Ball interception for soccer robots using reinforcement learning in Unreal Engine

Bachelor End Project
Mechanical Engineering Robotics

Full Name	Student ID
Tycho Brouwer	1753320

Supervisors:
A.S. Deogan
M.J.G. van de Molengraft

Eindhoven, January 26, 2025

Contents

Abstract

This study investigates the application of reinforcement learning (RL) to improve ball interception performance in soccer robots for the RoboCup Middle Size League, leveraging the Unreal Engine as a simulation environment. The research addresses the need for complex dynamical models compared to simple models in RL. It highlights the potential trade-offs between model complexity and performance. A simplified robot controller model is implemented, and training is performed using the Learning Agents plugin with the Proximal Policy Optimization (PPO) algorithm. Experimental results demonstrate that RL can produce efficient ball interception trajectories, achieving a success rate of up to 77% under varying conditions. The findings underscore the feasibility of using RL to adapt to dynamic and competitive environments. The study suggests future research to explore alternative simulation platforms and hybrid approaches that integrate RL with traditional control techniques to further enhance performance. The code and scripts are available at https://github.com/tychobrouwer/soccer-robot-rl-ball-interception.

1 Introduction

Robot soccer presents unique challenges for adaptive behaviour in dynamic and unstructured environments. It continues to drive innovation by creating an engaging and competitive platform for research groups to compete [1–3]. RoboCup states the ambitious goal of developing fully autonomous robot teams capable of competing at human professional levels by 2050 [4]. Tech United developed robots, called TURTLEs, based on three-wheeled omnidirectional robots. These robots are designed to move in all directions and are equipped with a specialized ball handling and shooting system [5].

Efficient ball interception is a fundamental capability for soccer-playing robots. There is a lot of literature on interception path generation for various applications [6–9]. For interception, the calculations required are quite complex as they need to account for the dynamic behaviour of the ball, therefore current solutions for ball interception in the RoboCup, such as [10], make various assumptions to reduce the computational costs. Other solutions proposed in the past for the simulation RoboCup league use quantitative methods for interception of the ball [11].

Reinforcement learning (RL) has emerged as a promising approach to addressing these challenges in robotics. By enabling systems to learn effective strategies through interaction with their environment, RL provides an alternative to traditional analytical methods. Its usage has also been proposed for use within the RoboCup soccer robots [12, 13]. Also, its feasibility is partially discussed in [14] for ball interception.

This study investigates the use of reinforcement learning (RL) to improve the ball interception performance of Tech United soccer robots. Traditional interception strategies often rely on estimations of the robot's movement capabilities to simplify the complex dynamics involved. RL offers an alternative approach by enabling robots to learn efficient interception behaviours through the training of a model, bypassing the need to design and calculate accurate analytical algorithms for solving the interception path. While RL still involves real-time calculations during inference, it removes the need for computationally intensive modelling. This allows for faster development and flexibility in dynamic scenarios, leveraging learned strategies rather than pre-defined equations.

One focus of this study is to examine the need for complex dynamical models compared to simple models for use in reinforcement learning. Simplified models, while less accurate, can reduce computational costs and training complexity by capturing the only required dynamics. This approach allows RL to focus on learning residual behaviours, potentially improving training efficiency and the interpretability of the resulting system. The research investigates whether simplified models are sufficient for effective ball interception, balancing accuracy with practicality.

Another aim is to create a reinforcement learning platform that can provide training a model for a practical and effective solution for real-time ball interception. By enabling the robots to learn optimal interception paths, RL has the potential to improve their speed and adaptability in dynamic environments.

2 Conceptual Framework

2.1 Characteristics of Omnidirectional Robots

The soccer robots currently used by TechUnited are three-wheeled omnidirectional robots, a schematic is shown in Figure 2.1. Omnidirectional robots have a key advantage in that they can perform holonomic motion. In other words, they can traverse in any direction without needing to reorient their base. This capability is achieved through specialized omnidirectional wheels like Mecanum or Omni-wheels.

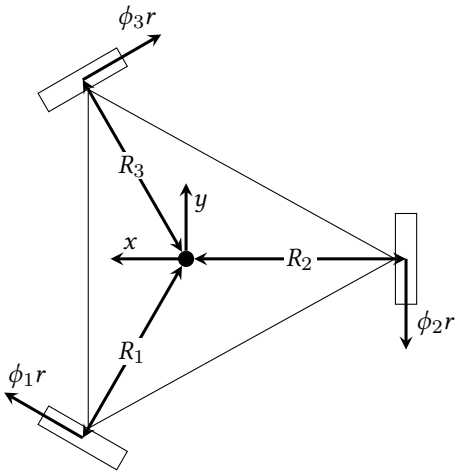


Figure 2.1: Diagram of the three-wheeled omnidirectional soccer robot used by TechUnited for the RoboCup Middle Size.

The current generation of TechUnited robots uses Omni-wheels enabling reliable omnidirectional movement. Omni-wheels consist of a central wheel with smaller rollers mounted around its circumference. These rollers allow the wheel to roll freely in a direction perpendicular to its primary axis of rotation.

Using three Omni-wheels at a 120-degree angle on the robot allows for movement on the x- and y-axes and rotation around the z-axis. The kinematics of this configuration can be determined by the projection of the robot's linear ($\dot{x}$, $\dot{y}$) and angular (ω) velocities onto the direction of the wheel's rotation axis. This gives the rotational speeds of the wheels ϕ as given in Equation 2.1.

$$\begin{bmatrix} \phi_1 \\ \phi_2 \\ \phi_3 \end{bmatrix} = \frac{1}{r} \begin{bmatrix} sin(\theta_1) & -cos(\theta_1) & -R_1 \\ sin(\theta_2) & -cos(\theta_2) & -R_2 \\ sin(\theta_3) & -cos(\theta_3) & -R_3 \end{bmatrix} \begin{bmatrix} \dot{x} \\ \dot{y} \\ \omega \end{bmatrix} = \frac{1}{r} \begin{bmatrix} \frac{\sqrt{3}}{2} & \frac{1}{2} & R_1 \\ 0 & -1 & R_2 \\ -\frac{\sqrt{3}}{2} & \frac{1}{2} & R_3 \end{bmatrix} \begin{bmatrix} \dot{x} \\ \dot{y} \\ \omega \end{bmatrix} \tag{2.1}$$

Inverse kinematics can be used to compute the robot's velocities from the wheel speeds as shown in Equation 2.2.

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \omega \end{bmatrix} = \frac{r}{3} \begin{bmatrix} \sqrt{3} & 0 & -\sqrt{3} \\ 1 & -2 & 1 \\ \frac{1}{R_1} & \frac{1}{R_2} & \frac{1}{R_3} \end{bmatrix} \begin{bmatrix} \phi_1 \\ \phi_2 \\ \phi_3 \end{bmatrix} \tag{2.2}$$

2.2 Principles of Reinforcement Learning

Reinforcement Learning is a machine learning approach in which an agent learns to make decisions by interacting with its environment. The agent takes actions, observes the resulting outcomes, and receives feedback in the form of a reward signal. Over time, the agent learns to optimise its actions to maximise rewards, developing an optimal policy through trial and error [15].

2.2.1 Proximal Policy Optimisation (PPO)

Proximal Policy Optimization (PPO) [16] is a popular algorithm in reinforcement learning where an agent learns to make decisions by interacting with its environment. The agent takes actions based on its current state, and in response, it receives feedback in the form of rewards or penalties.

2.3 Unreal Engine 5.5 for Reinforcement Learning

Unreal Engine (UE) 5.5 is a real-time 3D engine widely recognised for its capabilities in gaming, film, and visualisation [17]. For scientific simulation, it offers tools for creating detailed, interactive, and visually accurate environments.

The physics engine of Unreal Engine, Chaos, allows for simulation of the robot dynamics, providing a realistic environment for the agent to learn and optimize its performance. Chaos models various physical phenomena, including forces, torques, friction, inertia, and impact dynamics [18], which are crucial for replicating real-world interactions. This ensures the agent is trained in an environment that resembles real-world conditions, improving the transferability of the trained model to the real robot.

2.3.1 Blueprints

Blueprints in UE are a visual scripting system designed to simplify the creation of elements, interactions, and logic without requiring extensive coding knowledge [19]. They consist of a node-based interface where users can design and implement logic by connecting nodes that represent actions, events, and data flows. This system is built upon the C++ framework of Unreal Engine, offering both accessibility to beginners and interoperability with C++ code for advanced users.

2.3.2 Actors

Actors in UE possess multiple intrinsic functions that execute in response to specific events, as detailed in [20]. Two functions of relevance are as follows:

- **BeginPlay**: This function is called on the spawning of an actor. It serves to initialize objects, configure initial states, and execute required setup tasks before starting the simulation.
- **Tick**: This function is executed on every frame of the simulation or at predefined intervals. It allows for continuous updates to the actor during the simulation. However, excessive use of the tick function should be avoided when constant updates are unnecessary, as it can negatively impact performance. Instead, custom events are preferred for specific tasks.

2.3.3 Learning Agents Plugin

Reinforcement learning in UE is performed using the Learning Agents plugin [21], an experimental plugin that enables AI agent training using reinforcement learning and imitation learning. The Learning Agents plugin provides an integrated environment in which agents can learn optimal behaviours by interacting with their environment. Using reinforcement learning, agents can learn to make decisions through trial and error, optimizing their actions to achieve specific objectives.

The plugin consists of two main components:

- **C++ Library**: Integrated directly into Unreal Engine, this library provides functionality for defining observations, actions, rewards, and neural network structures. It supports flow control for both training and inference procedures.
- **Python Integration**: Training is performed using PyTorch in the background, a popular machine learning framework. Communication between the Unreal Engine process and the Python process enables real-time training and evaluation.

Users can utilize the C++ API for customized implementations and improved performance. For researchers, the Python integration of the plugin makes it compatible with tools such as TensorBoard and NumPy, which are widely used in machine learning experiments.

In this project, the Learning Agents plugin is particularly useful because it enables the training of reinforcement learning-based ball interception strategies within the simulated environment of Unreal Engine. The integration into Unreal Engine and compatibility with other reinforcement learning implementations make it a practical tool for use in this project.

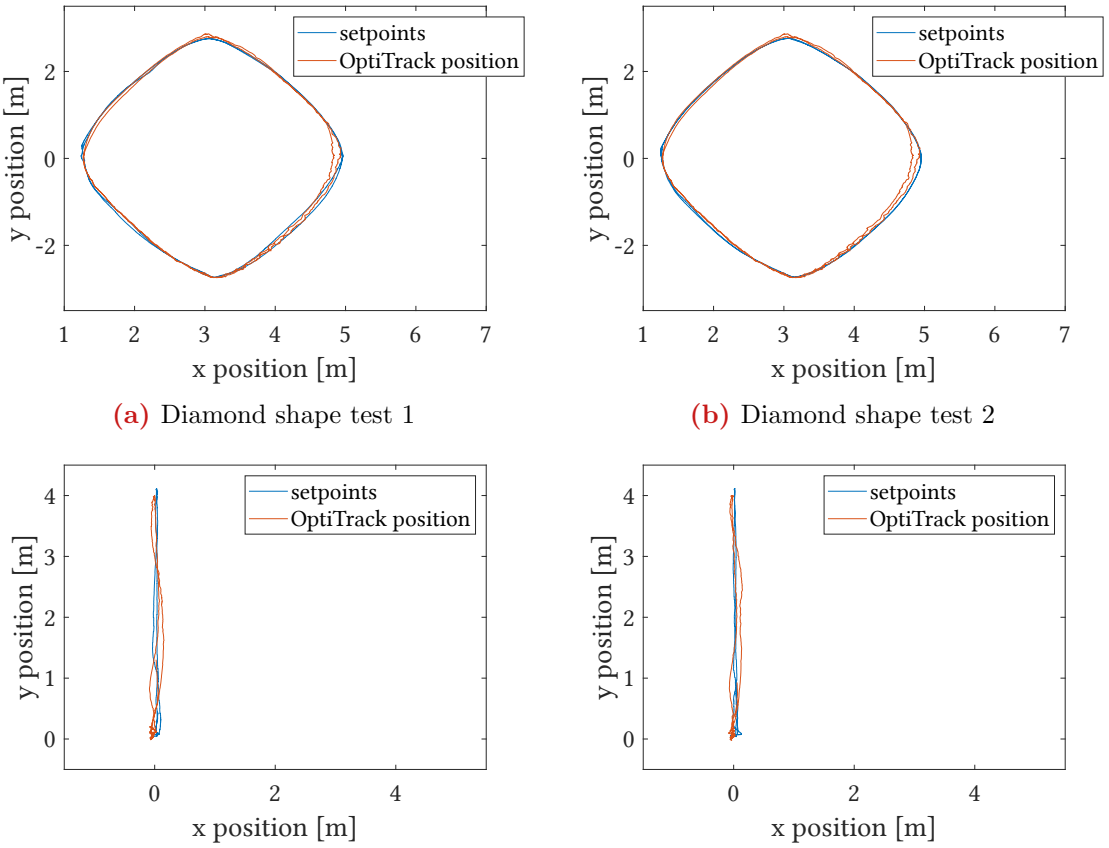
3 Dynamical Model of the Soccer Robots

A sufficiently accurate dynamical model of the soccer robots is required to create a realistic environment for reinforcement learning. By incorporating well-known dynamics into the model, the RL agent can focus on optimizing higher-level behaviours instead of learning basic system dynamics. This section discusses the chosen model's complexity and its role in evaluating position accuracy and implementing the robot controller within the simulation environment.

3.1 Evaluation of Position Accuracy

Position accuracy measurements were performed to create a reference for the robot's movement and validate the dynamic model, which was later implemented in Unreal Engine. During the experiment, the robot was instructed to traverse a straight line without rotation and a diamond shape with rotations. The robot firmware was modified to log significant data, including relative time, encoder values, DAC output, position, velocity, and acceleration setpoints. The robot's position was also tracked using a vision-based OptiTrack system, which will be used as the true location of the robot to verify the location data gathered from the robot.

Figure 3.1 shows a comparison between the OptiTrack positions and the setpoint data. The close alignment suggests good controller performance, with the average error at 0.05 [m] and the maximum error at 0.1 [m] for the tested cases. Several factors such as increasing the acceleration could increase the error, due to wheel slip or other system limitations. Therefore, modelling the controller and system dynamics is crucial for understanding behaviour under varying conditions.



(a) Diamond shape test 1

(b) Diamond shape test 2

(c) Straight line test 1

(d) Straight line test 2

Figure 3.1: Comparison between the OptiTrack position and setpoints for a diamond shape and a straight line plotted to observe the controller performance.

3.2 Implementation of the Robot Controller

To replicate the robot's behaviour in simulation, the controller used on the physical robot was implemented in Unreal Engine. The original Simulink-based controller was translated

into the Unreal Engine environment. The controller design, depicted in Figure 3.2, comprises three main components, a position, velocity, and feedforward controller. With D_x the position error gain matrix and D_v the velocity gain matrix. Only the rotational position error e^{ϕ} is integrated in the Laplace domain using the transfer function $1/s$. The final controller output is multiplied by $k = 0.0976$ before passing the values to the motor drivers.

It should be noted that the feedforward controller is currently disabled in the soccer robot firmware. Consequently, this component was removed from the Unreal Engine implementation to keep consistent with the real-world setup.

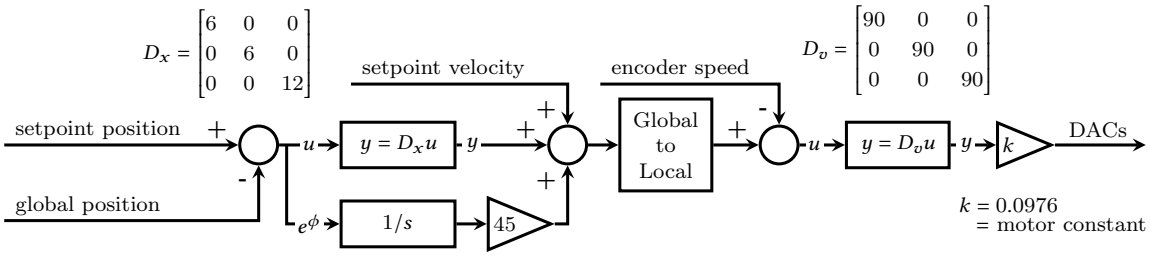


Figure 3.2: Controller design implementation in UE.

3.3 Implementation of Motor Dynamics

The TechUnited robots use three Maxon RE40 motors to drive the omni-wheels, which are used with a Maxon Planetary gearhead GP 42 C 12:1. The resistance, inertia, and motor constants are available in the Maxon catalogue [22]. The motors are used to a maximum of 24 volts in the current soccer robots.

Assuming that the inductance effects are negligible compared to the influence of mechanical motion, the inductance term is omitted, simplifying the motor dynamics equations outlined in [23]. The resulting equation is:

$$J_m \ddot{\theta}_m + \left(b + \frac{K_t K_e}{R_a} \right) \dot{\theta}_m = \frac{K_t}{R_a} v_a \tag{3.1}$$

where:

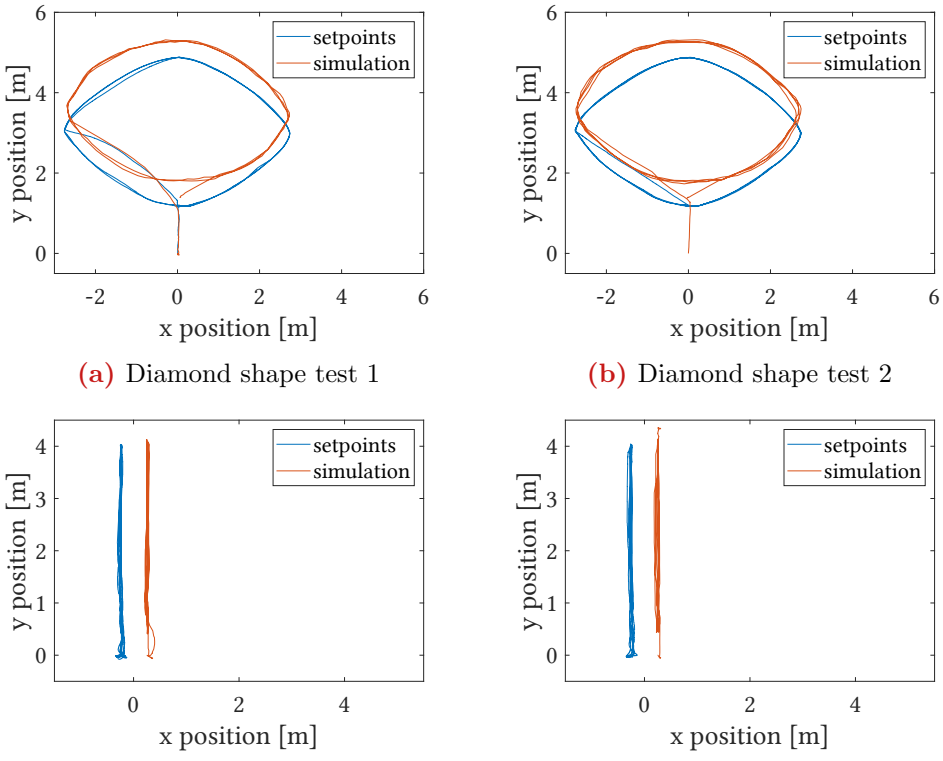
- J_m: Rotor inertia.
- $\ddot{\theta}_m$: Angular acceleration of the rotor.
- b: Viscous damping coefficient of the motor.
- K_t: Torque constant.
- K_e: Back electromotive force (EMF) constant.
- R_a: Armature resistance.
- $\dot{\theta}$: Angular velocity of the rotor.
- v_a: Applied armature voltage.

3.4 Accuracy of the Dynamical Model

Accurately implementing the motor dynamical model presents challenges due to the lack of precise knowledge about the robot's dynamic specifications. Parameters such as total inertia, frictional losses, and external influences on motor speeds, are complex and difficult to quantify. Accurately capturing these effects would require detailed testing and measurements on the complete robot system.

Given the constraints of this project, making a dynamical model of the motor was not pursued. This decision was influenced by time constraints and the primary focus of the project on evaluating reinforcement learning for the given use case. Instead, a simplified first-order dynamic model for the motors was implemented as an initial approach. This model was intended to determine whether a more complex representation of the robot's dynamics would be necessary to achieve sufficient simulation accuracy.

Tests were performed using this simplified model to evaluate its performance in replicating the robot's behaviour. The results, as shown in Figure 3.3, highlighted significant differences between the model's predictions and the observed performance of the robot. These findings indicate that the simple first-order model is insufficient for capturing the robot's dynamics.



(a) Diamond shape test 1

(b) Diamond shape test 2

(c) Straight line test 1

(d) Straight line test 2

Figure 3.3: Comparison between the position setpoints and the simulation position results using the simplified controller provided with the setpoints.

Furthermore, the controller implemented on the real robot demonstrates sufficient accuracy for the project's purposes as seen in Figure 3.1, making it reasonable for training the reinforcement model to neglect the controller dynamics and assume the robot will be able to follow the generated setpoints with sufficient accuracy.

Integrating the actual controller into the simulation environment could be an avenue for future work. This approach could reduce the sim-to-real gap, improving the fidelity of the simulations. Further discussion on this topic is provided in Section 6.

3.5 Implementation of Reinforcement Learning Controller

The reinforcement learning controller was implemented as an idealised controller, designed to follow the setpoints accurately as long as the maximum velocity limits of the robot are not exceeded. This idealized implementation assumes that the controller can precisely follow the generated setpoints, simplifying the system for the reinforcement learning-based approach. The implementation is visualised in Figure 3.4.

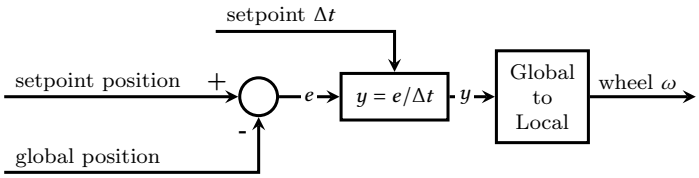


Figure 3.4: Idealised reinforcement learning controller design implementation in UE.

4 Reinforcement Learning Framework

The Learning Agents plugin for Unreal Engine (UE) 5.5 consists of three primary blueprint classes: the manager class, the interactor class, and the environment class, with each handling a distinct aspect of the learning process [24]. In the context of this project, the omni-robot will be referred to as the agent and the ball will refer to the target.

The implementation details of agent initialization and state updates are discussed further in Appendix A.

4.1 Learning Agents Manager

The Manager class controls the learning process. It coordinates the interaction between the components and runs the training loop, handling tasks such as initiating training, running inference, monitoring progress, and storing results. The flow of the manager is visualized in Figure 4.1.

During the begin play event, the learning agents manager performs the setup needed for the training or inference to start. It first ensures the actors tick after the manager's tick to ensure updates to the reinforcement learning process are run before the robot actor updates. After this, the manager binds the interactor, policy, critic, training environment, and shared memory instance to the PPO trainer instance.

The event tick runs either inference using the trained policy or runs training using the PPO trainer. The run training function has abstracted away running the functions for performing actions and gathering observations, rewards, and completion for each training iteration. The implementation of these functions is discussed in the next sections Section 4.2 and Section 4.3.

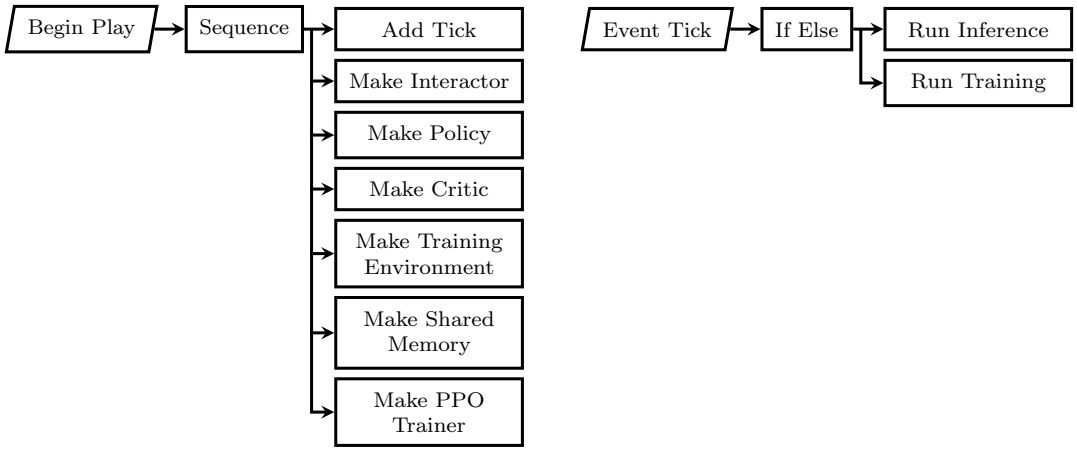


Figure 4.1: Event graph for the learning agents manager blueprint in UE.

4.2 Learning Interactor

The Interactor class enables the agents to interact with the environment. It specifies and gathers observation data and specifies and performs actions for the training agents. The Unreal Engine implementation is shown in Figure 4.2.

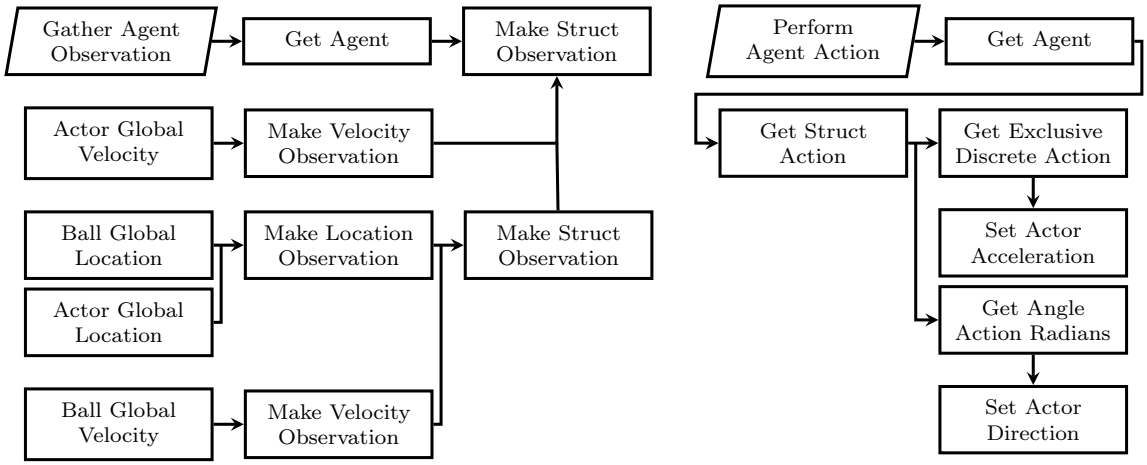


Figure 4.2: Event graph for the learning interactor blueprint in UE.

4.2.1 Agent Observation

The gather agent observation function collects the observations for a specific agent. The football robot agent has observations for the relative location of the ball, its velocity vector, and the velocity vector of the ball.

4.2.2 Agent Action

For perfect efficient ball interception, the robot will accelerate maximally, maintain a constant velocity, and decelerate maximally. Through this narrowing of the problem solution, the actions can be simplified into an integer ($\mathbb{Z} \cap [-1, 1]$) specifying the type of acceleration and an angle ($\mathbb{R} \cap [-\pi, \pi]$) specifying the direction of acceleration or deceleration. The perform agent action function generates a new action for the agent by generating these two values.

4.3 Learning Environment

The Training Environment class contains code specifically required for reinforcement learning, such as the reward, completion, and reset functions. The actions performed in the functions are visualized in Figure 4.3.

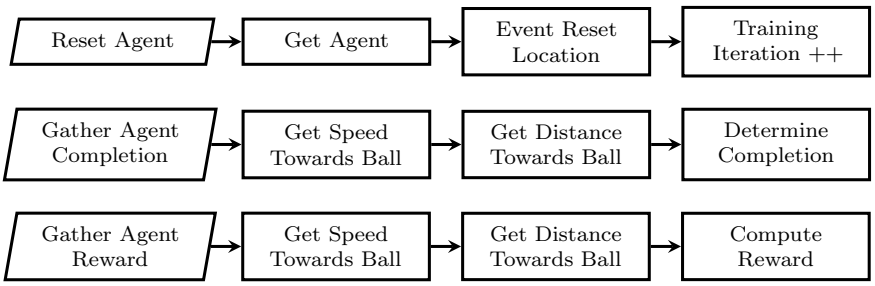


Figure 4.3: Event graph for the learning environment blueprint in UE.

4.3.1 Reset Agent Function

The robot location is reset to its initial starting position during the reset agent function, and a new ball location is generated. Additionally, a counter tracking the number of training iterations is incremented. This iteration count can be utilized to adjust the reward functions throughout the training process dynamically. For instance, at the beginning of training, the reward function may avoid penalizing time duration. Later, once the model more consistently achieves the goal, a time penalty can be introduced to encourage a more optimal interception path towards the ball.

4.3.2 Completion Function

The gather agent completion function determines when an agent should be reset. Agent resets occur for two primary reasons. The agent is deemed to have failed and is unable to reach its goal within a reasonable time, it allows for premature termination to save computational resources. Such failures should be associated with low cumulative rewards such that in later iterations they are excluded from policy updates. The other reason for termination is a successful completion of the goal, this should be paired with high cumulative rewards to encourage use in the next policy update. In the context of training the omni robot for path planning, the following termination criteria were implemented:

- **Time-Based Termination:** This condition resets the agent if a predefined time limit is exceeded. Initially, the time limit was set to a high value, providing sufficient time for the agent to explore and collect data for generating the new policy. As the model improved, the time limit was reduced to save computational resources.
- **Distance-Based Termination:** This condition monitors the agent's distance from the target ball. Since the maximum spawning distance between the agent and the ball is controlled by the initial position generation function and the agent is expected to decrease its distance to the ball over time the iteration can be terminated if deviating too far from the ball location.
- **Velocity-Based Termination:** During the later stages of training, this condition associates agents moving away from the ball with sufficient velocity as undesirable behaviour. Such agents are reset immediately. However, this condition is not applied

in the early stages of training to ensure the algorithm collects sufficient data on the performance of the agent before introducing stricter constraints.

- **Successful Completion:** An agent is deemed successful when it satisfies the conditions of low relative velocity and distance to the target ball. The thresholds for these metrics can be adjusted during training. Early in the process, less stringent thresholds are used to reduce task complexity and facilitate learning. As the model improves, they are tightened to refine the policy.

4.3.3 Rewards Function

The reward function is defined in the gather agent reward function. Rewards are given on each simulation tick of the agent manager. The cumulative reward is used at the agent's termination to determine its performance and influence on the new policy. Positive rewards should be used to encourage desired behaviour. In contrast, negative rewards should be used to discourage undesired behaviour where the magnitude of rewards should reflect the relative importance of actions or outcomes to the overall goal. Here, gradual and continuous reward structures are generally more effective, providing incremental feedback that helps the agent refine its policy step by step. The rewards used for training the interception path are given with the tuneable parameters, the scaling factor β and the decay factor α.

- **Distance-based Reward:** To give a reward based on distance, a non-linear reward is used to create a stronger incentive for the agent to reduce the remaining distance as much as possible. An exponential function was chosen with a negative exponent where the distance is normalized by dividing by the initial distance between the robot and ball, as shown in Equation 4.1.

$$r = \beta \cdot \exp\left(\alpha \frac{\Delta x}{x_{init}}\right) \tag{4.1}$$

- **Velocity-based Penalty:** Velocity away from the target should be discouraged and is therefore penalised. Velocity towards the target should not be directly rewarded to ensure that the slowdown close to the ball is not penalised. This was implemented using Equation 4.2.

$$r = \begin{cases} -\left(\beta \cdot \exp\left(\alpha |v|\right) - \beta\right) & \text{if } v \leq 0 \\ 0 & \text{otherwise} \end{cases} \tag{4.2}$$

- **Completion Reward:** To ensure the agent is correctly favoured in the next policy generation when the successful completion is reached, a relatively large reward should be given on the same conditions as this completion from Section 4.3.2. So, the conditions of low relative velocity and distance to the target ball are used for giving the reward, Equation 4.3.

$$r = \begin{cases} \beta & \text{if } v \leq 0.05 \text{ [m] and } \Delta x \leq 0.05 \text{ [m/s]} \\ 0 & \text{otherwise} \end{cases} \tag{4.3}$$

- **Time-based Penalty:** A time-based penalty is introduced when the current policy can consistently complete the goal. This further refines the model to reach the

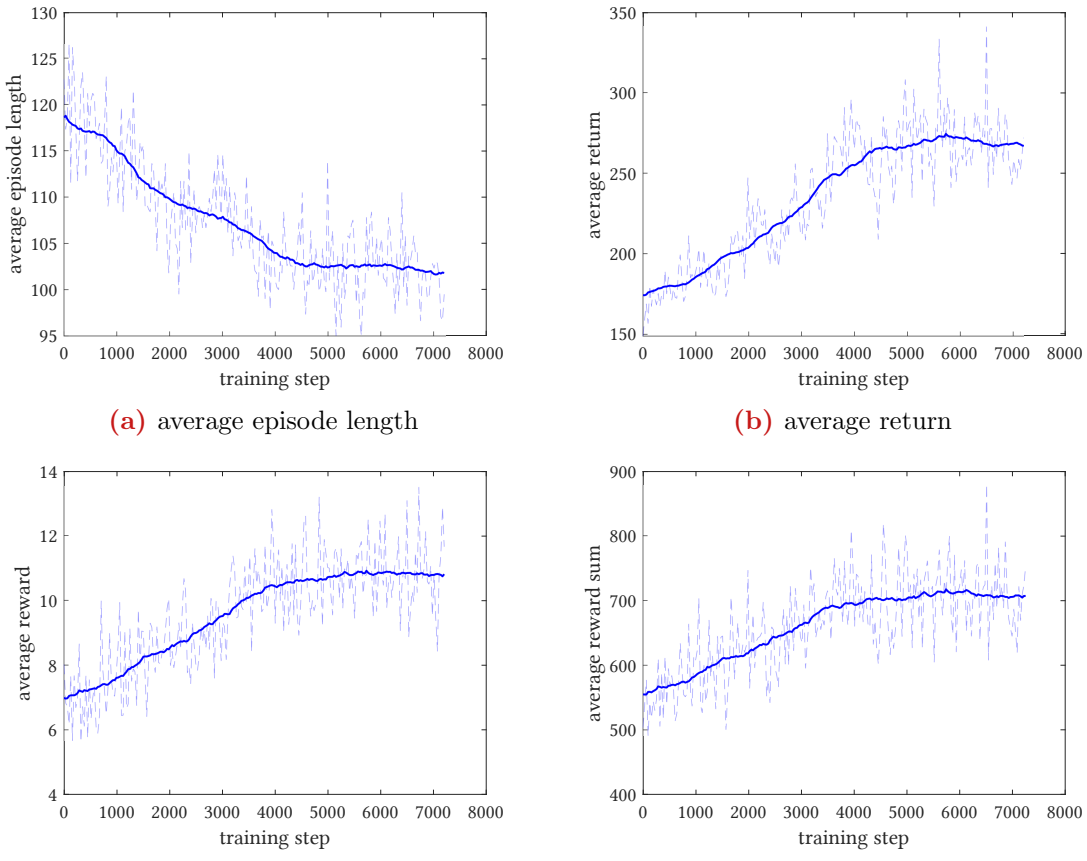
goal more efficiently and faster. The elapsed time is normalised using the starting distance between the agent and the ball, to ensure fair comparison between agents with different randomly generated ball positions and velocities.

$$r = -\left(\beta \cdot \exp\left(\alpha \frac{t}{x_{init}}\right) - \beta\right) \tag{4.4}$$

4.4 Training of Agents

In reinforcement learning, the limiting factor in training is often running agents in the simulated environment, rather than the computational cost of optimizing the neural network. This can be mostly eliminated by conducting the training in headless mode [25], which simulates without rendering. This significantly accelerates the training process by multiple factors. This is done by running with the following command

```
.\OmniWheelRobot\Saved\StagedBuilds\Windows\OmniWheelRobot
    \Binaries\Win64\OmniWheelRobot.exe Basic
    -nullrhi -nosound -log -log=omnirobot_learning.log
```



(a) average episode length

(b) average return

(c) average reward

(d) cumulative reward

Figure 4.4: Plots the gathered reinforcement learning metrics throughout training. The flattening of these metrics over time indicates convergence of the learning process.

The progress of the learning process can be monitored using TensorBoard, a visualization tool that is part of the TensorFlow framework [26]. TensorBoard enables the inspection of

logs generated during the reinforcement learning process, providing insights into average episode length, average return, average reward, and cumulative reward over time. These metrics give a view of the trained policy's performance and training progress. For instance, a declining cumulative reward may signal potential issues, such as an inappropriate reward function. Flattening of these metrics over time can indicate that the agent's learning process is converging.

Figure 4.4 illustrates an example of these metrics plotted throughout training. Initially, the metrics show significant variability, showing the agent's exploration of the environment. Over time, the average episode length, average return, and average reward stabilize, while the cumulative reward plot approaches a plateau. This behaviour suggests that the agent has reached a steady-state performance, successfully optimizing its policy for the given task and reward function. The performance of the policy on the task should be compared with other policies using separate test cases which is further discussed in Section 5.

5 Results of Reinforcement Learning

The neural network underlying reinforcement learning has so-called layers. More layers can improve behaviour in complex situations but may increase the time to train. For all policies, the number of layers used was 3, an increase from the default 1 layer. This increase significantly improved the performance of the learned policies.

All policies were initially trained using a reward function solely based on the distance to the target and the objective completion for three thousand episodes. This helped significantly in improving the initial learning speed of the more complex reward functions, as the policies learned to first move towards the ball. The values used for further training episodes were determined by tuning them, depending on observations made in the training results. For example, a policy that overshoots on many test cases might need a higher penalty for the velocity away from the ball. The values and the number of episodes trained used are given in Table 5.1.

	1		**2**		**3**		**4**	
Distance	$\beta = 5$	$\alpha = -2$	$\beta = 5$	$\alpha = -2$	$\beta = 6$	$\alpha = -2$	$\beta = 10$	$\alpha = -2$
Velocity	$\beta = 5$	$\alpha = 0.2$	$\beta = 3$	$\alpha = 0.2$	$\beta = 2$	$\alpha = 0.2$	$\beta = 2$	$\alpha = 0.2$
Completion	$\beta = 2000$		$\beta = 2000$		$\beta = 2000$		$\beta = 2000$	
Time	$\beta = 0.24$	$\alpha = 75$	$\beta = 0.24$	$\alpha = 75$	$\beta = 0.28$	$\alpha = 75$	$\beta = 0.28$	$\alpha = 75$
Episodes	58000		60000		42000		9000	

Table 5.1: Parameter values used the reward functions (Section 4.3.3) during training of the policies.

To evaluate the performance of the trained model, a series of test cases were designed, incorporating variations in ball velocity, initial distance, the direction of the initial ball position, and ball velocity direction. An angle of 0 means the ball is moving direction away from the ball. The test scenarios were constructed by combining all combinations of the following parameter values:

- **Initial Distance:** $\{100, 200, 300, 400, 500, 600\}$
- **Initial Ball Direction:** $\{-\pi/8, -\pi/4, -3\pi/8\}$

- **Ball Velocity:** {0, 100, 200}
- **Ball Movement Direction:** $\{\pi/2, 3\pi/4, 8\pi/9\}$

This resulted in 162 distinct test cases, ensuring the model was evaluated under diverse conditions. For each test, the time taken by the model to intercept the ball was recorded. In Table 5.2, the results are visualised for several different trained policies. Trained policy 2 shows the best results with the highest intercept percentage. Though policy 3 shows a lower mean time to intercept than policy 2, they cannot be compared due to the significant difference in success percentage. When having a closer look at the individual categories of test cases behaviour differences can be seen between parameter values.

	1	**2**	**3**	**4**
MIT [ms]	3675	3870	3560	3912
Percentage	70.4%	58.0%	64.2%	77.2%

Table 5.2: Summary of machine learning results for multiple trained policies, showing the mean intercept time (MIT) in milliseconds and the percentage of successful intercepts for the trained policies.

5.1 Successful Intercept Percentage

An analysis of the success rates for ball interception under varying test conditions, as presented in Figure 5.1, reveals notable patterns across the tested policies.

- **Distance:** When examining the percentage of successful intercepts at different distances, policy 1 does not show a clear relationship between distance and success rate. In contrast, policies 2, 3, and 4 show an increasing success rate with greater initial distances, especially within the first 400 [cm].
- **Velocity:** The second category considers the effect of varying ball velocities. At a velocity of 0 [cm/s], all trained policies achieve 100% success in interception of the ball for the generated test cases. However, at a velocity of 200 [cm/s], performance deteriorates significantly across all policies. Policy 4 outperforms the other policies, achieving higher intercept success rates at the 100 and 200 [m/s] test cases.
- **Initial Ball Location:** For the third category, which examines the direction of the initial ball location, policies 1, 3, and 4 show a decrease in success rate as the initial angle becomes more negative. This decline can be attributed to the increased relative velocity between the ball and the robot generated by the test conditions at larger negative angles. This increase in relative velocity results in a more challenging interception scenario, reducing success rates. Notably, policy 2 does not exhibit a clear relationship between the initial angle and success percentage.
- **Ball Movement Direction:** The final category evaluates success rates based on the movement direction of the ball. Policies 1 and 4 demonstrate significantly higher success rates for the $\pi/8$ ball direction, while policies 3 and 4 achieve higher success rates for the $\pi/4$ direction. The $3\pi/8$ direction yields similar results across all policies.

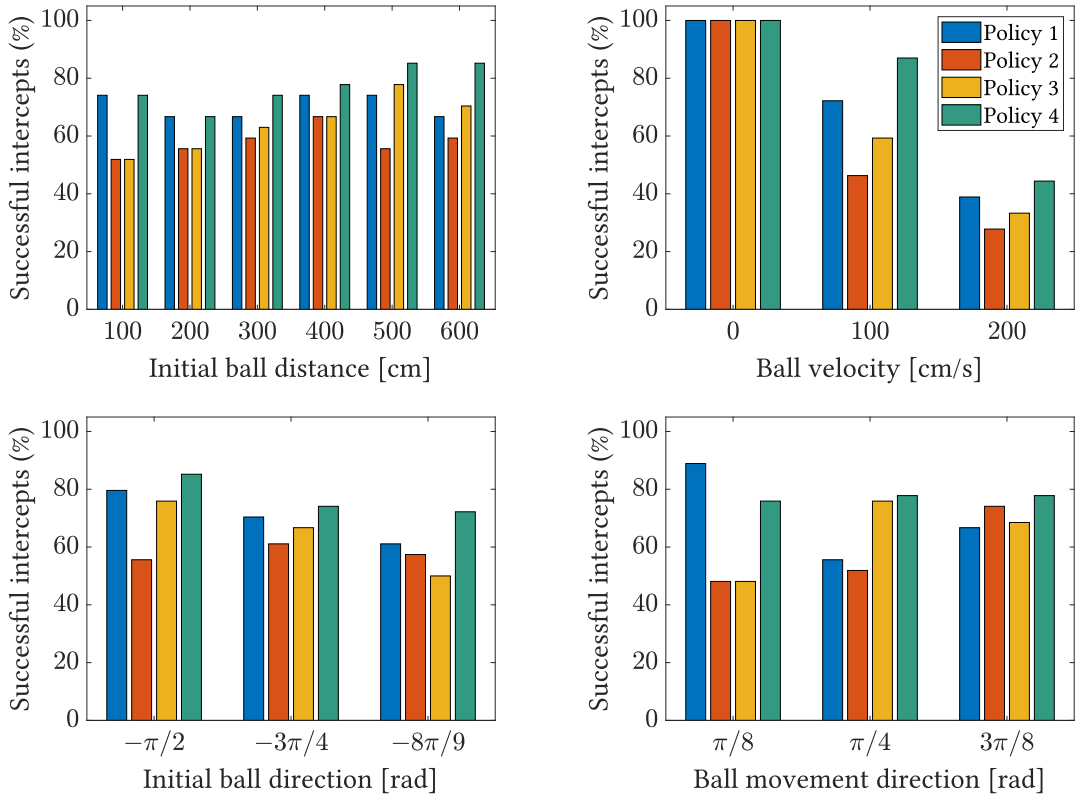


Figure 5.1: Results for the trained policies, showing the percentage of successful intercepts for the different variables starting distance, ball velocity, initial ball direction, and ball movement direction.

5.2 Mean Intercept Time

The analysis of intercept times, as presented in Figure 5.2, provides valuable insights into the performance of the trained policies under various test conditions. It is important to note that the calculations are based solely on successful intercepts, which introduces a bias favouring models with lower intercept percentages. These models often only succeed in simpler scenarios which require minimal robot movement, leading to shorter intercept times. This bias is particularly evident in cases with a ball velocity of 200 [cm/s], where successful intercepts typically occur when the ball moves directly toward the robot from a short distance, resulting in low intercept times.

- **Distance:** The relationship between distance and intercept time shows as the distance increases, the intercept time also increases. This result aligns with expectations, as greater distances inherently require more time. Policies 2 and 3 demonstrate significantly more efficient intercept times at most distances, but their lower success percentages must be considered. Policy 4 performs worse compared to policy 1 at distances of 500 and 600 [cm], however, achieves higher success rates at those distances.
- **Velocity:** The results for velocity-varying test cases are difficult to compare because of the greatly differing success percentages between the policies. The intercept times are ordered the same as the successful intercept percentages, policy 4 has the highest intercept rate and the worst intercept time.

- **Initial Ball Location:** Examining the relationship between the initial direction of the ball and intercept times, policies 1 and 4 show a decrease in intercept time as the angle becomes more negative. Policies 2 and 3 do not show a clear relationship with the angle and display significantly shorter intercept times for the $-\pi/2$ direction.
- **Ball Movement Direction:** For test cases involving ball movement direction, policies 1 and 4 outperform policies 2 and 3 in terms of intercept times for the $\pi/8$ direction. However, this comparison is complicated by significant differences in intercept percentages, making a direct comparison less meaningful. Similarly, Policies 3 and 4 perform better than policies 1 and 2 in the $\pi/4$ direction, but again, variations in success rates affect the interpretation of these results.

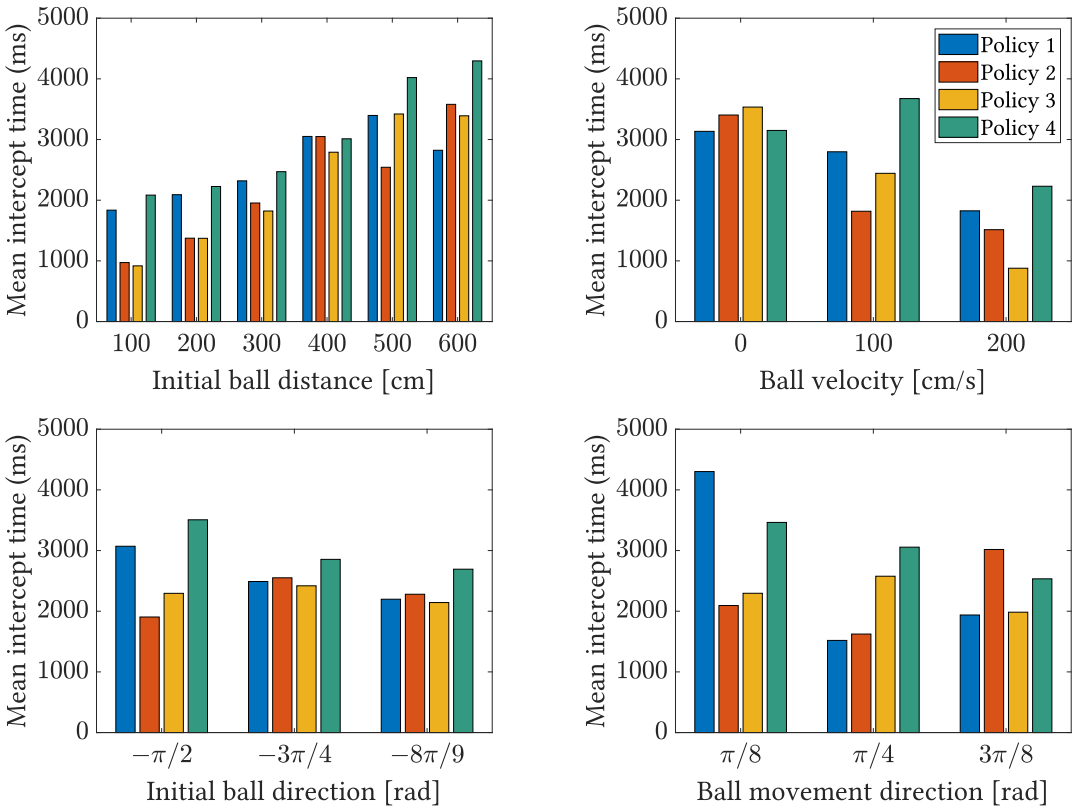


Figure 5.2: Results for the trained policies, showing the mean intercept time (MIT) of successful intercepts for the different variables starting distance, ball velocity, initial ball direction, and ball movement direction.

5.3 Intercept Path

The intercept paths for several test cases are shown in Appendix B to illustrate instances where the policies successfully intercept or fail to intercept the ball. These provide an understanding of the specific conditions under which the trained policies can intercept the ball efficiently.

6 Conclusion & Discussion

The analysis of the intercept paths in Appendix B demonstrates that reinforcement learning can successfully generate direct and efficient ball interception trajectories. These paths highlight the potential of reinforcement learning to achieve high intercept accuracy, as evidenced by the 77% success rate on the generated test cases achieved by the best-performing policy (Policy 4) across diverse conditions (Table 5.2). This indicates the adaptability of reinforcement learning-based approaches in dynamic scenarios.

However, the results in Figure 5.1 reveal notable challenges in scenarios involving high ball velocities. These difficulties suggest that the ability of the policy to adapt decreases under such conditions. Small adjustments to reward function parameters have already shown significant influence on performance at higher velocities, suggesting that further tuning could minimize these challenges.

Despite the overall positive outcomes, the results in Figure B.5 indicate room for improvement. While Policy 4 achieved the highest intercept percentage, Policies 1 and 3 intercepted without missing on the first attempt whereas policy 4 failed and needed to correct. This suggests that optimizing the training process could create a better-performing policy.

6.1 Directions for Future Research

Future studies could explore the impact of neural network composition on trained policy performance. Specifically, increasing the depth of the network may enable the models to better generalize and adapt to a wider variety of intercept scenarios, particularly those involving high-velocity scenarios.

For future research, alternative simulation environments can be considered such as NVIDIA Isaac and MuJoCo, which are widely used in robotic reinforcement learning research [27]. Exploring these environments could enhance the realism and effectiveness of simulations, potentially improving the development and transferability of learned policies to physical robots.

Integration of the real controller into Unreal Engine is another improvement which can be introduced, this would allow for a more realistic simulation of the system dynamics and enable validation of the reinforcement learning controller in an environment closer to reality and reduce the sim-to-real gap.

Research into hybrid solutions combining reinforcement learning with traditional methods offers an avenue for future exploration. For instance, a reinforcement learning model could be used to generate higher-level setpoints, which would then be handled by conventional trajectory planning algorithms. This combination could leverage the strengths of both methodologies, with reinforcement learning providing flexibility and adaptability, while traditional control ensures stability and predictability.

Finally, implementation and validation of the reinforcement learning-based model in physical robotic systems would be a crucial step. Real-world tests would provide a valuable understanding of how the model would handle the complexities of physical interactions, including friction, imperfect measurements, and latency.

References

[1] J.-H. Kim, Y.-J. Kim, D.-H. Kim, and K.-T. Seow, "1. Soccer Robotics," English, in *Soccer Robotics*, First edition, Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 1–26, ISBN: 978-3-540-40921-2. DOI: 10.1007/978-3-540-40921-2_1.

[2] RoboCup (Conference), C. Buche, A. Rossi, M. Simões, and U. Visser, *RoboCup 2023: Robot World Cup XXVI*, English, First edition. Cham, Switzerland: Springer Cham, 2024, ISBN: 978-3-031-55015-7. DOI: 10.1007/978-3-031-55015-7.

[3] RoboCup (Conference), A. Eguchi, N. Lau, M. Paetzel-Prüsmann, and T. Wanichanon, *RoboCup 2022: Robot World Cup XXV*, English, First edition. Cham, Switzerland: Springer Cham, 2023, ISBN: 978-3-031-28469-4. DOI: 10.1007/978-3-031-28469-4.

[4] RoboCup Federation, *RoboCup Objective*, English. [Online]. Available: https://www.robocup.org/objective (visited on 12/24/2024).

[5] R. M. Beumer *et al.*, "Tech United Eindhoven Middle Size League Winner 2023," English, in *RoboCup 2023: Robot World Cup XXVI*, C. Buche, A. Rossi, M. Simões, and U. Visser, Eds., Cham: Springer Nature Switzerland, 2024, pp. 428–439, ISBN: 978-3-031-55015-7. DOI: 10.1007/978-3-031-55015-7_36.

[6] F. Belkhouche and B. Belkhouche, "On the tracking and interception of a moving object by a wheeled mobile robot," English, in *IEEE Conference on Robotics, Automation and Mechatronics, 2004.*, vol. 1, 2004, 130–135 vol.1, ISBN: 0-7803-8645-0. DOI: 10.1109/RAMECH.2004.1438904.

[7] F. Belkhouche and B. Belkhouche, "Modified parallel navigation for ball interception by a wheeled mobile robot goalkeeper," English, *Advanced Robotics*, vol. 20, no. 4, pp. 429–452, 2006, Publisher: Taylor & Francis. DOI: 10.1163/156855306776562260.

[8] M. Verrijt, "Improving ball interception accuracy in an automated football table," English, M.S. thesis, Eindhoven University of Technology, Eindhoven, Nov. 2011.

[9] S. Khan *et al.*, "Active interception of moving ball: A multi-player strategy for humanoid soccer robots," English, *Multimedia Tools and Applications*, Dec. 2024, ISSN: 1573-7721. DOI: 10.1007/s11042-024-20491-6. (visited on 01/11/2025).

[10] J. Cunha, N. Lau, and J. Rodrigues, "Ball Interception Behaviour in Robotic Soccer," English, in *RoboCup 2011: Robot Soccer World Cup XV*, T. Röfer, N. M. Mayer, J. Savage, and U. Saranlı, Eds., Berlin, Heidelberg: Springer, 2012, pp. 114–125, ISBN: 978-3-642-32060-6. DOI: 10.1007/978-3-642-32060-6_10.

[11] F. Stolzenburg, O. Obst, and J. Murray, "Qualitative Velocity and Ball Interception," English, in *KI 2002: Advances in Artificial Intelligence*, M. Jarke, G. Lakemeyer, and J. Koehler, Eds., Berlin, Heidelberg: Springer, 2002, pp. 283–298, ISBN: 978-3-540-45751-0. DOI: 10.1007/3-540-45751-8_19.

[12] M. Riedmiller, T. Gabel, R. Hafner, and S. Lange, "Reinforcement learning for robot soccer," English, *Autonomous Robots*, vol. 27, no. 1, pp. 55–73, Jul. 2009. DOI: 10.1007/s10514-009-9120-4.

[13] C. Hu, M. Xu, and K.-S. Hwang, "An adaptive cooperation with reinforcement learning for robot soccer games," English, *International Journal of Advanced Robotic Systems*, vol. 17, no. 3, 2020. DOI: 10.1177/1729881420921324.

[14] T. Nakashima, M. Udo, and H. Ishibuchi, "A Fuzzy Reinforcement Learning for a Ball Interception Problem," English, in *RoboCup 2003: Robot Soccer World Cup VII*, D.

Polani, B. Browning, A. Bonarini, and K. Yoshida, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 559–567, ISBN: 978-3-540-25940-4. DOI: 10.1007/978-3-540-25940-4_52.

[15] R. S. Sutton and A. G. Barto, *Reinforcement learning: an introduction* (Adaptive computation and machine learning), English, Second edition. Cambridge, Massachusetts: The MIT Press, 2018, ISBN: 978-0-262-03924-6.

[16] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal Policy Optimization Algorithms," English, *CoRR*, vol. abs/1707.06347, Aug. 2017. DOI: 10.48550/arXiv.1707.06347.

[17] Epic Games, *Unreal Engine 5.5 Documentation*, English, Nov. 2024. [Online]. Available: https://dev.epicgames.com/documentation/en-us/unreal-engine/unreal-engine-5-5-documentation (visited on 12/24/2024).

[18] Epic Games, *Physics*, English, Nov. 2024. [Online]. Available: https://dev.epicgames.com/documentation/en-us/unreal-engine/BlueprintAPI/Physics (visited on 01/24/2025).

[19] Epic Games, *Blueprints Visual Scripting*, English, Nov. 2024. [Online]. Available: https://dev.epicgames.com/documentation/en-us/unreal-engine/blueprints-visual-scripting-in-unreal-engine (visited on 12/24/2024).

[20] Epic Games, *Actors*, English, Nov. 2024. [Online]. Available: https://dev.epicgames.com/documentation/en-us/unreal-engine/actors-in-unreal-engine (visited on 12/26/2024).

[21] B. Mulcahy, D. Holden, and G. Wang, *Learning to Drive (5.5)*, English, Nov. 2024. [Online]. Available: https://dev.epicgames.com/community/learning/courses/GAR/unreal-engine-learning-agents-5-5/7dmy/unreal-engine-learning-to-drive-5-5 (visited on 12/24/2024).

[22] Maxon, *Maxon group RE40*, product page. [Online]. Available: https://www.maxongroup.com/maxon/view/product/motor/dcmotor/re/re40/148866 (visited on 01/12/2025).

[23] G. F. Franklin, J. D. Powell, and A. Emami-Naeini, *Feedback control of dynamic systems*, English, Seventh edition. Boston: Pearson, 2015, ISBN: 978-0-13-349659-8.

[24] B. Mulcahy, D. Holden, and G. Wang, *Learning agents (5.5)*, English, Nov. 2024. [Online]. Available: https://dev.epicgames.com/community/learning/tutorials/bZnJ/unreal-engine-learning-agents-5-5 (visited on 12/24/2024).

[25] B. Mulcahy, D. Holden, and G. Wang, *Headless Training & Network Snapshots (5.5)*, English, Nov. 2024. [Online]. Available: https://dev.epicgames.com/community/learning/courses/GAR/unreal-engine-learning-agents-5-5/DPDd/unreal-engine-headless-training-network-snapshots-5-5 (visited on 12/24/2024).

[26] M. Abadi *et al.*, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems," *CoRR*, vol. abs/1603.04467, Mar. 2016. DOI: 10.48550/arXiv.1603.04467.

[27] M. Kaup, C. Wolff, H. Hwang, J. Mayer, and E. Bruni, *A Review of Nine Physics Engines for Reinforcement Learning Research*, Aug. 2024. DOI: 10.48550/arXiv.2407.08590. (visited on 01/24/2025).

A Implementation of Omni-Robot Agent

The omni-robot is implemented as an agent in Unreal Engine (UE). The event graph in UE is depicted in Figure A.1.

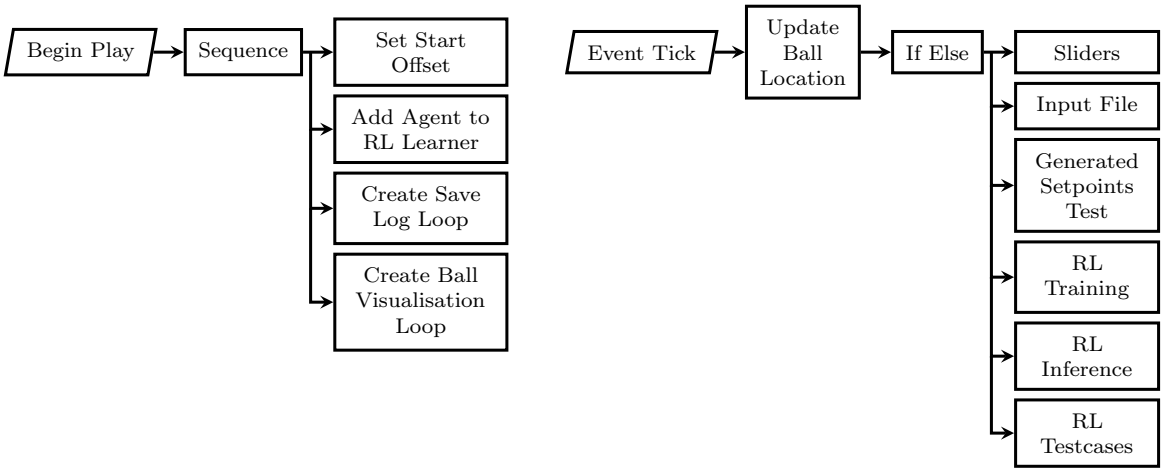


Figure A.1: Event graph for the omni-robot agent blueprint in UE.

During the begin play event, the blueprint first sets the start location offset variable, this is used to reset the robot location, normalize the location data for the reinforcement training, and generate the ball location centred around the robot location. After the agent is added to the reinforcement learning manager, the agent is initiated for use during training. Finally, loops on an interval for saving the log data and updating the ball visualisation are started.

The blueprint of the agent handles updating the robot from the selected operation mode and specified input data. In the interface, the mode can be selected using a dropdown menu, the available options are, sliders, input file, generated setpoints test, RL training, and RL inference.

- **Sliders:** The sliders option enables setting the wheel angular velocities using the sliders in the viewport. This can be used to verify the robot's kinematics code.

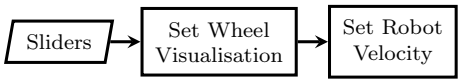


Figure A.2: Event graph for the omni-robot agent tick for sliders option.

- **Input File:** For the input file option, the blueprint reads the specified file and sets the setpoints. Using this, setpoints acquired from the real robot can be run and the model can be evaluated against the results from the real robot.

Figure A.3: Event graph for the omni-robot agent tick for input file option.

- **Generate Setpoints Test:** This option uses simple test values to validate the generation of setpoints from the input values generated by the reinforcement model.
- **RL Training:** The training option runs the reinforcement training. On the agent resetting, a new random ball location and velocity are automatically generated.
- **RL Inference:** The inference option runs the reinforcement model in inference mode. Similarly, as in training, a new random ball location and velocity are automatically generated.

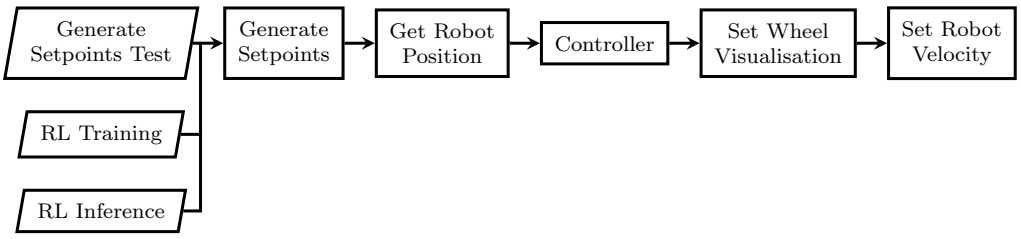
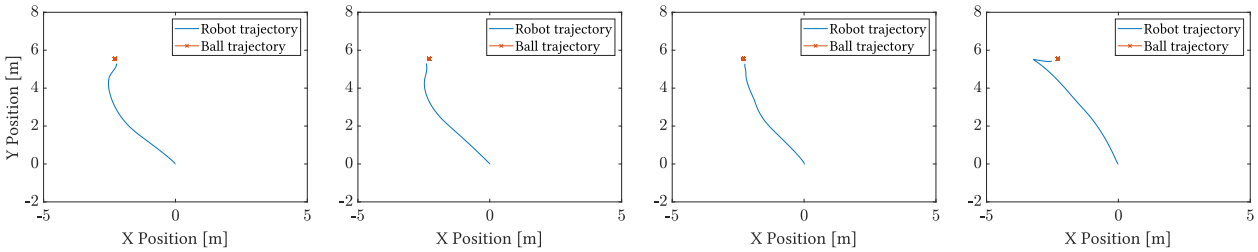


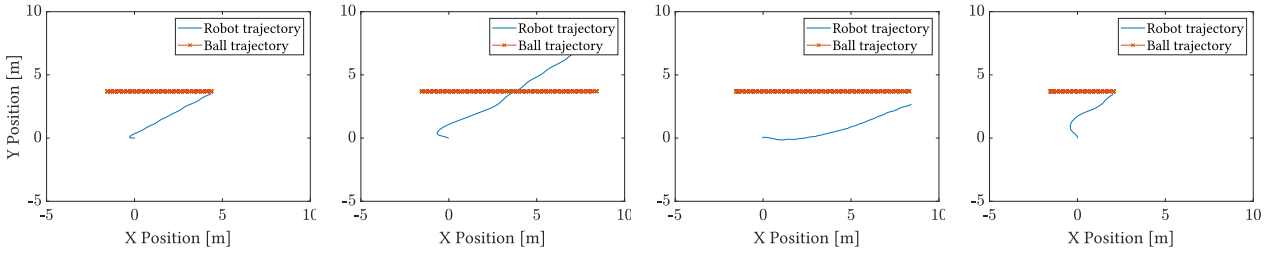
Figure A.4: Event graph for the omni-robot agent tick for options using the generate setpoints function.

B Ball Intercept Path for Test Cases



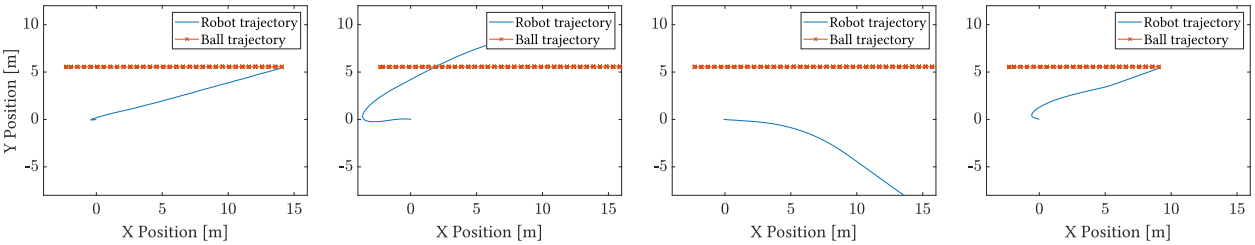
(a) Policy 1 **(b)** Policy 2 **(c)** Policy 3 **(d)** Policy 4

Figure B.1: Intercept trajectory for test stationary ball (test nr. 16) for the trained policies. The trajectories for a stationary ball show deviations from the optimal path.



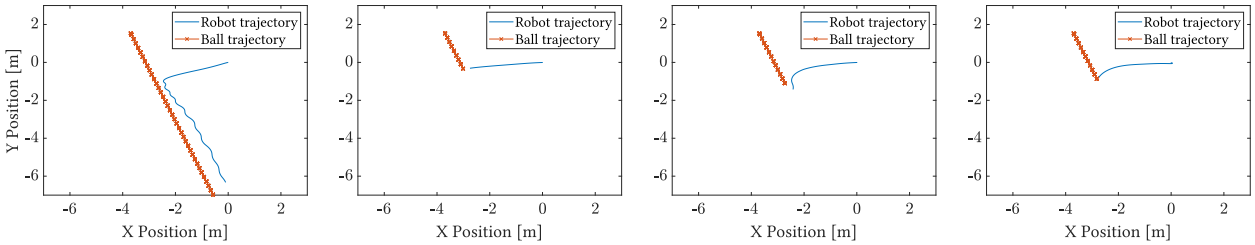
(a) Policy 1 **(b)** Policy 2 **(c)** Policy 3 **(d)** Policy 4

Figure B.2: Intercept trajectory moving ball at 100 [m/s] (test nr. 28) for the trained policies. Policy 4 shows the fastest intercept. The path of policy 1 shows a more direct path but it did not move at the maximum speed of the robot. Policy 2 misses the ball closely and does not correct its course. Policy 3 takes an inefficient path and intercepts significantly slower.



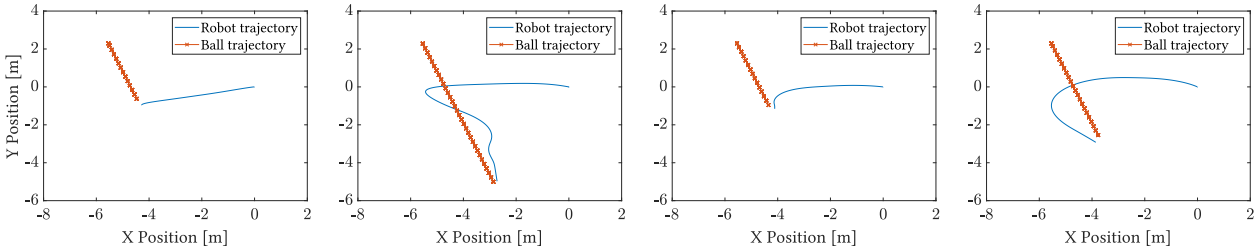
(a) Policy 1 (b) Policy 2 (c) Policy 3 (d) Policy 4

Figure B.3: Intercept trajectory for test moving ball at 200 [m/s] (test nr. 52) for the trained policies. Policy 4 again shows the fastest intercept. The path of policy 1 shows a more direct path but it did not move at the maximum speed of the robot. Policy 2 moves in the wrong direction and misses the ball after correcting once. Policy 3 did not intercept the ball.



(a) Policy 1 (b) Policy 2 (c) Policy 3 (d) Policy 4

Figure B.4: Intercept trajectory for test moving ball at 100 [m/s] (test nr. 138) for the trained policies. Policy 2, 3, and 4 show intercepts, policy 2 shows a slightly faster intercept compared to 3 and 4. Policy 1 undershoots the ball and tracks the ball after.



(a) Policy 1 (b) Policy 2 (c) Policy 3 (d) Policy 4

Figure B.5: Intercept trajectory for test moving ball at 100 [m/s] (test nr. 144) for the trained policies. All policies show intercepts, however, 2 and 4 initially miss the ball and correct themselves after. Policy 1 shows the fastest intercept.