificial Intelligence*, 2010.

[20] S. Behnke, M. Schreiber, J. Stuckler, R. Renner, and H. Strasdat, "See, walk, and kick: Humanoid robots start to play soccer," in *2006 6th IEEE-RAS International Conference on Humanoid Robots*, 2006, pp. 497–503.

[21] T. Hester, M. Quinlan, and P. Stone, "Generalized model learning for reinforcement learning on a humanoid robot," in *2010 IEEE International Conference on Robotics and Automation*. IEEE, 2010, pp. 2369–2374.

[22] M. Abreu, N. Lau, A. Sousa, and L. P. Reis, "Learning low level skills from scratch for humanoid robot soccer using deep reinforcement learning," in *2019 IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC)*. IEEE, 2019, pp. 1–8.

[23] E. Todorov, T. Erez, and Y. Tassa, "Mujoco: A physics engine for model-based control," in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2012, pp. 5026–5033.

[24] M. Plappert, M. Andrychowicz, A. Ray, B. McGrew, B. Baker, G. Powell, J. Schneider, J. Tobin, M. Chociej, P. Welinder *et al.*, "Multi-goal reinforcement learning: Challenging robotics environments and request for research," *arXiv preprint arXiv:1802.09464*, 2018.

[25] M. Andrychowicz, F. Wolski, A. Ray, J. Schneider, R. Fong, P. Welinder, B. McGrew, J. Tobin, O. P. Abbeel, and W. Zaremba, "Hindsight experience replay," in *Advances in Neural Information Processing Systems*, 2017, pp. 5048–5058.