Università degli Studi di Salerno

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE ED ELETTRICA E MATEMATICA APPLICATA

CORSO DI LAUREA TRIENNALE IN INGEGNERIA INFORMATICA

MACHINE LEARNING PROJECT WORK

Group 8

Marotta Giuseppe - 0622702302 - g.marotta31@studenti.unisa.it

Rea Gaetano - 0622702190 - g.rea7@studenti.unisa.it

Squitieri Giuseppe - 0622702339 - g.squitieri8@studenti.unisa.it

Tramice Davide - 0622702194 - d.tramice@studenti.unisa.it

Anno Accademico 2023/2024

Indice

1 Intruduction

The primary objective of this project is to develop a machine learning model that can effectively control a robotic arm to play table tennis within a virtual environment. This task requires a nuanced understanding of the dynamics involved in real-time interactions between the robotic arm and the ping pong ball, depending on the influence of the playfield and the opponent, necessitating a robust model that can predict and react to rapid changes within the game environment.

1.1 Problem Description

The game environment is composed by the playfield, the ball and two robotic arms where there's only one to be controled, in order to "win" the table tennis game against its "automatic" opponent: both the arms are designed to closely mimic human arm movements necessary for playing table tennis, in fact they feature a base that allows a major horizontal and a slight vertical movement, mimicking the waist's movement to cover various angles on the table. This base supports a structure consisting of articulated joints similar to a human's shoulder and elbow, providing the arm with the necessary reach and flexibility to extend towards or retract from the play area, meanwhile a wrist-similar joint at the end of the arm enables precise adjustments of the paddle's angle and rotation, crucial for effectively hitting the ball with desired spins and directions.

1.2 Problem Solution

In order to solve this problem and to accomplish the project objective we decided to split the resolution in multiple steps and the main ones split the main problem into two smaller subproblems: the first one is to define a supervised learning based model which allows the robotic arm to consistently follow and intercept the table tennis ball by setting the arm joints and base in the right way, meanwhile the second subproblem is to define aswell a reinforced learning based model which starts from the results of the first network and allows the robotic arm to effectively hit the intercepted ball in the right way in order to stack points to win the table tennis game.

2 Supervised Learning

In the first design phase, the goal is to build a network capable of positioning the robot's joints so that the paddle reaches a certain position. This is achieved through inverse kinematics, which determines the necessary joint movements starting from the desired paddle position. Therefore, this chapter will analyze the development of a network that takes 3 inputs (the desired position of the paddle). The network, with a hidden layer of 128 neurons, computes 11 output values corresponding to the robot's joints.

2.1 Dataset Generation

The dataset generation for our supervised learning model was systematically approached to ensure that the robotic arm could be trained under various scenarios representative of real gameplay conditions. Utilizing a Python script, `client_generateDataset.py`, we implemented a structured method to populate our dataset dynamically based on predefined joint configurations and range restrictions: dataset associates a large number of joint settings each to a specific point in the game field space, expressed by its tridimensional coordinates, in order to populate a dataset which can be used to train the robotic arm to move in function of the ball movements.

The script defines specific ranges for joint movements, both translational and rotational, where each joint of the robotic arm can vary within given limits: these ranges are crucial as they simulate the possible positions and movements the robotic arm might need during an actual table tennis match. Using the `numpy` library, the script is capable of generating random configurations within these defined limits for each joint, and this aspect is implemented by the function `get_joints()`, which creates these required random joint configurations, ensuring a comprehensive coverage of potential real-game scenarios in terms of spatial positioning.

This randomized generation is not entirely stochastic: certain joints, like those directly controlling the paddle, have their randomness confined to simulate realistic and strategically relevant paddle movements, and in fact, to validate each generated state's relevance, the script employs a function, `paddle_versor()`, which assesses the orientation of the paddle in comparison to a predefined correct orientation (versor). This ensures that the data collected only includes viable gameplay states where the paddle is adequately oriented to interact with the ball effectively.

Data collection is performed in a continuous loop, where each valid state, confirmed by the paddle's orientation, is recorded into a CSV file; more precisely, the generated CSV files are three:

- `Low_position_dataset.csv`
- `Mid_position_dataset.csv`
- `High_position_dataset.csv`

Each one generated under certain constraints related to the joints movement ranges, in order to define which part of the game space is going to be considered in the data collection process. Each one of the three CSV files is related to a specific arbitrarily defined "portion" of the mentioned game space, respectively to the low, the mid, and the high part of the game space in relation to the z-axis, in order to define a larger and more case-covered dataset which accumulates a wide variety of joint positions and paddle orientations, capturing a rich variety of data that provides a robust foundation for training the supervised learning model.

This structured approach to dataset generation not only ensures diversity in the data but also embeds a layer of pre-validation, making sure that every data point used for training the model is a plausible and game-relevant scenario. This methodology enhances the efficiency of our training process and increases the model's ability to generalize across different game states.

2.2 Matplot Graphs

In order to verify if the dataset generation was suitable for the planned training we created a series of Matplot graphs which give a view of the space data covered in the dataset generation, considering the three mentioned sections (low, mid and high), all in relation to the axis considered in the game environment.

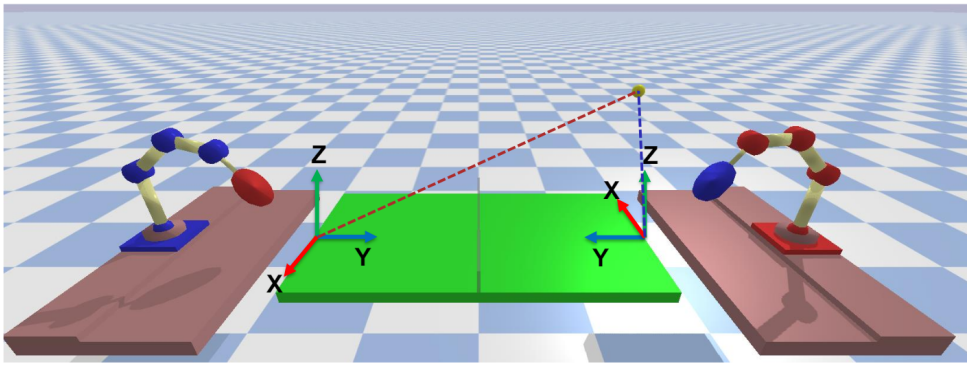

Figura 2.1: Game environment axis orientation

Low Position Scatterplot

These graphs showcase points clustered in the lower region of the Z-axis, which corresponds to the robotic arm handling shots that are closer to the table, reflecting defensive play or scenarios where the ball is low. This dataset is essential for training the model to handle low shots effectively, which are common in defensive strategies during a match.

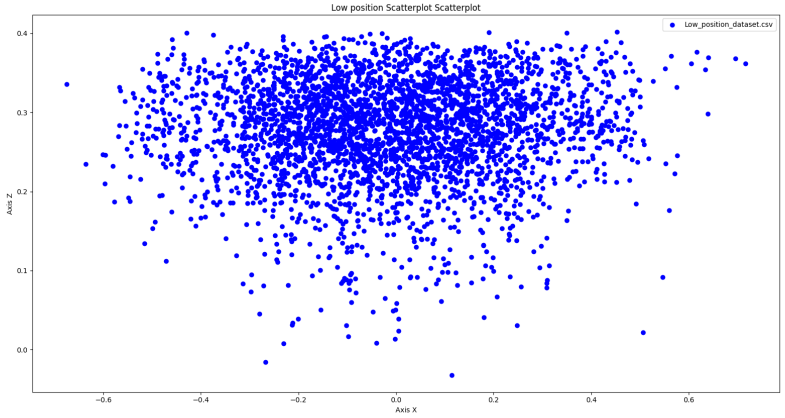

Figura 2.2: Low position scatterplot z/x

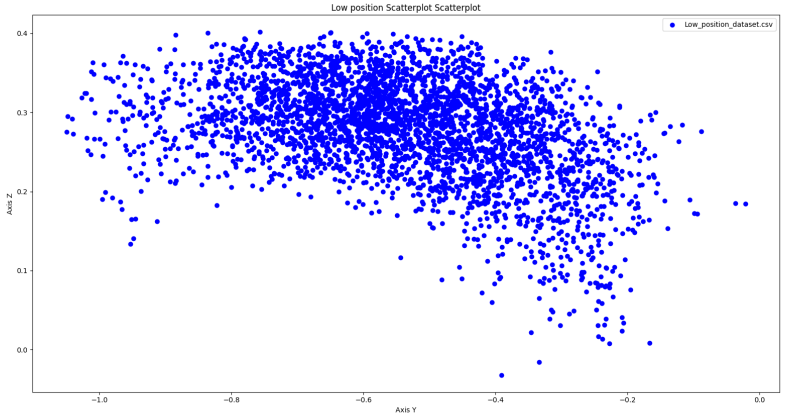

Figura 2.3: Low position scatterplot z/y

Mid Position Scatterplot

The mid position points are spread around the middle range of the Z-axis, representing the typical range of play where the ball passes in regular rally exchanges. This is the most common scenario in table tennis, making this dataset critical for ensuring the robotic arm performs well under usual playing conditions.

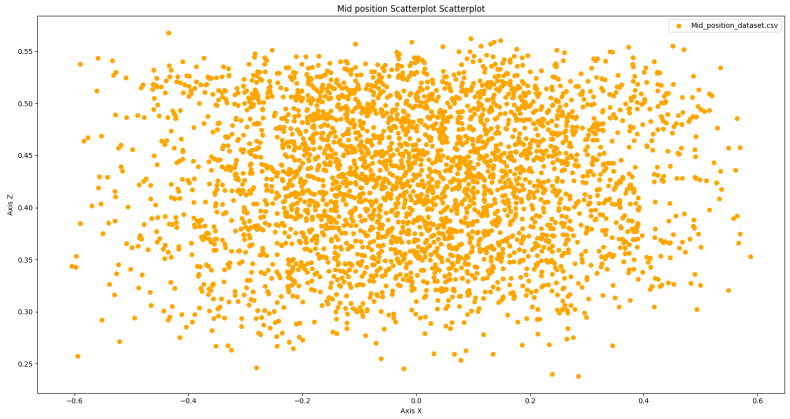

Figura 2.4: Mid position scatterplot z/x

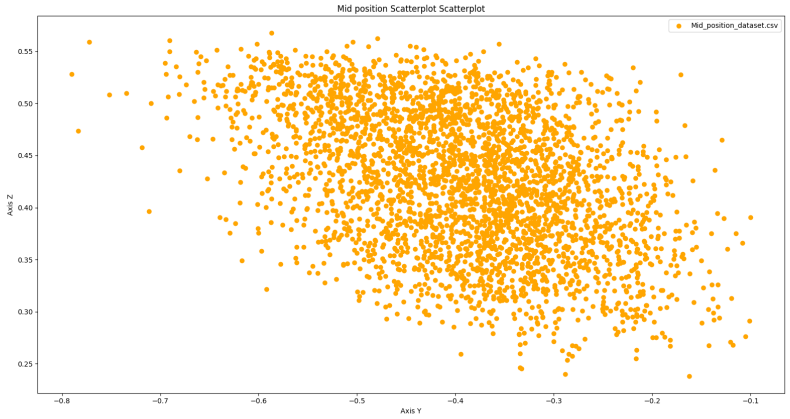

Figura 2.5: Mid position scatterplot z/y

High Position Scatterplot

The high position scatterplot points are located in the upper region of the Z-axis, indicating the robotic arm reaching up to handle high shots, which could be crucial during aggressive plays or when the ball is lobbed. Training the model with these data points ensures it can handle a wide range of shots which could be otherwise impossible to intercept

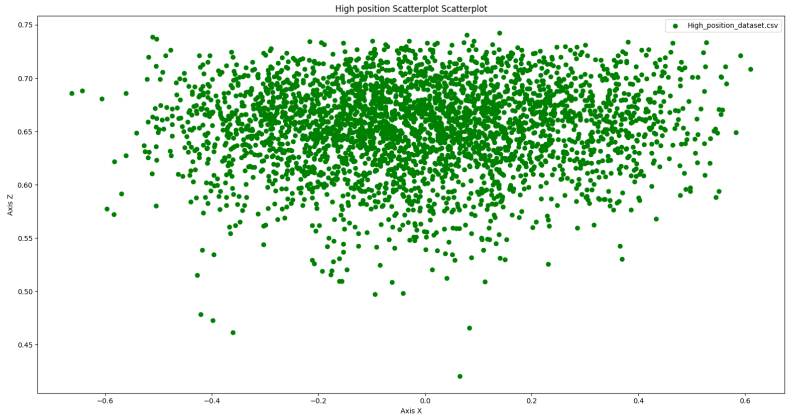

Figura 2.6: High position scatterplot z/x

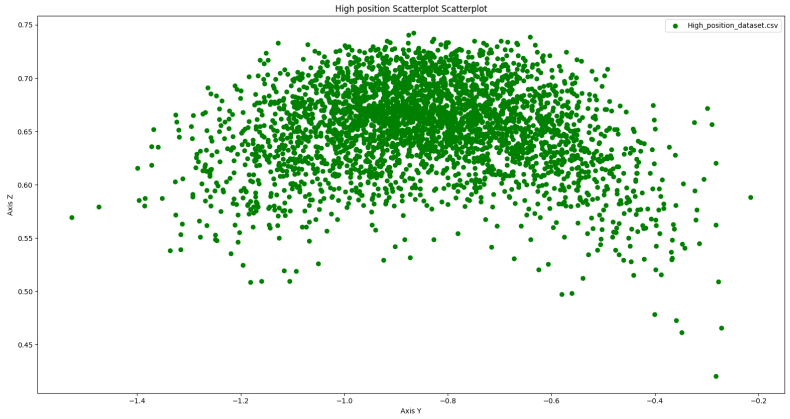

Figura 2.7: High position scatterplot z/y

Combined Scatterplot

The combined scatterplots provide an overview of all the data points across the high, mid, and low datasets, illustrating a comprehensive training spectrum that the robotic arm will encounter. This plot demonstrates the varied capabilities of the robotic arm across different playing conditions, showcasing the adaptability required for competitive gameplay.

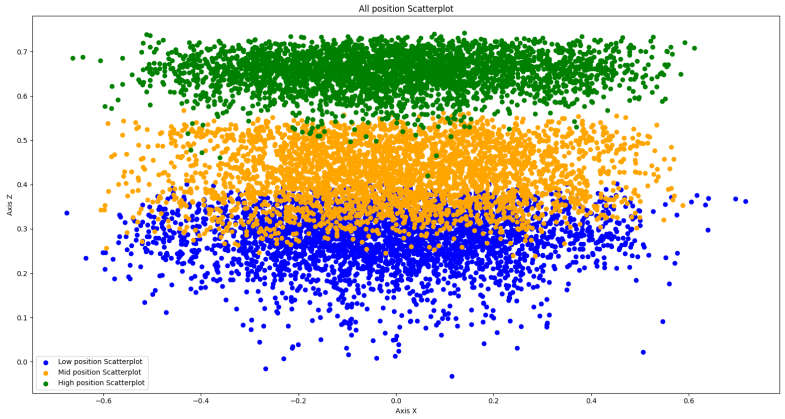

Figura 2.8: Combined scatterplot z/x

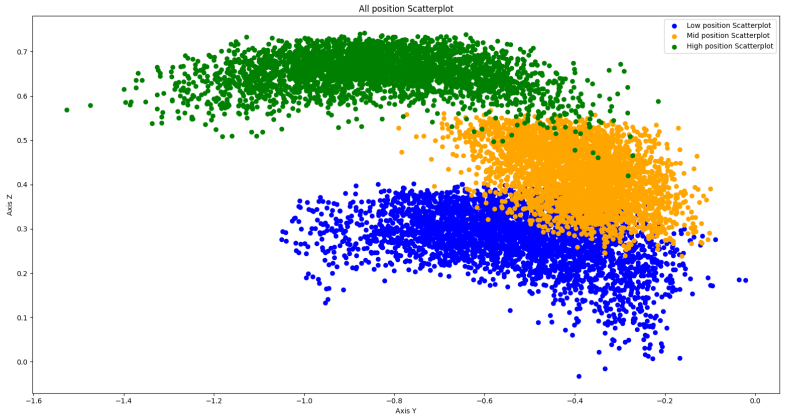

Figura 2.9: Combined scatterplot z/y

These plots are invaluable for assessing the comprehensiveness of the training data. By visually confirming that the data points are well-distributed across the intended ranges of motion, we can ensure the supervised model will be trained on a balanced dataset that mirrors real-world conditions closely. This approach not only enhances the robotic arm's performance but also its ability to react dynamically to unexpected game situations, enhancing its robustness and versatility.

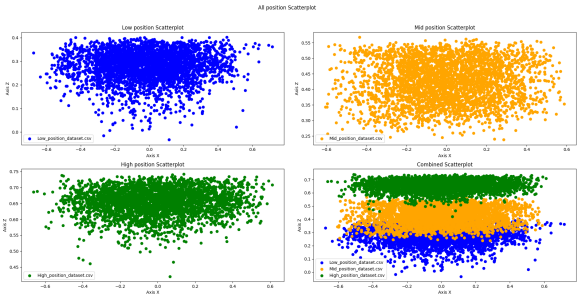

Figura 2.10: General Overview z/x

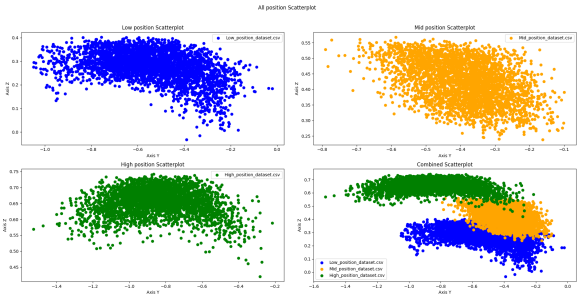

Figura 2.11: General Overview z/y

2.3 Training Details

2.3.1 Overview

Training a virtual robotic system for competitive environments such as table tennis requires a model capable of interpreting complex dynamics and responding with high precision. The purpose of our model is to anticipate the trajectory of the ball and adjust the robot's joints for optimal interaction, and this is where the application of inverse kinematics (IK) becomes crucial: Inverse kinematics is a method used in robotics to calculate the configurations needed for a jointed arm, like the agent we are interested to, in order to reach a desired position in space, which is essential for interacting with moving objects, in this case to meet the ball at certain points in the game field. This integration of IK within the neural network model allows for a translation of spatial and temporal features extracted from the game's state into actionable joint movements, ensuring the robot can dynamically respond to the ball with high precision.

Role of Inverse Kinematics in Model Training

Inverse kinematics transforms the problem of robotic control into a more manageable form for our neural network. By training the model to output joint positions directly, we effectively embed the complex calculations of IK into the learning process. This approach not only simplifies the real-time computational load but also enhances the model's ability to generalize from diverse gameplay data, adapting to new and unforeseen game scenarios effectively.

The neural network model used in this project, therefore, is not just learning from visual clues or simple direct control signals; it is learning to embody the principles of robotic movement dictated by the laws of physics and geometry, represented through the lens of inverse kinematics; this happens by utilizing advanced simulation tools to generate data that accurately reflects possible real-game scenarios, including varied ball trajectories and speeds, later transforming this data into a format that effectively teaches the model about the spatial relationships and

necessary joint configurations to interact with the ball correctly. At this point it is indeed necessary to design a network capable of processing the mentioned inputs and producing outputs that represent joint movements, grounded in the principles of inverse kinematics, to iteratively adjust the model's parameters to minimize error in joint position predictions, ensuring precise alignment with both the training data and the physical realities of robotic movement.

2.3.2 Model Architecture

The neural network model used in this project is a regression model designed to predict the joint positions of the robotic arm based on the incoming ball's position and velocity. The model architecture is crafted to process the spatial and temporal features extracted from the game's state and convert them into actionable joint movements. The network consists of several layers:

- **Input layer:** Accepts the ball's current coordinates, expressed by three continuous variables.
- **Hidden layers:** Two layers with 128 neurons each and ReLU activation to introduce non-linearity, enabling the model to learn complex patterns from relatively simple input data.
- **Output layer:** Provides the next set of joint positions to optimally intercept the ball, outputting 11 continuous variables that represent each joint's targeted position.

2.3.3 Training Process

The training process involves several steps designed to optimize the model's performance through iterative adjustments based on the training data:

1. **Data Collection:** Utilizing a custom dataset class, we collect a comprehensive set of possible scenarios that the robot might encounter during gameplay. This dataset includes data formatted directly from a CSV file, whose origin has been discussed earlier in the dataset generation section.
2. **Data Preprocessing:** The data from the CSV file is preprocessed to format the inputs and targets into tensors, ensuring that the model trains effectively without bias toward any particular range of input values.
3. **Loss Function:** We use the Mean Squared Error (MSE) as the loss function to quantify the difference between the predicted joint positions and the ground truth from the dataset, providing a clear metric for optimization.
4. **Optimizer:** An Adam optimizer is employed to adjust the weights of the neural network to minimize the loss, taking advantage of its efficient handling of sparse gradients and adaptive learning rate capabilities.

5. **Training Cycles:** The model is trained over multiple epochs, each involving running through the dataset, allowing for gradual and steady improvement in the model's accuracy.

6. **Validation:** The model's performance is periodically validated against a separate validation set extracted from the initial dataset to monitor for overfitting and ensure generalizability.

7. **Early Stopping:** To further prevent overfitting, training is halted if the validation loss does not improve for a consecutive number of epochs, ensuring that we do not continue to train a model past its point of optimal performance.

2.4 Training Implementation Details

This section provides an in-depth look at the implementation of the training process for the desired neural network model, that we remember is designed to predict joint positions of robotic arm based on the incoming ball's position and velocity. The discussion includes detailed breakdowns of the custom dataset preparation, model architecture, training loop, and results visualization.

2.4.1 Dataset Preparation

Custom Dataset Class

The custom dataset class is crucial for reading and preprocessing data from a CSV file, which contains both input features and target outputs.

```
class CustomDataset(Dataset):
    def __init__(self, file_path):
        with open(file_path, 'r') as file:
            reader = csv.reader(file)
            next(reader)  # Skip header
            data = list(reader)
        self.targets = []
        self.inputs = []
        for line in data:
            values = list(map(float, line))
            # First 11 columns are targets
            self.targets.append(values[:11])
            # Last 3 columns are inputs
            self.inputs.append(values[-3:])
        # Convert lists to PyTorch tensors
```

```
        self.targets = torch.tensor(self.targets, dtype=torch.float32)
        self.inputs = torch.tensor(self.inputs, dtype=torch.float32)

    def __len__(self):
        # Return the length of the dataset
        return len(self.targets)

    def __getitem__(self, idx):
        # Return input-target pair
        return self.inputs[idx], self.targets[idx]
```

This class manages the loading and structuring of the dataset. Data is parsed, separated into inputs and targets, and converted into tensors for efficient processing with PyTorch during the training and validation phases.

2.4.2 Model Architecture

Regression Neural Network

The neural network model, defined below, uses linear layers and ReLU activations to process input data and predict outputs.

```
class RegressionModel(nn.Module):
    def __init__(self):
        super(RegressionModel, self).__init__()
        self.fc1 = nn.Linear(3, 128)
        self.fc2 = nn.Linear(128, 128)
        self.fc3 = nn.Linear(128, 11)
    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

The model consists of an input layer that takes three features, consisting in the coordinates which express the ball position, an hidden layer with 128 neurons, and an output layer that predicts 11 values representing the joint positions and settings. ReLU activation functions are used to add non-linearity, enhancing the model's ability to learn complex patterns in the data.

2.4.3 Training Procedure

Training Function

The training function orchestrates the entire learning process, managing forward passes, loss calculations, backpropagation, and implementing early stopping based on validation performance. This comprehensive approach ensures robust training while preventing overfitting through careful monitoring and control mechanisms.

```
def train_model(model, train_loader, val_loader, criterion, optimizer,
                epochs=100, patience=10):
    best_loss = float('inf')
    patience_counter = 0

    for epoch in range(epochs):
        model.train()
        epoch_train_loss = 0.0
        for inputs, targets in train_loader:
            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, targets)
            loss.backward()
            optimizer.step()
            epoch_train_loss += loss.item()

        epoch_train_loss /= len(train_loader)
        train_losses.append(epoch_train_loss)

        model.eval()
        val_loss = 0.0
        with torch.no_grad():
            for inputs, targets in val_loader:
                outputs = model(inputs)
                loss = criterion(outputs, targets)
                val_loss += loss.item()

        val_loss /= len(val_loader)
        val_losses.append(val_loss)

        print(f'Epoch [{epoch + 1}/{epochs}], Train Loss: {epoch_train_lo
              'Val Loss: {val_loss:.4f}')
```

```
if val_loss < best_loss:
    best_loss = val_loss
    patience_counter = 0
    torch.save(model.state_dict(), 'model0.pth')
else:
    patience_counter += 1

if patience_counter >= patience:
    print('Early stopping triggered')
    break
```

- **Initialization of Best Loss and Patience Counter**: These are prepared before the epochs start, setting the stage for tracking improvements and early stopping.
- **Epoch Loop**: Manages training and validation in each cycle, adjusting model weights and evaluating performance.
- **Training Phase**:
    1. **Zeroing Gradients**: Essential for correct gradient computation by clearing old gradients.
    2. **Forward Pass and Loss Computation**: Processes inputs through the model and calculates loss against targets.
    3. **Backpropagation and Parameter Update**: Calculates gradients and updates model parameters, optimizing the network.
    4. **Tracking Train Loss**: Accumulates and averages the loss over batches within an epoch for analysis.
- **Validation Phase**:
    1. **Model Evaluation Mode**: Switches to evaluation mode, optimizing performance for validation data.
    2. **No Gradient Calculation**: Reduces computation load by disabling gradient tracking during validation.
    3. **Loss Computation and Tracking**: Assesses model's generalization capability by computing validation loss.
- **Early Stopping Check**: Monitors non-improvement in validation loss, stopping training to prevent overfitting.

This structured explanation provides a complete overview of the training function, ensuring clarity on how the model learns and generalizes, balanced with mechanisms to prevent overfitting.

2.4.4 Main Function Explanation

The ‘main‘ function acts as the entry point for the script, orchestrating the entire process from data handling to model training and results visualization. It ensures a streamlined execution of tasks in a logical sequence, crucial for the successful training of the neural network model.

Initialization

```
def main():
    global train_losses, val_losses
    # Path to the CSV file containing the data
    file_path = 'datasets/All_positions_dataset.csv'
```

The function begins by defining the path to the CSV file containing the dataset, initializing global lists to track training and validation losses. These will store the loss values over each epoch for later analysis and visualization.

Dataset Preparation and Loading

```
    train_losses = []
    val_losses = []
    dataset = CustomDataset(file_path)
    train_size = int(0.8 * len(dataset))
    val_size = len(dataset) - train_size
    train_dataset, val_dataset = random_split(dataset,
    [train_size, val_size])
    train_loader = DataLoader(train_dataset, batch_size=128,
        shuffle=True)
    val_loader = DataLoader(val_dataset, batch_size=128,
        shuffle=False)
```

The dataset is loaded and split into training and validation sets, constituting 80% and 20% of the total dataset, respectively. ‘DataLoader‘ objects are then created for both datasets with specified batch sizes and shuffling configurations to optimize the training process and ensure data variability.

Model Initialization and Training

```
    model = RegressionModel()
    criterion = nn.MSELoss()
    optimizer = optim.Adam(model.parameters(), lr=0.0001)
    train_model(model, train_loader, val_loader, criterion,
```

```
        optimizer, epochs=100, patience=10)
```

A regression model is initialized along with the MSE loss function and Adam optimizer. The model is then trained using the previously defined function, which includes mechanisms like early stopping to enhance training efficiency and prevent overfitting.

Results Visualization

Post-training, several plots are generated to visualize the training and validation loss trends across epochs. These plots are crucial for assessing the model's learning progress and ensuring that the training process is effective. The plots are not only displayed but also saved to disk for future reference or analysis.

```
plt.figure(figsize=(10, 5))
plt.plot(train_losses, label='Training Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training Loss')
plt.legend()
plt.tight_layout()
plt.savefig('Train_inverse_kinematics_graphic/train_loss.png')
plt.show()
```

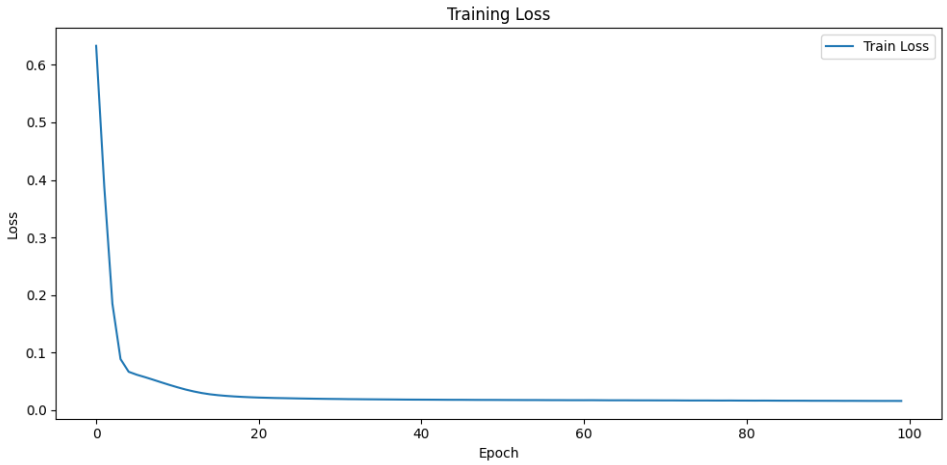

Figura 2.12: Train Loss Graph

This graph displays the training loss as the model progresses through each epoch. The loss sharply decreases initially and then flattens out, indicating that the model quickly learned the dominant patterns in the data and then made incremental improvements. The training loss approaches a low steady state, suggesting the model is fitting well to the training data.

```
plt.plot(val_losses, label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Validation Loss')
plt.legend()
plt.tight_layout()
plt.savefig('Train_inverse_kinematics_graphic/val_loss.png')
plt.show()
```

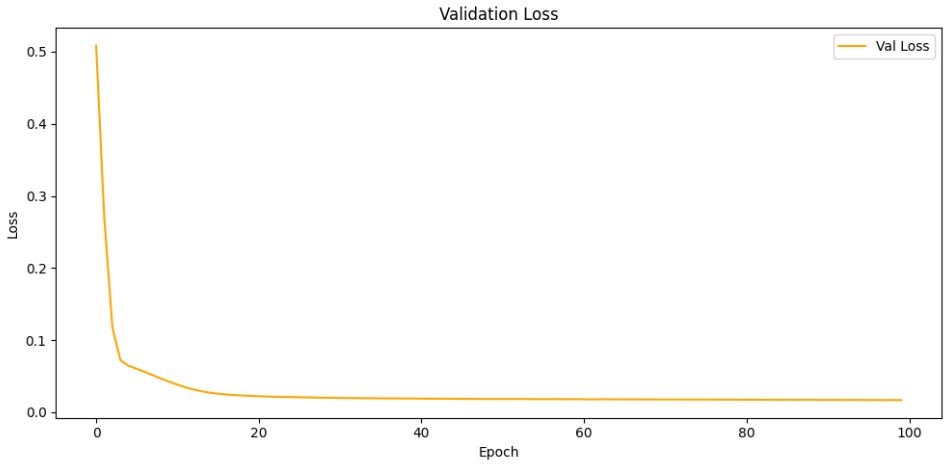

Figura 2.13: Validation Loss Graph

The validation loss graph shows how well the model generalizes to new, unseen data. Like the training loss, it decreases sharply at first and then levels off. The behavior of the validation loss being close to the training loss and remaining relatively stable after the initial drop is a good indicator that the model is not overfitting.

```
plt.plot(train_losses, label='Training Loss')
plt.plot(val_losses, label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training and Validation Loss')
plt.legend()
plt.tight_layout()
plt.savefig('Train_inverse_kinematics_graphic/combined_loss.png')
plt.show()
```

Figura 2.14: Combined Loss Graph (Loss and Validation)

This combined graph provides a direct comparison between training and validation losses over the epochs. The closeness of these two lines throughout the training process reaffirms that the model maintains a good balance between learning the training data and generalizing to new data.

2.5 Model Usage

In order to effectively visualize the results of the completed training, we set up the deployment of the model in the virtual environment provided: we in fact used a script designed to interact with a simulated table tennis environment on the basis of a trained model. The script incorporates a client-server architecture facilitated by the Client class, which allows our model to communicate in real-time with the simulation environment, retrieving the state of the game and sending back decisions made by the trained neural network.

Main Function

The core of the script is the run function, where the model is continuously evaluated. First, the script establishes a connection to the simulation server through the Client class, using connection parameters such as host and port, which can be customized via command-line arguments. Here's how the connection setup and model deployment are orchestrated:

```
def main():
    name = 'Wow math is useful!'
    if len(sys.argv) > 1:
        name = sys.argv[1]
    port = DEFAULT_PORT
    if len(sys.argv) > 2:
```

```
    port = sys.argv[2]
host = 'localhost'
if len(sys.argv) > 3:
    host = sys.argv[3]

cli = Client(name, host, port)
run(cli)
```

Crucial Functions

There are a few functions in the script that play crucial roles in enabling the robot to dynamically interact with the simulated environment based on the model's predictions. These functions are specifically designed to process the simulation data, compute the necessary physics for the ball's trajectory, and determine the optimal robot response. Let's delve into the specific utility functions used:

- `calculate_final_ball_position`: This function simulates the ball's trajectory within the game environment using physics principles like gravity, bounce dynamics, and friction, and based on the physical simulation of its environment it iteratively updates the ball's position and velocity. By predicting the ball's future position, it enables the robotic system to plan and execute movements strategically.

```
def final_ball_position(state):

    bx, by, bz = state[17:20]
    vx, vy, vz = state[20:23]
    g = 9.81  # Acceleration due to gravity
    d = 0.01  # Time interval for the simulation
    restitution = 0.7  # Coefficient of restitution for the bounce
    min_height = 0.001  # Minimum height to consider a bounce
    max_bounces = 2  # Maximum number of bounces
    lateral_friction = 0.3  # Coefficient of lateral friction

    bounces = 0
    minz = 999

    while by > -0.5 and bounces < max_bounces:
        # Update position
        bx += vx * d
        by += vy * d
```

```
            bz += vz * d
            minz = min(minz, bz)

            # Update velocity
            vz -= g * d

            # Check if the ball touches the table
            if bz <= 0:
                # Bounce with energy loss
                bz = -bz * restitution
                # Invert vertical velocity with energy loss
                vz = -vz * restitution
                # Apply lateral friction
                vx *= (1 - lateral_friction)
                # Apply lateral friction
                vy *= (1 - lateral_friction)
                bounces += 1

                # Stop the simulation if the bounce energy is very low
                if abs(vz) < min_height:
                    break

        return bx, by, bz
```

- `calculate_dist`: This function calculates the Euclidean distance between the robot's predicted joint positions and the ball's location. It serves as a crucial metric to assess how accurately the robot aligns itself with the predicted ball position, guiding adjustments to achieve optimal interaction. This function is fundamental for determining the immediacy and precision of the robot's responses. By computing the distance, the system can dynamically adjust its strategies to improve the robot's positioning relative to the ball's trajectory.

```
def calculate_dist(state):
    px, py, pz = state[11:14]  # Robot's joint positions
    bx, by, bz = state[17:20]  # Ball's predicted position
    # Euclidean distance
    dist = math.hypot(px - bx, py - by, pz - bz)
    return dist
```

Run Function

Within the run function, the model's predictions are generated in a loop that continues indefinitely, or until the simulation is stopped. During each iteration of the loop, the script requests the current state of the simulation from the server, calculates the predicted optimal joint movements based on the ball's trajectory using the trained model, and sends these movements back to the server to be executed within the simulation.

The simulation continuously updates and sends the ball and robot states to the client. The `calculate_final_ball_position` function predicts where the ball will land based on physics calculations, allowing the robotic agent to anticipate and respond more effectively, then using the function `calculate_dist` to adapt an evaluating metric to the effectiveness of the response.

Here is the process within the run function:

```
def run(cli):
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    model = RegressionModel().to(device)
    model.load_state_dict(torch.load("model0.pth", map_location=device))
    model.eval()

    while True:
        state = cli.get_state()
        j = [0.0] * JOINTS

        bx, by, bz = final_ball_position(state)

        ballPos_tensor = torch.tensor([bx, by, bz],
            dtype=torch.float32).unsqueeze(0).to(device)

        if state[31] < 0.9:
            j = model(ballPos_tensor)
            j = j.cpu().detach().numpy().flatten()

            if calculate_dist(state) < 0.2:
                j[9] += max(0, 1.5 - calculate_dist(state))
        else:
            j[0:11] = state[0:11]

        # Send the joint positions to the client
        cli.send_joints(j)
```

2.6 Testing

In this phase, the goal is to evaluate the performance of our trained machine learning model using a test data set. This involves making predictions with the model and calculating the mean square error (MSE) to measure its accuracy.

The `CustomDataset` class, mentioned earlier, is responsible for loading and preprocessing data from a CSV file. This class reads the data, separates it into input (ball position and velocity) and target (the desired positions of the robotic arm joints), and converts it into PyTorch tensors for efficient processing.

The `test_model` function is critical to evaluate the performance of the trained model. First, the model is set to evaluation mode with `model.eval()`, which ensures that certain layers behave appropriately during testing. This step is essential to ensure that the model's predictions are accurate and unaffected by training-specific behaviors.

Within a `torch.no_grad()` context, which disables gradient calculations to save memory and computing power during testing, the function iterates over the test `DataLoader`. For each set of inputs and targets, the model generates outputs (predictions). These predictions, along with the actual target values, are collected in lists. After all batches are processed, these lists are converted to numpy arrays to facilitate later analysis.

```
def test_model(model, dataloader_test):
    model.eval()
    predictions = []
    real_values = []

    with torch.no_grad():
        for inputs_test, targets_test in dataloader_test:
            outputs_test = model(inputs_test)
            predictions.extend(outputs_test.tolist())
            real_values.extend(targets_test.tolist())

    return np.array(predictions), np.array(real_values)
```

The main function orchestrates the entire testing procedure. It begins by specifying the path to the CSV file containing the test data. Using this file path, a `CustomDataset` instance is created, which loads and preprocesses the test data. A `DataLoader` is then instantiated for the test dataset, enabling efficient batch processing.

Next, the model is initialized and its pre-trained weights are loaded from a pre-trained model (`model0.pth`). This step ensures that the model parameters are set to the values that performed best during training.

With the model and data ready, the `test_model` function is called to generate predictions and collect the corresponding real target values. These predictions and real values are then used to compute the Mean Squared Error (MSE), a common metric for regression problems. The MSE measures the average squared difference between the predicted values and the actual values, providing a clear indication of the model's accuracy. A lower MSE indicates that the model's predictions are closer to the actual target values.

After conducting the model evaluation, the results have been encouraging, achieving a Mean Squared Error (MSE) value of just 0.8578. While this final result indicates a strong performance, there is potential for further improvement through additional techniques to enhance generalization. Despite this, the current performance is satisfactory for our specific problem context. It is also worth noting that the test dataset used for the evaluation is more diverse and generic compared to the one used during the training phase. This suggests that our model maintains robust performance even when exposed to a broader range of data, underscoring its reliability and effectiveness in real-world applications.

```
def main():
    file_path_test = 'datasets/All_positions_dataset_test.csv'

    dataset_test = CustomDataset(file_path_test)

    batch_size_test = 128
    dataloader_test = DataLoader(dataset_test,
        batch_size=batch_size_test, shuffle=False)

    model = RegressionModel()
    model.load_state_dict(torch.load('model0.pth'))

    predictions, real_values = test_model(model, dataloader_test)

    mse_loss = nn.MSELoss()
    mse = mse_loss(predictions, real_values).item()
    print(f"MSE: {mse:.4f}")
```

3 Reinforcement Learning

In this chapter multiple aspects of the project will be discussed: first of all it is indeed important to remember which is the focus of the reinforcement learning in this implementation and which objective has to be reached. In fact we already saw how, thanks to the implementation of inverse kinematics dynamics, the defined supervised model implemented for our robotic arm is able to locate and reach the game ball correctely: the focus for the reinforcement learning is now to correctely teach the agent (the robotic arm) how to succesfully hit the already located ball in order to score points to win table tennis games. It is so evident which kind of breakdown has been done, in order to divide the project ”problem” into multiple subproblems, and the ones related to the reinforcement learning approach will be discussed in the following: in order to begin with the explaination of the steps done it is necessary to introduce some aspects regarding the nature of the approach that has been done, beginning with an introduction to DDPG.

3.1 Deep Deterministic Policy Gradient (DDPG)

Deep Deterministic Policy Gradient (DDPG) represents a significant stride in reinforcement learning techniques, particularly tailored for complex environments where actions are not merely discrete but continuous: it is a sophisticated reinforcement learning algorithm that bridges the gap between the foundational principles of Q-learning (a predominantly value-based approach) and the versatility required in continuous action domains, employing policy gradient methods. This synergy allows DDPG to effectively manage tasks that demand a more sophisticated control than what traditional discrete action spaces provide.

3.1.1 Key Characteristics of DDPG

DDPG adapts the actor-critic model, integrating two primary components (networks) within its framework:

- Actor Network: This neural network model, also known as the policy network, directly maps states to actions. It is termed ”deterministic” in this context because, unlike stochastic policies that produce a distribution over actions, it outputs the specific action that maximizes the policy’s performance for a given state. This characteristic is crucial in

continuous action spaces where the aim is to identify precise actions rather than selecting from a set of possibilities.

- Critic Network: While the actor proposes actions to take, the critic network evaluates these actions by computing the value function. This model assesses the actor's proposed actions within the context of the current environment, providing crucial feedback on the quality of the actions suggested by the actor. Essentially, the critic's role is to estimate the expected reward of taking a particular action in a given state, hence guiding the actor toward more optimal behaviors.

Given this architectural design it's crucial to note that Deep Deterministic Policy Gradient uniquely combines Q-learning principles within a policy gradient framework, meaning that involves directly optimizing the policy function by adjusting the parameters of the policy in a way that maximizes the expected reward over time; in fact the DDPG algorithm updates its policies using gradient ascent, a core technique in policy gradient methods focused on maximizing a reward signal. Concurrently DDPG employs a value-based approach, where the gradient updates depend on Q-values provided by the critic network: this integration allows DDPG to optimize actions continuously in environments with continuous action spaces while considering both immediate and future rewards. This dual strategy enables DDPG to effectively balance and enhance the learning process, improving both the stability and efficacy of the training.

3.1.2 DDPG Implementation

DDPG Class: Actor and Critic Models

The DDPG class initializes two pairs of models: the actor and its target, and the critic and its target. Each model pair facilitates the learning process, with the target models providing a stable baseline for updates.

```
class DDPG(object):
    def __init__(self, gamma, tau, hidden_size, num_inputs,
    action_space):
        self.actor = Actor(hidden_size, num_inputs,
            action_space).to(device)
        self.critic = Critic(hidden_size, num_inputs,
            action_space).to(device)
        self.actor_target = Actor(hidden_size, num_inputs,
            action_space).to(device)
        self.critic_target = Critic(hidden_size, num_inputs,
            action_space).to(device)
        ...
```

The primary purpose of having separate target networks (actor target and critic target) is to stabilize the training process: in reinforcement learning, especially in environments with continuous action spaces, directly updating the actor and critic networks from highly correlated inputs and outputs can lead to significant instability. This instability arises because updates to the networks can drastically change the policy and the value estimates, leading to feedback loops between the policy and value function updates. In fact, to solve this kind of problems, target networks are copies of the main networks, but their parameters are updated less frequently or more slowly: since the target networks lag behind the main networks, they provide a set of older but more ”stable” parameters to generate target values for updating the main networks. This temporal decoupling means that the learning updates for both the actor and the critic are based on slightly outdated but more stable data, rather than on the most current, possibly highly fluctuating data.

Later in the class definition, after setting up the Adam optimizer fot both the critic and actor networks and before setting up the directory to save the models, it has to be ensured that both target networks presents coherent weights, and that is realized by executing a so called ”hard update”.

```
hard_update(self.actor_target, self.actor)
hard_update(self.critic_target, self.critic)
```

Hard updates, differently from soft updated which will be later discussed, directly copy the weights from the main networks (actor or critic) to their corresponding target networks. This process is in fact typically done at the initialization stage to synchronize the target networks with the main networks right at the start of training.

Action Evaluation

The `calc_action` method evaluates the best action to take in a given state by querying the actor model. It optionally adds noise for exploration purposes and ensures the action remains within valid bounds.

```
def calc_action(self, state, action_noise=None):
    self.actor.eval()
    mu = self.actor(state.to(device))
    self.actor.train()
    if action_noise is not None:
        noise = torch.Tensor(action_noise.noise()).to(device)
        mu += noise
    mu = mu.clamp(self.action_space.low, self.action_space.high)
    return mu
```

Learning Process

The `update_params` function adjusts the actor and critic networks based on sampled batches from the environment. This function applies a soft update rule to gradually align the target networks with the primary networks, stabilizing learning.

```
def update_params(self, batch):
    state_action_values = self.critic(state_batch,
    self.actor(state_batch))
    policy_loss = -state_action_values.mean()
    ...
    soft_update(self.actor_target, self.actor, self.tau)
    soft_update(self.critic_target, self.critic, self.tau)
```

Model Persistence

To facilitate training across multiple sessions, the DDPG class includes methods for saving and loading model checkpoints, ensuring that training progress can be preserved and resumed.

```
def save_checkpoint(self, last_timestep, replay_buffer):
    checkpoint = {'actor': self.actor.state_dict(), 'critic':
        self.critic.state_dict(), ...}
    torch.save(checkpoint, self.checkpoint_dir + '/model.pth')

def load_checkpoint(self, checkpoint_path):
    checkpoint = torch.load(checkpoint_path)
    self.actor.load_state_dict(checkpoint['actor'])
    self.critic.load_state_dict(checkpoint['critic'])
    ...
```

Operational Modes

The class provides methods to toggle between training and evaluation modes, optimizing performance and behavior depending on the current phase of the learning process.

```
def set_eval(self):
    self.actor.eval()
    self.critic.eval()

def set_train(self):
    self.actor.train()
    self.critic.train()
```

This detailed implementation of the DDPG algorithm showcases the complexity and robustness required for effective reinforcement learning in continuous action environments, allowing the system to learn nuanced strategies over extended interactions.

3.2 Crucial Classes

There are a couple of classes which turned out to be pivotal in achieving a coherent reinforcement learning process. These include mechanisms for managing ”replay memory,” adding learning ”noise,” and defining the network architectures themselves:

- **Replay Memory:** Replay Memory plays a fundamental role in reinforcement learning by breaking the temporal correlations within the sequence of observed data. This technique is essential for stabilizing the learning updates and preventing the deleterious effects of highly interdependent data inputs. The replay memory stores transitions collected from the environment, each consisting of the current state, the action taken, the reward received, the next state, and a flag indicating whether the episode has terminated. These transitions are stored in a circular buffer, ensuring that the memory does not exceed its defined capacity.

```
Transition = namedtuple('Transition', ('state', 'action',
'done', 'next_state', 'reward'))
    class ReplayMemory(object):
        def __init__(self, capacity):
            self.capacity = capacity
            self.memory = []
            self.position = 0

        def push(self, *args):
            """Saves a transition."""
            # Expands the memory array if below capacity
            if len(self.memory) < self.capacity:
                self.memory.append(None)
            self.memory[self.position] = Transition(*args)
             # Circular buffer
            self.position = (self.position + 1) % self.capacity

        def sample(self, batch_size):
            """Randomly samples a batch of transitions for
            training."""
```

```
        return random.sample(self.memory, batch_size)
```

The ability to randomly sample from the replay memory allows for the training process to benefit from learning across a diverse array of scenarios, rather than being restricted to sequential order, which could lead to suboptimal learning dynamics. This method also mimics the idea of experience replay, where the learning algorithm can learn from past experiences repeatedly, thus significantly improving the learning efficiency and effectiveness.

- **Noise:** The implementation of noise in the learning process is critical for ensuring sufficient exploration of the environment by the learning agent. There were multiple kinds of noises to possibly adapt throughout the implemenation, but one of them only will be considered in this explaination as a reperentative example of the others.

```
class OrnsteinUhlenbeckActionNoise:
def __init__(self, mu, sigma, theta=.15, dt=1e-2, x0=None):
    self.theta = theta
    self.mu = mu
    self.sigma = sigma
    self.dt = dt
    self.x0 = x0
    self.reset()

def noise(self):
    x = self.x_prev + self.theta * (self.mu - self.x_prev)
        * self.dt \ + self.sigma * np.sqrt(self.dt) *
        np.random.normal(size=self.mu.shape)
    self.x_prev = x
    return x

def reset(self):
    self.x_prev = self.x0 if self.x0 is not None
    else np.zeros_like(self.mu)
```

This method updates the noise value based on the Ornstein-Uhlenbeck process: it calculates new noise values by considering the current state of the noise, the mean level it should revert to, and the defined volatility. The resultant noise is not just random; it incorporates elements from its previous state, state that can be resetted with a dedicated

function, ensuring that the noise generated is correlated over time. This temporal correlation enhances the effectiveness of the exploration process by preventing excessively unexpected behaviors that could interfere with the learning process.

3.3 Reinforcement Learning Implementation

3.3.1 Crucial Functions

Function: `get_input_actor`

The `get_input_actor` function is designed to prepare the relevant state information for the actor model by extracting specific features from the overall state of the environment. These elements retrieved from the state array are then concatenated into a single array, `relevant_state`, which serves as the input to the actor network. This array contains the critical information needed by the actor model to determine the next action based on the current state of the environment.

```
def get_input_actor(state):
    paddle_position = state[11:14]  #Paddle center position(x, y, z)
    paddle_normal = state[14:17]  #Paddle normal vector(x, y, z)
    ball_position = state[17:20]  #Current ball position(x, y, z)
    ball_velocity = state[20:23]  #Current ball velocity(x, y, z)
    paddle_pitch_joint = [state[9]]  #Convert to list to concatenate

    relevant_state = np.concatenate((paddle_position,
                                     paddle_normal,
                                     ball_position,
                                     ball_velocity,
                                     paddle_pitch_joint))

    return relevant_state
```

Function: `where_ball_will_touch`

The `where_ball_will_touch` function simulates the trajectory of the ball to determine where it will touch the ground based on its current state and considering the game field dimensions. The function retrieves the ball's current position and velocity from the state array and defines constants for gravity, the time step for the simulation, and the maximum number of iterations to prevent infinite loops.

During the simulation loop, the function iteratively updates the ball's position and velocity until the ball touches the ground or the maximum number of iterations is reached: this involves

calculating the ball's new position and velocity at each time step, taking into account the physics effects accounted in the simulation environment, making it possible to predict where the ball will land, which is crucial for the robot's decision-making process (that we remember is also based on inverse kinematiks).

```
def where_ball_will_touch(state):
    opponent_table_length = 1.2
    my_table_length = 1.2
    opponent_table_width = 1.4

    bx, by, bz = state[17:20]
    vx, vy, vz = state[20:23]
    g = 9.81  # Acceleration due to gravity
    d = 0.05  # Time step for simulation
    max_iterations = 1000  # To prevent infinite loops

    iterations = 0

    while bz > 0 and iterations <= max_iterations:
        bx += vx * d
        by += vy * d
        bz += vz * d
        vz -= g * d
        iterations += 1

    if iterations == max_iterations:
        logger.warning("Trajectory overflow")

    return bx, by, bz
```

Function: `final_ball_position`

The `final_ball_position` function predicts the final position of the ball after it has potentially bounced multiple times. This function is crucial for simulating the ball's behavior in a table tennis game, accounting for gravity, bounces, and lateral friction. Here's how it works:

First, the function extracts the ball's current position and velocity from the `state` array:

```
def final_ball_position(state):
    bx, by, bz = state[17:20]
    vx, vy, vz = state[20:23]
```

Several constants are defined to simulate the physical environment:

```
g = 9.81  # Gravity acceleration
d = 0.01  # Simulation time interval
restitution = 0.7  # Restitution coefficient for bounce
min_height = 0.001  # Minimum height to consider a bounce
max_bounces = 2  # Maximum number of bounces
attrito = 0.3  # Lateral friction coefficient
```

The function then initializes counters for the number of bounces and the minimum height:

```
bounces = 0
minz = 999
```

A loop is used to simulate the ball's trajectory until it hits the ground and potentially bounces:

```
while by > 0 and bounces < max_bounces:
    # Update position
    bx += vx * d
    by += vy * d
    bz += vz * d
    minz = min(minz, bz)

    vz -= g * d
    if bz <= 0:
        # Bounce with energy loss
        bz = -bz * restitution
        # Invert vertical velocity with energy loss
        vz = -vz * restitution
        vx *= (1 - attrito)
        vy *= (1 - attrito)
        bounces += 1
        if abs(vz) < min_height:
            break
```

Finally, the function returns the predicted final position of the ball:

```
return bx, by, bz
```

The primary difference between `final_ball_position` and `where_ball_touched` lies in their purposes and implementations.

`final_ball_position` is designed to simulate the ball's entire trajectory, including bounces, until it comes to rest or is no longer relevant for the simulation: this function accounts

for the ball's position updates, gravity's effect on vertical velocity, and the impact of bounces by considering a restitution coefficient and lateral friction. It also ensures that the ball's behavior after each bounce is physically accurate, making it crucial for scenarios where multiple bounces and detailed ball behavior are relevant.

`where_ball_touched`, on the other hand, is focused on predicting the ball's landing position when it touches the ground for the first time: this function iterates until the ball hits the ground, ignoring subsequent bounces and finer details of the ball's behavior post-impact. It is simpler and serves to quickly determine where the ball will touch down without the need for detailed bounce simulations.

Function: `get_joint_position_supervised`

The `get_joint_position_supervised` function leverages the supervised learning model we already discussed about to predict the joint positions needed to intercept the ball based on its predicted final position (inverse kinematiks). Initially, the function sets up a list, `j`, to store the joint positions: it then calls the `final_ball_position` function to compute where the ball is expected to land, in order to convert this final position into a PyTorch tensor that will be transferred to the appropriate device (either CPU or GPU).

If the ball has not yet touched the robot, indicated by `state[31] < 0.9`, the function uses the inverse kinematics model (`ik_model`) to predict the necessary joint positions: these predicted positions are then processed and assigned to the list `j`.If the ball has already touched the robot, the function retains the current joint positions from the state array: this ensures that the robot's movements are updated accurately to interact with the ball based on real-time data.

```
def get_joint_position_supervised(ik_model, state):
    j = [0.0] * JOINTS

    bx, by, bz = final_ball_position(state)

    ball_pos = torch.tensor([bx, by, bz], dtype=torch.float32).
        unsqueeze(0).to(device)

    if state[31] < 0.9:
        j = ik_model(ball_pos)
        j = j.cpu().detach().numpy().flatten()
        j[10] = math.pi / 2
    else:
        j[0:11] = state[0:11]
    return j
```

Function: `min_distance_paddle_ball`

The `min_distance_paddle_ball` function calculates the Euclidean distance between the current positions of the ball and the paddle. This distance is essential for determining how close the ball is to the paddle, which can inform decisions about whether to attempt a hit or move the paddle for better positioning.

```
def min_distance_paddle_ball(state):
    ball_position = state[17:20]
    paddle_position = state[11:14]
    return np.linalg.norm(paddle_position - ball_position)
```

This function works by extracting the positions of the ball and the paddle from the `state` array and then using the `np.linalg.norm` function from the NumPy library to compute the Euclidean distance between these two points.

Function: `robot_paddle_touched_ball`

The `robot_paddle_touched_ball` function determines whether the robot's paddle has touched the ball. This is crucial for updating the game state and potentially awarding points.

```
def robot_paddle_touched_ball(state):
    vx, vy, vz = state[20:23]
    paddle_position = state[11:14]
    ball_position = state[17:20]
    ball_touched_your_robot = state[31]

    return ball_touched_your_robot > 0.9 and vy > 0
```

This function checks two conditions: The `ball_touched_your_robot` flag must be greater than 0.9, indicating that the ball has indeed touched the robot's paddle and the ball's y-velocity (`vy`) must be positive, meaning the ball is moving away from the robot after the touch.

Function: `opponent_field_is_touched`

The `opponent_field_is_touched` function checks if the ball has landed in the opponent's side of the table. This is useful for determining whether a serve or return has successfully landed in play.

```
def opponent_field_is_touched(bx, by, bz):
    opponent_table_length = 1.2
```

```
    my_table_length = 1.2
    opponent_table_width = 1.4

    return by > my_table_length and by <= my_table_length +
        opponent_table_length and bx >= -opponent_table_width
        / 2 and bx <= opponent_table_width / 2
```

This function checks whether the ball's y-position (`by`) is within the range that corresponds to the opponent's half of the table and whether the x-position (`bx`) is within the table's width.

Function: `my_field_is_touched`

The `my_field_is_touched` function checks if the ball has landed on the player's side of the table. This is important for determining if a serve or return has failed to cross the net or has been returned back to the player's side.

```
def my_field_is_touched(bx, by, bz):
    my_table_length = 1.2
    table_width = 1.4

    return 0 <= by <= my_table_length and -table_width / 2
        <= bx <= table_width / 2
```

Similar to the previous function, it checks whether the ball's y-position (`by`) is within the player's half of the table and the x-position (`bx`) is within the table's width.

Function: `missed_ball`

The `missed_ball` function determines whether the ball has been missed by the robot's paddle. This can be used to decide if a point should be awarded to the opponent or if the game state should be updated to reflect a missed shot.

```
def missed_ball(state):
    bx, by, bz = state[17:20]  # Ball position
    vx, vy, vz = state[20:23]  # Ball velocity
    px, py, pz = state[11:14]  # Paddle position

    # Define a buffer for the paddle's reach in each direction
    buffer_x = 0.1
    buffer_y = 0.1
    buffer_z = 0.1
```

```
    # Check if the ball is past the paddle's y position
    if by < py:
        # Further check if the ball is within the buffer
        range in x and z direction
        if not (px - buffer_x <= bx <= px + buffer_x
            and pz - buffer_z <= bz <= pz + buffer_z):
            # Consider the ball missed if it's not
            within the paddle's reach
            return True

    # Additional check if the ball's velocity indicates
    it is moving away from the paddle
    if vy < 0:
        return True

    return False
```

This function first checks if the ball's y-position (`by`) is less than the paddle's y-position (`py`), indicating the ball has passed the paddle. It then checks if the ball's x and z positions (`bx` and `bz`) are within a buffer range around the paddle's x and z positions (`px` and `pz`). If the ball is not within this buffer, it is considered missed. Additionally, if the ball's y-velocity (`vy`) is negative, indicating it is moving away from the paddle, the ball is also considered missed.

Function: `compute_reward`

The `compute_reward` function is designed to calculate the reward for the agent based on the current state of the game. This function evaluates where the ball landed and assigns a reward value accordingly. The reward system is crucial for guiding the agent's learning process in reinforcement learning.

```
def compute_reward(state):

    bx, by, bz = where_ball_touched(state)

    opponent_table_length = 1.2
    my_table_length = 1.2
    opponent_table_width = 1.4
    target_x = 0.0
    target_y = my_table_length + opponent_table_length / 2
```

```
# Calculate distance from opponent field center
center_opp_field = np.sqrt((bx - target_x)
** 2 + (by - target_y) ** 2)


if my_field_is_touched(bx, by, bz):
    logger.info("Ball in your field")
    if by > 0.7:
        return -100
    return -500

# Check if the ball landed in the opponent's field
if opponent_field_is_touched(bx, by, bz):
    logger.info("Ball in")
    return max(0, 100 - center_opp_field)


logger.info("Ball out")
return -300
```

First, the function calls `where_ball_touched` to determine the final position of the ball. It then calculates the distance from the center of the opponent's field, which is used to determine the reward.

- If the ball lands in the player's own field, the function checks the ball's y-coordinate to decide the penalty. A ball landing deep in the player's field results in a smaller penalty (`-100`), while a ball landing closer results in a larger penalty (`-500`). - If the ball lands in the opponent's field, a positive reward is given, which is higher the closer the ball is to the center of the opponent's field. - If the ball goes out of bounds, a penalty of `-300` is assigned.

This reward structure encourages the agent to aim for the opponent's field and penalizes mistakes that cause the ball to land in the player's field or out of bounds.

Function: `check_done`

The `check_done` function checks whether the current episode of the game should be terminated. This function helps manage the flow of the game by identifying when a point has been scored or when the ball is out of play. In the final version of the project this function has been unused because within the training cycle has been implemented a new logic to chek if the episo can be considered done.

```
def check_done(state, next_state):
```

```
    current_score_robot = state[34]
    current_score_opponent = state[35]
    next_score_robot = next_state[34]
    next_score_opponent = next_state[35]

    if next_score_robot > current_score_robot or
    next_score_opponent > current_score_opponent:
        return True

    ball_in_field = next_state[29] == 1 or next_state[32] == 1
    if not ball_in_field:
        return True

    return False
```

This function compares the scores before and after an action: - If the robot's score has increased or the opponent's score has increased, it means a point has been scored, so the function returns `True` indicating the episode is done. - It also checks if the ball is still in play by examining the state flags that indicate whether the ball is in either player's half of the field (`next_state[29]` and `next_state[32]`). If the ball is not in the field, the function returns `True`.

By determining when an episode is finished, this function ensures that the game state is appropriately managed, allowing for new episodes to begin and the learning process to continue smoothly.

3.3.2 Run Function

The `run` function is the core component that orchestrates the reinforcement learning process for training the robotic arm to play table tennis. It involves setting up the necessary parameters, initializing the DDPG agent, managing the replay memory, and running the training episodes.

The function starts by defining the directory to save checkpoints and with the setting of several hyperparameters, including the seed for reproducibility, the discount factor (`gamma`), the target update rate (`tau`), the hidden layer sizes for the neural networks, the size of the replay memory, the standard deviation for noise, and the batch size. The random seed is set for both PyTorch and NumPy to ensure consistent results across runs.

```
def run(cli):
    ...
    setting of checkpoint directory and necessary parameters
    ...
```

The DDPG agent is then defined and built with the specified parameters. The observation space and action space are set based on the environment's requirements, and the DDPG agent is initialized with these parameters.

```
# Define and build DDPG agent
observation_space = 13
action_space = np.array(joint_ranges)

agent = DDPG(gamma,
             tau,
             hidden_size,
             observation_space,
             action_space,
             checkpoint_dir=checkpoint_dir
             )
```

Next, the replay memory and the Ornstein-Uhlenbeck noise process are initialized. The replay memory stores past experiences to facilitate experience replay, while the noise process ensures exploration by adding noise to the actions. Additionally, an inverse kinematics (IK) model is loaded and set to evaluation mode.

```
# Initialize replay memory
memory = ReplayMemory(int(replay_size))

# Initialize OU-Noise
nb_actions = action_space.shape[0]
ou_noise = OrnsteinUhlenbeckActionNoise(mu=np.zeros(nb_actions),
    sigma=float(noise_stddev) * np.ones(nb_actions))

# Initialize IK model
model = RegressionModel().to(device)
model.load_state_dict(torch.load("model0.pth",
    map_location=device))
model.eval()
```

Moreover is possible to consider eventual saved checkpoints in a specified directory,so that the function loads the latest checkpoint to resume training from where it left off or, otherwise, it starts training from scratch. Later various counters and variables needed for training are then initialized, such as lists to store rewards and losses, and the number of training episodes and maximum timesteps per episode are setted.

```
    # Define counters and other variables
    rewards, policy_losses, value_losses, mean_test_rewards = [], [], [],
    epoch = 0
    t = 0
    time_last_checkpoint = time.time()

    num_episodes = 1000  # Number of episodes to train
    max_timesteps = 1000  # Maximum timesteps per episode
```

For each episode, the function retrieves the initial state from the client, processes it to extract relevant features, and resets the noise process. Various flags and counters are also reset. The function then iterates through each timestep of the episode, updating the joint positions based on the IK model and calculating actions using the DDPG agent when appropriate.

```
for episode in range(num_episodes):
    init_state = cli.get_state()
    filtered_state = get_input_actor(init_state)
    filtered_state = torch.Tensor(filtered_state).unsqueeze(0)

    ou_noise.reset()
    episode_reward = 0

    action_done = False
    push_done = False
    done = False

    for timestamp in range(max_timesteps):

        j = get_joint_position_supervised(model, init_state)

        if min_distance_paddle_ball(
            init_state) < 0.2 and not action_done:
            logger.info("Action done")
            action = agent.calc_action(filtered_state,
                action_noise=ou_noise)
            new_joints = action.squeeze(0)
            j[9] = new_joints[0]
            action_done = True

        cli.send_joints(j)
```

At each timestep, the function retrieves the next state from the client, processes it, and checks if the robot's paddle touched the ball or missed it. If either event occurs, it computes the reward, updates the replay memory, and breaks out of the loop. The state is updated, and if the replay memory contains enough experiences, the agent updates its parameters using a batch of transitions. Rewards and losses are logged, and checkpoints are saved periodically.

```
# Assuming you get the next state after performing the action
    next_state = cli.get_state()
    filtered_next_state = get_input_actor(next_state)
    filtered_next_state = torch.
         Tensor(filtered_next_state).unsqueeze(0)

    if robot_paddle_touched_ball(next_state)
        and action_done and not push_done:
        logger.info("Touched robot paddle")

        done = True
        reward = compute_reward(next_state)
        reward = torch.Tensor([reward])
        done_tensor = torch.Tensor([done])
        logger.info("Pushing to replay buffer, reward: %s", reward)
        memory.push(filtered_state, action,
            done_tensor, filtered_next_state, reward)
        push_done = True
        episode_reward += reward.item()
        break  # Exit the loop immediately

    elif action_done and missed_ball(next_state) and not push_done:
        logger.info("Missed robot paddle")
        done = True
        reward = -200
        reward = torch.Tensor([reward])
        done_tensor = torch.Tensor([done])
        logger.info("Pushing to replay buffer, reward: %s", reward)
        memory.push(filtered_state, action, done_tensor,
            filtered_next_state, reward)
        push_done = True
        episode_reward += reward.item()
        break  # Exit the loop immediately
```

```
    filtered_state = filtered_next_state
    init_state = next_state
    t += 1

    if len(memory) > batch_size:
        transitions = memory.sample(batch_size)
        batch = Transition(*zip(*transitions))
        value_loss, policy_loss = agent.update_params(batch)
        value_losses.append(value_loss)
        policy_losses.append(policy_loss)

rewards.append(episode_reward)
writer.add_scalar('Reward/Episode', episode_reward, episode)

logger.info('Episode %d/%d, Reward: %f\n', episode + 1,
    num_episodes, episode_reward)

# Save checkpoint every 100 episodes
if (episode + 1) % 100 == 0:
    agent.save_checkpoint(t, memory)

writer.close()
logger.info("Training completed.")
```

Overall, the `run` function integrates the DDPG agent with the table tennis simulation environment, managing the training process, updating the agent, and logging performance metrics.

4 Failed Approaches

During the course of this project, our team has significantly deepened our understanding of Machine Learning principles and applications. This journey was not without its challenges; in fact, we made several fundamental errors along the way, even initially, when we failed to decompose the primary problem into manageable subproblems, which led to conceptual and implementation errors.

However, these mistakes were valuable learning opportunities: for instance, our first attempt at implementing the inverse kinematics model was unsuccessful, but this failure paved the way for a successful second attempt. This iterative process underscored the importance of thorough data analysis and proper problem structuring in Machine Learning projects.

Moreover, we discovered that integrating inverse kinematics with specific mathematical functions was an effective strategy, and in fact this approach enabled us to accurately time the ball strikes, demonstrating the practical benefits of combining theoretical models with precise computational methods. Through these experiences, we have gained a clearer understanding of the critical role that data and structured methodologies play in the successful application of Machine Learning techniques.

During our initial attempt with the mathematical approach, we developed and tested numerous functions, such as those for ball trajectory prediction, and this process involved extensive trial and error, allowing us to refine our methods and deepen our understanding of the underlying principles.

In conclusion, the 8th group can confidently state that we have gained substantial knowledge throughout this project. Although we may not win the competition, the lessons we've learned are invaluable and can be applied to future endeavors. This experience has reinforced the importance of perseverance, continuous learning, and the practical application of theoretical concepts.

Elenco delle figure