

Training Futsal AIs in Unity with Reinforcement Learning

Pau Comas Herrera
University of Twente, The Netherlands

Abstract- Reinforcement learning is a modern unsupervised learning approach that allows to create intelligent players for game theory scenarios. With the recently created Unity ML-Agents Toolkit, developers can create games in the Unity game IDE that support the application of state-of-the-art machine learning algorithms for AI player trainings. In this project, we create different Futsal based game environments, design and train RL intelligent models and assess different RL techniques. This work contributes to the Unity and RL community with a methodology to create games that support the toolkit, research and results of different test configurations for agent's trainings and an open repository of the work that can be freely accessed and used.

1. INTRODUCTION

Artificial intelligence (AI) in games has evolved impressively since its first appearances in the early 1950s. From hardcoding to deep neural networks, the techniques and results have been in constant growth since the early appearance of AIs in Nim [1] or Pong [2]. DeepBlue managed to beat world chess champion Kasparov in 1997 with an algorithm based in alpha-beta minimax search, AphaGo defeated Go champion Lee Sedol in 2016 with a complex Monte-Carlo tree search [3] and AlphaStar [4] is winning against StarCraft champions with a combination of supervised [5] and unsupervised learning [6] techniques in deep neural networks. The affordable complexity by the algorithms and computers has raised until the point of being competitive in Real-Time Strategy (RTS) games like StarCraft, where we also don't alternate turns and the states and possible future states are considerable and higher than in chess or Go.

This evolution has also led to making AI creation more open to everyone. Even though a lot of work has been done, further research in ML algorithms behavior in a lot of games is needed and the possibilities are uncountable. In this paper, we challenge ourselves to create a Futsal [7] AI based in a deep neural network (NN) approach with reinforcement learning. Futsal is a two-team, 5 vs 5 game played in a 40 x 20 m field where teams compete to score in the opponents' goal. We create an environment with AI players receiving similar information for decision-making as in real-life, allowing us to analyse if our deep machine learning (ML) policies will behave similar to human players. In addition to evaluating the gameplay results of our AI, we also aim to analyze the challenges and most important issues presented in the process, bringing us some interesting insights for reinforcement learning projects. With Unity ML-Agents we can use and evaluate some reinforcement learning (RL) training techniques, such as curriculum learning, self-play or environment parameter randomization. Remark that in this project we want our models to be trained fully with reinforcement learning, without going into imitation learning or supervised learning options that could support the process.



Figure 1. Photo of a futsal game in Albania

This work has three main contributions:

- We present a game creation methodology for the Unity environment, focusing in implementing a simulation environment for reinforcement learning context creation, configuration for the episodic trainings and model assessment.
- We build an open Github repository [38] with four different Futsal game environments, all of them ampliable and testable for the general public. Further work and research can be done in unsupervised learning trainings or games in Unity. The environments are also open to be implemented as a game apart from a simulation.
- We present conclusions and insights regarding our research/work with neural network configuration, RL algorithms, RL additional training techniques, raycast observations and etc.

The rest of this paper is structured as follows: in section 1 we introduce the lector to the main elements of our project and techniques involved in its course, in section 2 we examine the related work done by the community until now, in section 3 we describe our methodology in creating and testing game environments, in section 4 we display and argue the results obtained in our progressive approach ending with the Futsal 5VS5 game and finally we end the report with the conclusions of our work.

A. Reinforcement learning

Reinforcement learning (RL) [8] is a machine learning (ML) area where, following unsupervised learning, we aim to train intelligent agents to make the best decisions that maximize rewards designed in a certain environment. Unlike the traditional ML models where the aim is to fit a model to a dataset to solve a problem, in RL the models/agents learn by trial and error receiving “rewards” as feedback. These “rewards” will determine the policy and the frequency of every action made in the future. Simply explained, the state (observations) of the environment is given to a RL algorithm, this algorithm outputs the best action at the moment as an output, and after applying the action the algorithm is fed with a reward. A vicious cycle will be repeated where the model receives observations and rewards to output the dynamic actions. At the end, if we achieve convergence, the outputted actions will be the ones that maximize the overall cumulative rewards.

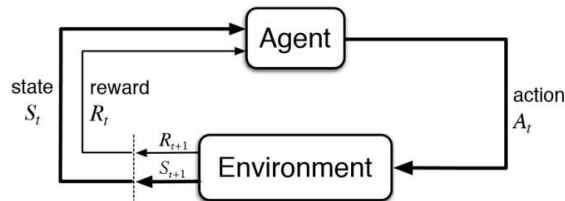


Figure 2. RL functioning scheme

As we discussed and taking as an example Figure 2, the goal is to find a suitable action model that would maximize the total cumulative reward of the agent. The rewards are ideated by the engineers, seeking to achieve concrete objectives inside the decision process. We will see that the rewards’ modelling is not trivial and plays an important part to get good performances. Famous and known deep RL algorithms are TD() algorithms [9], Q-learning [10] and PPO [11]. There are several differences between them, but they all are part of the RL theory. In this work we’ll work with the mentioned PPO and MA-POCA [12], a recent algorithm made by Unity.

B. Proximal Policy Optimization

Proximal Policy Optimization (PPO) is an on-policy RL algorithm made by OpenAI and released in 2017. It was created to become the state-of-the-art RL algorithm and has become very popular for its performance, simplicity and polyvalence. It is based in the formula of traditional policy gradient methods [13] and their successor Trust Region Policy Optimization (TRPO) [14] with some relevant modifications.

The complete explanation about how it works is properly done in [15], analyzing how we evolved from policy gradient to TRPO and then PPO. Nonetheless, here is a summary of the main ideas.

The full objective function that is aimed to be maximized in every learning iteration is displayed in Figure 3. We can distinguish three terms and two free coefficients, c_1 and c_2 . The first term is called the clipped surrogate objective (Fig. 4) and is the main term that directs learning. It performs a minimum in the expectation of two terms, with the first as the TRPO optimization term, with $r_t(\theta)$ being the probability ratio and \hat{A}_t the advantage. The probability ratio is higher in case the action chosen in this state is more probable than in a saved past policy version, whilst the advantage is higher if the action is better than we estimated. The second term of the minimum is a “clipped” (more conservative) version of the first, and it dominates in two cases: 1. When the action was already more probable and is better than expected and 2. When the action was already less probable than before and is worse than expected. This gives learning stable and conservative modifications without making it substantially slower.

$$L_t^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}_t [L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t)],$$

Figure 3. Full objective function of PPO

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t [\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon)\hat{A}_t)]$$

Figure 4. Clipped main objective of PPO

Going back to Figure 3, the second term is a squared-error loss of value. If the actual value is higher than what we expected, it will be positive. Finally, the third term is an entropy bonus that will ensure exploration of states and actions in the first part of the training. At the beginning the entropy of the policy will be high but will decrease over time so the other terms will progressively dominate. PPO is a pragmatic and relatively simple algorithm that is backed up with its heuristic results against other algorithms [15].

C. MA-POCA

Multi-Agent Posthumous Credit Assignment (MA-POCA) is a RL algorithm recently created by Unity [12]. It has been designed specifically for multi-agent adversarial games with dynamic number of players. It is based in the Counterfactual Multi-Agent (COMA) algorithm [16], with main differences as the handling of “dying” players observations and impact in the learning process. Both follow a centralized training, decentralized execution philosophy with a “critic” neural network that performs the training, and outputs the actions receiving all kinds of observations from the “actors” neural networks that execute them. In Figure 5 we can see the COMA visual scheme.

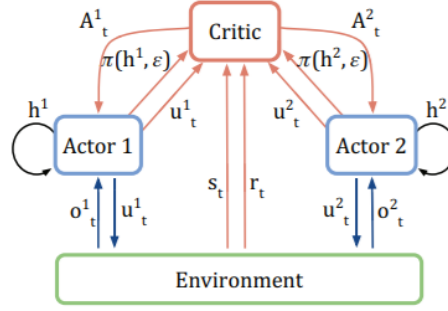


Figure 5. Basic scheme of the actor-critic architecture

MA-POCA focuses on polishing training for games with changing number of active players. MA-POCA uses self-attention [17] over active agents in the critic network, thereby addressing the issue of posthumous credit assignment without the need for absorbing states [12]. Additionally, self-attention enables a network architecture that can efficiently compute counterfactual baselines for groups of homogeneous and heterogeneous agents. According to Unity developers, the fact that COMA uses absorbing states for dying players limits the capability of function approximation and the efficiency of the model. With attention we can avoid this while maintaining the full-observability of the actor-critic architecture. MA-POCA has been released not long ago and the results available have been conducted by the Unity team.

D. Unity and ML-Agents

We will create our game environment in one of the most popular game engines and Integrated Development Environment (IDE), Unity. We have complex physics, 3D rendering and collision detection. With Unity, we're able to create and test 2D and 3D games in a pragmatic and embedded fashion. That means that we can design and assess almost equally faster, even if it is in forms of coding or graphic interaction. In Figure 6, we can see some of the main features of Unity: in the *Hierarchy* tab we can create and edit the various components of the game; the *Scene* tab allows us to control and see the graphic visualization of the game; the *Game scene* is the real behaviour of the game and can be played/paused; the *Inspector* lets us change and connect the properties and attributes of the components, also allowing to connect code documents to them.

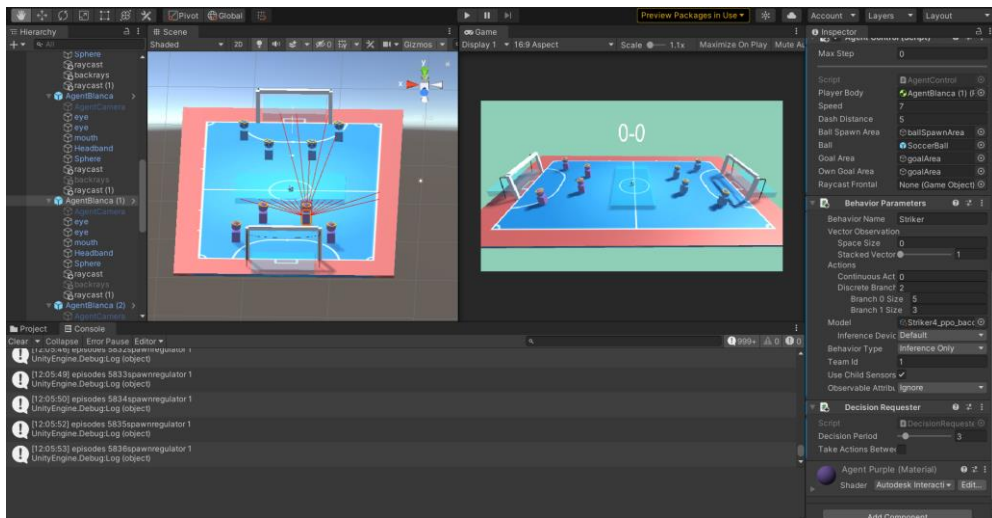


Figure 6. Unity environment framework interface

These are the main elements of Unity for the classical approach of game creation, but is not enough for the aim of this project. To create and test RL models in our game, Unity has the exact

needed toolkit: ML-Agents [18]. The Unity Machine Learning Agents Toolkit (ML-Agents) is an open-source project that enables Unity games and simulations to serve as environments for training intelligent agents. It provides implementations of state-of-the-art algorithms - such as PPO or SAC [19] - to make things easier for developers. It was created both for AI researchers and videogame developers with the implementations based on PyTorch and an open Python low-level API, if the user desires to create or edit his own RL implementations to apply in-game. In Figure 7 we can see a simple overview of the processes in ML-Agents. In Unity we create the environment and “agents” in our games. The Python API works as a link between the data sent by the Unity Communicator and the ML implementations in the Python Trainer.



Figure 7. Graph representing Unity ML-Agents working flow

The key in the Unity process are the “Agents” [20]. In our case, an agent would be any of the soccer players. An agent is an entity that can observe its environment, decide on the best course of action using those observations, and execute those actions within its environment. Agents can be created in Unity by extending the Agent class in coding in C# files to control components. The most important aspects of creating agents that can successfully learn are the observations the agent collects and the rewards assigned to estimate the value of the agent's current state toward accomplishing its tasks. The Agent class has all the necessary functions to implementate this in a simple way [21].

An Agent passes observations to its Policy model (specified in the Behavior Parameters in the Inspector). The Policy then makes a decision and passes the chosen action back to the agent. The agent code receives and executes the action, for example, moving the agent in one direction or another. In order to train an agent using RL, the agent calculates a reward value at each action. The reward is used to discover the optimal decision-making policy and is designed entirely by us. The class that controls training for all agents in Unity is called *Academy* [22] and is a global accessible Singleton class. It is in charge of communicating with the external Python process, has a set of global parameters accessible for the agents and ensures that all live agents are in sync.

E. Training in ML-Agents

ML-Agents provides implementations of some important RL algorithms. Our focus will be in two specific options, PPO and MA-POCA. It has been discussed about how ML-Agents allows us to link the Unity environment and the ML implementations with a Python API and the “Agents” class in C#, but how training can be executed and monitored is yet to be pointed out.

mlagents-learn is the main training utility provided by the ML-Agents Toolkit. Opening a command window in a Anaconda environment the trainings can be directed and played in Unity, out of the Unity app and with or without graphical visualization. It accepts a number of CLI options in addition to a Yet Another Markup Language (YAML) [23] configuration file that contains all the options and hyperparameters to be used during training. The set of configurations and hyperparameters to include in this file depend on the agents in our environment and the specific training method we wish to utilize. The hyperparameter values change between RL algorithms and have a impact on the training performance. In Methodology is included how is operated with them and all the possible options.

This process creates Open Neural Network eXchange (ONNX) [24] files that will represent our deep RL models that afterwards will be appended to agents as a behaviour. It will also enable analysis through TensorBoard dashboards [25].

F. Self-play

In multi-agent competitive environments, especially in symmetric team-based games, it is the engineers responsibility to choose how models are going to be trained and who are they going to face in the process. An opponent team could be hardcoded, the model could face itself, etc. In every moment, the best opponent is a model with a similar level of play. Self-play [26] is a RL learning technique where this exact idea is applied, saving past policies of a deep model to train it (Fig. 8).

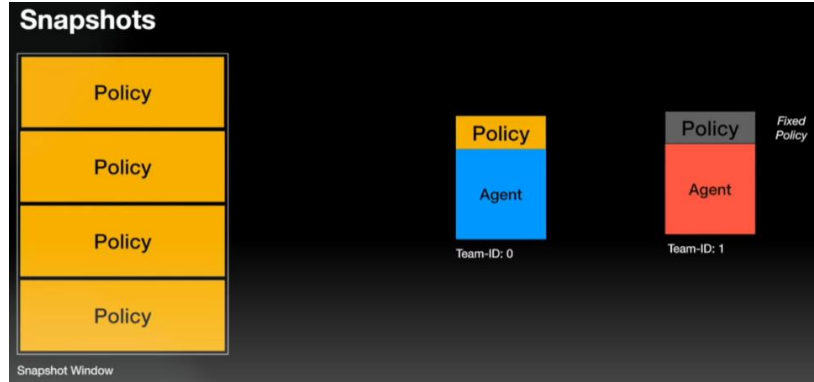


Figure 8. Self-play elements in Unity

With self-play, there is a training team and a test team in a environment, and they swap sides after a given number of steps. This and other parameters can be widely configured. The training team has the learning policy (*fixed policy*) and is actively training it, whilst the test team will behave according to a randomly selected past policy (*snapshot*) of a policy stack. In theory, this prevents overfitting of our model by having different styles of play/saved policies to train against and improved stability as the training is done with an opponent of similar level. Unity allows us to apply this technique in training. We'll explain and implement it in many tests of this project.

G. Curriculum learning and parameter randomization

If we think about training in humans, we do it progressively in difficulty. We always start easy and, in the extent of our improvements, we raise the complexity of our work. This concept can be applied in RL and is called curriculum learning [27]. We start with a basic environment and increase the difficulty of it according to the progress of our agent. This can be controlled with the ELO, time of training, score or any desired metric. In complex environments, it presumably helps to a smoother and effective training.

Unity also allows to apply this technique in ML-Agents training, and we can combine it with environment parameter randomization to aim for robust models that can generalize for tweaks or variations on the environment.

2. RELATED WORK

Unity helps the developers by giving some succesful examples using the ML-Agents toolkit. This collection of example environments [28] that come within downloading the package become fairly useful as an inspiration for our project. There we can find a lot of exportable components and examples of ML-Agents class coding and training configurations (observations, rewards and actions). For instance, the “Wall Jump” and “Strk vs Goalie” examples have been very useful for

understanding the basics of reward shaping, action creation and application of self-play and curriculum learning.

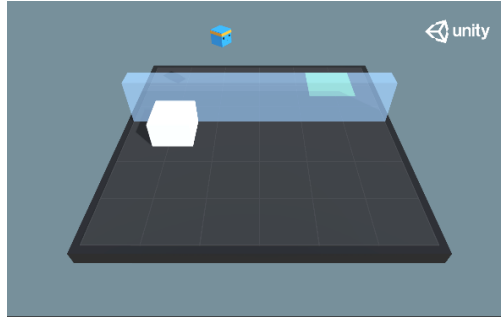


Figure 9. Wall jump game episode example

There are no known futsal RL projects, but instead some projects were found involving RL AIs in football environments. Google Research has presented an advanced, physics-based 3D open-source simulator of football [29] for RL training and customization. Additionally, the English football club Manchester City presented a competition in this C++ environment awarding the best RL algorithms [30]. Nonetheless, in our project the environments are made by us, and we'll focus in assessing multiple configurations and training rather than designing new RL algorithms or variations.

There have also been other interesting RL projects in Unity: a successful application of PPO was done in a 1 vs 1 volleyball simulation [31], Unity presented its brand new MA-POCA algorithm with "Dodgeball" [32] and good baseline examples are being openly shared by Internet content creators such as Sebastian Schuchmann [33] or ImmersiveLimit [34].

3. METHODOLOGY

In this project we've followed a progressive environment methodology [Fig. 10], starting from the easiest training environment (1 player) until finishing with the full Futsal game environment (5 vs 5). This increase in complexity allowed us to extract relevant insights from the very beginning that we carried on as we advanced, avoiding the potential confusion of starting in too advanced scenarios. At the end we managed to analyze various problem environments, performing in each one of them the same functioning philosophy:

- Design in the game environment
- Neural network configuration
- Model assessment

Firstly, design in the game environment compresses all the work in making the actual game simulation and programming it so it could work with or without training RL agents. Secondly, neural network configuration contains the management of inputs (observations, rewards), outputs (actions) and internal deep NN configuration or hyperparametrization. Finally, assessing our created models was performed with Tensorboard results and in-game testing.

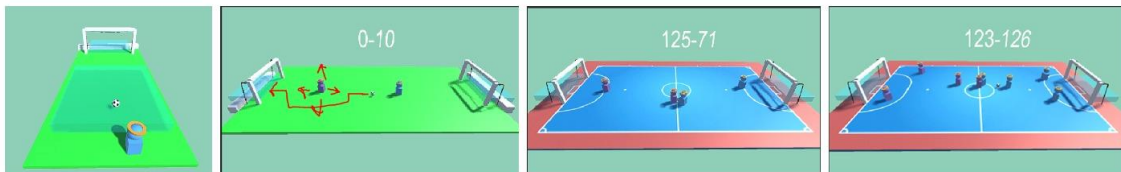


Figure 10. Evolution of our Unity game environments

A. Problem environments (game design and programming)

For the posterior trainings to be executed properly in our environments we need to ensure that the game domains we work in are suited to our intentions and are free of bugs. That's why there has been always a notable work in programming the environments and creating the 3D scenes. The behavior of the players, the ball, the physics of every component... were always an important part at the beginning. We decided and configured them taking in mind that we wanted to replicate a real Futsal game.

Unity allows us to do this in a intuitive way, letting us create elements in which we can assign a mass, drag, colliders, triggers and more. Some elements in our project were based in example environments created by the Unity team - prefabs [35] - and helped this process to success. Without getting into RL yet, we could assign scripts to the players - *MonoBehaviours* [36] - and test their actions and interaction with the ball or the scene with the keyboard. We'll discuss about specific details of every environment implementation in the Results section.

The general approach in programming was to have few scripts that managed everything: a scene controller called *FieldController*, the script for the players' control and RL training *AgentController* and a script for the ball interaction and triggering events called *Ball*. Nonetheless, in the early stages of the project we didn't need *FieldController* and controlled the environment and the agents in *AgentController*. In the scripts we determined all the rules applied to the game and that directed the training. For instance, common rules that all our environments shared are:

- If a player falls out of bounds, the game/training ends
- If a player shoots the ball out of bounds, the game/training ends
- If a player scores a goal, the game/training ends
- If we reach the limit of game duration before anything happens, the game/training ends
- A new ball appears randomly in a spawn zone at the beginning of every game episode

Most of the rules are applied based on a trigger/collision approach. Unity has multiple functions that allow us to deploy triggerable elements in the game scene that cause functions in scripts to be executed [37]. All the different game environments scenes can be consulted in the GitHub repository [38].

B. Neural network configuration

This step includes all the work that has to be done to configure and activate a RL training in a well-programmed game scene. As we discussed before, with the ML-Agents package we have access to the "Agent" C# class and we can apply it to our players. With this class we get all the functions needed to communicate our Unity environment with the NN models later. That's exactly what we have done in all the iterations of this project managing observations, actions and rewards.

- Observations

Except in certain tests that we'll discuss in the Results section, our observations provided to the deep models have always been obtained with Raycasts [39]. Several rays are cast into the game physics world, and the objects that are hit determine the observation vector that is produced. The total size of the created and fed observations to the NN per training step is obtained with the following formula:

$$(\text{Stacked Raycasts}) * (1 + 2 * \text{Rays Per Direction}) * (\text{Num Detectable Tags} + 2)$$

Where the rays per direction are determined by the total number of rays that an agent spreads, the detectable tags are certain game elements that we want to detect and "stacked raycasts" specifies how many previous observations to stack together to the deep NN. For each ray we obtain if he

detects a tag, the coding of the tag and the distance [40]. In Fig.11 we can see an example of raycasts of the purple player, each one of them detecting one of various tags such as the ball, the bounds of the futsal field, the goal...

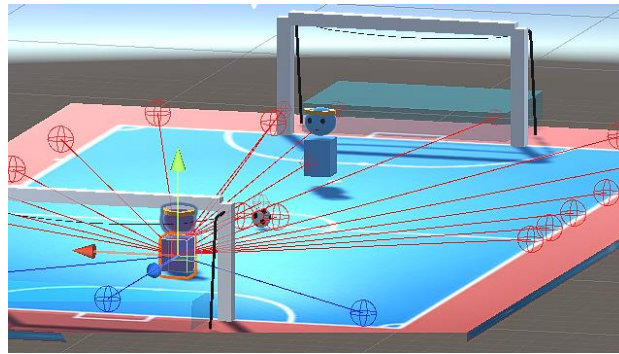


Figure 11. Rays of the purple agent and collisions

These are probably not the type of observations that maximize the gameplay results but the ones that could be closer to human inputs in Futsal. We wanted to simulate real players' observations, and there's nothing more similar than simulating the player's vision.

- Actions

In order to make training as faster as possible, we always have to try to have as few actions as possible. We have taken this approach and have chosen only two groups of actions during the project: WASD and WASD + rotation right and left. We'll talk about the decision process later, but both of them worked according to the context and provided good results. We also configured the speed of every direction to simulate reality, for instance making the forward velocity higher than the others. Unity lets us choose between discrete and continuous actions and we decided to work only with discrete actions for simplicity.

Actions are decided and received from the neural network in the "Agent" class of every player. The outcome of those actions will be used to send rewards as feedback.

- Rewards

The feedback necessary for the convergence of our trainings is generated and sent in the scripting of our games. It's our design who allows agents and the scene controller to do that task as we intend to when we do extrinsic rewarding [27]. Unity also has the option to do intrinsic rewards but we won't focus on that. Rewards are generated after action outcomes in our environment and are the key to direct our trainings in the good way. Reward modelling [41] has been one of the most important topics of research in the Futsal project.

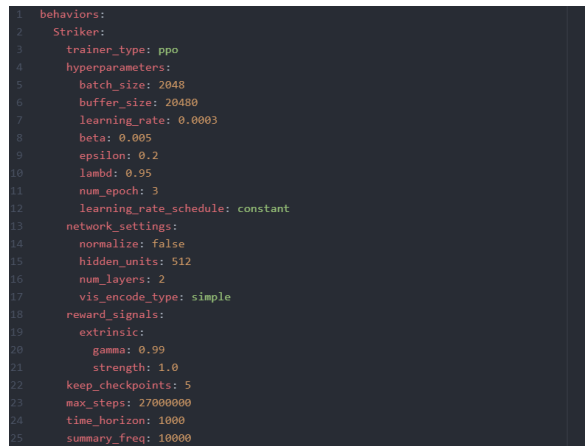
For every game domain, we've had to design thoroughly our rewards. RL rewards theory and the experimental results (obtained with trainings) were the two inputs to base our decisions on. It's important to have rewards that can speed the process but not ruin it. If you structure positive rewards, the system will want to accumulate as much as possible. This can lead to interesting behavior. Negative rewards are different. Negative rewards incentivize you to get done as quickly as possible because you're constantly losing points when you play this game. That's an important distinction at the moment of choosing them. Common rewards that have been employed in our project are the negative "existential penalty" (negative reward every step to incentivize agents to move on to get positive) and the positive "goal reward" (positive reward for when you score a goal). Reward types and values have changed a lot in the project, but the progressive environment methodology permitted us have early feedback about what rewards are stable in the long run.

- Training configuration

We've discussed over the things we can control outside the RL models, however it is also relevant to configure the neural networks from inside and module how the training process is performed. We have trained our models using *mlagents-learn* training utility and here we can see an example of training execution:

```
mlagents-learn config\ppo\Futsal2_selfplay3_curr_upg.yaml --run-id=Futsal4_ppo_bacc_3spawn2 --env="Builds/2VS2_1801_nobacspawn" --num-envs=3 --no-graphics --resume
```

We need a YAML config. file that determines the configuration of our neural network and the training mode. In the previous example we trained a model with no Unity graphics with 3 parallel environments. ML-Agents allows us to do things like this that come handy for training faster in case it's needed. Also, inside the Unity game scene we could duplicate the scene create multiple Futsal games in one Unity environment to speed up training, at cost of more demanding computation power. The YAML file does the hyperparametrization of implementations of RL algorithms that ML-Agents brings us. As we said, we work with PPO and MA-POCA, and both of them are implemented. However, configuring our deep models to obtain our best results was important. In Fig.12 we can see a YAML configuration file example of a PPO model.



```
1 behaviors:
2   Striker:
3     trainer_type: ppo
4     hyperparameters:
5       batch_size: 2048
6       buffer_size: 20480
7       learning_rate: 0.0003
8       beta: 0.005
9       epsilon: 0.2
10      lambda: 0.95
11      num_epoch: 3
12      learning_rate_schedule: constant
13    network_settings:
14      normalize: false
15      hidden_units: 512
16      num_layers: 2
17      vis_encode_type: simple
18    reward_signals:
19      extrinsic:
20        gamma: 0.99
21        strength: 1.0
22    keep_checkpoints: 5
23    max_steps: 27000000
24    time_horizon: 1000
25    summary_freq: 10000
```

Figure 12. PPO YAML configuration file example

The YAML file lets us activate self-play, curriculum learning or environment parameter randomization metrics as we can see exemplified in Annex 1. To complete our training preparation, we needed the component Behavior Parameters [42] and Decision Requester [43] in our agents' components. Implementation can be seen in the repository [38] and that was the way to indicate Unity that those players would need to communicate with the Python API for training.

C. Model assessment

After our models had been created, we assessed them with Tensorboard and in-game testing. TensorBoard [25] is a TensorFlow utility that displays different generated statistics of the training process in a graphic dashboard. We could then compare different policy and environment results of various models trained. In-game testing refers to games between different models and scoring + behavior comparisons. Analyzing how the agents played was logically important in order to improve or evaluate them.

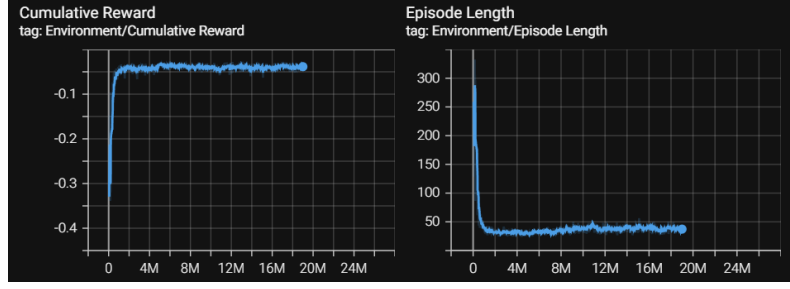


Figure 13. Tensorboard graphics example

In Fig. 13 are two of the recurrent graphics that we used to assess the quality of trainings. The first is the mean cumulative episode reward over all agents, and during a successful training session until the stabilization moment. Because the rewards can change, the importance of this graphic resides in the shape. A successful training, regarding of the gameplay results, will increase in reward in the first steps and then settle.

Assessment with in-game testing can be done thanks to the Behavior Parameters component what we need to append to agents. There we can choose the behavior type of our agents: heuristic, learning and inference. With inference we can control the agent with trained models without affecting the neural connections of the algorithm, just for testing. That's what we have done to evaluate gameplay demeanors.

4. RESULTS

In this section the discussion of the results in every environment is assembled. Starting from the simplest environment, the assessment in the design and decisions made has been important in every case, allowing us to build and argue conclusions for the end of the project.

A. Striker vs Goal

The main elements of the game environment consist of one player, a futsal ball and a futsal goal. Even though this is the simplest domain of the project, we faced many challenges at the beginning. At this first step was very important to model the collisions, physics and rules of the game so we could update our game based on this. The actions of our player were tested and assessed graphically so the interaction with the ball was realistic. The mass, drag and angular drag of both the player and the ball were chosen according to this reasoning. They can be changed and tested easily in the Unity environment. The heuristic mode in the Behavior Parameters component of the Agent permitted us to test the player without RL training, such as in user-controlled games.

We based the creation of our Agent, Ball model in prefabs found in example environments [28]. The first action group we decided to implement was a WASD scheme of 4 options. The rules of this environment are simple: if there's a goal, the episode ends; if the ball or the player fall out, the episode ends. To achieve this we placed different trigger zones in the environment and controlled the system and the scoreboard via scripting. We decided to spawn the ball randomly in a designated area so we could train the player to score in multiple situations (Fig. 14).

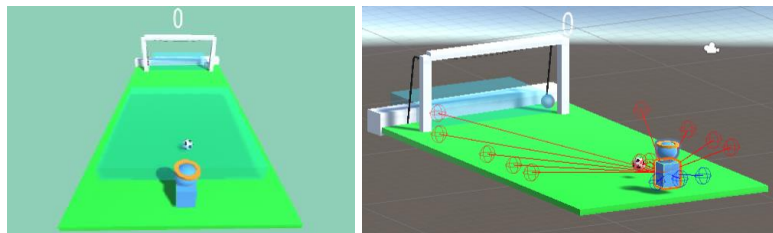


Figure 14. Agent environment and spawn zone (left) and display of rays of the agent (right)

We conducted several tests to evaluate different reward schemes plus the effect with actions, observations or curriculum learning. Reward shaping is a big deal. If you have sparse rewards, you don't get rewarded very often [41] and just rewarding goals wasn't enough for the agent to learn. We wanted to instead shape rewards that get gradual feedback and let it know it's getting better and getting closer. Also, not being careful with rewards can produce unexpected outcomes. In Figure 15 we have the summary table of our tests and rewards.

	Actions	Observ.	Rewards	Curr.Learn	Notes	Goal Ratio
Futsal1_1	WASD	4 rays	Goal +1 Fall -1 BFail -0.5 Exist. -0.002	No	Player hides in the goal	0%
Futsal1_2	WASD	4 rays	Goal +1 Exist. -0.002	No	Player suicides	0%
Futsal1_3	WASD	4 rays	Goal +1 Fall -0.1 BMC +0.05 Exist. -0.001	No	Goals + suicides	34%
Futsal1_4	WASD	4 rays	Goal +1 Fall -100 BMC +0.01 Exist. -0.001	No	Hiding in goal + goals	38%
Futsal1_5	WASD	4 rays	Goal +100 Fall -100, InGoal -100 BMC +10 Exist. -0.01	No	Coward goals	46%
Futsal1_6	WASD	4 rays	Futsal1_5	Yes	Not necessary	28%
Futsal1_7	WASD+RR	5+3 rays	Futsal1_5	No	Great	95%

Figure 15. Table of test models in StrikervsGoal environment. BFail: ball fall reward. BMC: Ball is a meter closer to the goal reward. InGoal: player touching goal penalty

An episode has a maximum length of 1000 steps and ends automatically if no terminal rewards have been given. In this environment, we trained each of our models for 1M steps and analyzed the gameplay and numerical results in TensorBoard. Each of the entries on the table managed to converge but only watching the gameplay allowed us to analyze the quality of play. In addition, the Goal Ratio metric was created [Annex 2], given that we challenged us to achieve at least a 80% of goals in the total of episodes to advance to the next level.

The RL algorithm used in all models was PPO, in a NN with 2 layers of 256 hidden units. We parametrized PPO by the default values given by developers in example environments. The initial observation rays consisted of a ray for every WASD direction. This should make the RL training to proceed smoothly and if more complexity is needed to do it after result analysis.

In Futsal1_1 and Futsal1_2 models we experimented the danger of messing with rewards. In the former, we abused of negative penalties and as a result our agent didn't want to take risks and hid in the goal. In the latter, the lack of rewards to help the training process made the agent be unsuccessful. We improved the reward scheme by creating a reward for bringing the ball 1 meter closer to the goal and bigger penalties for abrupt episode endings for falling out of the field. This improved the training process and alongside a penalty for entering the goal brought the best results. We implemented this penalty because the agent conformed itself with scoring some goals and missing others by entering the goal until the episode ended abruptly.

At this point we achieved a 46% of goals, which wasn't enough. In this model the agent lacks of an extra level to score the difficult goals and acts cowardly when the ball is in risky spots. Then we started to look at other options to improve the RL models. It was decided to test curriculum learning, a technique we already introduced. We did it by restricting the size of the spawn zone more at the beginning and making it bigger according to the training progress. We didn't do just one quick test, but several configurations where we modelled the curriculum learning parameters

[Annex 1]. It didn't improve the results in the same number of steps, which was our goal. We believe that the problem environment isn't complex enough so that it makes a difference, and the next model demonstrated it to us. Futsal1_7 received an improvement in the number of actions performable by the agent and the RayCast observations. We added rotation (right and left, slower than other movements) to the WASD controls and boosted the observations fed to the NN from 32 to 256, creating more rays (from 4 to 5 in front and 3 behind) and stacking them in groups of 3 steps. The number of observations generated can be calculated by the RayCast formula discussed previously in this document. With this model we achieved a 95% of goals. The agent positions itself well after each goal and scores slowly to maintain the effectivity. This is a succesful start and base for the next environments.

B. Striker vs Striker (1VS1)

The addition of an opponent agent brought the futsal field design to the table. This game domain contained 2 agents, 2 goals and 1 ball in a 40x20 m field. The incorporation of a new agent meant that we couldn't control anymore the game from the script inside our agent, but we needed to create a new script that managed both of the agents actions in episodic terminations or rewards (*FieldController.cs*). From this general script we could extrapolate our rules to the 1VS1, adding the relocation of the players in every new episode as a new directive. Moreover, the possible tags detected by the rays had to be completed so they contained all the new elements in the environment, such as the other player or the other goal. The new detectable tags went from 4 to 7 (e.g., for purple agent: *goalB*, *goalP*, *field_limits*, *ball*, *blueAgent*, *goal_partsP*, *goal_partsB*) and consequently the number of observations created increased by 1,5x.

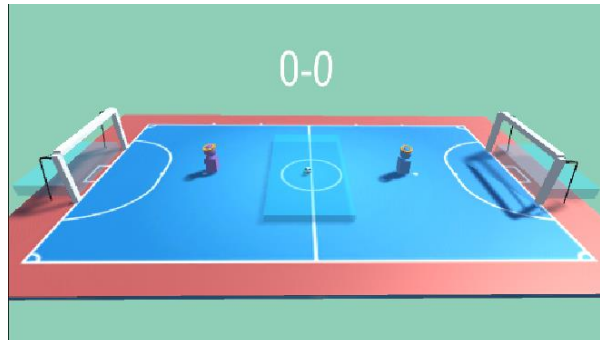


Figure 16. Striker vs Striker game environment in Unity.

Another important change was to implement compatibility with testing with MA-POCA through registering the agents in SimpleMultiAgentGroups [44] that enable cooperative training and cooperative rewards. This allows us to scalate our teams to have more players easily in the future, whilst mantaining compatibility both with PPO or MA-POCA. PPO only works with individual rewards, but group rewards are translated automatically if we apply PPO in order to avoid issues. In Figure 16 we have the graphical overview of the 1VS1 environment. The ball spawn area and the goal triggers can be seen. Also, there are triggers in the borders of the futsal field to terminate episodes in case the ball goes out.

This was the environment where we spent more time given its criticality: it is the simplest of the adversarial environments and any problems that may have in the complex cooperative adversarial trainings, with more players per team, we thought it could be solved here beforehand. In addition, all the succesful testing that we did here regarding the environment and scripting would be very useful for the next steps, with very similar conditions. A lot of testing has been done and the relevant points of interest are organized in different subsections. At the end we englobe all with the key RL models.

- Self-play testing

One of the challenges in this game domain was to evaluate the effects of the self-play technique and if it could induce better training results, in terms of speed and gameplay against other models or styles of play to assess possible overfitting. With ML Agents we can configure and add to the YAML file these self-play parameters:

- *Savesteps (SVS)*: trainer steps between snapshots. If bigger, wider variety of policy /opponents. Will take more time to optimize though.
- *Team_change (TC)*: number of trainer_steps between switching the learning team.
- *Swapsteps (SWS)*: Number of ghost steps (not trainer steps) between swapping the opponents' policy with a different snapshot. If higher, ghost will change less times his policy.
- *Window (W)*: how many policies do we save as snapshots. If bigger, wider variety of policies/opponents. Will take more time to optimize though.

With that in mind we generated several RL models with different parametrizations. Most of them failed to beat models without selfplay, but one of them performed better against every model in the same training conditions: SVS of 10000, TC of 100000, SWS of 5000 and W of 10 snapshots in trainings of more than 2M steps. The results were good against other selfplay versions (100-67, 102-77, 103-44) and the version without selfplay (100-88). However, the difference wasn't usually high and we expected it to grow in more complex environments.

- Reward modelling. Progressiveness

The reward scheme we had built at the previous environment worked pretty well, however didn't extrapolate to the 1vs1 domain. In 1vs0 we gave a penalty to the agent for getting into a goal because not doing it limited training performance. Here and from now on it's not applicable, given that the agents can get into goals to protect them against the opposing team. This was the first of many changes we tested and assessed in various converging models (Fig. 17). At the end, we obtained a reward model that brought good and intuitive behavior to the players, as we desired.

	Rewards	Notes		Rewards	Notes
Striker vs Goal	Goal 100 Fall -100, InGoal -100 BMC +10 Exist. -0.01	Not logical in 1vs1	Strik vs Strik Less Rew. Cut	Goal +1 GoalAg -1 Fall -1 BallOut -0.1 BMC +0.03 ABMC -0.03	Progressive learning. Discutible results.
Strik vs Strik Rehearsal 1	Goal +1 GoalAg -1 Fall -1 BMC +0.02 BallOut -0.05	A lot of balls out. Mediocre scoring	Strik vs Strik Progressive BMC	Goal +1 GoalAg -1 Fall -1 BallOut -0.1 BMC +0.05 --> +0 ABMC -0.05 --> -0	Stable learning. Good behavior
Strik vs Strik Rew. Cut	Goal +1 GoalAg -1 Fall -1 BallOut -0.1	Doesn't learn. Too sparse rew.			

Figure 17. Table of reward schemes used in 1VS1 testing experiments

The first rehearsal on 1vs1 didn't go well. Firstly, we scalated all rewards so a Goal was a reward of 1 by recommendation of the Unity developers [27]. We included a penalty for receiving a goal and maintained the BMC reward that incentivized shooting the ball closer to the other's goal. The players prioritized shooting the ball forward rather than scoring, resulting in a lot of lost balls and

quick episodes. We realized that now the BMC reward, with opposition in front, made the model converge to a behavior that conformed itself with gaining some positive rewards as BMC rather than making goals. The consequent decision of deleting BMC in the reward cut model didn't work properly either. Now the rewards were too sparse for the agent to learn, so we had to try working a "right in the middle" solution with the Less Reward Cut model. In Fig.18 we can see the difference in value function loss of both trainings. A successful training will have a value loss that raises at the beginning but then stabilizes. Here we rewarded shooting the ball 1 meter closer to the opponent's goal by 0.003 and penalized the opposite by 0.003. Now the RL algorithm could converge properly, but the convergence point was still mediocre. The players were still conforming with shooting the ball to the sidelines and acting conservatively. The Less Rew. Cut model lost several times against the Rehearsal 1 model, much more aggressive attacking the ball. It is apparent that different reward schemes, most of the times, led to different styles of play.

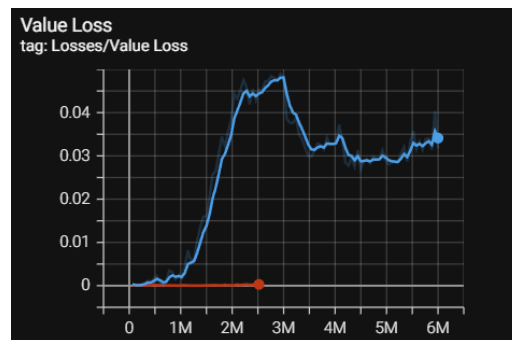


Figure 18. Value loss function of Rew. Cut (brown) and Less Rew. Cut (blue) training sessions

The key change that made the difference was making the BMC progressive over time. We used the curriculum learning technique to indicate in the YAML how we wanted BMC and ABMC to be in our training. The first 10% of the total steps would be 0.005/-0.005, until 40% 0.003/-0.003, until 70% 0.001/-0.001 and it would disappear until the end and in future extra trainings. This provided us the perfect system to train our agents, helping them at the beginning to recognise the task and redirecting them to value the importance of goals as they became better. Shorter Progressive BMC models (3M steps) are better than longer (6M) versions of other models trained.

Finally, we also attempted adding a reward based on ray detection that rewarded the agent seeing the ball. We did that because in later stages the players had problems reacting to losing sight of the ball and becoming confused. Nevertheless, this didn't help. It was counterintuitive to us to force this reward, because with more teammates it is not always the best strategy to look at the ball in all times, at least in real futsal. Anyway, the consequences of the reward were that the agent was just content by seeing the ball and playing conservative.

- Action/Observation variations

The action scheme at the beginning consisted of a branch of 7 different possibilities (forward, backwards, left, right, rotation right, rotation left and nothing). The fact that we could only perform one of those at the same step became restrictive as we added complexity and thus we decided to change to a 2 discrete branches design, where the players could choose a direction and rotation at the same time (forward, backwards, left, right or nothing and rotate right, left or nothing). This increase in possible outputs of the NN didn't ruin the convergence in trainings and made a beneficial difference in gameplay.

In the observations side, we started with the Striker vs Goal design, with a total of 297 generated observations. We found after a while that this wasn't enough, and the first problem that we detected is that in situations where a player had the ball and the opponent player in front, couldn't detect both of them. This goes against our aim of making the observations similar to the human

vision, so we decided to reinforce the observations with an additional group of rays located on the superior part of the agent. This necessary upgrade [Fig. 19] raised the number of observations to 621 although it gave better gameplay results.

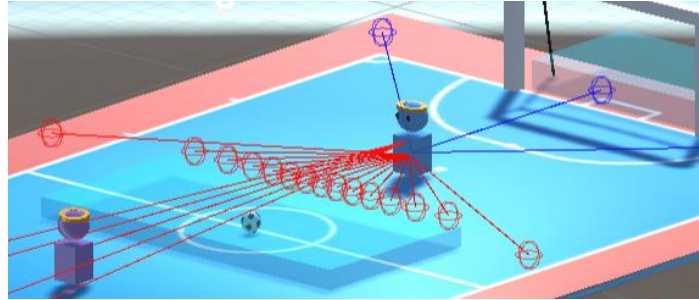


Figure 19. Visualization of the two layers of front rays

At a advanced stage of the analysis, we found ourselves with a recurrent problem: agents struggled to react in long episodes and when they lost the sight of the ball. A test with speed of the ball or position as a discrete observation didn't improve things in multiple tests: we believe that in a big observation space caused by raycasts they didn't manage to make that of a difference. Also, because we stack three raycast observations, we have some indirect memory and the NN can infer the direction of where the ball is going. An extra recurring issue we had was that the blue team was always performing better, but this was just solved by reviewing the positioning of both players. A pixel of difference on the starting position made an impact on which player won in the long run.

The alternative chosen to improve the agents' gameplay was to prune the observation space to make training more efficient. We called this the "*rebuild*" and consisted of the junction of the tags "goal_partsB" and "goal_partsP" with "goalB" and "goalP" respectively. This diminished the number of observations from 621 to 483. A different prune considered was erasing the "ball" and "goal" tag detection from the high raycasts, but it only decreased observations a bit more to 453. The "*rebuild*", together with other modifications proved to be effective and improved the decision-making of agents.

- NN size and ball spawning

We carried our NN design from 1vs0. It seemed to be as effective in 1vs1 that in 1vs0, but we tried to test if we could apply any notable improvements that affected training. The `learning_rate` [23] corresponds to the strength of each gradient descent update step. This should typically be decreased if training is unstable, and the reward does not consistently increase. This problem didn't arise in our case, but we decided to test a *bigger PPO* NN of 2 layers of 512 hidden units with more conservative parameters. For this, we decreased the beta value of the PPO algorithm a little, we halved the `learning_rate` and we increased the batch and buffer size. The tests employed with more ambitious NN configurations didn't go well. Seems that for the 1vs1 configuration our previous PPO YAML worked almost perfectly for this use case. Nonetheless, in future environments this would be reevaluated again and the conclusions will be different.

Finally, another change regarding training was implemented and tested. We tried increasing the spawn zones of the ball by making the ball appear behind the blue or purple team in a 20% of the episodes. This was carried out by a random parameter in a script that moved the spawn zone according to probability in every episode. The aim of the change was to check how a more *varied spawn* was going to affect the training results, if it would make the decision-making of the players more complete. The results were mixed in 6M step trainings. The new-spawn models behaved better when playing with the new spawn against traditional models, but this didn't translate into old-spawn games, where the traditional models stood strong. That may make sense, but our goal

was to demonstrate if a more complete training would make an agent better in all kind of situations (including long or unconventional episodes) and here wasn't the case.

- Putting all together: the best model

Every point of interest discussed wasn't tried individually but in waves combined with other modifications and assessed based on the gameplay and convergence statistics. The outcome of the analysis is that the best model is the one that is trained with the BMC progressive reward scheme, 2 branches of actions, 483 observation space with the *rebuild* applied and the PPO algorithm configuration file of the NN we used in the 1vs0 phase, all trained with the original random spawn zone in the middle.



Figure 20. Gameplay of our best model playing itself.

The agents don't miss easy goals and have learnt to position themselves intelligently by facing the ball [Fig. 20], protecting the goal and forcing out of bounds of the other player in plays next to the sidelines. Because of the less speed of the agents going backwards, agents try to dribble by surprising the other agent with a self-pass to overcome them. There are still some occasional confusions, but overall the results are satisfactory. Most of humans would lose to this agent - setting the behavior type to heuristic in Behavior Parameters lets us try that with hardcoded controls. The optimum model is included in the repository of the project.

C. 2 VS 2

At this stage of the project, with the inclusion of a teammate in both of our teams a new point of interest comes into play: cooperation. In terms of the game environment and programming most of the work is done, thanks to our scalable approach at the time of implementing the past features. The focus shifts to how good our models can be in a cooperative adversarial environment, and this opens up the inclusion of MA-POCA and group rewards to the already implemented PPO configurations. In summary, the aim of this phase is to obtain training insights in order to scalate stably with more players.

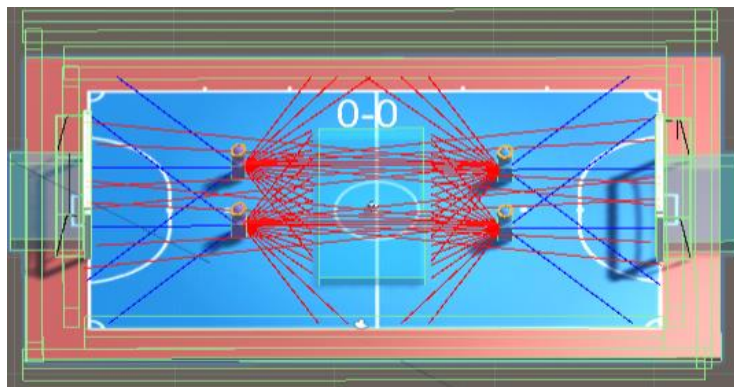


Figure 21. 2VS2 game environment elements

- Actions, observations and rewards

As introduced, we maintained most of the 1VS1 results in the 2VS2 initial game configuration. In Fig.21 we can observe the positioning of players and a full disclosure of all the rays involved in a game episode, following the 1VS1 *rebuild* structure with two layers of front rows and 3 rays at the back of every player in order to emulate the awareness of the field elements location. Because of the need for a detectable tag for the teammate of each agent, the number of observations generated per experience is instantly augmented to 552 – recall that we’re always following the formula introduced in the Methodology section.

It was decided that actions and rewards remained equal originally and later on we saw that they were already suitable to achieve good results. However, the adaptation of the rewards to the MA-POCA algorithm was necessary. In MA-POCA, we can use at the same time individual and group rewards for a team of X agents. We already explained in the *Introduction* how it allows to train cooperative behaviors in multiagent adversarial games, following the philosophy of centralized learning with decentralized execution (actor-critic). MA-POCA is built for environments that also need to handle a dynamic number of players in-game, but in our case the number of players in a training episode remains fixed. Group rewards are meant to reinforce agents to act in the group’s best interest instead of individual ones, and are treated differently than individual agent rewards during training [45]. In Fig.22 the chosen separation of rewards is displayed. The episodic ending rewards were defined as groupal rewards for MA-POCA, excepting the fall from the environment of one player, which should just penalize the actual player.

	Group Rewards	Individual Rewards
MA-POCA	Goal +1 GoalAg -1 BallOut -0.1	Fall -1 BMC ABMC
PPO	--	All of MA-POCA

Figure 22. Rewards table of 2VS2

- Neural network configurations

One of the advantages of MA-POCA in ML-Agents is that uses the same configurations as PPO and there are no additional POCA-specific parameters. This feature permitted us to train RL models with the same YAML configuration that we dragged from previous stages. We maintained our progressive rewards (BMC and ABMC) implemented with curriculum learning and the self-play module design. The increment in complexity in our game context made us reconsider the NN configuration and assess a more ambitious design – as we’re going to explain hereafter – which was already put at test before but didn’t make an impact.

The parameters where chosen based in various tests of different values separately. This new configuration design (*bigger YAML*) differed from the previous NN and training (*original YAML*) as we can examine in Fig. 23.

	original YAML	bigger YAML	Notes
batch size	128	2048	experiences for any iteration of gradient descent update
buffer size	2048	20480	experiences before learning
learning rate	0.003	0.006	strength of each gradient descent update step
beta	0.01	0.005	strength of the entropy regularization (+ randomness)
hidden units	256	512	how many units are in each fully connected layer in NN
nr of layers	2	2	internal layers of the NN
SP: save steps	10000	25000	--
SP: team change	100000	100000	--
SP: swap steps	2000	5000	--
SP: window	10	10	--

Figure 23. Comparison between our two NN designs in 2VS2

The main differences relapse into the size of the network layers (doubled) and magnitudes in gradient descent. An error we committed was lowering the beta and reducing the randomness regularization, an effect we didn't want to apply. The self-play (SP) parameters that we already explained before were scaled a little bit to adapt to a more complex training process. With these two configuration files we performed most of our trainings in this stage. We could adapt the *max_steps* variable to determine how progressive rewards worked and limit the steps for opposing models. For instance, we could train until 12M steps with progressive BMC and then to 24M in the same model with BMC and ABMC at 0 (Fig. 24).

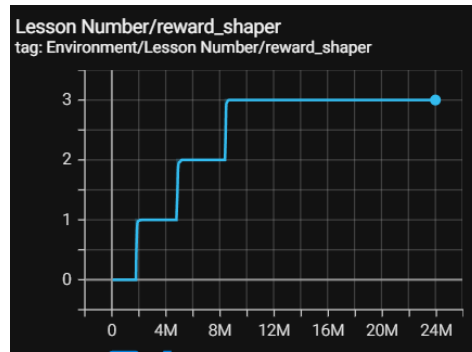


Figure 24. *Reward_shaper* variable for BMC and ABMC stages in a 24M training. The progressivity from 0.005 to 0.003 to 0.001 to 0 is done until 12M.

- Result models

Our testing began with 4M and 8M trainings of models in the original YAML configuration that we dragged from previous stages. The *rebuild* observation change that we introduced in 1VS1 wasn't yet made, since we decided to get some results in 2VS2 to finalize our testing in the Striker vs Striker game environment. In this models, we got the first signs of cooperation both in PPO and MA-POCA with coverages. A coverage [46, Fig.26] is a term that refers to a player defending the back of a teammate when he has to abandon its position. PPO agents were better, winning in direct games against POCA thanks to a better finishing ability against the goal. Nonetheless, the behavior was far from ideal when the episodes took long enough.

	steps	REBUILD	Notes
PPO original YAML	4M, 8M	NO	First signs of cooperation, one attacks other covers Bad at balls close to out of bounds
POCA original YAML	4M, 8M	NO	Coverages + 2 players go aggressively. Miss easy goals
PPO original YAML	12M, 24M	YES	Good coverages and goal scoring. Bad permutation
POCA original YAML	12M, 24M	YES	Tough losses against 1 player of PPO (26-20) and 2 players (42-3)
POCA bigger YAML	12M, 24M	YES	Very quick episodes, good tactically, bad technically. Mediocre in long eps.
PPO bigger YAML	12M, 24M	YES	Slightly better than PPO original (100-88, 160-120). Not much change from 12M to 24M

Figure 25. Table of relevant models in 2VS2

We inferred that the clumsy behavior in long episodes may get better as the training when further, so we decided to extend the executions until 12M and 24M. It was also the time when we applied the *rebuild* to be more effective with observations. PPO with the original YAML managed to get good results in most of the episodes, with constant coverages and players being able to score the easy goals. They also showed good understanding in positioning. The bad news were that we didn't see much permutations [47], which refer to overcomed players getting back to help their teammate and are usual in futsal. We questioned ourselves if we needed a more ambitious NN model to solve this type of issues. POCA in 12M or 24M with the original YAML wasn't good and suffered embarassing results. This could also be seen in Tensorboard. This was another incentive to step up the neural network size.

POCA improved a lot with the *bigger YAML*, demonstrating that for the first time in the project we had witnessed a limitation of our original NN configuration in the problem context. PPO didn't suffer that much from it, but with the *bigger YAML* we obtained a slightly improved performance. POCA based models showed good tactical awareness, quick decision making yet worse technical abilities at the time scoring than PPO. Finally, our best model was PPO with the bigger YAML and we think the gameplay results are satisfactory. The lack of permutations in some instances is still an issue, but in most episodes the actions taken are very logical. Another aspect that we missed is passing, but the difference in our game design and action possibilities compared to the real-life game makes it understandable, and we assume it is a limitation of our Unity game context.



Figure 26. Example of coverages to a teammate that goes to the ball

D. 4VS4 and 5VS5

This stage was the last of our project and consisted of applying the previous relevant ideas in the new main game context that we aimed for at the beginning, the futsal game of 4vs4 or 5vs5. Until now, we obtained solid models in every environment. Nevertheless, this two new game contexts bring the need of cooperativity at the next level - at least it's what we expected at the starting point. We quickly saw that the dimensionality of the challenge was higher than anticipated.

As we concluded from the 2VS2 environment, PPO behaved slightly better with the *bigger* configuration file than with the original. After more testing in the 4VS4 environment, we can affirm that the added complexity makes the original YAML non-viable with PPO anymore. In Fig.27 it is shown the cumulative reward evolution of PPO with the original YAML (green) and with the bigger YAML (pink). We always used this graphic, not to assess the quality of a model but to assert its convergence to something. A shape such as the pink case demonstrates quick evolution of reward and then stabilization to a certain mean maximum as expected, which doesn't mean that the agent isn't still getting better - remember that the models are always facing past versions of themselves, the overall reward cannot always increase. The green line shows that the training has failed, with the same problem at hand but with the original YAML. The conclusion is that the original YAML has become too simple for us.

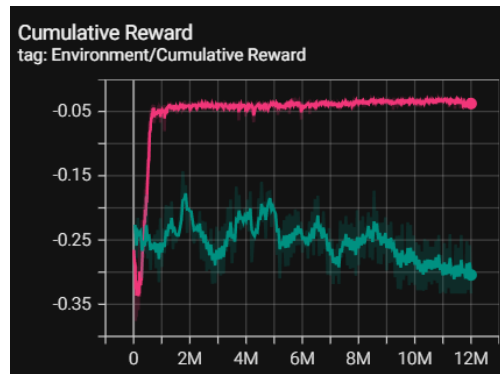


Figure 27. Cumulative reward graphic of two trainings of 12M steps

Another point of interest of this chapter is to assess if PPO is really always more effective than MA-POCA - in equal conditions. With more agents coming into play, our hypothesis was that MA-POCA would make a difference as the complexity and needed for teamwork grows. We'll see how it went and the various test contexts that we presented to solve the question.

- Changes to training: old and new ideas

In past phases we tested the *varied spawn*, a software mechanism that randomized even more the spawn and added two fictional zones behind one and the other team. The ball could then appear there a 20% of the trainings. The goal was to make agents stronger in unusual situations and reactive against them. We recovered this idea in this stage because the first trainings showed that the agents had a strong lack of response capability in strange game states. We decided to do it here with a 20% probability that the ball appears behind blue team and a 20% for purple team. Another modification we tested thoroughly and tried to address the same issue was the deletion of the back rays of the agents. Through this project we had worked with the back rays to simulate the human's constant awareness of the field: our hypothesis was that this made the players conform themselves with no turning the character, even though it's faster running forward, given that they already possess some kind of vision behind and they were not properly acknowledging that is better to turn quickly.

- Results: training and relevant models

In Fig. 29 we have the table of relevant models of the 4VS4 environment. The models are labeled with a number in parenthesis. They are distinguished by the learning algorithm, if the back rays (BR) were used and how was the spawn used for training. The first two important models were the PPO and MA-POCA versions of maintaining the BR and not using varied spawn. The gameplay was pretty simple, with one of the 4 players always reaching the ball, one covering and two trying to protect the goal. We can see a visual screenshot in Fig. 28, as the blue team. There were no signs of passing and the way they behave is reminiscent of the beginnings of football and futsal, where the game was much more individually oriented. PPO overplayed MA-POCA in this and the subsequent configurations. MA-POCA showed sluggishness and slow individual reactions to events in the game.

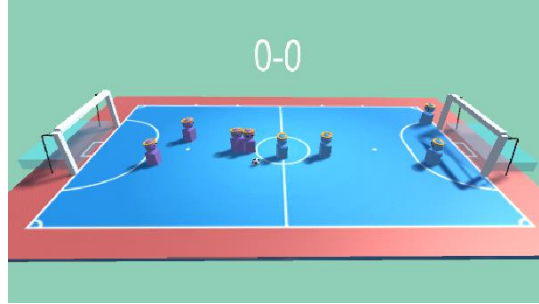


Figure 28. Game between agents with model (2) in purple and (1) in blue

The problem we had with (1) was that in strange states the agents got confused easily. With the back rays deletion, we improved performance in models until 12M steps - thanks to a lower number of observations/complexity - but as we trained further the original configuration (more complex) got better and better. On the other hand, the varied spawn forced a different behavior of the players at the back, making them step behind the goal to achieve a better panoramical view of the field (Fig. 30). It didn't translate as an improvement in gameplay or scoring performance. In tested games with the normal spawning in the middle, they were worse than original models, whilst in games with the varied spawn (not realistic to futsal) they were just slightly better. Finally, we made a divergent test that consisted of adding a lot more observations at the back of the players. This clashes with our requisite of having realistic observations, but we wanted to test how much of an impact this made. The output was that the model generated was just a little improvement of our best model (1).

4VS4	steps	back rays	trained with varied spawn	notes
PPO bigger YAML (1)	12M, 19M, 27M	Yes	No	2 keepers, 1 covers, 1 goes. Good (only) in frequent states. Improves over steps
POCA bigger YAML (2)	12M, 19M, 27M	Yes	No	Loses against PPO in every model trained. Sluggish play.
PPO bigger YAML (3)	12M, 19M, 27M	No	No	Prioritizes having the ball in front. Better than (1) until 12M, then worse.
POCA bigger YAML (4)	12M, 19M, 27M	No	No	Same symptoms as (1) and (2) difference. Inferior quality.
PPO bigger YAML (5)	12M, 19M, 27M	No	Yes	(3) wins in normal spawn games. (5) wins by little in varied spawn. Different behaviour from players at back.
PPO bigger YAML (5) + rays	12M, 19M, 27M	Yes, much more	No	Slightly better than (1), but not much of a difference

Figure 29. Table of relevant models in 4VS4

As mentioned, the best model of all for us is the first one we trained: with the *bigger YAML*, maintaining the back rays and not training with the varied spawn. That also means that at the end we weren't able to solve some long episode confusions with our testing, even though we obtained several insights about the reasons why our attempts failed.

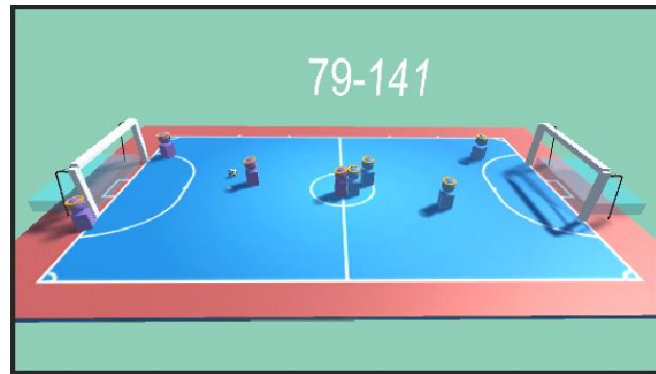


Figure 30. Screenshot of game played between purple model (trained with varied spawn) and blue

The movement to 5VS5 was very straightforward, as we only had to include one more player per team (Fig.32). In this definitive environment we conducted the last tests of the project, with our *bigger YAML* with PPO and MA-POCA and then creating a new *biggest YAML* for further testing. We didn't touch much of any other elements, given that all the work we previously did was to smooth the transition to this moment. The reason why we created a *biggest YAML* was because for the first time, the MA-POCA model in the testing (Fig. 31) was very close to PPO's level (lost 100-84, 100-88, 101-84). Maybe we thought it was as a consequence of the latest raise in complexity and we wanted to test further with a NN design with a few tweaks.

5VS5	steps	YAML file	Notes
PPO 1	19M	bigger YAML	Simple futsal gameplay, a lot of player under goal. Decent positioning. Wins everything. No passing
POCA 1	19M	bigger YAML	Much more direct. Almost as good as PPO1 in games.
PPO 2	19M	biggest YAML	Much worse than PPO 1. Slower and more confused players. Much better than POCA 2.
POCA 2	19M	biggest YAML	Good positioning but very slow reaction time of agents.

Figure 31. Table of relevant models in 5VS5

The *biggest YAML* differed from the other in a few things: beta from 0.05 to 0.1 to slightly increase randomness phase, epsilon from 0.2 to 0.15 to slow down training speed, layers from 2 to 3 and batch size from 32 to 512. We think that this changes went too far and that's why the models PPO and POCA 2 in Fig.32 weren't competitive. We already had a suitable configuration in *bigger YAML*, and at the end we concluded that PPO was also more competitive. POCA 2 lost the improvement signs we had seen in the 1st version of both of the algorithms, and performed notably worse with the change.

Our best model was PPO 1. The gameplay was interesting because it had a resemblance with old fashioned football, where one player tries to overcome the opposition and the teammates just try to position themselves wisely. After some pondering, we can say is understandable that we didn't see any passing, since three players almost cover all the goal and high shots cannot be made. The

results aren't as good as in other early stages of this project but there is room for improvement and polishing measures can be tested.

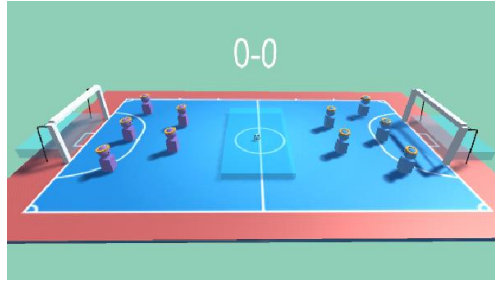


Figure 32. Initial disposition of a 5VS5 episode

5. CONCLUSIONS

In this work we have built different Futsal AIs in Unity game environments designed and implemented by us. After an introduction to reinforcement learning, the ML-Agents Unity toolkit and the main focuses of our project, we detailed the methodology used for creating and assessing our machine learning models, to then analyze thoroughly all the results and design decisions to extract relevant insights.

We can say that we generated very solid AI behavior in the 1vs0, 1vs1 and 2vs2 environments, having in 4vs4 and 5vs5 more difficulties as a consequence of the jump in complexity, even though we obtained simple but sometimes effective team gameplay that reminds us of how the game was played in the beginnings of Futsal. We believe that there is room for improvement, and our focus would be first in modifying the environment design to make the game states as similar as possible to the real human game. In this work, for simplicity and skill reasons, we created a simpler version of Futsal with action and state possibilities more prioritized to allow easy testing and succesful trainings, and we think that a consequence we limited the learning potential of the agents in the full game environment.

Our research extracts that, in similar configuration conditions and equal training time, the PPO models ended being always better than the MA-POCA ones, what contradicted our initial hypothesis. Even though PPO didn't have a cooperative oriented design, the agents managed to react mostly well to the teammates actions in the big environments. Only the 5VS5 MA-POCA model showed itself closely to the competitiveness of PPO, which can be a good point of start.

We've experienced the difficulties of training in RL, having a lot of variables that come into play and have to be managed. Changes or tweaks in rewards, actions, observations and neural network configuration can all make the difference and make the process not trivial. The inclusion of a progressive decreasing reward for pushing the ball to the opponent's goal was one of the best solutions we proposed and verified in this report. In addition, we believe that our design in the ray observation space ended being very efficient and we saw that can compete against agents with a much higher unrealistic observation space. It is not easy to recreate human vision and we sure think that we could still improve it with more time. Also, we consider that this work was useful for assessing some RL techniques that ML-Agents offered, such as self-play or curriculum learning (that we used for the progressive reward).

6. ACKNOWLEDGEMENT

Much appreciation to Ernst Moritz Hahn for supervising the work and giving me the necessary guidelines to move forward. In addition, I'd like to thank the University of Twente for allowing me to do this placement project so I could stay for a semester at this great learning place.

REFERENCES

- [1] Introduction to NIM <https://brilliant.org/wiki/nim/>
- [2] Nork, B., Lengert, G. D., Litschel, R. U., Ahmad, N., Lam, G. T., & Logofătu, D. (2018, September). Machine learning with the pong game: a case study. In *International Conference on Engineering Applications of Neural Networks* (pp. 106-117). Springer, Cham.
- [3] Deep Blue algorithm <https://www.professional-ai.com/deep-blue-algorithm.html>
- [4] AlphaStar: Mastering the real-time strategy game StarCraft II <https://deepmind.com/blog/article/alphastar-mastering-real-time-strategy-game-starcraft-ii>
- [5] Cunningham, P., Cord, M., & Delany, S. J. (2008). Supervised learning. In *Machine learning techniques for multimedia* (pp. 21-49). Springer, Berlin, Heidelberg.
- [6] Barlow, H. B. (1989). Unsupervised learning. *Neural computation*, 1(3), 295-311.
- [7] What is Futsal. <https://www.edpsoccer.com/page/show/1491802-what-is-futsal>
- [8] A guide of reinforcement learning <https://deepsense.ai/what-is-reinforcement-learning-the-complete-guide/>
- [9] Tesauro, G. (1995). Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38(3), 58-68.
- [10] Watkins, C. J., & Dayan, P. (1992). Q-learning. *Machine learning*, 8(3), 279-292.
- [11] Proximal Policy Optimization <https://openai.com/blog/openai-baselines-ppo/>
- [12] Cohen, A., Teng, E., Berges, V. P., Dong, R. P., Henry, H., Mattar, M., ... & Ganguly, S. (2021). On the Use and Misuse of Absorbing States in Multi-agent Reinforcement Learning. *arXiv preprint arXiv:2111.05992*.
- [13] Policy gradient methods. http://www.scholarpedia.org/article/Policy_gradient_methods
- [14] Schulman, J., Levine, S., Abbeel, P., Jordan, M., & Moritz, P. (2015, June). Trust region policy optimization. In *International conference on machine learning* (pp. 1889-1897). PMLR.
- [15] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.
- [16] Foerster, J., Farquhar, G., Afouras, T., Nardelli, N., & Whiteson, S. (2018, April). Counterfactual multi-agent policy gradients. In *Proceedings of the AAAI conference on artificial intelligence* (Vol. 32, No. 1).
- [17] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... & Polosukhin, I. (2017). Attention is all you need. *Advances in neural information processing systems*, 30.
- [18] ML-Agents Toolkit repository <https://github.com/Unity-Technologies/ml-agents>
- [19] Soft-Actor Critic algorithm and background <https://spinningup.openai.com/en/latest/algorithms/sac.html>
- [20] Agents documentation https://github.com/Unity-Technologies/ml-agents/blob/release_18_branch/docs/Training-ML-Agents.md
- [21] Class Agent <https://docs.unity3d.com/Packages/com.unity.ml-agents@1.0/api/Unity.MLAgents.Agent.html>
- [22] Class Academy <https://docs.unity3d.com/Packages/com.unity.ml-agents@1.0/api/Unity.MLAgents.Academy.html?q=academy>
- [23] YAML configuration file documentation <https://github.com/Unity-Technologies/ml-agents/blob/main/docs/Training-Configuration-File.md>
- [24] ONNX documentation and examples <https://pytorch.org/docs/stable/onnx.html>
- [25] Using Tensorboard to observe training <https://github.com/Unity-Technologies/ml-agents/blob/main/docs/Using-Tensorboard.md>
- [26] Competitive Self-Play <https://openai.com/blog/competitive-self-play/>
- [27] ML-Agents overview. https://github.com/Unity-Technologies/ml-agents/blob/release_18_branch/docs/ML-Agents-Overview.md#solving-complex-tasks-using-curriculum-learning
- [28] Example environments in Unity https://github.com/Unity-Technologies/ml-agents/blob/release_18_branch/docs/Learning-Environment-Examples.md
- [29] Kurach, K., Raichuk, A., Stańczyk, P., Zajac, M., Bachem, O., Espeholt, L., ... & Gelly, S. (2019). Google research football: A novel reinforcement learning environment. *arXiv preprint arXiv:1907.11180*.
- [30] Google Research Football with Manchester City F.C. <https://www.kaggle.com/c/google-football>
- [31] <https://towardsdatascience.com/ultimate-volleyball-a-3d-volleyball-environment-built-using-unity-ml-agents-c9d3213f3064>
- [32] ML-Agents play Dodgeball. <https://blog.unity.com/technology/ml-agents-plays-dodgeball>
- [33] Sebastian Schuchmann <https://www.youtube.com/c/SebastianSchuchmannAI>
- [34] Immersive limit website <https://www.immersivelimit.com/tutorials/unity-ml-agents-tutorial-list>
- [35] Unity prefabs <https://docs.unity3d.com/Manual/Prefabs.html>
- [36] MonoBehaviour class <https://docs.unity3d.com/ScriptReference/MonoBehaviour.html>
- [37] Unity forum question about triggers <https://forum.unity.com/threads/ontriggerenter-ontriggerstay-ontriggerexit.529328/>
- [38] Game environments open repository <https://github.com/ilChapo/Futsal-MLAgents/tree/main2>
- [39] Raycast observations <https://github.com/Unity-Technologies/ml-agents/blob/main/docs/Learning-Environment-Design-Agents.md#raycast-observations>
- [40] Ray sensors code implementation https://github.com/Unity-Technologies/ml-agents/blob/release_18_branch/com.unity.ml-agents/Runtime/Sensors/RayPerceptionSensor.cs
- [41] Tips and tricks for rewards <https://medium.com/@BonsaiAI/deep-reinforcement-learning-models-tips-tricks-for-writing-reward-functions-a84fe525e8e0>
- [42] Behavior parameters element <https://docs.unity3d.com/Packages/com.unity.ml-agents@1.0/api/Unity.MLAgents.Policies.BehaviorParameters.html>
- [43] Decision requester element <https://docs.unity3d.com/Packages/com.unity.ml-agents@1.0/api/Unity.MLAgents.DecisionRequester.html>

- [44] SimpleMultiAgentGroups in Unity <https://docs.unity3d.com/Packages/com.unity.ml-agents@2.0/api/Unity.MLAgents.SimpleMultiAgentGroup.html>
- [45] Design Agents <https://github.com/Unity-Technologies/ml-agents/blob/main/docs/Learning-Environment-Design-Agents.md#rewards>
- [46] Coverage in soccer (Spanish) <https://entrenadorfutbol.es/la-cobertura-en-el-futbol-definicion-y-ejercicios/>
- [46] Permutation in soccer (Spanish) <https://entrenadorfutbol.es/la-permuta-en-el-futbol-definicion-y-ejercicios/>

ANNEXES

1. YAML examples of RL techniques add-ons

<pre>environment_parameters: my_environment_parameter: curriculum: - name: MyFirstLesson # The '-' is important as this is a list completion_criteria: measure: reward behavior: Striker signal_smoothing: true min_lesson_length: 200 threshold: 20 value: 0.1 - name: MySecondLesson # This is the start of the second Lesson completion_criteria: measure: reward behavior: Striker signal_smoothing: true min_lesson_length: 400 threshold: 70 require_reset: true value: sampler_type: uniform sampler_parameters: min_value: 0.2 max_value: 0.8 - name: MyLastLesson value: 1.0</pre>	<pre>environment_parameters: reward_shaper: curriculum: - name: Lesson0 completion_criteria: measure: progress behavior: Striker signal_smoothing: true min_lesson_length: 1 threshold: 0.1 value: 0.005 - name: Lesson1 completion_criteria: measure: progress behavior: Striker signal_smoothing: true min_lesson_length: 1 threshold: 0.4 value: 0.003 - name: Lesson2 completion_criteria: measure: progress behavior: Striker signal_smoothing: true min_lesson_length: 1 threshold: 0.7 value: 0.001 - name: Lesson3 value: 0.000</pre>
---	---

```
self_play:
  save_steps: 25000
  team_change: 100000
  swap_steps: 2000
  window: 10
  play_against_latest_model_ratio: 0.5
  initial_elo: 1200.0
```

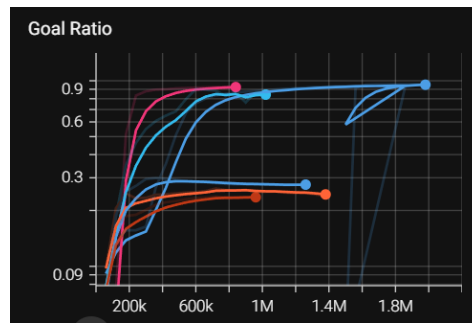
Examples of environment parameter randomization (left), curriculum learning with environment parameter randomization (middle) and self-play add-on (right).

The first case is taken from our YAML in a 1vs0 training, where we had a variable called “my_environment_parameter” that was in charge of controlling the size of the ball spawn zone. “MyFirstLesson” was the first 20% of steps and the size of the spawn zone was put to 10%. Then in “MySecondLesson” we randomly chose a size between 20% and 80% of the maximum until the 70% of steps were completed. “MyLastLesson” put the size to the maximum until the end of the training period.

In the case of the middle, we can see the same modus operandi with a different internal approach. Here we’re controlling our “pushing forward the ball” reward with the variable “reward_shaper”. There is no randomization, but a progressive decrement of the reward from % threshold to threshold of steps.

In the right we see an example of how to indicate a YAML file that we want to apply self-play to our training process. The attributes meaning are properly explained in 1vs1 self-play section of the report.

2. Goal Ratio graphic



With the creation of the Goal Ratio variable in 1vs0, we could visualize the variable progression over steps in TensorBoard. In the image, the blue line that was trained further represents our best model and reaches 95,1% of goals. The purple model was one that we had to cut because of an environment mistake that invalidated the results.