Reinforcement Learning Volleyball

Sacha Cerf, Virgile Foussereau, Quentin Lao and Victor Baillet

***Abstract*—This project focuses on the application of Reinforcement Learning techniques to the domain of volleyball. Specifically, it aims to improve the existing Unity environment called Ultimate Volleyball that uses the ML Agents library to train agents able to play 1v1 volleyball games. The project has two main goals: (1) enhance the existing environment using reward engineering techniques to speed up the training process and (2) develop and compare several methods for implementing 2v2 volleyball games using Reinforcement Learning. To achieve these goals, the project will explore various ways to improve training in Reinforcement Learning and to make agents cooperate in a complex and dynamic environment.**

I. Introduction

The aim of this article is to present our method and results regarding the training of 1v1 and 2v2 volleyball players using Reinforcement Learning. More specifically, we started from an existing Unity environment called *Ultimate Volleyball* [1] originally designed for 1v1. This environment uses the Unity ML Agents Toolkit, which is a Python library to train RL agents in the Unity engine [2]. Using reward engineering, we sped up the 1v1 training process, and then tried various methods for the training of 2v2 volleyball agents. For the 1v1 set-up we aim at maximizing the number of passes over the net while for the 2v2 set-up we want to train competitive teams which try to win the game. To evaluate our performances in 2v2, we developed a hard-coded baseline.

The aspects of RL that we are specifically looking at in this project are : (1) reward engineering and (2) cooperation, with application in the domain of team sports. This is interesting and relevant to RL as team sports pose unique challenges such as coordination, communication, and cooperation between agents. Reward engineering is also a crucial aspect of Reinforcement Learning as it plays a vital role in shaping the behavior of RL agents. In RL, the agent's objective is to maximize the expected cumulative reward over time, and the reward function is the mechanism through which the agent receives feedback on its actions. Therefore, the reward function is critical as it determines the agent's behavior and influences its learning process.

Previous work in the field of RL for team sports has focused mainly on soccer or basketball [3], with limited studies in volleyball. Volleyball is a difficult case for RL as not letting the ball fall is a complex action that requires the agent to anticipate and act fast. Relevant research has been conducted for 2D volleyball, in the *Slime Volleyball* environment [4] [5]. In this project we use a 3D environment named *Ultimate Volleyball*. In [1], the author was able to train an agent to play in a one versus one situation, with the goal to obtain the longest rally possible (i.e maximizing the number of passes).

In our work, we vastly improve these results, both in resulting episode length (length of a rally) but also in training speed. We also step up from this initial environment by adding a second player to each team, in order to study how RL can manage cooperation in the complex case of volleyball. To make the cooperation valuable, we will no longer aim at getting long rally but rather winning the game.

The specific limitations of our study include the limited scope of the Ultimate Volleyball environment. Also, to overcome reward sparsity in 2v2, we made the choice to define separate models for the two players and give them rewards that are slightly tainted with our knowledge of the game, which orients their strategy. Further studies could probably improve the performance of our 2v2 team with more complex models allowing a from scratch approach.

II. Background

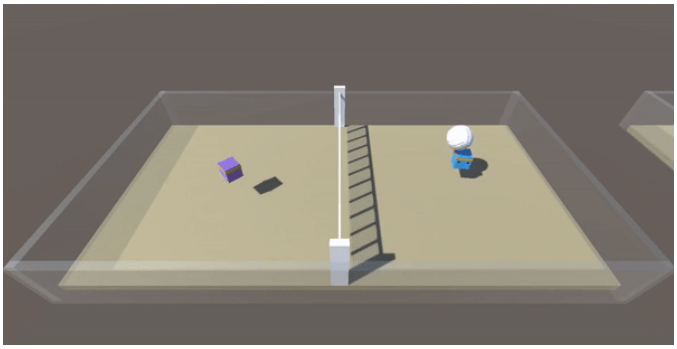

Fig. 1. Illustration of the volleyball environment

The *Ultimate Volleyball* environment has been developed using Unity ML-Agents platform. Two players, a blue one and a purple one, play volleyball over a net. The agent is a 1 meter sided cube, and the terrain is a rectangle that is 15 meters large and 30 meters long. The ball is a 0.15 meter radius sphere.

The action space consists of 4 discrete action branches:

- Forward motion (3 possible actions: no action, forward, backward)
- Rotation (3 possible actions: no action, rotate left, rotate right)
- Side motion (3 possible actions: no action, left, right)
- Jump (2 possible actions: no action, jump)

These actions allow a player to move freely across the field. The agents look like cubes, but have sphere colliders to help them control the ball trajectory. Collisions are handled by Unity physics.

The observation space consists of 11 floating point values:

- Agent Y-rotation (1 float)
- Normalised directional vector from agent to ball (3 floats)
- Distance from agent to ball (1 float)
- Agent X, Y, Z velocity (3 floats)
- Ball X, Y, Z velocity (3 floats)

In the next parts, especially for the 2v2 study, we will change some of the actions and observations from this initial set-up.

The base reward function from [1] gives a +1 reward each time the agent hits the ball over the net. Using this reward function, the author reaches satisfying results after 20 millions steps ($\sim$ 7 hours of training), with a resulting mean reward between 2 and 3 (which corresponds to 2-3 exchanges above the net).

Regarding the algorithm used for training, we will use Proximal Policy Optimization (PPO). It was originally developed by OpenAI [6]. PPO is an actor-critic algorithm that updates policy in *proximity* to the previous by clipping its objective function with a maximum difference of ϵ [7]. In our configuration, we choose $\epsilon = 0.15$, a little bit lower than the default value of ML-Agents. Indeed, big policy updates are risky while small policy updates are more likely to converge. As our environment is quite complex and can have difficulties to converge, a lower ϵ is necessary. For the neural network in the the PPO algorithm we choose empirically to use two hidden layers of 256 neurons and a learning rate of 0.0006. Other choices for the tuning of the PPO algorithm are summarized in the the configuration provided in appendix V.

III. Methodology/Approach

A. Improve 1v1 training

Before changing the environment to a 2v2 set-up and introducing cooperation, our first goal is to improve the training in the simpler 1v1 case by doing reward engineering. Effective reward engineering can significantly improve the training speed and stability of RL algorithms [8]. The main challenge is to introduce as many rewards as possible to increase the learning opportunities of the agent (preventing *reward sparsity*) without denaturing too much the resulting optimal strategy. On one hand, a well designed reward function can guide the agent to learn the desired behavior faster and to avoid undesired ones. On the other hand, a badly designed reward function might make the optimization problem even more difficult, creating more local optima, or have an "exploit" that draws away the attention of the agent towards a simple reward pattern that does not help him winning the actual game. Moreover, a reward function that is too oriented and tainted with presupposed knowledge of the game might guide the agent to an effectively suboptimal strategy, preventing him from finding original ones that may be more efficient.

In our case, we believe that the base reward function is too sparse for an efficient learning. The set of actions needed to obtain the +1 reward is very large and complex : computing where the ball is going to fall, going under it, jumping to push the ball at the right time, making sure to push it in the right direction over the net. Our first idea is to introduce a negative reward N_f whenever the agent let the ball fall on the ground. This negative reward is not constant but depends on the distance D_f between the agent and the ball when it touched the ground. We use this formula:

$$N_f(D_f) = -1 + \exp\left(-\frac{D_f}{2}\right) \tag{1}$$

This function gives a negative reward close to -1 when the agent is far from the impact point of the ball, and gets closer to 0 when the agent is closer to the ball impact point. This gradation allows the agent to quickly understands that he needs to go at the expected impact point of the ball. Note how this behaviour is inevitable to reach an optimal policy. Thus, this reward should not change the resulting optima, simply make the training faster. A graph of the function is provided in figure 2.

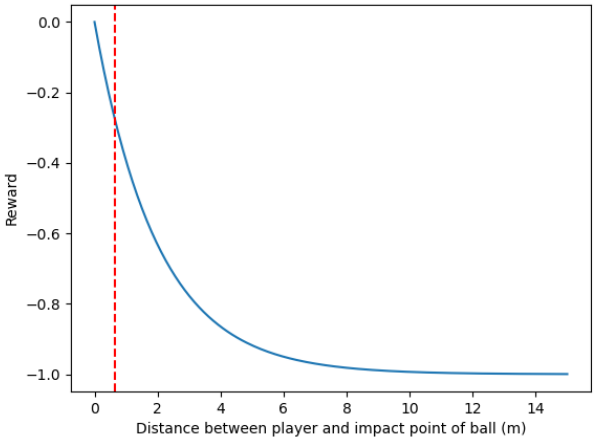

Fig. 2. Negative reward for letting the ball fall, depending on the distance from agent to fall point

In figure 2 we plotted a vertical line for $D_f^R = 0.65m$. This corresponds to the player radius (0.5) plus the ball radius (0.15). As these two objects are both not deformable objects, we always have $D_f > 0.65m$ which means that the negative reward for letting the ball fall is always more negative than $N_f^R = -0.3$ more or less. This fact will be useful for our second idea of reward engineering.

By jumping under the ball before it falls on the ground, the agent will make the ball bounce in the agent-ball direction, but at the beginning the agent does not know how to position himself, so the agent-ball direction will typically head out of the terrain. If the ball is pushed hard enough (by jumping before impact for example) to go out of the player court, it will totally prevent the negative reward N_f. The positive +1 reward for sending the ball over the net will teach the agent to bounce the ball in the correct direction. However this may take some time to be learned. Also, in the initial version hitting the ball out of bounds will end the episode but will not add a negative reward to the agent. Consequently, agents trained this way have a tendency to send the ball out of bounds. The author of [1] states that she tried to add a -1 negative reward

for sending the ball out of bounds, but it led to "shy" agents who prefer letting the ball fall than playing it and risking to send it out of bounds. This was true in the author set-up with not negative reward for letting the ball fall, but it is not our case. By setting a negative reward for sending the ball out of bounds N_{out} between 0 and N_f^R we can avoid this shyness issue while still penalizing hitting the ball out of bounds : the reward will always be bigger for a ball hit. To make the agent learn faster we decide to make N_{out} dependant on the angle θ_{out} between the net direction and the ball direction when it goes out of bounds. The formula is given in (2)

$$N_{out}(\theta_{out}) = \frac{N_f^R}{2} * (\cos\theta_{out} - 1) \tag{2}$$

For instance, if the player hits the ball out of bounds in the complete opposite direction of the net, the negative reward will be very harsh. The closer the agent is to make a correct hit (towards the net), the smaller the penalty will be. The important fact is that hitting the ball out of bounds will never leads to a harsher negative reward than letting the ball fall. A plot of the function is provided in figure 3.

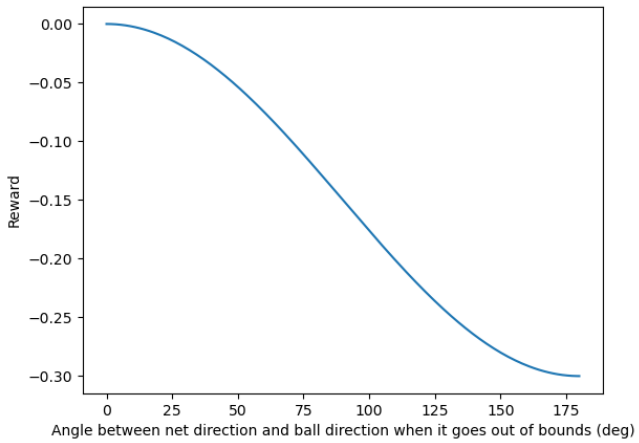

Fig. 3. Negative reward for hitting the ball out of bounds, depending on the direction

B. Explore 2v2 confrontation

After improving the 1v1 training we decided to study how we could implement a 2v2 environment and train agents on the same team to collaborate. Simply adding a new player to each team proved not to be sufficient, even when adding observations concerning the position of the teammate. It would typically end in the two players wanting to push the ball to the other team court, with little to no cooperation. To make cooperation interesting for agents we decided to add two key components : a competitive aspect and a touch action. Instead of just sending the ball back and forth over the net, the overall goal of a team is now to score a point (making the other team to fail). The new spike touch is a discrete branch with 2 values (no action, touch). The touch simulates the action of the player's arms.

- If the player is on the ground, it is a set. The ball goes in the ball-player direction at speed of 20 m/s.
- If the player is in the air (after a jump) it is a spike. The ball goes at right angles to the ball-player direction, towards the opponent's court (this mimics the action of an arm that would go in circular motion), at a speed of 30 m/s. A spike is much easier to make after a set since the timing of the touch needs to be frame perfect when the ball goes at a high speed towards the hitter. Hence the need for collaboration.

As a spike is much more powerful than a regular touch, being able to spike in the opponent court gives a competitive advantage.

This action is however quite complex. To avoid too much complexity, we remove the rotation action from the initial environment. This does not reduce the player ability to move across the field, as translation is still possible in four directions. This compensate a part of the new action complexity but could be considered a bit less realistic. However it not completely unrealistic either because real volleyball players often try to be always facing the net, using a lot of side or backward motion. Note that these new "motion" controls are bang-bang second order controls, that is that they set the forward and side acceleration to $+a$ or $-a$, where $a = 3m/s^2$. The forward and side acceleration are added, and the sum is normalized so that a diagonal movement is not faster.

We also change the observation space. First, we add the directional vector and distance to team mate. Second, we also add an int between 0 and 2 which describes the last player to touch the ball : 2 if it is an opponent, 1 if it is the player's team mate and 0 if it is the player itself. Finally we remove the agent current rotation as it is now constant (always facing the net).

To summarize these changes, here is a the new environment description :

The action space consists of 4 discrete action branches:

- Forward motion (3 possible actions: no action, forward, backward)
- Side motion (3 possible actions: no action, left, right)
- Jump (2 possible actions: no action, jump)
- Touch (2 possible actions: no action, touch)

The observation space consists of 15 elements:

- Normalised directional vector from agent to ball (3)
- Distance from agent to ball (1)
- Normalised directional vector from agent to team mate (3)
- Distance from agent to team mate (1)
- Ball X, Y, Z velocity (3)
- Player X, Y, Z velocity (3)
- Last player to touch the ball (1)

Several ways to train the two players to cooperate could be used. We distinguish three main approaches :

- (A) Single network controlling both players
- (B) Use an identical network for each player
- (C) Use different networks for each player

The approach (A) is actually a single agent controlling both players. This may lead to the best strategies but needs a complex model and is highly unrealistic in our case. The approach (B) is to train an identical network for each of the two players. It means that, in an identical situation, both agents would do the exact same action. In theory, good results are possible with this approach. As the player knows his team mate position, we can imagine a trained agent who has learned to let the player closest to the ball defend and set, while the other goes for the spike. However, this is highly complex to learn. Indeed, not only the agent has to learn very different behaviours (defend & set or spike) but he also needs to knows which behavior adopt depending on the situation and his team mate actions. The approach (C) makes things easier by allowing to specialize each agent. In this approach, an agent only need to learn one behavior. In our case we separated the tasks in two different roles : a setter and a hitter. The setter's goal is to defend the opponent ball and set it for his team mate, while the hitter's goal is to spike the ball in the opponent court. This is quite realistic as in reality, volleyball players often have a specialized role.

For our team with specialized roles, we adapt our reward function depending on the role. For the setter we have :

- $N_f(D_f)$ for letting the ball fall if the last touch was the opponent team
- $N_{out}(\theta_{set})$ for hitting the ball out of bounds, with θ_{set} being the angle between the ball direction and the team mate direction (instead of the net direction in 1v1 case)
- -1 for jumping
- +0.5 for a successful set to the hitter, leading to a spike
- +1 for a successful set to the hitter, leading to a successful spike over the net

For the hitter we have :

- $N_f(D_f)$ for letting the ball fall if the last touch was his setter team mate
- $N_{out}(\theta_{out})$ for hitting the ball out of bounds or spiking the ball in his own court, with θ_{out} being the angle between the ball direction and the net direction (as in 1v1 case)
- -0.1 for jumping if the last touch is a set from his team mate, -0.2 otherwise
- +0.3 for a spike after a set from his team mate
- +1 for a successful spike over the net after a set from his team mate

This set of positive and negative rewards aims to teach the strategy "set & spike" to the players. The small penalty attributed to the hitter for jumping is an idea from our reviewers. Indeed, several of them were disturbed by our first iteration where the hitter was basically constantly jumping around to spike the ball. In reality this is not possible, as jumping cost a lot of energy. The penalty we added represents this energy cost. However, it was quite hard to find the exact penalty value to use. A penalty too harsh leads to a hitter never wanting to jumping while a penalty too soft has no effect on the hitter behavior. The values we used in our implementation allowed us to obtain a hitter jumping only when necessary.

C. Model evaluation against a baseline in the 2v2 setting

The question of how to evaluate our 2v2 model is very important. One could think of just measuring the episode length to see how long the agents can make a rally last, but that is not an objective metric of the performance of the agent. Indeed, a mediocre agent can have a long rally against another mediocre agent as long as it knows how to hit the ball over the net. The issue is that here, the adversary is the model itself. In the 1v1 setting, this was not a problem because the margin of progress after having learned how to hit the ball over the net is not very large. However in the 2v2 setting, many more factors influence the performance of the agent : knowing how to set correctly and spike at the right moment, place the spike correctly on the other court, preferably with a critical trajectory (almost vertically, or in the corners)... Measuring the mean reward is not a satisfying metric either, because we want to observe the effect of adding and removing some rewards on the result. For this reason, we decided to use a metric based on the model's performance against a hard-coded baseline. Our metric for a given model is the probability that it wins a point against the baseline (see Fig 8), determined empirically on a high number of points. A part of our team was dedicated to its implementation. Here, we describe its strategy and computations in detail, as well as the difficulties met in its development, to highlight how complex the challenges the RL agent has to overcome are. We even had to do some simplifications on the physics of the environment along the way to implement the baseline in a reasonable amount of time.

1) Move target prediction and agent control: Here, we explain how the agents decide where to move to aim at a given target, and how they move to a given point. The aiming of the agents is *theoretically* perfect in the sense that it can compute the exact position where it needs to be to aim at a particular target. Let us describe these computations. We denote by $(\vec{u_x}, \vec{u_y}, \vec{u_z})$ the world's coordinate system. First, suppose that we know the position H of the ball when it will be hit. We introduce :

- T, the target point.
- $\vec{u_x}' = (\vec{u_y} \times \frac{\overrightarrow{HT}}{||\overrightarrow{HT}||}) \times \vec{u_y}$, the vector that is orthogonal to u_y in the plane spanned by $\overrightarrow{HT}$ and $\vec{u_y}$ (which will be the plane of the trajectory).
- (x'_T, y_T), the coordinates of T in the coordinate system $(H, \vec{u_x}', \vec{u_y})$.
- v_0, the ball touch velocity (here, 20 m/s)
- $g = 9.81 m/s$, the gravitational acceleration
- $\theta \in [0, \frac{\pi}{2}]$, the unknown, which is the angle of propulsion.

Then, ignoring air resistance, we get the following equation:

$$\tan\theta = \frac{1}{x_T} \times \left(y_T + \frac{g x_T^2}{2 v_0^2 \cos^2\theta} \right)$$

It is important to note that when v_0 is high enough (compared to x_T and y_T), this equation has two solutions : one in $[0, \frac{\pi}{4}]$ (the "straight" solution), and one in $[\frac{\pi}{4}, \frac{\pi}{2}]$ (the "bell" solution). When it is the case, we prefer the "bell" solution, because it will give more time for the hitter to position itself. For this reason, we first try a dichotomy in $[\frac{\pi}{4}, \frac{\pi}{2}]$, then apply Newton's method if no solution is found. From θ, one can find the correct position for the agent by projecting the position of the ball along the vector $\cos\theta\vec{u_x}' + \sin\theta\vec{u_y}$ (which is the set direction) on the plane $(y = 0.5)$ (which is the plane of the agent when it is not in the air), which gives the point P_H (depending on H). Now, one has to find H, the position of the ball when it will be hit. Many solutions are possible, but there is one constraint : H must be in the range of the agent from P_H. Indeed, P_H is just the projection of H on a plane following the set direction, there is no guarantee that this point is close enough to H (think about when the set direction is very horizontal). However, if we take an H that is low enough, we can guarantee that P_H will be close, but if H is too low then the ball speed will be high at H, and the ball will travel a non negligible distance in one frame, which drastically undermines precision from our observations. The previous reasoning led us to the following protocol to find a convenient H.

- For y in $[0, r]$, where r is the agent's range, let H_y be the unique point of the ball trajectory that has ordinate y and negative y ball speed.
- Find $y_0 = \max\{y \in [0, r], ||\overrightarrow{H_y P_{H_y}}|| \leq r\}$.
- The hit point is H_{y_0}.

We do not give the formula for H_y, because it is simply parabolic trajectory prediction. Finding y_0 can be done with a dichotomy for example. Using this method, we get a 100% hit rate when the target is a 0.2 meter radius ball (we did not try with a smaller target), and the agent is placed perfectly on $P_{H_{y_0}}$. However, in practice, reaching $P_{H_{y_0}}$ at a sufficient precision is very hard. As an illustration of this, when we first tried, as a simplification, to switch from second to first order bang-bang control (that is, control the speed rather than the acceleration), we could not reach $P_{H_{y_0}}$ precisely enough, because the distance that the agent travelled in a single time step was too high (for a speed that allowed the agent to cover the court in a reasonable amount of time). Thus, second order control is mandatory to get sufficient precision and high average speed. In 1D, the optimal solution to this control problem would be to accelerate towards the target, and to start decelerating when the distance to the target point is exactly the breaking distance, which can be easily calculated. However here, the things are more complicated due to the fact that the deceleration on x depends on the deceleration on z because of normalization. For this reason, we decided to increase the agent's rigidbody's drag and to rely on it for breaking. To reach a given target point, the algorithm calculates at each frame its stopping point in free motion (that is, if no control is applied),

$$\vec{S} = \frac{m}{\lambda}\vec{v} + \vec{P},$$

where m is the player's mass, λ the drag, $\vec{v}$ its current velocity, $\vec{P}$ its current position, and T its target point. Then, it adapts its acceleration on the x and z coordinates depending on the position of S and P relative to T. For instance, if $P_x > T_x$ and $S_x > T_x$, it will accelerate negatively on the x-axis. We realize that this control is not optimal and needs further improvement.

2) Strategy of the team: Here, we explain the strategy of the team, which uses the prediction and control tools described in the previous section. The idea is to realise a simple set-spike scheme. First, the setter predicts if the ball will land on its court. If it is the case, it uses the aiming and control module to place itself in order to set the ball towards the attack spot, which is in front of the center of the net. In the meantime, the hitter uses the control module to place itself just before the attack point. It then jumps exactly 0.2s (the time it takes to reach its spike when jumping) before the ball reaches its spike height. When the ball is in its range, it predicts in each frame the trajectory of a potential spike, and waits until the critical moment when the ball will land on the back boundary line. If the hitter could not reach the attack spot in time to jump and hit, it will simply touch the ball from the ground and send it towards the opponent court.

IV. Results and Discussion

A. 1v1 results

In 1v1, our modifications to the environment leads to a much faster training, especially in the early part. First, our engineered reward function allows us to reach an episode length of 20 steps (2s) two times faster than the original model (4M steps againt 8M steps). Using also our other modifications in the PPO configuration, observations and actions we can speed up the early part of the training by a factor of 8. This result is visible on 4.

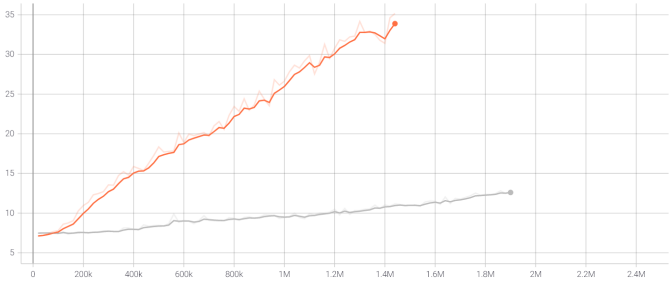

Fig. 4. Episode Length as function of number of steps for original model (gray) vs ours (orange) during early phase of training

These changes also allow us to reach an episode length of 200 steps (20s) in 20M training steps against 150 steps (15s) in 60M training steps for the original model as seen in figure 5.

B. 2v2 results

1) 2v2 training: In 2v2, we tried several approaches. Our first attempt was using approach (B) as described in subsection III-B, which is an identical network duplicated for each agent. In practice, we did not manage to make the agent differentiate his behavior depending on his team mate using approach (B). We believe that this is because it is very complex for an agent to learn completely different behaviors and switch between them. Therefore, our result using this approach was

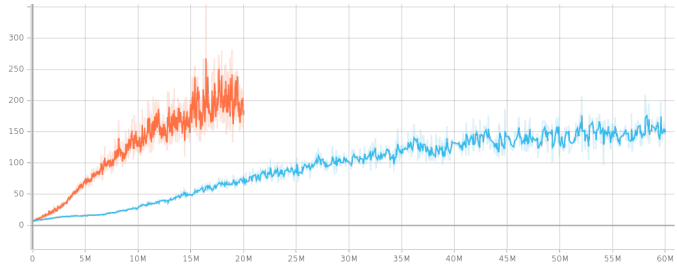

Fig. 5. Episode Length as function of number of steps for original model (blue) vs ours (orange)

no very satisfying with two agents always going for the ball to push it back to the opponent court without any cooperation.

We decided to focus on approach (C) : separate and specialized networks for each agent. Our first attempt was to replicate exactly a real volleyball strategy: player 1 defends the ball after the opponent attack, player 2 sets it and player 1 spikes. However, this attempt was not successful and the agents were never able to learn this full sequence. The player 1 role was probably the cause of this failure: depending on the situation, he either needs to make a defensive touch towards its setter or he has to spike towards the opponent court. Doing these two very different set of actions was too hard to learn for our models. Therefore, we decided to simplify the strategy by ”merging” the defense and the set part: player 1 defends the ball and player 2 immediately spikes it. This is the set-up described at the end of III-B.

The result of this final approach are more satisfying. The sequence of set and spike is successfully learned. We even see very interesting and unexpected behavior such as the spike strategy. Our trained agents converge towards a strategy called ”quick spike” in volleyball. Basically, the hitter waits very close to his setter and spike the ball directly after the set (in the ascending phase). In a more ”classic” strategy, that we use in our baseline, the hitter waits for a set at a specific point and spike in the descending phase of the set. On the reward curve in figure 6 we can see that the setter is the first to learn that he needs to make a set, while the hitter reward is decreasing because he has not yet learn that he needs to attack the set given by his setter. After this initial phase, both reward functions increase together as the agents learn to collaborate successfully.

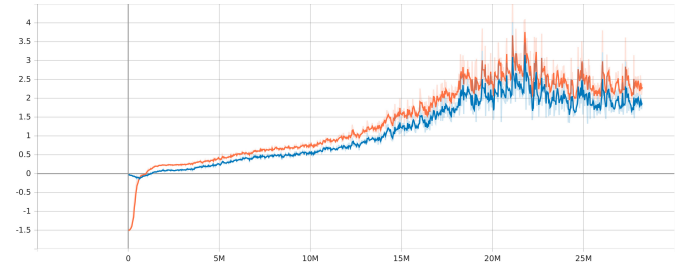

Fig. 6. Reward as function of number of steps for Hitter (dark blue) and Setter (orange)

The training in figure 6 was done before receiving the reviews on our initial work, thus before implementing the jumping penalty for the hitter. We added it to our model and started a new training. However, instead of starting from scratches, we used the previous trained agents as the initial model. The results are presented in figure 7.

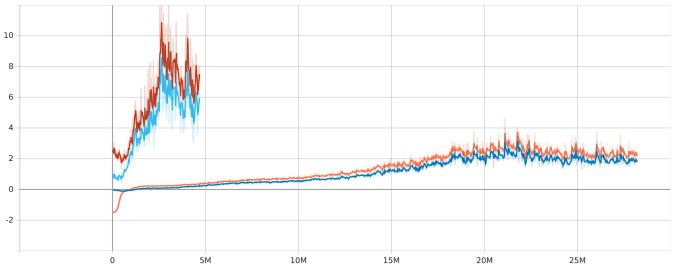

Fig. 7. Reward as function of number of steps for the Hitter with jump penalty (light blue) and without (dark blue) and the Setter with jump penalty (brown) and without (orange)

We can see that, at first, the mean reward of the Hitter has dropped in comparison with the end of the curve where he had no penalty for jumping. This is logical as he has not learned yet to no jump too much. However, as he learn it, the reward keep going up and we end up with agents that are a lot better than the previous ones. We were really surprised by these results. The jump penalty was added to prevent the hitter for jumping tirelessly as it was unrealistic and unpleasant to the eyes, but we did not expect an improvement in performance. With hindsight, our interpretation is that jumping constantly caused the Hitter to sometimes miss the ball due to bad timing. By penalizing the jump, we force the Hitter to jump only at the right timing. Therefore, it increases his chances to do a good spike.

2) 2v2 competition against a baseline: To evaluate the performance of our trained agents, we oppose them to our baseline team described in subsection III-C. We start a match between the team and the baseline and count the number of points won by each team, as in a real volleyball game. We let the match run for more or less a thousand point to verify that the ratio of the scores converges to a constant. We test 3 match-up: a dummy team with two agents only returning the ball as a set over the net, our RL team without jump penalty and finally our complete RL team (with jump penalty). The results are provided in table I.

TABLE I
PROBABILITY OF WINNING A POINT AGAINST THE BASELINE

	Probability of winning a point against the baseline
Dummy team	38 %
RL team without jump penalty	60 %
RL team	70 %

As we can see, the complete RL team we developed outperforms the hard-coded baseline by a very large margin. Indeed, a few more % in the probability of winning a point leads to a very high probability of winning a game. We note p

the probability of winning a point and we play a game where the goal is to be the first to score n points. Given p, what is the probability P_n to win the game ? The exact formula can be determined and is provided in equation 3.

$$P_n = p^n \sum_{k=0}^{n-1} \binom{n+k-1}{k} (1-p)^k \tag{3}$$

We plot this formula for n=25 (a volleyball set is 25 winning points) in figure 8. As we can see, our RL teams have a very high chance of winning. With its probability $p = 0.7$ of winning a point, our complete RL team reaches a probability $P_{25} = 0.998$ of winning a match set against the hard-coded baseline!

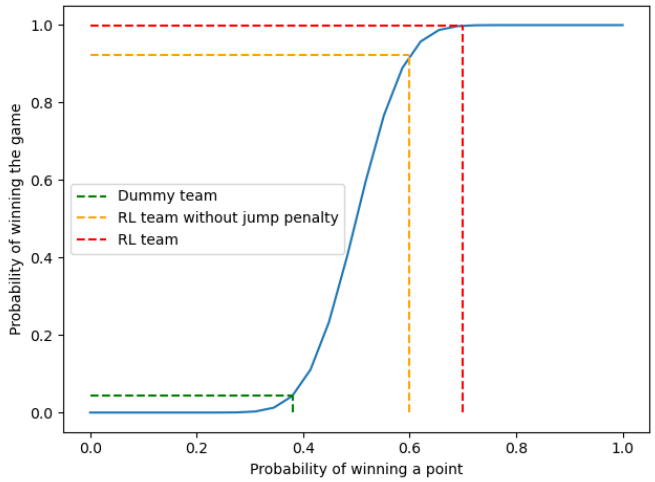

Fig. 8. Probability of winning the game as a function of the probability of winning a point

V. Conclusions

Using reward engineering we have been able to greatly improve the training of agents in a 3D volleyball environment. Final performance is increased and the training is two times quicker if we only change the rewards, 4 times quicker using our custom PPO configuration and 6 times quicker by changing observations and actions of the agent in the environment. In this paper, we also explored different ways to train agents to collaborate in two versus two competitive games. We found that, in our case, the best option was to train each player of the team for a specialized role. Our final trained model is able to consistently beat a hard-coded baseline with 70% chance of winning a point, which means a 99.8% chance of winning a 25 points set. Further developments of our work could try to develop more complex strategies, for example in a 4v4 set-up.

References

[1] Joy Zhang. *Ultimate Volleyball: A multi-agent reinforcement learning environment built using Unity ML-Agents*. 2021. URL: https://www.gocoder.one/blog/3d-volleyball-environment-with-unity-ml-agents/.

[2] Unity Technologies. *ML Agents Toolkit Release 20*. 2022. URL: https://github.com/Unity-Technologies/ml-agents.

[3] Unity ML-Agents. *Soccer Twos*. 2021. URL: https://deepanshut041.github.io/Reinforcement-Learning/mlagents/05_soccer_twos/.

[4] Jaleh Zand, Jack Parker-Holder, and Stephen J. Roberts. *On-the-fly Strategy Adaptation for ad-hoc Agent Coordination*. 2022. DOI: 10.48550/ARXIV.2203.08015. URL: https://arxiv.org/abs/2203.08015.

[5] New Jun Jie et al. "Bayesian Multi-Agent Reinforcement Learning for Slime Volleyball". In: *17th STePS* (2020).

[6] John Schulman et al. *Proximal Policy Optimization Algorithms*. 2017. DOI: 10.48550/ARXIV.1707.06347. URL: https://arxiv.org/abs/1707.06347.

[7] Jesse Read. "Lecture VI - Reinforcement Learning III." In: *INF581 Advanced Machine Learning and Autonomous Agents*. 2023.

[8] Daniel Dewey. "Reinforcement learning and the reward engineering principle". In: *2014 AAAI Spring Symposium Series*. 2014.

APPENDIX

Appendix A

```yaml
    trainer_type: ppo
    hyperparameters:
      batch_size: 2048
      buffer_size: 20480
      learning_rate: 0.0006
      beta: 0.003
      epsilon: 0.15
      lambd: 0.93
      num_epoch: 4
      learning_rate_schedule: constant
    network_settings:
      normalize: true
      hidden_units: 256
      num_layers: 2
      vis_encode_type: simple
    reward_signals:
      extrinsic:
        gamma: 0.96
        strength: 1.0
    keep_checkpoints: 5
    max_steps: 20000000
    time_horizon: 1000
    summary_freq: 20000
```

Fig. 9. Training configuration (documentation on this page)

Appendix B

You can access our code here for 2v2 set-up and here for 1v1 set-up. Small animations are provided in the readme of each branch to illustrate the results, in case you do no want to install everything.