UNIVERSITÀ DEGLI STUDI DI SALERNO

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE ED ELETTRICA
E MATEMATICA APPLICATA

Project Work Report - Machine Learning

Tennis Table Tournament

Professors

Pasquale Foggia - pfoggia@unisa.it
Diego Gragnaniello - digragnaniello@unisa.it
Mario Vento - mvento@unisa.it

Team members

Name and Surname	**Student ID**	**E-mail**
Ciaravola Giosuè	0622702177	g.ciaravola3@studenti.unisa.it
Conato Christian	0622702273	c.conato@studenti.unisa.it
Del Gaudio Nunzio	0622702277	n.delgaudio5@studenti.unisa.it
Garofalo Mariachiara	0622702173	m.garofalo38@studenti.unisa.it

ACADEMIC YEAR 2023/2024

Contents

Introduction

In this report, we outline the process used to train a Machine Learning model aimed at controlling a simulated robotic arm in a virtual environment to play Tennis Table. This project was completed as the final assignment for the Machine Learning course in the 2023/2024 academic year. The following sections provide an analysis of the approach taken and the results achieved.

1 Initial approaches

This chapter examines the initial approaches we adopted to tackle the problem, discussing the reasons they were ultimately discarded. We then present the first viable solution we arrived at through this process.

1.1 Disastrous start

A perfect start was never expected. Indeed, the initial approaches were primarily exploratory, aimed at building confidence with a machine learning project, quite different from the individual scripts covered in class. After spending some time deciding how to organize the code and outlining a few initial ideas, we decided to start training the robotic arm using a reinforcement learning network.

Inspired by Professor Gragnaniello's examples, we developed the appropriate code and began training the network using a DDPC (Deep Deterministic Policy Gradient) architecture with two layers of four hundred neurons each, similar to the inverted pendulum balancing example from class. Naively, we fed all the information provided by the system as input to obtain all the positioning values for the arm. The reward was proportional to the ball's distance, whether the ball was hit, and whether the arm scored a point by hitting the ball. As expected, the result was disastrous. After a night of training, we ended up with an arm huddled in a corner for the entire match. Assuming this was due to a bias error, we incrementally added up to two hidden layers, increasing their size: eventually, we had four layers with five hundred neurons each.

The results were never satisfactory. In fact, the more we progressed, the more we realized that networks of such size would require training times that were simply unavailable. We had to simplify the problem

1.2 Disastrous continuum

It quickly became clear that we needed to modify the network's inputs and outputs. We realized that the network should only be given the necessary inputs to produce a limited set of outputs. Even at this early stage, the idea of using multiple networks for specific tasks was beginning to take shape.

Reflecting on how to divide the problem, we concluded that the cart should move algorithmically, tracking the ball on the horizontal axis (using the ball's x-coordinate when it lands) and moving on the vertical axis to maintain a constant distance from the ball (less than the fully extended length of the arm) so that the arm could always reach it.

Consequently, we decided to lock the arm's roll rotation, as every ball position would be accessible without changing the roll (to further simplify the problem). This was achieved using only the pitch of the joints since the ball was followed along the x-axis. The only challenge in reaching the ball was its height and distance along the y-axis, which could be managed by adjusting the arm joints' pitch appropriately. We used a reinforcement learning network to control the arm in this configuration (paddle fixed forward, no roll rotation, and algorithmic cart movement thanks to the trajectory prediction function discussed in the Section 1.3), aiming to minimize the distance between the ball and the paddle. The idea was that once the network was trained to follow the ball, another network would be trained to actually hit the ball.

However, even with this configuration, the arm's behavior remained unsatisfactory, characterized mainly by immobility and frequently ending up in strange, huddled positions, despite varying hyperparameters such as architecture size.

We then identified a possible reason for this behavior: since the reward was based on reducing the distance between the paddle and the ball, the distance naturally decreased as the ball approached the arm, so the arm receives a reward by standing still and waiting for the ball. At this point, we could have modified the reward, but we decided to take a different approach.

1.3 Trajectory function

To enable algorithmic use of the cart, a function was devised to predict the future position of the ball in terms of x and y coordinates when it reaches a specified height z. This predictive model relies on principles of kinematics and uniformly accelerated motion applied to the z-coordinate of the ball's trajectory.

The ball's motion was treated as uniformly accelerated along the z-axis, where acceleration is influenced by gravitational effects (simulated using PyBullet). The equation governing the motion along the z-axis, under the influence of gravity without air resistance, is represented as:

$$z(t) = bz + vz \cdot t - \frac{1}{2} g \cdot t^2 \tag{1.1}$$

where:

- $\mathbf{z(t)}$ is the vertical position of the ball at time;
- **bz** is the initial position of the ball along the z-axis;
- **vz** is the initial velocity of the ball along the z-axis;
- **g** is the gravitational acceleration (9.81);
- **t** is the time elapsed since launch.

The objective is to determine the time t at which the ball reaches the specified target height $target\_z$. This involves solving the quadratic equation derived from setting $z(t) = target\_z$:

$$-\frac{1}{2} g \cdot t^2 + vz \cdot t + (bz - target\_z) = 0 \tag{1.2}$$

If the equation does not have real solutions ($\Delta < 0$), it indicates the ball never reaches that height (possible only if it is struck below that height without rebounding). If the equation does have real solutions, it could yield two times due to the parabolic nature of the motion (one for ascent and one for descent), but we are interested only in the later time corresponding to descent.

Once the time is determined, it is used to solve uniformly accelerated motion equations for the other two axes (x and y):

$$x = bx + vx \cdot t \qquad\qquad y = by + vy \cdot t \tag{1.3}$$

By doing so, we have obtained a prediction of where the ball will bounce, which is accurate given the absence of air friction in our system. To validate the function's correctness, measurements were conducted by predicting the ball's landing after being struck and comparing the actual coordinates at the time of landing.

```python
def trajectory(state, target_z=0.1):

    bx, by, bz = state[17:20]
    vx, vy, vz = state[20:23]
    g = 9.81  # Acceleration due to gravity

    # Solve the quadratic equation to find the time t when
    # z(t) = target_z
    a = -0.5 * g
    b = vz
    c = bz - target_z

    # Calculate the discriminant
    discriminant = b**2 - 4 * a * c

    if discriminant < 0:
        # The ball never reaches z = target_z
        return None, None

    # Calculate the possible times
    t1 = (-b + math.sqrt(discriminant)) / (2 * a)
    t2 = (-b - math.sqrt(discriminant)) / (2 * a)

    # Take the positive time
    t = max(t1, t2)

    if t < 0:
        # Both times are negative, the ball never reaches
        # z = target_z
        return None, None

    # Calculate the x and y positions where the ball reaches
    # z = target_z
    x = bx + vx * t
    y = by + vy * t

    return x, y
```

Listing 1.1: Trajectory function.

Note

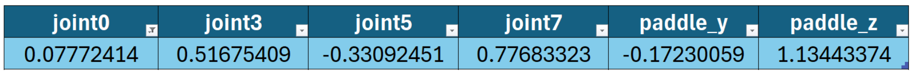
The function described above is located in the project's source files within the *utilities* folder, specifically in the file named *trajectory.py*.

2 Supervised: Arm

Following the failure of the reinforcement learning approach to move the arm in order to track the ball with the paddle, and after repeated advice from the professor, we transitioned to a supervised approach, which proved to be satisfactory. The subsequent sections will delve into the dataset construction and the learning process with the adopted structure.

2.1 Dataset construction

The arm was fixed at a specific x position relative to the cart (since this would be algorithmically adjusted based on the function described in Section 1.3), and a dataset of its various positions was created. To achieve this, random values were generated for the cart's y position (for forward and backward movements) and for the inclinations of joints 3, 5, and 7 (to reach different heights and lean forward or huddle), recording in the dataset the resulting positions and the coordinates (y and z) of the paddle obtained.



joint0	joint3	joint5	joint7	paddle_y	paddle_z
0.07772414	0.51675409	-0.33092451	0.77683323	-0.17230059	1.13443374

Figure 2.1: Dataset sample.

The code was designed to prevent the arm from recording positions that placed the paddle below the table or in positions too far backward, as these were deemed irrelevant for the final use. Additionally, joint movements were restricted: the first joint could not extend completely forward to the ground, limited to approximately 80° at most, and could not move backward beyond -17°; the second joint was restricted to an inclination of 90° both forward and backward; whereas the third joint had no limitations.

After an entire night of recording poses, a dataset comprising approximately 450,000 positions was obtained. From this dataset, a supervised neural network was successfully trained to position the arm precisely at desired coordinates, where the ball's landing was anticipated.

2.2 Network architecture and learning

The architecture used for the supervised network was discovered through trial and error, aiming to find a configuration that achieved low validation loss with minimal resources. Starting with 3 hidden layers of 300 neurons each (each linear with normalization and ReLU activation function) and a final fully connected linear layer, we eventually streamlined it to just 2 hidden layers with 100 and 50 neurons respectively, maintaining the same structure. The model takes 2 inputs (y and z to where to move the paddle), and produces 4 outputs (the position of the carriage y and the angles of joints 3, 5, and 7 to move the paddle to the desired position).

The validation loss during training quickly reached an optimal level and showed little improvement with further epochs, likely due to the dataset's size. With Early Stopping implemented, training halted at epoch 49 when the validation loss had not improved for 20 epochs. Subsequent attempts to adjust this parameter to extend training did not yield significant improvements. Moreover, when applying this model to the arm in the virtual environment, the results aligned perfectly with expectations: the arm moved, positioning the paddle precisely at the required y and z coordinates (within its operational range).

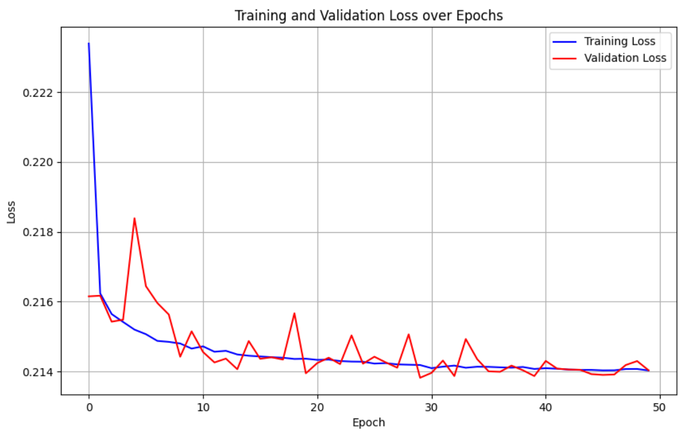


Figure 2.2: Loss trend.

At this stage, the challenge shifted to determining when to activate this network to position the arm and continuing to train the remaining network to manipulate the ball.

Note

The dataset creation and supervised training files are present in the project's source files within the *train_test* folder, specifically in the file named *dataset_builder.py* and *arm_supervised_train.py*.

3 Reinforcement: Paddles

Having ensured a "perfect" arm that positions the paddle as required, the focus shifted to studying the paddle's movement.

Based on our previous experience with reinforcement learning, we decided to break down the problem as much as possible. Initially, we considered four distinct scenarios, but due to the time required to train four networks, we reduced this to just two scenarios and two networks: one network to hit the ball from below and another for executing a smash from above.

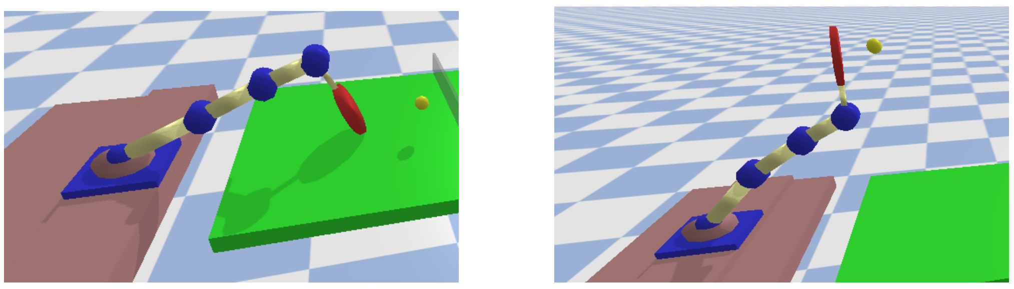


Figure 3.1: Paddle positions.

To simplify the networks' tasks as much as possible, they are activated only when the ball is within a close range of the paddle (distance < 0.3). Their specific role is to decide a single movement of the paddle necessary to hit the ball and propel it into the opponent's court.

3.1 Utilisation distinctions

As soon as the ball starts moving towards the arm, several algorithmic evaluations of its trajectory are made:

- If the landing point is outside the court, the ball is left to go out to score a point (the arm moves to the opposite side);
- If the ball is heading towards the home court and the bounce won't occur in the front part of the court (at $y < 0.2$), a decision is made whether to smash the ball or hit it from below immediately. For a smash, if the ball is in an upward trajectory (positive z-axis velocity), the peak point is calculated using kinematics as done for the trajectory (discussed in the Section 3.2). If this peak exceeds an empirically chosen threshold ($z\_max > 0.75$, below which the ball doesn't rise enough after bouncing to make a good smash), the system prepares for a smash.

Preparing for a smash involves positioning the arm in a backward stance, waiting for the ball to bounce on our side of the court. After the bounce, the new trajectory is calculated by scanning various heights, starting from the peak of the new parabolic arc and descending until a reachable point is found. The arm then positions itself at the chosen spot with the paddle raised, ready for the ball to arrive.

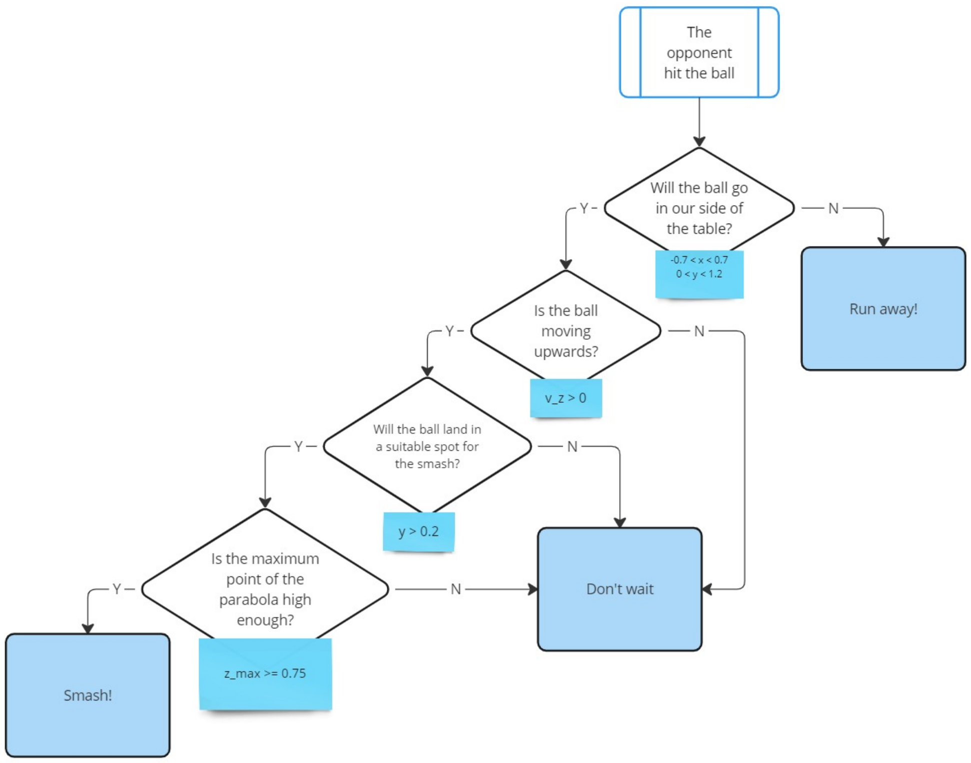

The opponent hit the ball
Will the ball go in our side of the table?
-0.7 < x < 0.7
0 < y < 1.2
Y
N
Run away!
Is the ball moving upwards?
v_z > 0
Y
N
Will the ball land in a suitable spot for the smash?
y > 0.2
Y
N
Is the maximum point of the parabola high enough?
z_max >= 0.75
Y
N
Don't wait
Smash!

Figure 3.2: Position decision tree.

3.2 Max height point function

The function that calculates the maximum point of the parabola of the ball's trajectory is computed similarly to how the landing point was determined, except now we need the time t at which the ball reaches its maximum height.

Since its trajectory is influenced by gravity, which acts downward with a constant acceleration g, the maximum height is reached when the vertical component of the velocity (vz) becomes zero, as this is the point where the projectile starts reversing its vertical movement from upward to downward.

Thus, given the previously discussed equation of motion 1.1, vertical velocity at time t is:

$$vz(t) = vz - g \cdot t \tag{3.1}$$

Setting $vz(t) = 0$, we have:

$$t\_max = \frac{vz}{g} \tag{3.2}$$

Which, as with the trajectory function, is used to calculate the coordinates by substituting it into the equations of motion.

```python

def max_height_point(state):
    bx, by, bz = state[17:20]
    vx, vy, vz = state[20:23]
    g = 9.81  # Acceleration due to gravity

    # Time at which the ball reaches the apex of the trajectory
    t_max = vz / g

    if t_max < 0:
        # If time is negative, the ball is already falling
        return None, None, None

    # Calculate x, y, z positions at the maximum height
    x_max = bx + vx * t_max
    y_max = by + vy * t_max
    z_max = bz + vz * t_max - 0.5 * g * t_max ** 2

    return x_max, y_max, z_max
```

Listing 3.1: Max height point function.

Note

The function described above is located in the project's source files within the *utilities* folder, specifically in the file named *trajectory.py*.

3.3 Reinforcement Learning

Given the difference in activation between the two networks, an ”opponent” was developed to facilitate simultaneous learning by ensuring both scenarios occurred alternately. Specifically, the ”Auto” player (the algorithmic player using kinematics) was modified to behave advantageously:

- When the serve is directed towards the training arm, the opponent completely disengages, moving to a corner and turning away, ensuring the situation calls for the smashing network due to the serve’s trajectory;
- When the serve is directed towards the opponent, the opponent uses the Auto algorithm to position itself and attempt to return the ball. This results in Auto positioning itself to send a very low ball, perfect for activating the network that handles low balls. After returning the serve, the opponent again disengages;

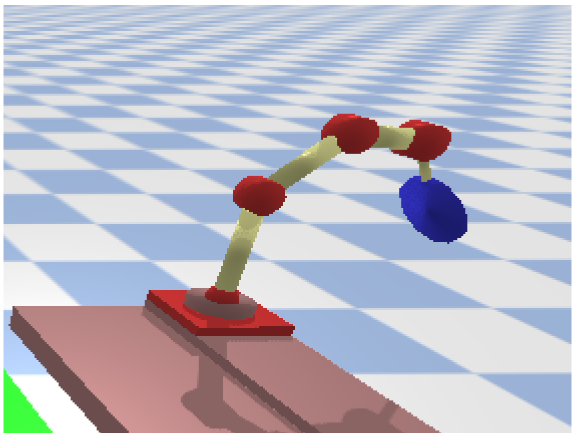


Figure 3.3: Disengaged opponent.

By making the opponent disengage, points scored or lost can be used for the reward system. When the serve is towards the home side, the subsequent point awarded depends solely on how the trained arm handles it. Conversely, when the serve is directed towards the opponent and the opponent misses, an episode is lost, which is not recorded since the ball never reaches the paddle’s activation range. If the opponent successfully returns the ball to our side, the point depends only on the trained arm’s response.

This setup allows for a reliable reward calculation:

- First, it is evaluated whether the ball was hit. If not, the reward is negative (-15);
- If the ball was hit, the landing trajectory in the opponent's court is calculated.
  - If it lands within the court, the reward is scaled positively based on proximity to the side edges (ranging from 15 at the center to 24 at the extreme side edges);
  - If it lands outside the court, a negative reward is given (-10);
- Thanks to the opponent disengaging at the right moments, it is possible to rely on the points, ensuring they depend solely on the actions of the training arm. After considering the previous two points, the score change following the action for which the reward is being calculated is evaluated:
  - If the score for the training arm increases and the reward was negative (for example, in cases where the trajectory was slightly out of bounds and the ball touched the edge, or there were strange net touches behavior), the reward is adjusted to a positive value (20);
  - Conversely, if the opponent's score increases and the reward is different from the missed ball penalty (-15), which is considered more significant and should remain at that value, the reward is set to the same value as an out-of-bounds penalty (-10), in cases where it was mistakenly positive due to strange net touches behavior or other factors.

```python
def calculate_paddle_reward(prev_state, state, point_state):

    # Ball has changed direction in y-axis
    if prev_state[21] * state[21] < 0:
        # Calculate trajectory to reach z = 0
        x, y = trajectory(state, 0)
        if x is not None and y is not None:
            if -0.7 < x < 0.7 and 1.2 < y < 2.4:

                x = abs(x)

                # Calculate reward based on ball's x position
                reward = (x * 20) + 10

                if reward < 15:
                    reward = 15  # Ensure minimum reward

            else:
                # Ball is out of opponent field
                reward = -10
        else:
            reward = 0  # Invalid trajectory calculation
    # If the episode is ended without catch the ball
    else:
        reward = -15

    # Point state is available (e.g., scoring point conditions)
    if point_state is not None:

        # Reward for scoring a point (if the reward is not
        already positive)
        if point_state[34] > prev_state[34] and reward <= 0:
            reward = 20

        # Reward for losing a point (if we don't miss the ball)
        if point_state[35] > prev_state[35] and reward != -15:
            reward = -10

    return reward
```

Listing 3.2: Reward function.

With the opponent and reward function defined, the training proceeded as a classic reinforcement learning process, alternating between recording episodes (with actions affected by noise for the exploration) into the replay buffer and learning via the DDPG.

Both networks aim to select a inclination offset for the paddle (bottom-to-top for the lower paddle, top-to-bottom for the upper one), starting from the empirically chosen position, in a single transition triggered when the ball enters the designated range for the paddle. In addition to the inclination offset, both networks can provide a second output indicating the absolute rotation of the paddle (for any angled shots).They take the ball's speed and position as input.

The action space of both networks is constrained both for the maximum offset and for the possible rotations:

- For the upper paddle, the maximum inclination offset (top-bottom) is about 70°, while the maximum rotation is about 14° on both sides;
- For the lower paddle, the maximum inclination offset (bottom-top) is about 14°, while the maximum rotation is about 9° on both sides.

Note

The training file is *paddle_train*, and the opponent is described in *auto_example*, both located in the project's source files directory *train_test*. The reward function can be found in the *reward_calculator* file in the *utilities* directory.

4 Results achieved

In this chapter, the results obtained will be analyzed in order to select the best networks achieved.

For both networks, multiple attempts were made varying almost all hyperparameters. In general, the same combinations were tested for both networks. Specifically, regarding the network structure, given the complexity of the input to process compared to the network used for the arm, it was decided to use at least one additional layer. Thus, we started with a model where both Actor and Critic had 3 layers of 600 neurons each, and gradually arrived at the final solution, achieving variable behaviors depending on the input, with three layers consisting of 200, 100, and 50 neurons.

4.1 Smash agent

This agent was trained for practically half of the training, as essentially every time the serve was hit towards the arm, it resulted in a smash. For this reason, it was possible to analyze many more models, evaluating their changes after many epochs had passed.

Evaluation was primarily done visually, observing the model's behavior to initially eliminate models that moved substantially incorrectly: consistently smashing too hard without distinguishing the zones of the court, consistently smashing too softly, etc.

During this phase, we realized that when receiving the serve, regardless of all the assessments that lead us to choose the smash, it is always advantageous to do so. Therefore, an additional flag was introduced in the arm's usage code to always choose the smash when receiving the serve.

After selecting a series of models that appeared to behave appropriately by adjusting the force based on the speed and position of the incoming ball, a more objective evaluation method was established: the environment was set up with the serve always directed towards the arm to be tested, with the opponent completely disengaged. This way, the result of the smash on the serve translated into a point lost or gained. It was decided to classify the agents based on the score achieved before making 10 errors (10 points lost).

After evaluating dozens of smash agents, the best one turned out to be based on a structure of 3 layers with 200, 100, and 50 neurons in the hidden layers, achieving a score of 140 upon the occurrence of the eleventh error (an average of 1 error every 15 serves, which is less than 1 error per game).

4.2 Don't wait agent

Learning for this second agent, instead, was a bit more complex. This was due to the nature of the dynamics it was designed for: low trajectory balls. The only way to obtain balls of this type using the tools of the environment is to use Auto as an opponent (the modified one with the disengagement). The latter has some issues:

- Often (about 50% of the time), it immediately makes a mistake by hitting the ball into the net. This prevents the activation of the agent under examination, thus causing slower learning (about half compared to the smash agent);
- When it does manage to hit the ball, due to the fixed position of the paddle, the trajectories produced in terms of ball position and velocity are very similar each time;

For these two reasons, the learning of this agent has been slower and less effective. In all combinations of hyperparameters tested, it has consistently learned to deliver a gentle tap that, when responding to Auto, sends the ball to the back edge of the table, scoring a point.

However, when used against a "normal" Auto on different types of balls, this shot tends to be slightly too strong, causing balls to potentially go beyond the table (although this does not happen because Auto intercepts them).

This problem, like others caused by specific training scenarios, could have been resolved by constructing tailored opponents to generate the situations. Unfortunately, however, the available time has run out.

4.3 Final Consideration

In general, a set of three models was built to position the paddle in all possible situations, using preselected methods, namely **smash** or **don't wait** (for the bounce), and hitting the ball to try to score.

Evaluating the set of 3 models with the same criterion of the maximum score before 10 errors in an infinite game against Auto, a score of about 300 was achieved.

List of Figures