UNIVERSITY OF SALERNO

DEPARTMENT OF INFORMATION ENGINEERING, ELECTRICAL ENGINEERING AND APPLIED MATHEMATICS

Machine Learning

Project Report

Table Tennis 2024

prof. Foggia Pasquale
prof. Gragnaniello Diego
prof. Vento Mario

Group 06

Students	Id number
Alberti Andrea	0622702370
Attianese Carmine	0622702355
Capaldo Vincenzo	0622702347
Esposito Paolo	0622702292

Academic Year 2023 – 2024

Contents

Abstract

This project report outlines the development and training of neural networks to control a robotic arm for playing table tennis. The project employs supervised learning for inverse kinematics and reinforcement learning to optimize the robot's movements.

The supervised learning component focuses on training two distinct neural networks for the high player and low player modes. For the high player, the neural network predicts joint angles for joint[5] and joint[7] based on the z coordinate. For the low player, the network predicts joint angles for joint[3], joint[5], and joint[7] based on the y and z coordinates. The data is normalized, and the networks are trained using the mean squared error loss function and Adam optimizer.

Reinforcement learning is used to further enhance the robot's performance by training an agent with an Actor-Critic model. The Actor determines the optimal actions based on the current state, while the Critic evaluates these actions. The training involves interacting with the environment, receiving rewards, and updating the network parameters to maximize long-term benefits. Hyperparameters are carefully chosen, and a replay buffer is utilized to store and sample transitions for training. The reward function is iteratively refined to improve the agent's decision-making capabilities.

Through this comprehensive and iterative training process, the neural networks and reinforcement learning agent effectively predict joint angles and optimize the robot's movements, ensuring precise control and accurate table tennis gameplay. The results demonstrate the successful application of machine learning techniques to robotic control in a dynamic and competitive environment.

1 Networks architecture

This chapter discusses the architecture of the networks involved in controlling the robotic arm. The player has the following architecture:

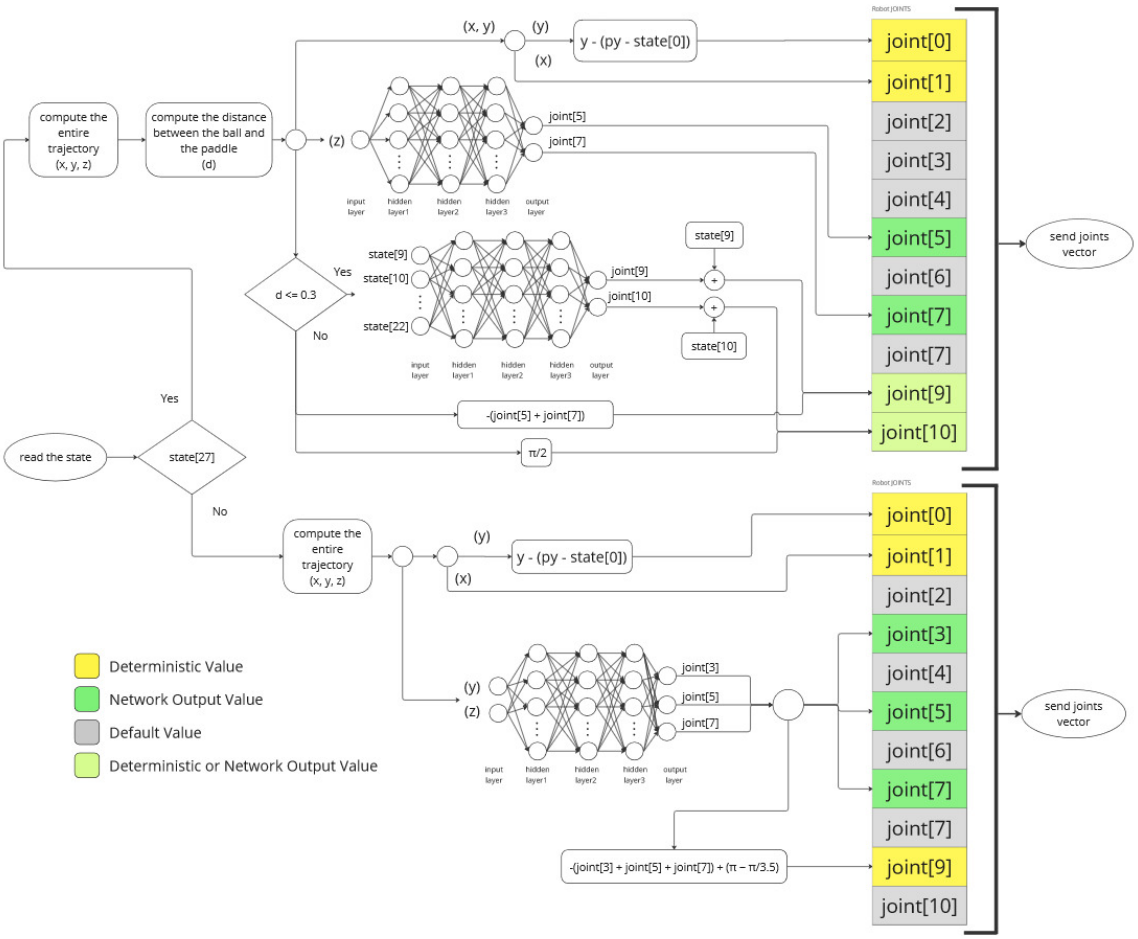

Figure 1: Player's architecture

The image shows what happens at each instant of the game, from reading the state of the environment to sending the action by setting the values of each joint.

The player has two modes: the '*high_player*' mode is activated when the opponent has to make a serve and allows the player to make a smash; the '*low_player*' mode is active for the rest of the game, allowing the player to respond correctly to the opponent's shots.

At each time the state of the environment is read, the player's mode choice is made via a condition, observing the state variable 27 (waiting for the opponent service). In the affirmative branch, '*high_player*' is selected, in the negative case '*low_player*' is selected.

Starting with the first player, the first thing that needs to be done is the calculation of the trajectory, which, observing the x, y, z coordinates of the ball and the x, y, z components of the velocity vector, constructs the entire trajectory by providing the x, y, z coordinates of the response point.

The x-coordinate is used directly as the $joint[1]$ value, allowing the player to easily follow the ball along the x-axis. The $joint[0]$ is processed according to the relation:

$$joint[0] = y - (py - joint[0])$$

where y is the y-coordinate of the ball after the trajectory calculation, py is the y-coordinate of the current paddle position, and $joint[0]$ is the current position along the y-axis of the platform. At this point, the resulting value is used to set the optimal position of $joint[0]$.

The z-coordinate, on the other hand, is used as the network input for the inverse kinematics, obtaining as output the angles of $joint[5]$ and $joint[7]$ to correctly reach, with the paddle center, the z-coordinate of the ball impact point.

After the trajectory calculation, the distance between the actual center of the ball and the actual center of the paddle is calculated. This distance is used to activate or deactivate the net used to move the paddle. If the distance is less than or equal to 0.3 the net is activated, allowing it to control $joint[9]$ and $joint[10]$; conversely, if the condition is not met, the net is not activated and the $joint[9]$ is set according to the relation:

$$joint[9] = -(joint[5] + joint[7])$$

and $joint[10]$ uses a default value. When the network is active it takes as input the state variables 9 to 22 that describe the current state of the paddle and the ball and returns the pitch and roll of the paddle as output.

Joints that are not controlled by the networks are set to a default value. Once all values are obtained, the value of the joints is sent to the environment defining the player's action at each instant.

As for the second player, again the first thing to do is to calculate the trajectory. The x and y coordinates are used, in the same mode, to command the platform. The y and z coordinates, on the other hand, are given as input to the network that deals with the inverse kinematics. Taking y_target and z_target as input, the network outputs $joint[3]$, $joint[5]$ and $joint[7]$ ensuring that the center of the paddle is reached at the point of ball impact. Finally, $joint[9]$ is controlled according to the relation:

$$joint[9] = -(joint[3] + joint[5] + joint[7]) + (\pi - \pi/3.5)$$

Once all joint values have been collected, they are combined with the default values for unused joints and sent to the environment.

The pattern described is repeated at each instant of play to always obtain the optimal decision on what action to take.

1.1 Networks Architecture for Inverse Kinematic

The project employs two distinct feed-forward neural networks to address the inverse kinematics problem: one for the *low_player* and one for the *high_player*. Each network is designed to map the desired paddle positions to the corresponding joint angles needed to achieve accurate robotic arm movements during the game of table tennis.

1.1.1 Neural network for the *low_player*

The neural network for the *low_player* is designed to handle the downward-facing paddle. This network takes as input the y and z coordinates of the paddle's target position. The network then processes these inputs through three hidden layers (each of 64 neurons) with *ReLU* (Rectified Linear Unit) activation functions, which allow the network to learn complex mappings. The output layer of this network has three neurons representing $joint[3]$, $joint[5]$ and $joint[7]$ angles. These angles are needed to correctly position the joints of the robotic arm and achieve the desired position of the paddle.

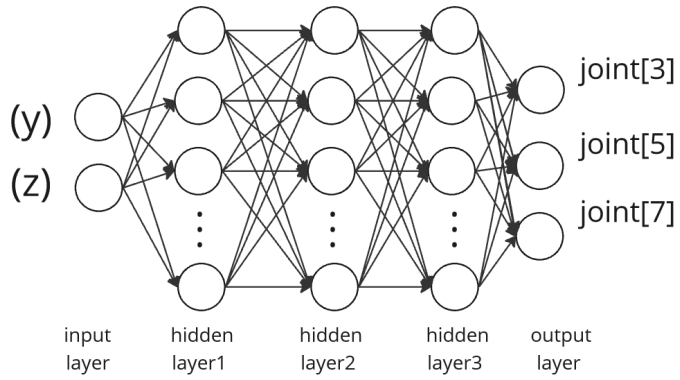

Figure 2: *low_player* Network

1.1.2 Neural network for the *high_player*

The neural network for *player_high* is configured to handle the upward-facing paddle. This network uses the z coordinate of the target position of the paddle as input. Like the network for the *low_player*, this network also includes three hidden layers (each of 64 neurons) with *ReLU* activation functions to capture the relationships between the required inputs and outputs. The output layer of the *high_players* network has two neurons, which correspond to the $joint[5]$ and $joint[7]$ angles. These angles are critical for adjusting the robotic arm's joints to maintain the correct paddle position.

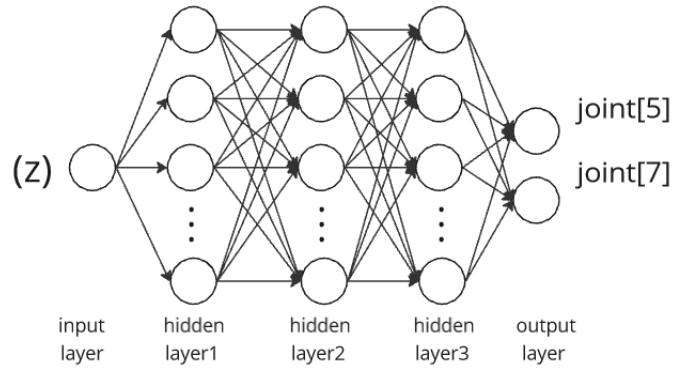

Figure 3: *high_player* Network

1.2 Network Architecture for Reinforcement Learning

The network architecture used for the paddle control is an Actor-Critic model, which includes two main neural networks: the Actor and the Critic. These networks work together to learn an optimal control policy based on the benefits obtained from interactions with the environment.

1.2.1 Actor

The actor is responsible for determining the actions the paddle of the robotic arm should take at any given time. It receives as input the current state of the environment, which includes state variables from 9 to 22: the paddle joints configuration, current position and normal versor, and the ball position and velocity vector. The actor processes this information using three fully connected hidden layers (of size 128, 128, and 64, respectively). Each hidden layer is followed by a normalization layer to improve training stability and the ReLU (Rectified Linear Unit) activation function. The final output defines the offset values of the joint[9] and joint[10] and it is passed through the Tanh activation function, ensuring that the actions produced fall within a range between -1 and 1. This allows the result to be scaled within the range of interest , in particular, from $-\pi/2$ to $+\pi/2$ to control the pitch of the paddle and from $-\pi/36$ to $+\pi/36$ to control the roll of the paddle.

1.2.2 Critic

The Critic evaluates the established actions by the Actor. It receives as input the current state of the environment, which includes state variables from 9 to 22 and the actor's established actions. It calculates the Q value, which represents an estimate of the "quality" of that specific action in that particular state. The Critic uses three fully connected hidden layers (of size 128, 130 and 64 respectively). Each hidden layer is followed by a normalization layer to improve training stability and the ReLU (Rectified Linear Unit) activation function. In the second hidden layer, the input is not just the current state but a concatenation of the state and the established actions, allowing the Critic to consider both information simultaneously. The output of the Critical is the calculated Q value.

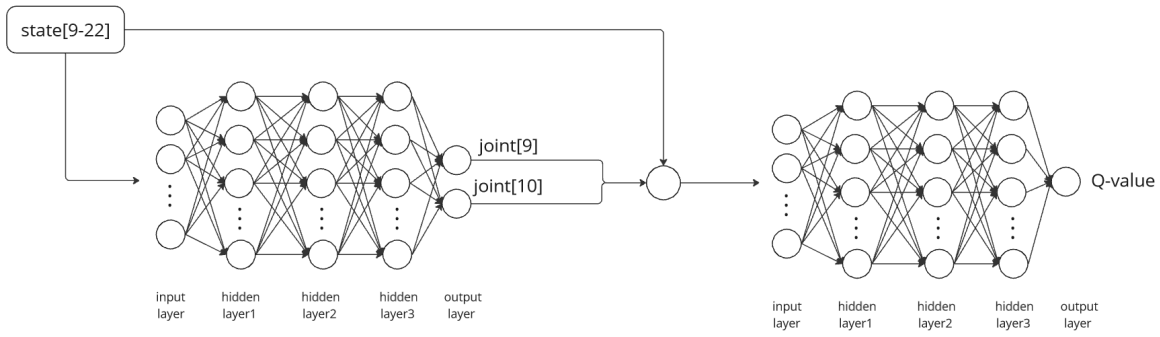

Figure 4: Actor-Critic Network

2 Inverse kinematics

To address the inverse kinematics problem and effectively control the robot's movements to hit the tennis table ball, the proposed solution involves using two supervised neural networks. One network is designed for the high player, with the paddle facing upwards, and the other for the low player, with the paddle facing downwards. This method ensures precise control of the robot's movements based on the player's position, allowing the robot to accurately intercept and return the ball.

The process is divided into two main phases: dataset creation and training. Both phases are carried out separately for the high player and the low player.

During the dataset creation phase, the system gathers data by simulating the robot's movements in a controlled environment without the presence of a ball. This helps to generate a comprehensive dataset of various joint configurations and corresponding paddle positions.

In the training phase, the collected datasets are used to train the neural networks. Each network learns to predict the necessary joint angles to achieve the desired paddle position for the high and low player scenarios. This training enables the robot to execute complex movements accurately, maintaining optimal paddle orientation to effectively play tennis table.

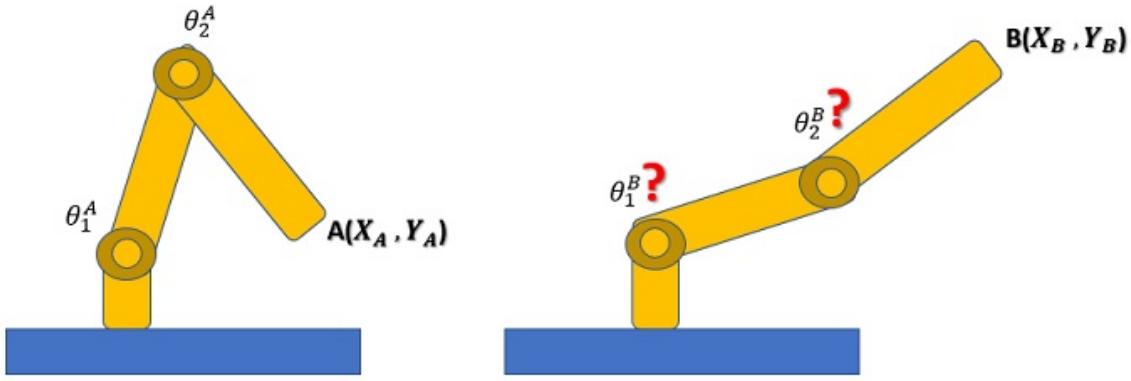

Figure 5: Inverse Kinematic problem

2.1 Datasets creation

2.1.1 High Player

The high player's dataset was created under the assumption that the paddle faces upwards. The purpose of the neural network is to determine the appropriate angles for joints $joint[5]$ and $joint[7]$ (outputs) based on the given z coordinate (input). This dataset was generated by systematically varying the angles of these joints and scanning the relevant area of the robot's workspace. To cover all possible configurations within these ranges, two nested loops were used, each iterating through the joint angles with a step size of $\pi/180$. This fine granularity ensured that a comprehensive dataset was created, capturing subtle variations in joint angles. The specific ranges for the joint angles were defined as follows:

- $joint[5]$ (pitch of the second arm link): 0 to $\pi/2$
- $joint[7]$ (pitch of the third arm link): 0 to $\pi/2$

The joint $joint[9]$ was calculated to maintain the paddle angle constant at 0 (vertically upwards), using the following formula:

$$joint[9] = -(joint[5] + joint[7])$$

This calculation ensure that the paddle maintained the correct angle even when the joints $joint[5]$ and $joint[7]$ varie. Any combinations of $joint[5]$ and $joint[7]$ that resulted in a calculated $joint[9]$ value outside its acceptable range of $-\frac{3\pi}{4}$ to $\frac{3\pi}{4}$ is excluded from the dataset, as the paddle would not be held in the correct position.

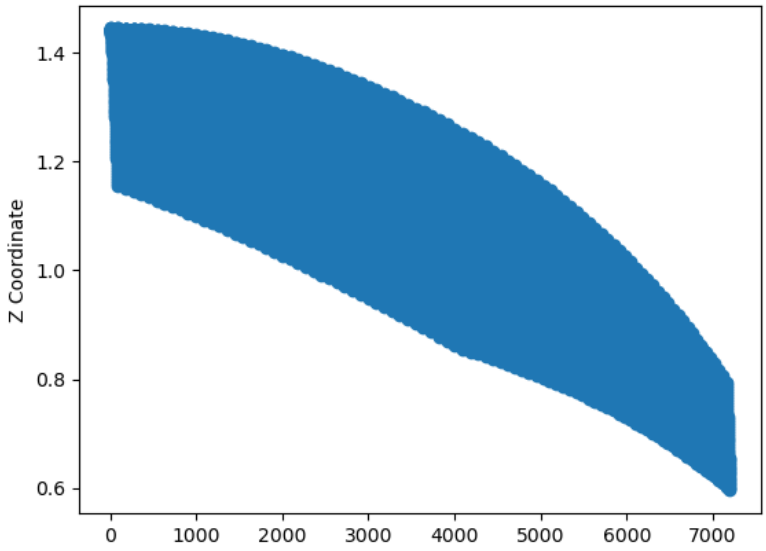

Figure 6: Plot of the z coordinate varying the rows of the high player dataset.

2.1.2 Low player

The low player's dataset was created under the assumption that the paddle faces downwards. The purpose of the neural network is to determine the appropriate angles for joints joint[3], joint[5], and joint[7] (outputs) based on given y and z coordinates (inputs). This dataset was generated by systematically varying the angles of these joints and scanning the relevant area of the robot's workspace. To cover all possible configurations within these ranges, three nested loops were used, each iterating through the joint angles with a step size of $\pi/90$. This fine granularity ensured that a comprehensive dataset was created, capturing subtle variations in joint angles. The specific ranges for the joint angles were defined as follows:

- $joint[3]$ (pitch of the first arm link): 0 to $\pi/4$
- $joint[5]$ (pitch of the second arm link): 0 to $\pi/2$
- $joint[7]$ (pitch of the third arm link): 0 to $\pi/2$

The joint joint[9] was calculated to maintain constant the paddle angle at $(\pi - \pi/3.5)$, using the following formula:

$$joint[9] = -(joint[3] + joint[5] + joint[7]) + (\pi - \pi/3.5)$$

This calculation ensure that the paddle maintained the correct angle even when the joints $joint[3]$, $joint[5]$, and $joint[7]$ varie. Any combinations of $joint[3]$, $joint[5]$, and $joint[7]$ that resulted in a calculated $joint[9]$ value outside its acceptable range of $-\frac{3\pi}{4}$ to $\frac{3\pi}{4}$ is excluded from the dataset, as the paddle would not be held in the correct position. Additionally, any configurations of $joint[3]$, $joint[5]$, and $joint[7]$ that caused the z-coordinates of the paddle to become negative is excluded to prevent the paddle from going under the table.

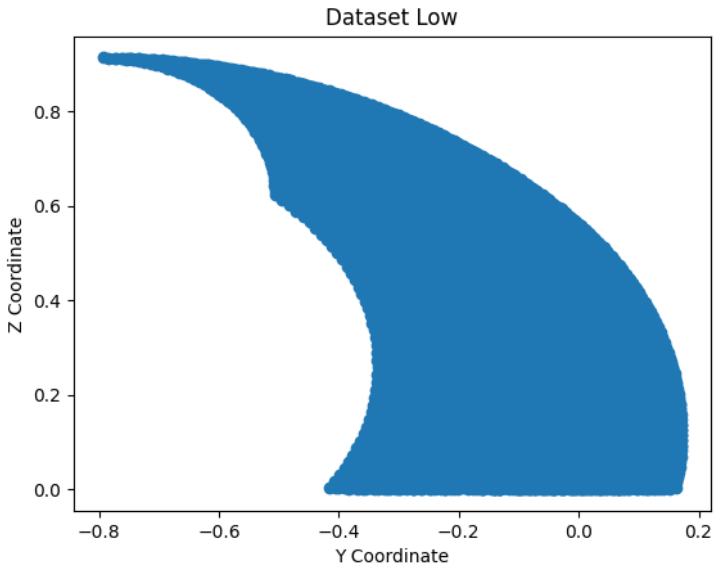

Figure 7: Plot of y and z coordinate of the low player dataset.

2.2 Training

The training of the neural networks for both the high player and the low player involves a systematic and detailed process to ensure accurate prediction of joint angles required for the robot to play tennis table effectively.

Firstly, the data is prepared by loading the respective datasets for the high and low players. This data is then split into training, validation, and test sets. A common practice is to allocate 70% of the data for training, 15% for validation, and 15% for testing. To ensure that the input features and target outputs are on a similar scale, normalization is performed. This involves scaling the data to have zero mean and unit variance.

The neural network architecture is then defined. For the low player, the neural network takes the y and z coordinates as inputs and outputs the angles for joints joint[3], joint[5], and joint[7]. For the high player, the network takes the z coordinate as input and outputs the angles for joints joint[5] and joint[7]. This architecture includes an input layer corresponding to the number of input features, one or more hidden layers with the ReLU activation functions, and an output layer that corresponds to the number of joint angles to be predicted.

Training begins by defining a loss function and an optimizer. The mean squared error (MSE) loss function measures the difference between the predicted and actual joint angles, while the Adam optimizer updates the network weights based on the computed gradients. The training process runs for a specified number of epochs, during which the model is iteratively improved. Early stopping is implemented to prevent overfitting by monitoring the validation loss and halting training if it does not improve for a certain number of epochs.

During each epoch, the model is trained on the training set and evaluated on the validation set. The training loop involves a forward pass to compute the predicted joint angles, loss computation to calculate how far off the predictions are from the actual values, a backward pass to perform backpropagation and compute gradients, and a weight update to adjust the model parameters. Validation involves evaluating the model on the validation set and recording the validation loss. If the validation loss improves, the model's state is saved; otherwise, the patience counter for early stopping is incremented.

Once training is complete, the best model (i.e., the one with the lowest validation loss) is loaded for final evaluation on the test set. The model's performance is then valued using the test set, with the mean squared error (MSE) used as the evaluation metric. This evaluation helps determine how well the model generalizes to new, unseen data.

Despite the relatively high training loss, the neural network accurately predicts the joint values, enabling the paddle to reach the desired point, as demonstrated in the tests carried out. This phenomenon occurs because there are multiple correct solutions for the joint angles that can position the paddle at the target location. Consequently, the predicted solution, while different from the specific target in the test set, is still valid and achieves the desired outcome. This flexibility in possible solutions means that the model performs effectively in practical scenarios, even if the mean squared error (MSE) is not minimized to an extremely low value.

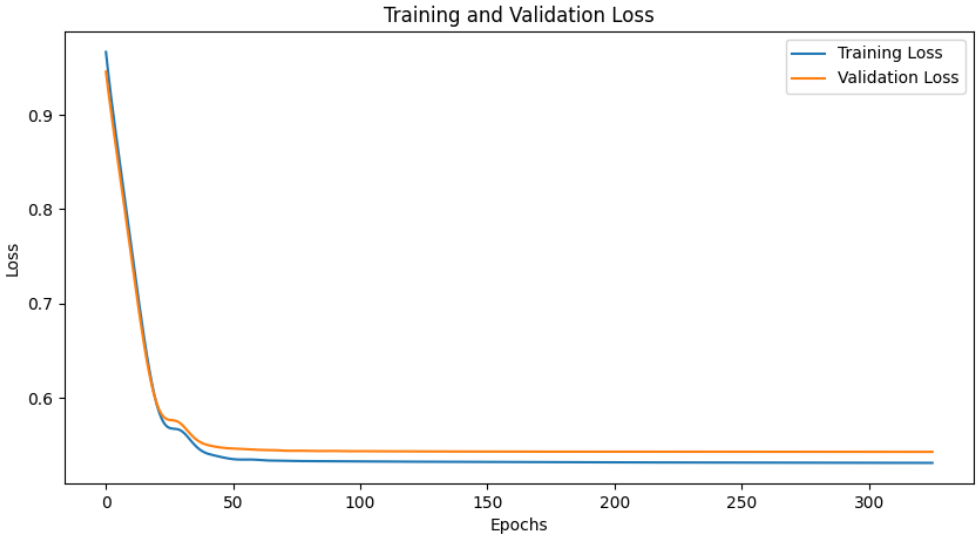

Figure 8: Plot of loss of the high player's training.

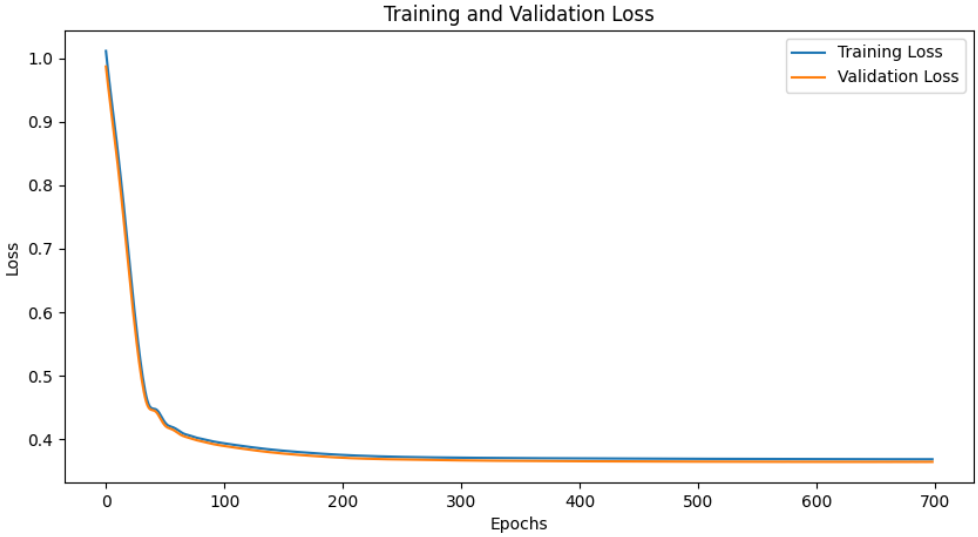

Figure 9: Plot of loss of the low player's training.

3 Reinforcement Learning

3.1 Agent Functioning

During the training phase with Reinforcement Learning, the agent interacts with the environment by executing actions that influence the state of the environment. At each interaction, the agent receives a reward that guides it in learning the optimal policy. The agent acts on the robot's joint[9] and joint[10], which respectively control the pitch and roll of the paddle. It is activated only when the ball is close to the paddle, managing the movement to hit the ball optimally, aiming to maximize the received reward. The agent calculates the actions using the policy network (actor). The states of the environment are passed through the actor network, which returns a continuous action. The actions are then limited through the `CustomActionSpace` class, which applies lower and upper bounds to ensure that the actions are valid within the simulated environment.

The agent uses the parameter τ and soft updates to update the actor and critic networks. The soft updates blend the weights of the target networks with those of the main networks, according to the formula:

$$\theta_{\text{target}} \leftarrow \tau\theta + (1-\tau)\theta_{\text{target}}$$

Additionally, the agent is equipped with methods to save the model weights during training and load them later. This allows resuming training from a specific point and preserving the best models.

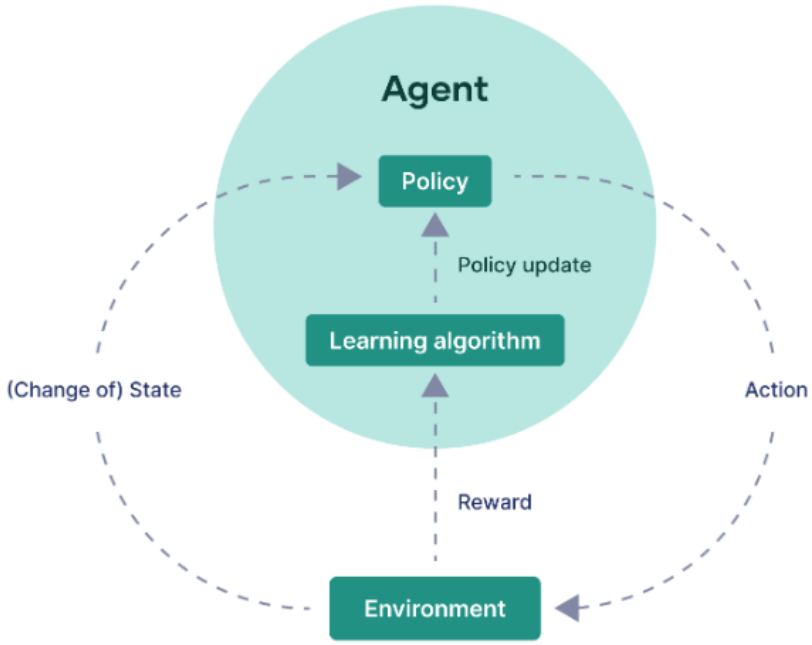

Figure 10: Reinforcement learning

3.2 Training Process

The training process of the DDPG agent consists of various steps that are repeated iteratively. The agent interacts with the environment by taking the state, calculating actions, and observing the results. Subsequently, the reward of the calculated action is computed, and the transitions are stored in the replay buffer. Each batch sampled from the replay buffer is used to update the parameters of the neural networks. To explore the environment effectively, noise was added to the agent's actions. This noise, of the Ornstein-Uhlenbeck type, helps in exploring new actions that could lead to better rewards. The critic network is updated by minimizing the mean squared error between the predicted Q-values and the targets. The actor network is updated by maximizing the mean Q-value.

3.2.1 Hyperparameters

The hyperparameters used during training are:

- **Gamma (γ)**: Discount factor for future rewards, set to 0.99.
- **Tau (τ)**: Update factor for soft updates of the target networks, set to 0.001.
- **Hidden Layer Size**: The size of the hidden layers of the neural networks, set to [128, 128, 64].
- **Replay Buffer Size**: The size of the replay buffer, set to 1e6 transitions.
- **Batch Size**: The batch size used during training, set to 32.
- **Learning Rate**: The learning rate for the actor and critic, respectively set to 1e-4 and 1e-3.
- **Standard Deviation of Noise (noise_stddev)**: Standard deviation of the Ornstein-Uhlenbeck noise, set to 0.2.

3.2.2 Replay Buffer

The replay buffer is used to store transitions (state, action, reward, next state) during the agent's interaction with the environment.

Each transition is represented by a named tuple `Transition` that includes state, action, done, next state, and reward, where done indicates whether the system has reached a terminal state. Once the ball is out of range, the episode is considered finished, whether the ball was hit or not.

Transitions are randomly sampled from the buffer to update the parameters of the neural networks.

3.2.3 Reward

The reward function is fundamental to guide the agent's learning, defining the quality of an action in a given state. It provides feedback that helps the agent understand the consequences of its actions, ultimately leading to improved performance in the task. The final reward function is:

$$reward = \begin{cases} +25 & \textit{if the agent hits the ball and then the ball hits the opponent's court} \\ -15 & \textit{if the agent hits the ball and then the ball doesn't hit the opponent's court} \\ -1 & \textit{if the agent doesn't hit the ball but the ball is in front of the paddle} \\ -6 & \textit{if the agent doesn't hit the ball but the ball is behind the paddle} \end{cases}$$

By utilizing this reward function, the agent is guided through a process of trial and error, learning to maximize positive outcomes while minimizing negative ones.

3.2.4 Monitoring and Evaluation

The details of the training, such as rewards and losses, are recorded using a logger and TensorBoard, producing graphs that indicate the progress of the training. Models are saved periodically when performance exceeds a predefined threshold, allowing the best agent state to be restored. After several trials, choices have been optimized, particularly the reward function, which has been modified over time to achieve better performance.

Initially, the reward function was based on the scores of the player and the opponent as follows:

$$reward = \begin{cases} +20 & \textit{if the agent scores a point} \\ -20 & \textit{if the opponent scores a point} \end{cases}$$

However, since the function was too general and heavily dependent on the opponent's skills; the agent was unable to learn correctly. The reward function was then modified several times, following poor results, until reaching the final version. The final version gives importance to the moment of impact between the paddle and the ball, with the aim of improving the quality of the stroke performed.

Additionally, the logic for saving transitions has been refined to try to save only relevant transitions in the buffer, where the reward is closely linked to the action taken and the state reached by the system. For example, previously, the transition was recorded at the moment when the score change occurred, even though that point could have been achieved due to an action performed in the past. In the final version of the training, the transition is saved with the initial state, the action performed, and the subsequent state at the moment of the stroke, while the calculation of the reward waits for the moment of impact with the opponent's court. This ensures that the saved transition is consistent and that the reward is closely linked to the other fields of the transition.

Other parameters were also changed during the process, such as the network inputs, and the size and number of hidden layers. Initially, all thirty-seven state variables were used as inputs, with an attempt to control all the robot joints as outputs. This task proved to be too complex for the agent, leading to a drastic reduction in the number of inputs and outputs, focusing only on managing joints nine and ten as outputs.

3.2.5 Final Results

Figure 11 shows the trend of the average reward during the initial phase of the training process. Initially, the average reward is negative, indicating that the agent was performing many random or bad actions. As the training progresses, the average reward begins to increase but remains negative, suggesting that while there is some improvement, the agent's policy is still ineffective and unable to achieve positive rewards, indicating suboptimal performance.

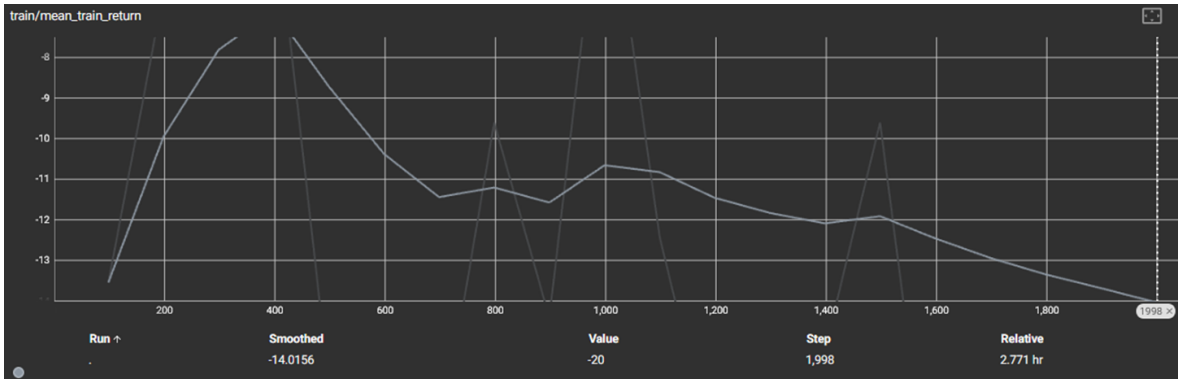

Figure 11: Average training initial reward

Figure 12 provides a more detailed view of the average training reward over the entire training process. As observed, the initial phase of training is characterized by a negative average reward, signifying a period of exploration where the agent tries various actions. With more training steps, there is a noticeable increase in the average reward, and around 2000 steps, the reward becomes positive. Although there are some fluctuations in the reward values thereafter, the general trend indicates stabilization at positive values. This pattern suggests that the agent has successfully resolved the initial inefficiencies and has learned an effective strategy for the task at hand.

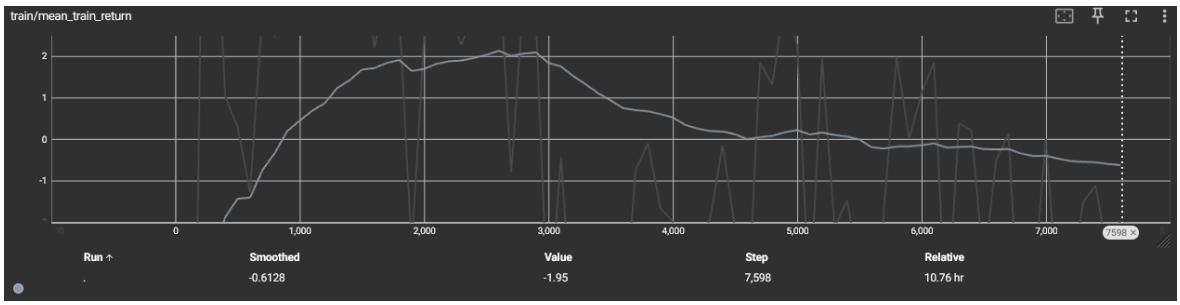

Figure 12: Average training final reward

List of Figures