Fundamentals of Artificial Intelligence: vollAIball

Author

Gianfranco Demarco
Francesco Ranieri

Introduction

This project explores the realm of applying artificial intelligence (AI) techniques to train autonomous agents at the game of volleyball. It consists of a Unity 3D environment, crafted on purpose, where agents are trained using the ML-Agents library and the Proximal Policy Optimization (PPO) algorithm. Another vital component of the project is the Prolog Knowledge Base, acting as a narrator that provides detailed insights into the game's events. These two main components are glued together by using a Python backend and rest APIs. This project allowed to explore the possibility of integrating different AI systems, using modern tech stacks and providing variegated features.

1 Background

In this section we go through all of the technologies and components used in this project.

1.1 Unity

Unity 3D[3] is a widely recognized game engine renowned for its versatility and ability to create immersive gaming experiences.

Unity 3D has gained popularity among developers, from independent hobbyists to professional studios, due to its cross-platform compatibility and extensive toolset. It offers robust support for both 2D and 3D game development, empowering developers to create engaging gameplay mechanics and stunning visual effects.

Unity 3D provides a wide range of features for game development, including physics simulation, advanced rendering techniques, and dynamic lighting systems. It offers a comprehensive workspace that combines artist-friendly tools with a component-driven design approach, streamlining the development process.

Unity 3D incorporates artificial intelligence (AI) techniques, such as machine learning and behavior trees, to enable intelligent agent behaviors and adaptive gameplay experiences.

The integration of AI in Unity empowers developers to create virtual environments with autonomous characters that can learn, reason, and make decisions, enhancing the realism and complexity of game worlds.

1.1.1 ML-Agents ML-Agents is a powerful framework integrated into Unity for implementing reinforcement learning (RL) algorithms in game development. ML-Agents enables the training of intelligent agents that can learn and improve their behaviors through interactions with the game environment.

The core components of ML-Agents include the *Agent*, the *Environment*, and the *Academy*.

The Agent represents an intelligent entity within the game, capable of making decisions and taking actions.

The Environment defines the virtual world in which the agent operates, providing the necessary observations and rewards.

The Academy acts as a central manager, coordinating the training process and facilitating the communication between multiple agents and environments.

ML-Agents supports a range of RL algorithms, including *Proximal Policy Optimization (PPO)*, which is commonly used for training agents in Unity. PPO is an actor-critic algorithm that optimizes policy networks to maximize rewards and improve agent performance over time. By iteratively learning from experiences, agents can develop strategies that lead to successful gameplay outcomes.

The training process in ML-Agents involves iterations of interactions between agents and environments. During each iteration, agents observe the environment, select actions based on their learned policies, receive rewards or penalties, and update their policies accordingly. This iterative learning process allows agents to improve their decision-making capabilities and adapt to changing game conditions.

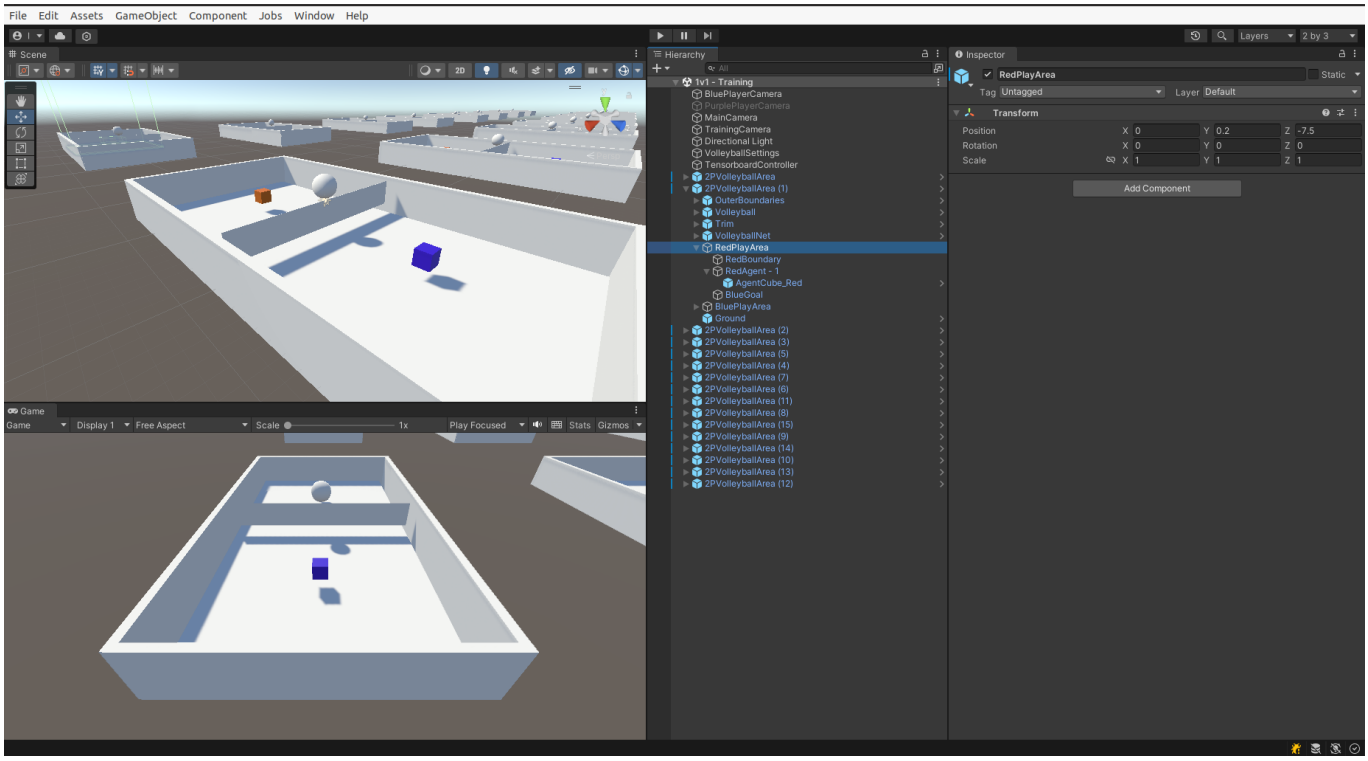

Figure 1: A scene in Unity3D for the vollAIball project

1.2 Reinforcement Learning

Reinforcement learning[1] is a subfield of machine learning that focuses on enabling agents to learn optimal behavior through interaction with an environment. Unlike supervised learning, where labeled examples are provided, reinforcement learning agents learn from feedback in the form of rewards or penalties. The agent explores the environment, takes actions, and receives feedback, which guides its learning process. Through a trial-and-error approach, reinforcement learning algorithms aim to maximize the cumulative reward obtained over time by learning the optimal policy, i.e., the best sequence of actions in different states.

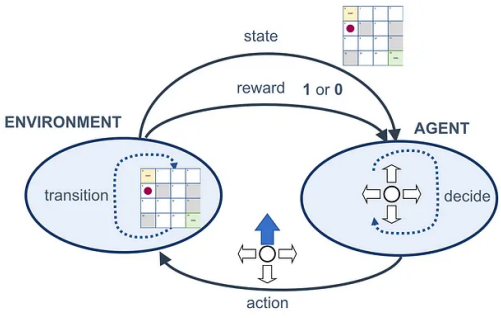

Figure 2: The Reinforcement Learning cycle

In its simplest form, the policy is a table with states as rows and actions as columns. This table is called Q-table. The table is initialized randomly, and the agent starts to interact with the environment and measures the reward for each action. It then computes the observed rewards and updates the Q-table accordingly.[2] This process is called Q-Learning.

1.2.1 Deep Reinforcement Learning Deep reinforcement learning combines artificial neural networks with a framework of reinforcement learning that helps software agents learn how to reach their goals. That is, it unites function approximation and target optimization, mapping states and actions to the rewards they lead to.

While in classical Reinforcement Learning algorithm the policy is approximated using tabular methods, in Deep Reinforcement Learning the task of approximating an optimal policy is demanded to a Neural Network. Deep Reinforcement Learning is especially useful when the input is high-dimensional, for example when dealing with visual input.

1.2.2 Proximal Policy Optimization Proximal Policy Optimization (PPO)[4] is a reinforcement learning algorithm widely used for training intelligent agents.

PPO builds upon *policy gradient methods*, which optimize the policy of an agent by iteratively adjusting its parameters to maximize cumulative rewards. PPO employs a surrogate objective function that guides the

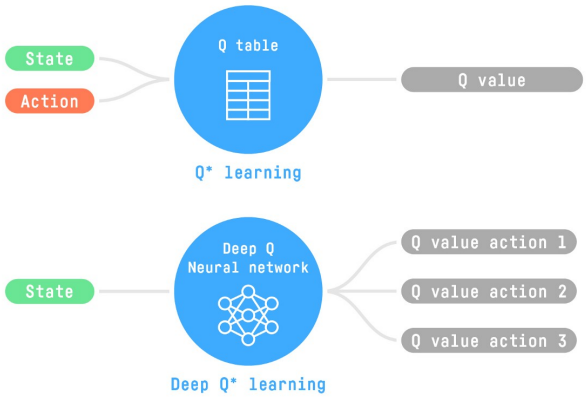

Figure 3: Q-Learning vs Deep Q-Learning

policy update process. The surrogate objective is a combination of the policy's probability ratio and a clipping function.

The policy's probability ratio represents the likelihood of taking a particular action under the current and previous policy parameters. It measures how much the updated policy deviates from the old policy. The clipping function constrains the policy update to a specified range, preventing large policy changes that could destabilize training.

PPO operates in two main stages: data collection and policy updates. In the data collection stage, the agent interacts with the environment, collecting trajectories of state-action pairs and corresponding rewards. These trajectories are then used to estimate the policy gradient.

During the policy update stage, the surrogate objective function is optimized to improve the policy. The objective is to maximize the expected value of the surrogate objective, which approximates the performance improvement. PPO employs optimization techniques such as stochastic gradient ascent to iteratively update the policy parameters, seeking to find the optimal policy that maximizes the expected rewards.

The key innovation of PPO lies in its use of a clipping mechanism. By applying a clipping function to the surrogate objective, PPO limits the policy update to a range where the improvement is guaranteed. This helps to ensure stable training and prevents drastic policy changes that could lead to poor performance.

PPO improves upon the limitations of earlier policy optimization algorithms, such as Vanilla Policy Gradient (VPG) and Trust Region Policy Optimization (TRPO). VPG often suffers from high variance in policy updates, leading to slow and unstable learning. TRPO, on the other hand, requires computationally expensive second-order optimization techniques, making it less practical for large-scale applications.

1.3 Prolog

Prolog[5] is a declarative programming language that allows developers to express a problem's logic and constraints rather than specifying the control flow. It employs a rule-based approach where knowledge is represented as facts and rules. This declarative nature makes Prolog particularly suited for applications involving knowledge representation, expert systems, and logical reasoning.

1.3.1 Logic Programming Paradigm Prolog follows the logic programming paradigm, based on first-order predicate logic. Programs in Prolog consist of a collection of facts and rules that define relationships and logical implications. Through a process called resolution, Prolog can infer solutions by unifying the input with the logical rules, providing a powerful mechanism for automated reasoning.

1.3.2 SWI-Prolog SWI-Prolog[6] is one of the most popular and widely used implementations of Prolog. It provides a robust and efficient environment for developing Prolog programs. SWI-Prolog offers an extensive set of built-in predicates, modules for organizing code, and support for various input/output operations. Its interactive interpreter enables developers to test and debug Prolog code efficiently.

1.3.3 Pyswip Pyswip[7] is a Python library that serves as a bridge between the Prolog language and Python. It enables seamless integration of Prolog programs within Python code, allowing developers to leverage the power of Prolog in their Python-based applications. Pyswip provides a convenient interface for calling Prolog predicates, asserting facts, and querying the Prolog Knowledge Base, enabling a smooth interaction between Prolog and Python.

2 The vollAIball Environment

The goal of this project is to train autonomous agents to play the game of vollAIball.
The environment has been created using Unity3D, where also the rules of the game are coded.
The Unity environment communicates with a Knowledge Base implemented using SWI-Prolog through a Python backend, which exposes APIs to store facts and retrieve the commentary of the game.
The commentary is then printed in a screen placed on the side of the field in the Unity scene.
Most of the efforts have been focused on a 1vs1 scenario, but some exploration also has been made for a 2vs2 setup.

2.1 Unity environment

The Unity environment consists of the field where the game takes place, the agents and the ball. The field is a complex environment, consisting of the terrain, the walls, the net, and some invisible areas which are used to trigger events.

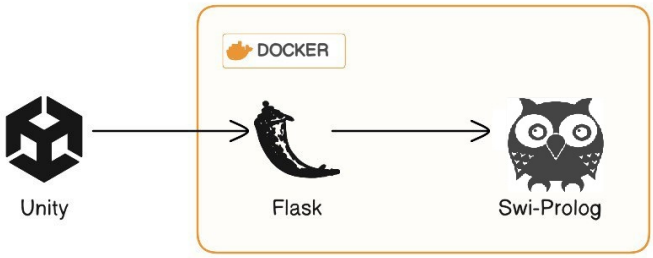

Figure 4: The project software architecture. The Unity environment communicates with a Flask server through REST APIs. The Flask server uses SWI-prolog to assert fact and retrieve the result of queries, which are then returned to the Unity environment to be used as commentary.

Some physics properties have been overridden, e.g. the gravity force has been made almost 3 times stronger to provide a more realistic simulated environment The rules of the game are coded into the Unity environment. During the experiments, numerous versions of the rule have been tested. The final rules are the following:

- A server is randomly chosen among all of the agents. The server is placed near the corner. The ball is placed near the server, above him.
- The same agent cannot touch the ball two times in a row.
- The ball can bounce on the walls and the games continues.
- If the ball touches the ground, the point is assigned to the proper team and the scene is reset.
- If the ball goes over the walls, the point is assigned to the proper team and the scene is reset.

During the game, two screens are placed into the field. One of them shows the commentary of the game, the other shows the score (the score is also computed by the Knowledge Base).

2.2 Prolog Narrator

To provide dynamic and informative commentary during the game of vollAIball, a Knowledge Base implemented using SWI-Prolog is utilized.

To achieve real-time commentary in the Unity environment, a REST API is utilized to communicate with the Prolog Knowledge Base. The Unity environment periodically polls the commentary endpoint using a 1-second interval. This ensures that the commentary information is continuously updated and available for display.

The commentary endpoint serves as the interface between the Unity environment and the Prolog Knowledge Base. It provides information about all the game events, allowing the Unity environment to retrieve the latest commentary based on the registered events. This

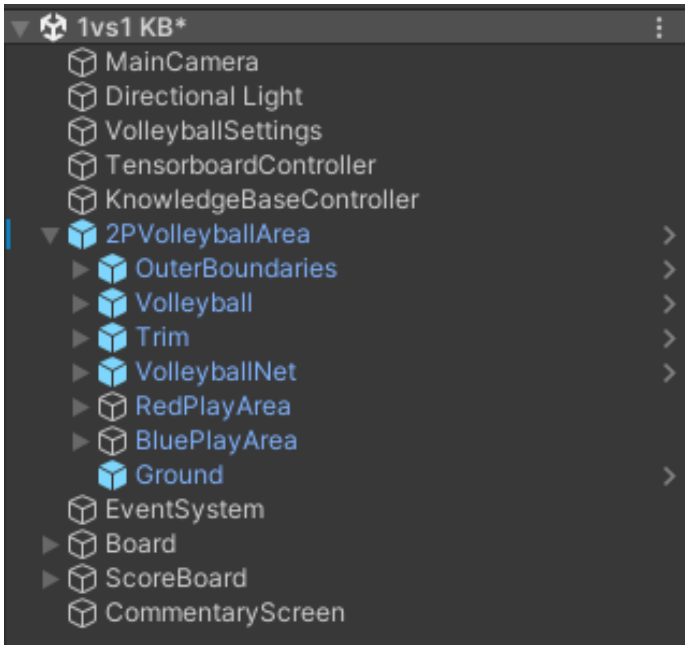

Figure 5: Hierarchy of the objects in the Unity environment. Not all the layers have been expanded for spacing reasons.

endpoint execute the *all_narratives* query against the Knowledge Base, which wraps all of the other queries, that in turn return a textual commentary of the game.

Because the query is general and doesn't take into account the current action, the cumulative commentary of the whole game is returned each time that the endpoint is called. To address this problem and obtain a real-time commentary, the narrative sentences are stored on the Unity side in a hash-set, which inherently avoids duplicates. Each time that the endpoint is called, only the new comments are displayed and the stored commentary is updated.

The commentary generation process involves several steps:

- **Knowledge Base Initialization**: When the backend server is bootstrapped, all of the rules containing the reasoning and commentary logic are defined into the Knowledge Base
- **Event Registration**: As the game progresses in the Unity environment, relevant game events, such as a successful score, a player's action or the ball falling outside of the field are registered in the Prolog Knowledge Base. These events are stored as facts, representing the history of the relevant events of the game
- **Commentary Retrieval**: Using the exposed APIs, the Unity environment retrieves commentary from the Prolog Knowledge Base based on the registered game events and the pre-defined rules. The extracted information are already in a human readable format thanks to the formatting capabilities of the Prolog

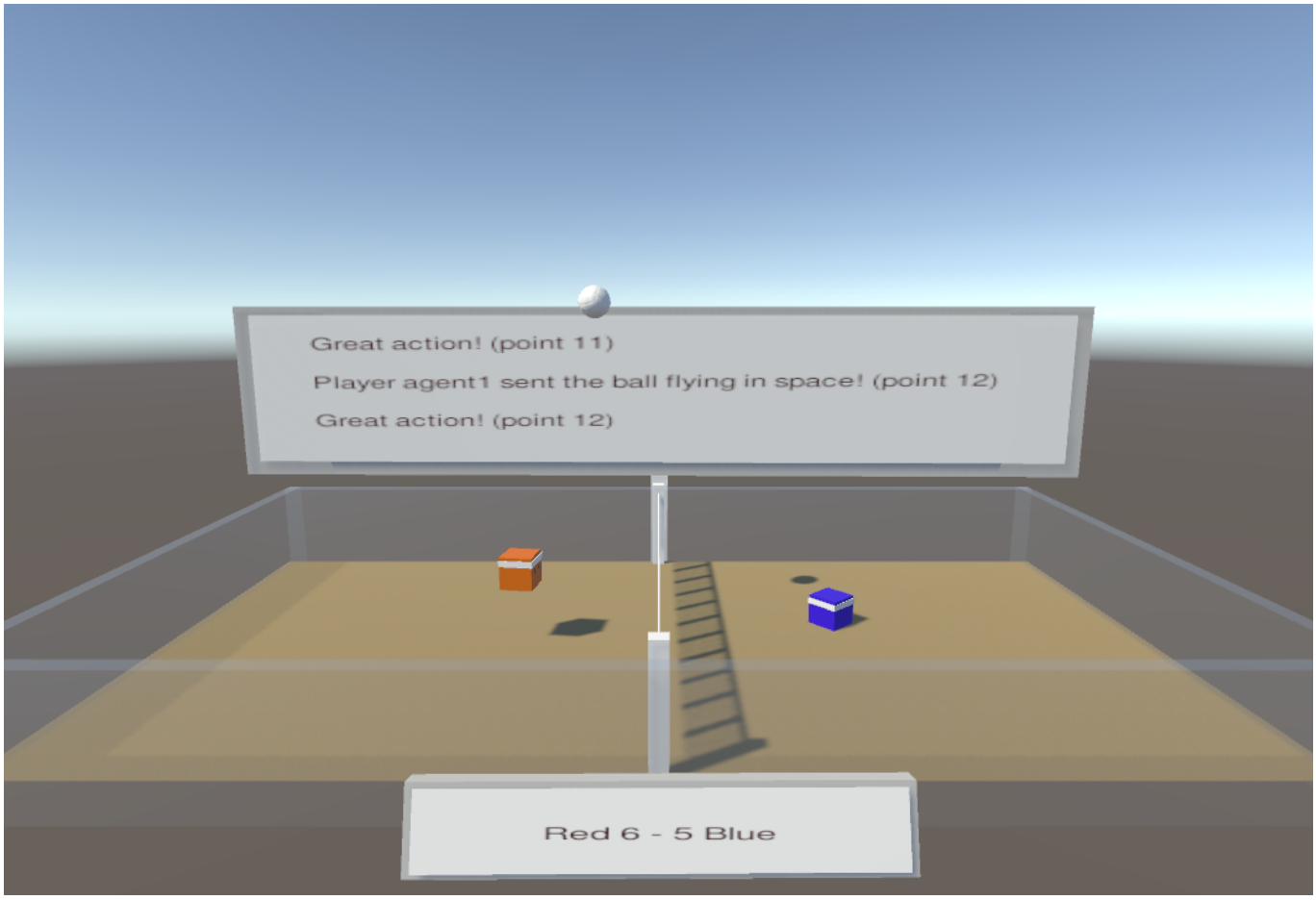

Figure 6: Frame captured during a game run with the final version of the project.

language, which are exploited by the custom predicates detailed in the next section.

- **Display in Unity Scene**: The commentary text is displayed on the screens placed in the Unity scene.

The Prolog commentary process enhances the overall gaming experience by providing real-time insights and analysis of the game events. It adds depth and immersion to the vollAIball game.

2.3 Knowledge Base and Queries

In the volleAIBall project, a Prolog Knowledge Base has been developed to capture and represent the events and information related to the volleyball game. The Prolog language provides a powerful framework for knowledge representation, logical reasoning, and querying. In this section, we explore the process of querying the volleAIBall Prolog Knowledge Base using SWI-Prolog, a widely used Prolog implementation.

2.3.1 Facts The Unity environment sends the game events to the Knowledge Base, which are asserted as facts. The tracked events are the following (*for more details consult the appendix*):

- team
- player
- playsinteam
- hitbluegoal
- hitredgoal
- hitIntoBlueArea
- hitIntoRedArea
- hitoutofbounds
- hitwall
- doubletouch
- touchplayerataction

2.3.2 Narratives At startup time, some predicates are defined to produce the narrative for specific game events. For example, the following are the narratives for when a goal or a own-goal is scored:

```
narrative(Text) :-
    hitbluegoal(X, Y, _),
    playsinteam(X, blue),
    format(atom(Text),
    "Player ~w scored! (point ~w)", [X, Y])
```

```
narrative(Text) :-
    hitbluegoal(X, Y, _),
    playsinteam(X, red),
    format(atom(Text),
    "Player ~w made a own-goal -.-' (point ~
    w)", [X, Y])
```

The predicates, when resolved, produce a textual result like *Player Agent1 scored! (point 1)* and *Player Agent2 made a own-goal -.-' (point 2)*".

2.3.3 The all_narratives predicate To retrieve all of the narratives, the **all_narratives/1** query is defined to act like a wrapper:

```
all_narratives(Narratives) :-
    findall(Text, narrative(Text),
    Narratives).
```

Let's break down the components of the query:

- **Head**: The head of the query, *all_narratives(Narratives)*, defines the predicate *all_narratives/1*. The single argument Narratives represents the list that will hold the retrieved narratives.

- **Body**: The body of the query utilizes the *findall/3* predicate to gather all instances of Text where the narrative(Text) predicate holds true. The *findall/3* predicate takes three arguments: the variable to collect (Text), the goal to satisfy (narrative(Text)), and the resulting list (Narratives).

3 Experiments

Numerous experiments have been run to try to achieve the best outcome. Many iterations have been made to craft the final punishment/reward schema, and to fine-tune the agents observations, the physics environment and the rules of the game.
One of the crucial aspects in Reinforcement Learning environments is to avoid to give to the agents the ability to exploit the environment and the reward schema. For example, when a positive reward was given to the agents at each time step to reward their ability to keep the game going, they learned to block the ball between their body and the wall. If this was done with only one touch, it didn't trigger the double touch rule and so the agent kept getting rewards without actually playing. Another example is when a positive reward was given for each time the agent touched the ball. Even if it was smaller than the reward obtained by throwing the ball in the opposite field, the agents learned to bounce the ball on their head. This was the reason the double touch rule was introduced as an hard constraint.

The training is conducted using the ML-Agents library and the PPO algorithm. The final configuration for the algorithm is the following:

```
behaviors:
  Volleyball:
    trainer_type: ppo
    hyperparameters:
      batch_size: 2048
      buffer_size: 20480
      learning_rate: 0.0002
      beta: 0.003
      epsilon: 0.15
      lambd: 0.93
      num_epoch: 4
      learning_rate_schedule: constant
    network_settings:
      normalize: true
      hidden_units: 256
      num_layers: 2
      vis_encode_type: simple
    reward_signals:
      extrinsic:
        gamma: 0.96
        strength: 1.0
    keep_checkpoints: 100
    max_steps: 200000000
    time_horizon: 1000
    summary_freq: 20000
```

3.1 Metrics Tracking

When the reward schema is frequently changing, it is difficult to compare different experiments using the cumulative reward as a metric. Also, if the reward schema is complex, it is not always clear what the cumulative reward value actually means, and it is difficult to spot bugs or problem in the environment. For this reason, a module was developed to track, in Tensorboard, statistics about the various events in the game. Some of the statistics tracked are the number of times each agents successfully throws the ball in the other field, the number of times the ball touches the wall or goes outside of the field etc. *(For a more comprehensive list, see the Appendix)*

3.2 Observations and Actions

When working with Reinforcement Learning, two of the most fundamentals aspects to design are the observations and the actions.
The *observations* are what the agents can see about the environment in which they are placed. Using a simulated environment like Unity allows to directly access properties like positions and rotations of the components of the environment, which simplifies a lot the task of learning. In a real-word scenario, the agent would probably have to rely on cameras or physical sensors (which is also possible in Unity, using components like cameras and Raycasts).
The *actions* are what an agent can do every time it is requested to act. With ML-Agents, a *Decision-Requester* component asks each agents which action in the environment it wants to perform with a fixed time interval. A short decision interval (many decision request in a short timeframe) makes the agent more

reactive, but makes the task of associating each action with the given reward harder, in turn making the training difficult. Actions can be continuous (like a force vector to apply to the agent) and discrete (like the direction in which to move the agent). The output of the policy, in this case a neural network, is collected and used to perform the proper action in the environment.

In the final version, the agents observation are:

- the relative position of the agent with respect to the ball (3 floats);
- the magnitude of the above vector (1 float);
- the agent rotation in the y axis (1 float);
- the agent velocity (3 floats);
- the ball velocity (3 floats);
- *(in the 2vs2 scenario)* the relative position of the agent with respect to its ally (3 floats);
- *(in the 2vs2 scenario)* the velocity of the agent's ally (3 floats);

for a total of 11 floats in the 1vs1 scenario and 17 floats in the 2vs2 scenario. *The observations setup was inspired by Ultimate Volleyball*[8]

In the final version, the agents can perform actions from 4 discrete branches:

- move forward (3 values: forward, backward, stay)
- rotate (3 values: right, left, stay)
- move to the side (3 values: right, left, stay)
- jump (2 values: jump, no jump)

The output layer of the neural network is so composed of 4 neurons, each taking one of the possible values described above.

3.3 Rewards schema

The final reward schema is the following:

- +1 if the agents sends the ball into the other half of the field
- -1 if the same agent touches the ball 2 times in a row; ends the episode
- -1 if the agents sends the ball out of the field; ends the episode
- -1 if the agent receives a goal; ends the episode

The rewards schema, in reality, sets up a collaborative game instead of an adversarial one. This is because the agents are rewarded to keep the ball in the game instead of them scoring goals.

In a symmetric game like volleyball, it is important to take into account the orientation of the agents, both for the actions and the observations; otherwise, agents would learn to perform the same actions using opposite values, effectively cancelling each other out.

3.4 Basic Experiments

The very first experiments where conducted with spherical model for the agents and continuous actions. In this setup, the actions represented a force vector to apply to the agents to make them move. This design was soon abandoned once it was clear that a spherical model made it impossible to control the ball in a reliable way. The next (and final) model has a cubical design. Giving the model the ability to rotate along its y axis enables directioning of the ball and provides enough agility and freedom to the agent. The choice of having discrete instead of continuous actions helped to keep the complexity of the problem lower.
In order to further decrease the problem complexity, attempts were made to fully discretize the environment: instead of having continuous positions, the game field was divided in a grid. In this setup, the agents could chose in with cell and in which cell send the ball, by drawing from a finite number of possibilities. Also the interaction with the ball was forced so that the ball would always perform a perfect parabola to the desired cell. This design was also abandoned because it gave too much restrictions to the play style.

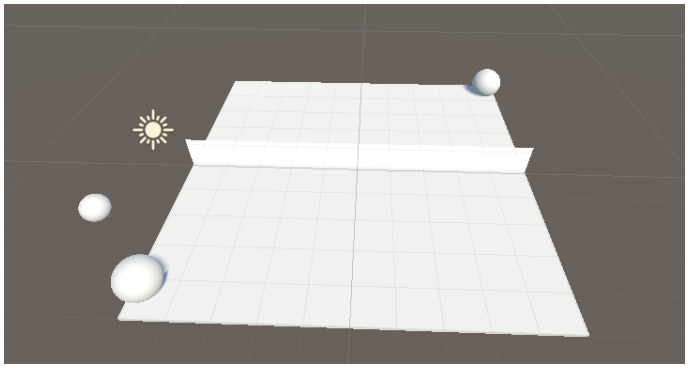

Figure 7: One of the first environment designs. The spherical form of the agents made it so that a small error in the impact angle caused the ball to fly far away.

3.5 1vs1 game

The 1vs1 setup is were most of the efforts were focused. The final model was trained for 77,5M steps (25.5h) on a i7-1165G7 CPU. The training ground was replicated 16 times to speed up the training.

The Cumulative Reward graph shows that the training proceeds faster in the first 20M iterations, almost flattening at around 70M. At first, the ball falls directly to the ground resulting in negative cumulative rewards. Then, the agents learn to touch it and the ball starts going on the walls or off of the field. Occasionally, the agents are able to throw the ball into the other half, and start obtaining positive rewards. This reinforces the actions that allow to throw the ball to the opponent and the Cumulative Rewards starts to grow pretty steadily.

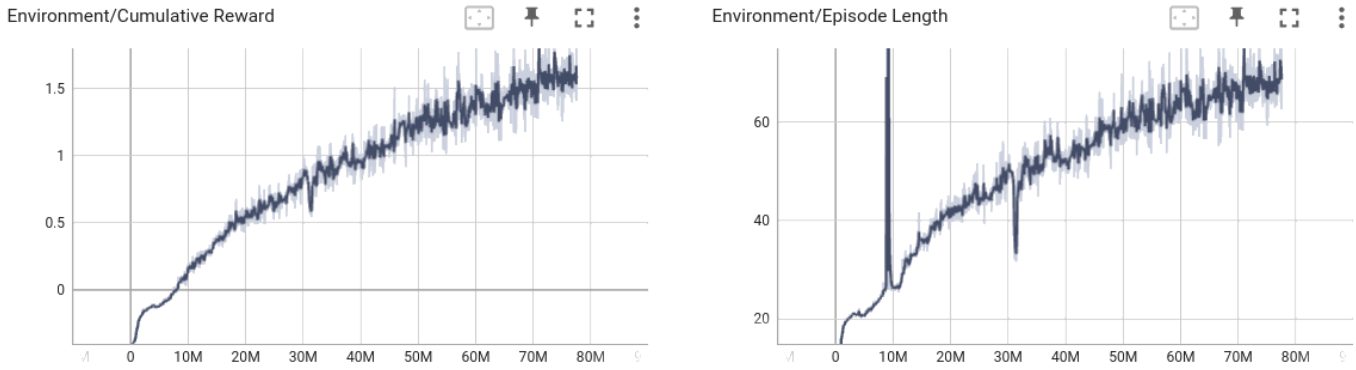

Figure 8: Cumulative reward and Episode Length during training in 1vs1 games. The 2 spikes are due to rare bugs in the environment that have been then fixed.

At 70M iterations, the agents are able to throw the ball back and fort a number of time, which varies depending on their starting positions, which are slightly randomized. At this point, many actions end because the ball bounces on the net or on the walls, making it difficult for the agents to defend. Continuing the training could make the agents learn to respond to these situations; also giving the agents some observations about the walls or the net could help them improve.

3.6 2vs2 game (no training)

Some efforts were made to model a 2vs2 game. The most simple way to attempt to this is to place 2 agent per team, each having the model learned in the 1vs1 setup. As expected, this is not very effective, since each agent tries to get to the ball and ends up obstructing his ally.

3.7 2vs2 game (with training)

A more correct way to approach the problem is to train the model directly in a 2vs2 environment. This involves adding some observations so that the agents can account for their teammate moves.

With the default reward schema, however, the agents learn a very lazy strategy. The server serves the ball, then stays at the edge of the field, leaving the other agent make all of the efforts (even for the receiver team, only one agent actually plays the game).

Altering the rewards schema to promote passes between players of the same team, has some nasty effects too. In particular, the agents which is not serving runs to the server, trying to touch the serving and send the ball into the other half, so that they would get both the rewards (same team touch + pass into the other half). This, however, most of the times results in the ball falling of to the ground or anyway not getting to the other half, making the training fail.
This suggests that more complex reward schemas or rules should be enforced to effectively develop a 2vs2 (or generally, multiple players per team) games.

4 Results and Conclusions

The vollAIball project focused on training autonomous agents to play the game of volleyball in both 1v1 and 2v2 environments using Unity3D and Reinforcement Learning techniques. Through the integration of machine learning algorithms and the utilization of a Prolog Knowledge Base for narration and commentary, the aim was to create immersive and intelligent gaming experiences.

The project highlighted the necessity for integrated solutions for the experiment tracking, and that custom statistics are needed to effectively compare different experiments when the reward schema is frequently changing.

In the 1v1 environment, the agents demonstrated promising performance and showed the ability to learn good behavior and strategies. The Proximal Policy Optimization (PPO) algorithm and the deep reinforcement learning approach allowed the agents to improve their skills over time and give life to some interesting games. It can be speculated that more training would lead to better results, since the reward curve was not completely flat when the training was stopped.

In the 2v2 environment, the aim was to expand the environment and introduce team dynamics. However, significant new challenges emerged in the achieving of strong teamwork and coordination between the agents. The agents often exhibited suboptimal behavior, such as colliding with each other and lacking effective team strategies. Despite attempts to incentivize teamwork through reward mechanisms, the agents did not show significant improvements in their performance. However, it must be noted that the cumulative reward kept increasing during training. Being the environment more complex than the 1vs1 setup, the computational effort was many times bigger, which made the training slower. Moreover, to account for the additional complexity of the environment, the task to learn is intrin-

sically harder. For these reasons, it can't be excluded that continuing the training for a sufficient amount of time would lead to the emersion of significative patterns or strategies.

Overall, the Unity3D environment confirmed its flexibility and capacity of modelling environments of increasing complexity. By exploiting this kind of tools, training and simulations can be run in virtual environments before deploying the models into the real world, making approachable also those scenarios where security, time and costs are among priorities.

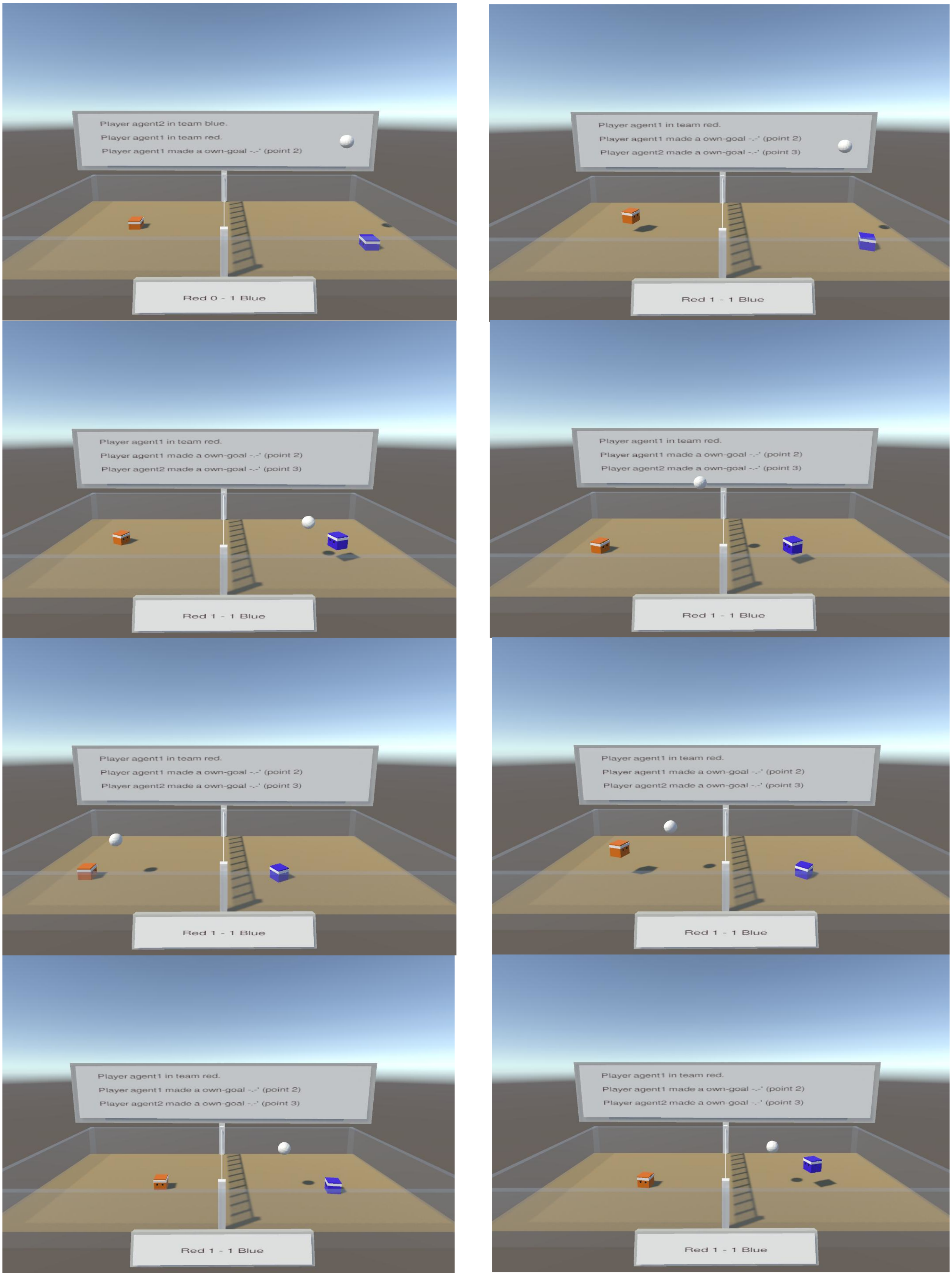

Figure 9: A sequence of frames from a vollAIball game.

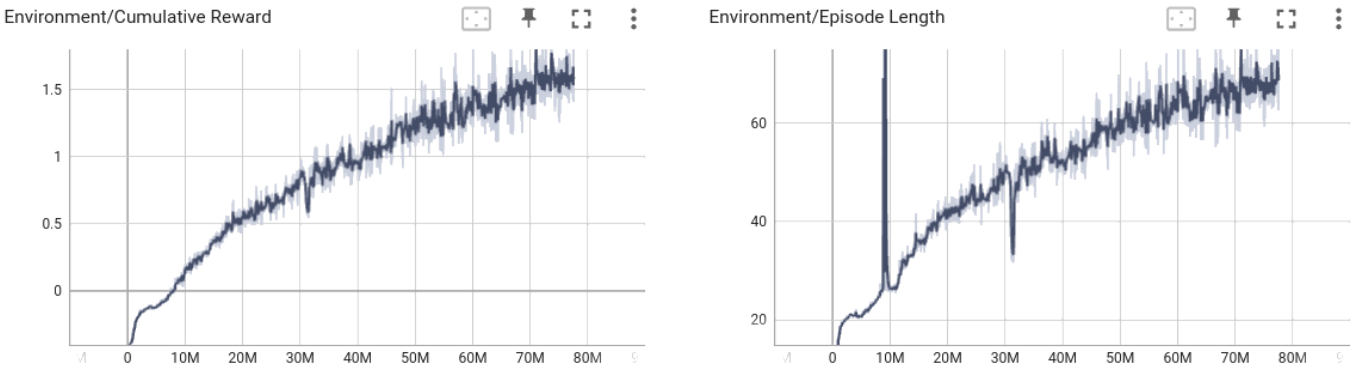

Figure 10: Cumulative reward and Episode Length during training in 1vs1 games. The 2 spikes are due to rare bugs in the environment that have been then fixed.

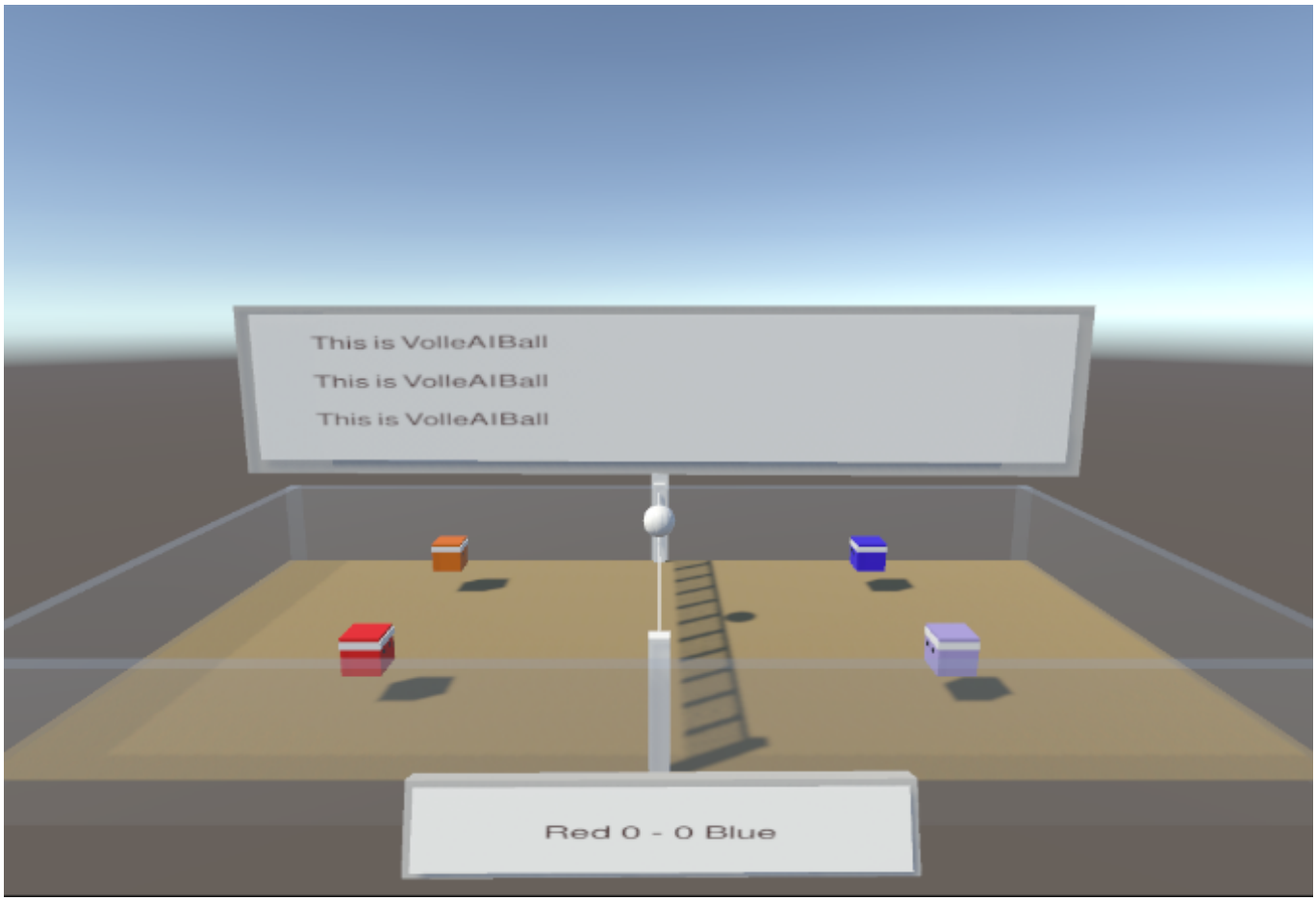

Figure 11: The scene setup for a 2vs2 game.

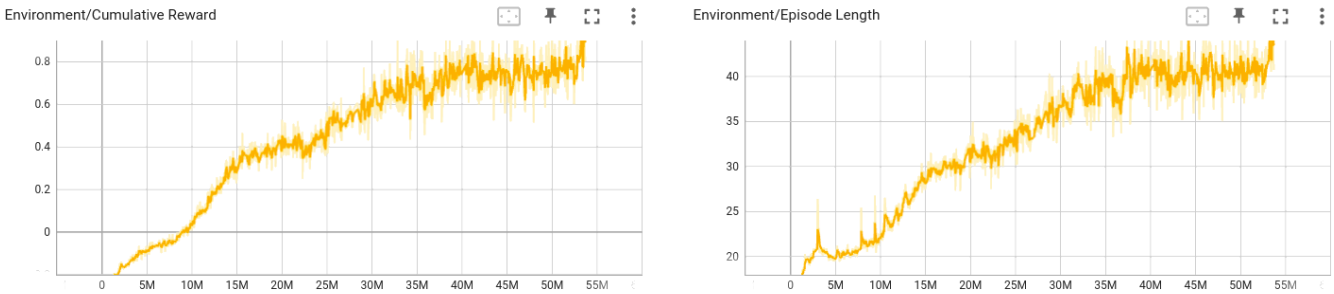

Figure 12: Cumulative reward and Episode Length during training in 2vs2 games.

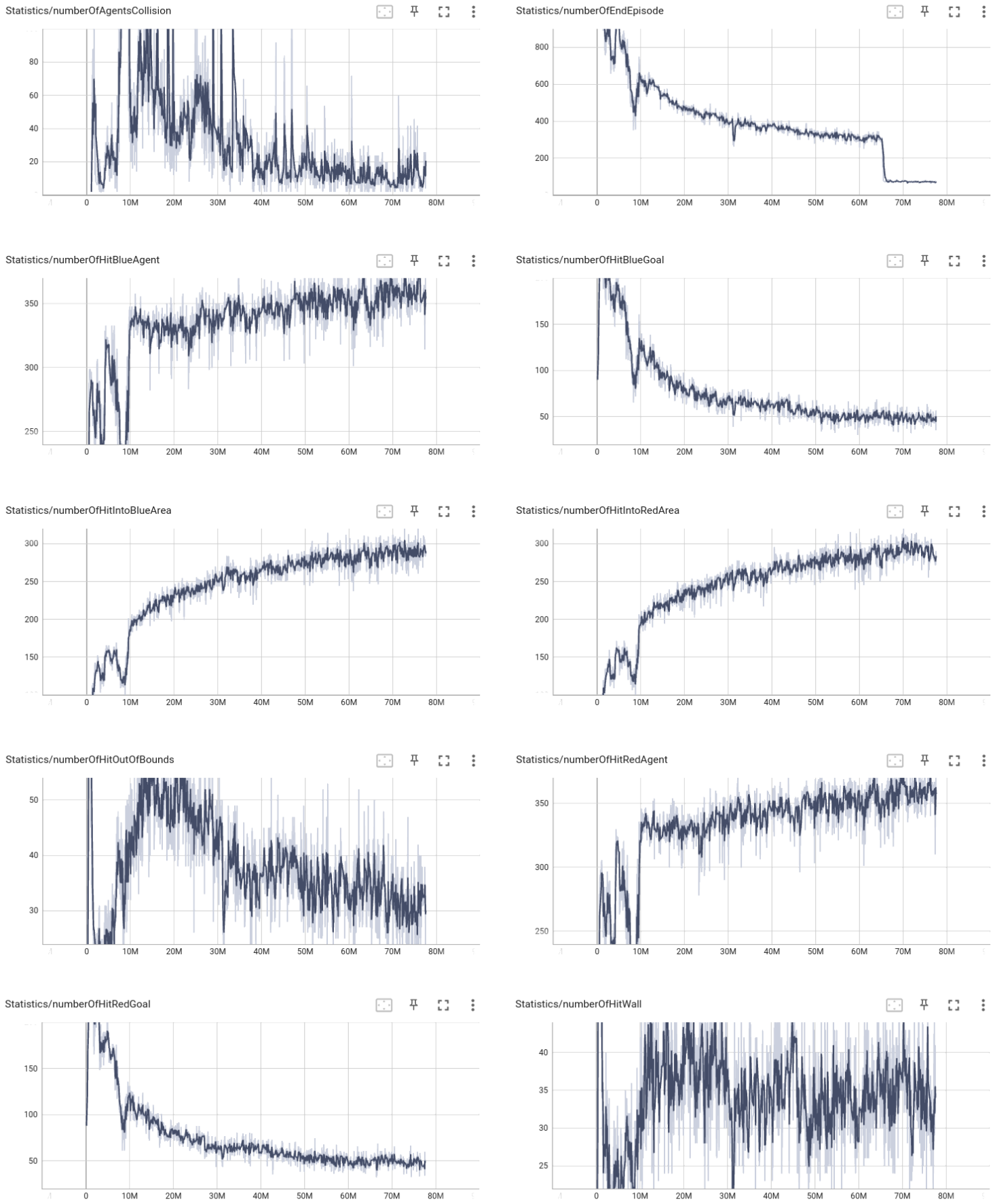

Figure 13: Custom statistics tracking for the final 1vs1 model. The suddon drop of numberOfEndEpisode is due to a change in the statistic tracking method; the data should be intended linear with the previous points in the series.

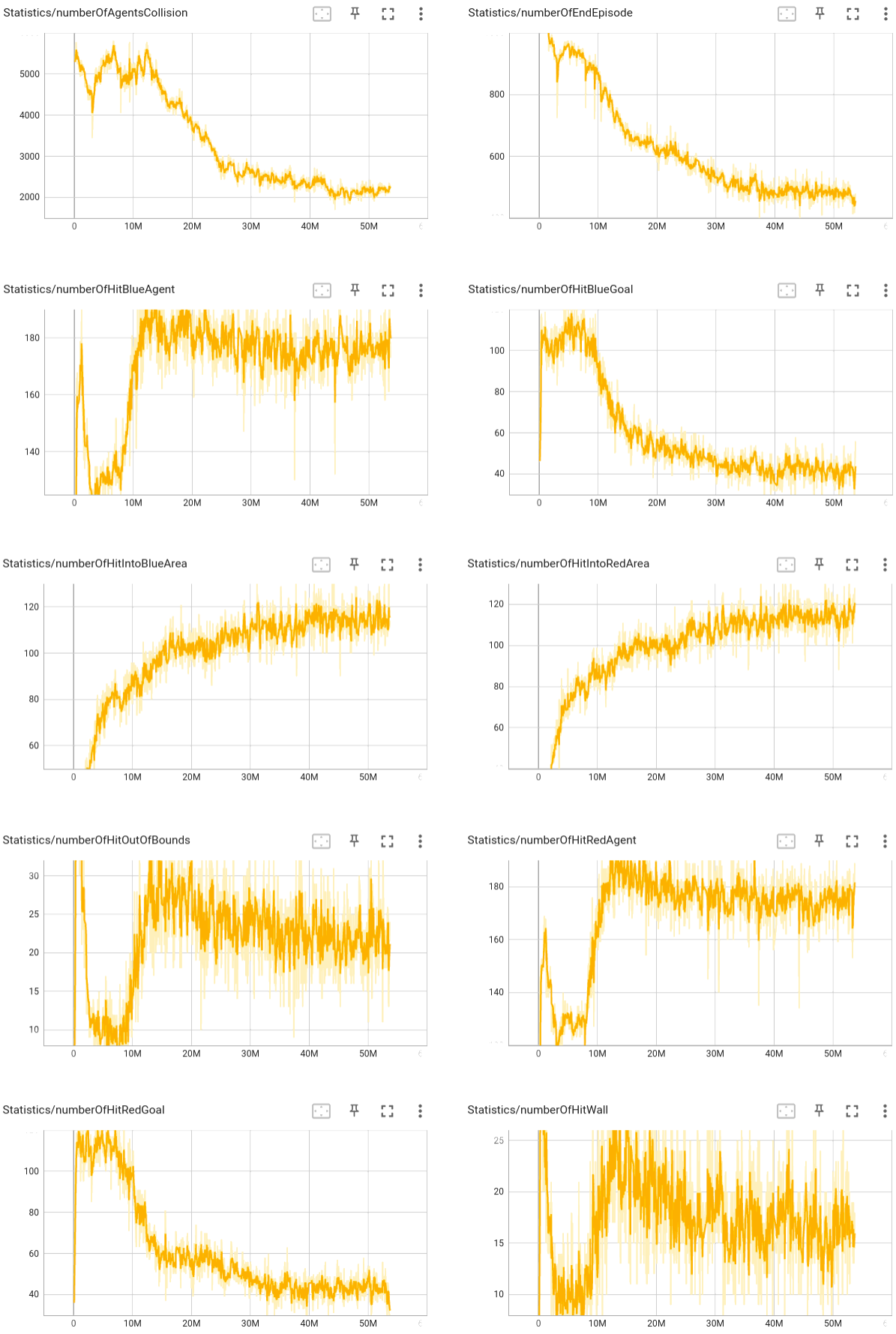

Figure 14: Custom statistics tracking for the final 2vs2 model.

Knowledge Base Insigths

The following sections reports the definition of the Knowledge Base for the vollAIball project, using the *pyswip* library.

Dynamics The narratives query use the definition of facts that may not be present in the Knowledge Base when the query is run, causing the execution to fail. The **dynamic/1** predicate is used to declare that certain predicates can be modified and updated during runtime, allowing for the addition and removal of facts (data) associated with these predicates. The usage of the **dynamic/1** predicate allows to fail silently when querying for facts that don't exist yet in the Knowledge Base.

```
dynamic("team/1")
dynamic("player/1")
dynamic("playsinteam/2")
dynamic("hitbluegoal/3")
dynamic("hitredgoal/3")
dynamic("hitoutofbounds/3")
dynamic("hitwall/3")
dynamic("doubletouch/3")
dynamic("narrative/1")
dynamic("touchplayerataction/4")
```

Players and team composition Predicates used to assert information about players and team composition

```
assertz("""narrative(Text) :-
    team(X), format(atom(Text), "Team: ~w.", [X])""")

assertz("""narrative(Text) :-
    playsinteam(X, Y), format(atom(Text), "Player ~w in team ~w.", [X, Y])""")
```

Narratives The following snippet adds rules to generate narrative commentary for various game events, including goals, own goals, hitting out of bounds, hitting the wall, and performing a double touch.

```
# Score goals and own goals
assertz("""narrative(Text) :-
    hitbluegoal(X, Y, _),
    playsinteam(X, blue),
    format(atom(Text), "Player ~w scored! (point ~w)", [X, Y])""")

assertz("""narrative(Text) :-
    hitbluegoal(X, Y, _),
    playsinteam(X, red),
    format(atom(Text), "Player ~w made a own-goal -.-' (point ~w)", [X, Y])""")

assertz("""narrative(Text) :-
    hitredgoal(X, Y, _),
    playsinteam(X, red),
    format(atom(Text), "Player ~w scored! (point ~w)", [X, Y])""")

assertz("""narrative(Text) :-
    hitredgoal(X, Y, _),
    playsinteam(X, blue),
    format(atom(Text), "Player ~w made a own-goal -.-' (point ~w)", [X, Y])""")

# Special cases that score a point
assertz("""narrative(Text) :-
    hitoutofbounds(X, Y, _),
    format(atom(Text), "Player ~w sent the ball flying in space! (point ~w)", [X, Y])""")

```

```
assertz("""narrative(Text) :-
    hitwall(X, Y, _),
    format(atom(Text), "Player ~w tried to break the wall! (point ~w)", [X, Y])""")

assertz("""narrative(Text) :-
    doubletouch(X, Y, _),
    format(atom(Text), "Player ~w wants the ball all for himself :O (point ~w)", [X, Y])""")
```

4.0.1 Score calculation

The code segment below performs a calculation to determine the current score of the game. The score of each team is made up by adding:

- The goal made by the team
- The own goal made by the opposite team
- The hit out of bounds of the opposite team
- The number of double touch of the opposite team

The resulting scores are then formatted into a narrative string of the form "Red X - Y Blue", where X represents the score of the red team and Y represents the score of the blue team.

```
# Compute current score
assertz("""narrative(Text) :-
    playsinteam(X1, blue), aggregate_all(count, hitbluegoal(X1, _, _), BlueGoals),
    playsinteam(X2, red), aggregate_all(count, hitbluegoal(X2, _, _), RedOwnGoals),
    playsinteam(X3, red), aggregate_all(count, hitoutofbounds(X3, _, _), RedOutOfBounds),
    playsinteam(X4, red), aggregate_all(count, doubletouch(X4, _, _), RedDoubleTouches),

    playsinteam(X5, red), aggregate_all(count, hitredgoal(X5, _, _), RedGoals),
    playsinteam(X6, blue), aggregate_all(count, hitredgoal(X6, _, _), BlueOwnGoals),
    playsinteam(X7, blue), aggregate_all(count, hitoutofbounds(X7, _, _), BlueOutOfBounds),
    playsinteam(X8, blue), aggregate_all(count, doubletouch(X8, _, _), BlueDoubleTouches),

    plus(BlueGoals, RedOwnGoals, BlueScorePartial1),
    plus(BlueScorePartial1, RedOutOfBounds, BlueScorePartial2),
    plus(BlueScorePartial2, RedDoubleTouches, BlueScore),

    plus(RedGoals, BlueOwnGoals, RedScorePartial1),
    plus(RedScorePartial1, BlueOutOfBounds, RedScorePartial2),
    plus(RedScorePartial2, BlueDoubleTouches, RedScore),

    format(atom(Text), "Red ~w - ~w Blue", [RedScore, BlueScore])"""
)
```

Nice actions The code defines two types of nice actions: "ace" and "great action".

An *ace* occurs when the server scores a point without the opponent team ever touching the ball. For the *ace* narrative, some utility predicates are defined to reduce the complexity of the overall query.

```
assertz("""ace(X1, Y) :-
    hitredgoal(X1, Y, _),
    playsinteam(X1, red),
    touchplayerataction(X1, Y, _, _),
    playsinteam(X2, blue),
    \+touchplayerataction(X2, Y, _, _)
""")

assertz("""ace(X1, Y) :-
    hitbluegoal(X1, Y, _),
    playsinteam(X1, blue),
    touchplayerataction(X1, Y, _, _),
    playsinteam(X2, red),
    \+touchplayerataction(X2, Y, _, _)
""")
```

```
assertz("""narrative(Text)
    :- ace(X, Y),
    format(atom(Text),
    "Ace for player ~w (point ~w)", [X, Y])
""")
```

A *great action* is defined as an action with more than 3 total touches by the players:

```
assertz("""greataction(Y) :-
    player(X1),
    \+doubletouch(X1, _, _),
    touchplayerataction(_, Y, _, _),
    aggregate_all(count, touchplayerataction(_, Y, _, _), DribblesInAction),
    DribblesInAction > 2""")

assertz("""narrative(Text) :-
    greataction(Y),
    format(atom(Text),
    "Great action! (point ~w)", [Y])
""")

This query uses the concept of \textit{negation-as-failure} to assert that none of the
    players touched the ball two times in a row, which wuold result in a faul.
```

Nice actions The code segment below defines the *all_narratives* predicate, which retrieves all the narratives from the Knowledge Base. The *all_narratives* predicate is implemented using the *findall* predicate provided by Prolog.

```
assertz("""all_narratives(Narratives) :- findall(Text, narrative(Text), Narratives)""")
```

Knowledge Base Assertions
List of game events registered in the Knowledge Base as facts

Predicate	Parameter	Description
team/1	team_name	Assert team in game
player/2	player_name	Assert player in game
playsinteam/2	team_name, player_name	Specifies the player's membership in a team
hitbluegoal/3	player_name, point, action	Assert that the blue team scored a goal
hitredgoal/3	player_name, point, action	Assert that the red team scored a goal
hitIntoBlueArea/3	player_name, point, action	Assert that the ball entered the blue team half
hitIntoRedArea/3	player_name, point, action	Assert that the ball entered the red team half
hitoutofbounds/3	player_name, point, action	Assert that a player sent the ball out of the field
hitwall/3	player_name, point, action	Assert that the ball hit the wall
doubletouch/3	player_name, point, action	Assert that a player touched the ball two times in a row
touchplayerataction/4	player_name, point, action, touch	Assert that a player hit the ball

Table 1: Parameter Insight, player_name: the player that performed the action, usually it is the last one to hit the ball; point: current point being played, action: current action being played; an action is defined as consecutive ball touches of players in the same team, number of touch: progressive counter of ball touches inside an action

References

[1] Shweta Bhatt. "Reinforcement Learning 101". In: 4 (). URL: https://towardsdatascience.com/reinforcement-learning-101-e24b50e1d292.

[2] Quang Trung Luu. "Q-Learning vs. Deep Q-Learning vs. Deep Q-Network". In: 3 ().

[3] Josh Petty. "What is Unity 3D What is it Used For?" In: 1 (). URL: https://conceptartempire.com/what-is-unity/.

[4] Thomas Simonini. "Proximal Policy Optimization (PPO)". In: 2 ().

[5] Wikipedia. "Prolog". In: 6 (). URL: https://en.wikipedia.org/wiki/Prolog.

[6] Wikipedia. "SWI-Prolog". In: 7 (). URL: https://www.swi-prolog.org/.

[7] yuce. "Pyswip". In: 7 (). URL: https://github.com/yuce/pyswip.

[8] Joy Zhang. "Ultimate Volleyball: A multi-agent reinforcement learning environment built using Unity ML-Agents". In: 5 (). URL: https://towardsdatascience.com/ultimate-volleyball-a-3d-volleyball-environment-built-using-unity-ml-agents-c9d3213f3064.