Unity ML Agents: Wall Jump and SoccerTwos environment using Reinforcement Learning (RL) tecnique

Thesis · January 2024

DOI: 10.13140/RG.2.2.32633.65125

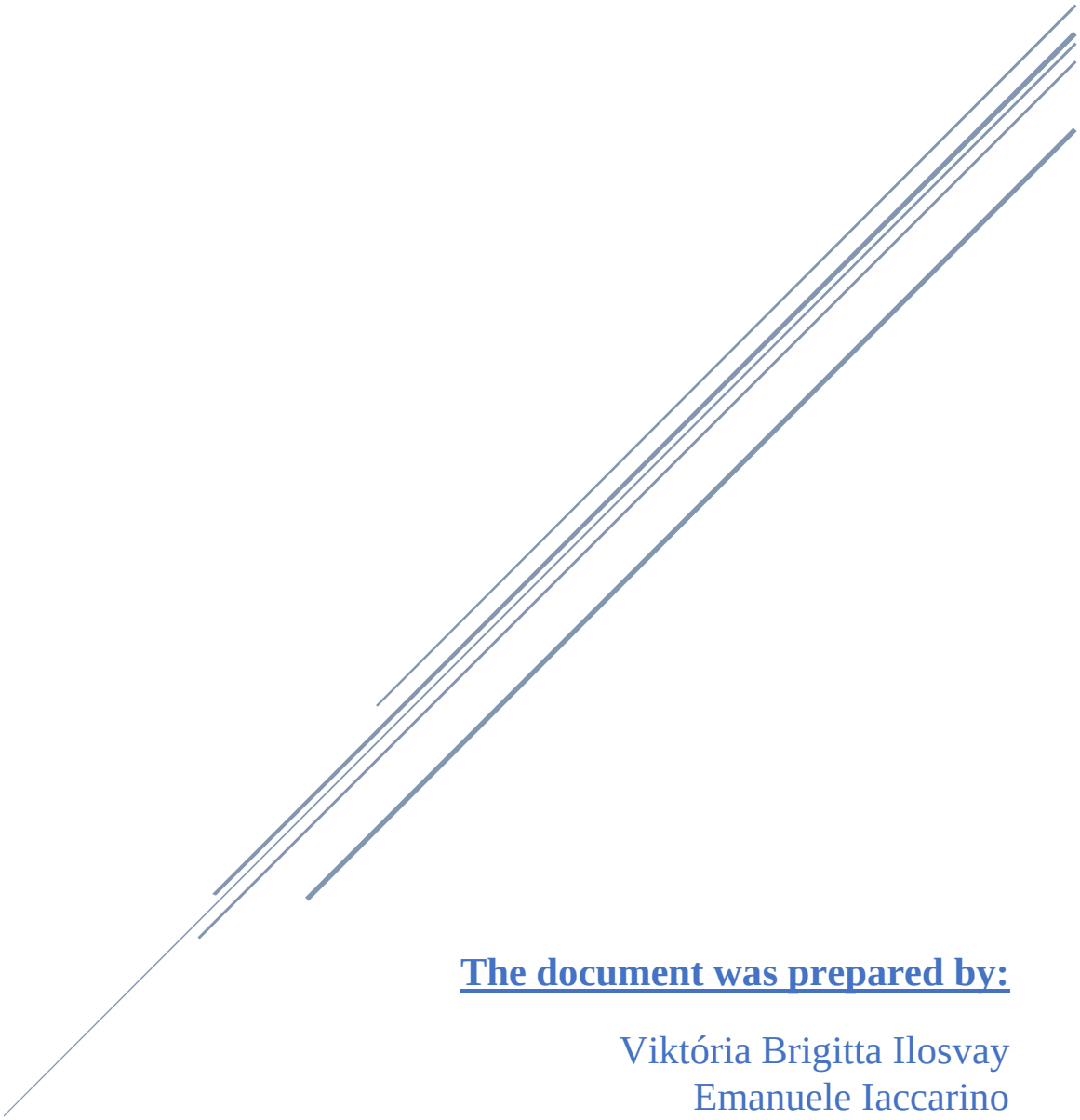
2 authors, including:

Emanuele Iaccarino
University of Naples Federico II
3 PUBLICATIONS **2** CITATIONS

SEE PROFILE

Reinforcement Learning, Final Project

Implementation and exploration of reinforcement learning techniques with Unity



The document was prepared by:

Viktória Brigitta Ilosvay
Emanuele Iaccarino

Deusto
Universidad de Deusto

Table of contents

Introduction to Unity

Nowadays, many developers use Unity, the development environment released by Unity Technologies in 2005, which offers a wealth of possibilities. Unity has grown to become one of the best, if not the best, multiplatform game engines, allowing programmers to write programs for Windows, MAC OS or Android. It also offers the possibility to develop for consoles such as Xbox, PlayStation or even Nintendo Switch and developers can also create games for VR glasses.

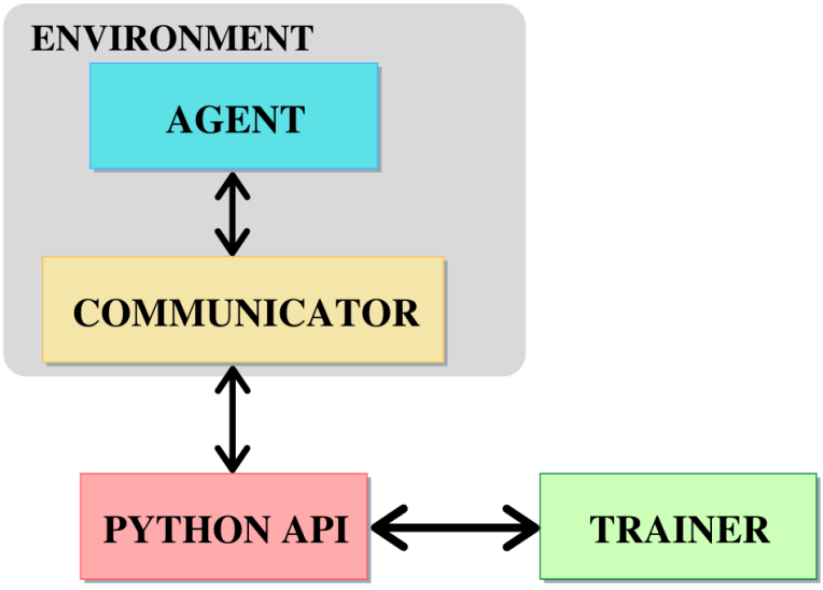
First of all, game development has grown over the years into a very significant industry. The coronavirus pandemic has also significantly increased the consumer market, as many of the people who were forced to stay at home have turned to this form of entertainment. It is worth noticing that game developers have a very large pressure, as the market is highly competitive due to consumer expectations focusing on quality. The global consumer market creates demand, but it is on companies if they exploit this opportunity with success or eventually, they fail to do so. To summarize, it is very important that a game is well developed and has an exciting plot. As for the competitors, it is essential that they are well adapted to every situation, as they should challenge the user as much as possible in order to provide the best possible gaming experience. In addition, a stable runtime is crucial to retain the consumer market. Unity is a perfect platform for game development. This has been proven by many successful programs developed in this environment. Games like Hearthstone, Rust, Pokémon Go, Cuphead and Untitled Goose Game have all been made in Unity and have been successful.

Apart from game development, robotics holds significant importance in the market. If we think about the future, almost everyone imagines that many more things will be automated. Starting with all the fixed tracked public transport vehicles, but also self-driving cars on the roadways and later eventually buses will take place. In many areas of life, robots are likely to play a much bigger role to make life easier. Since we want to give them a high level of responsibility, programmers cannot afford to have an accident due to any programming error. At the same time, the law also comes into play: if a robot makes a mistake and causes a major accident, it is worth considering who can be held to blame. However, this is something that will be answered only in the future.

In both cases, it is very important how agents adapt to different circumstances, how they will act in different situations. To achieve the best possible outcome, the programmers must choose between Supervised, Unsupervised and Reinforcement Learning. From these three the main advantage of the Reinforcement Machine Learning paradigm over the other two is that it allows the agents to adapt easily to changing environments, as they can update their knowledge of the environment based on their experience.

Unity Machine Learning Toolkit

The Unity Machine Learning Toolkit is an open-source add-on package to the Unity game engine. It enables the developers to train behaviors using reinforcement learning by utilizing the open-source PyTorch reinforcement learning library. PyTorch enables training AI agents using CPU and GPU and the resulting Neural Network (NN) behavior model can be used to control the agents in the Unity engine. The ML-agents toolkit consists of three main components: the agent, the environment, and available actions for the agent. Agents share their experiences during the training. Their policy must access player inputs, scripts, and neural networks through Python. For this purpose, the agent uses Communicator to connect to the trainers through the Python API.



***1. Figure**: Communication between the agents and the trainers*

Deusto
Universidad de Deusto

Installation and Setup

Installing Unity Hub and Unity Editor

1) Downloading Unity Hub

Visit the official Unity website and click on the "**Download Unity Hub**" button. Once the download is complete, double-click the installer to initiate the installation process.

2) Installing Unity Hub

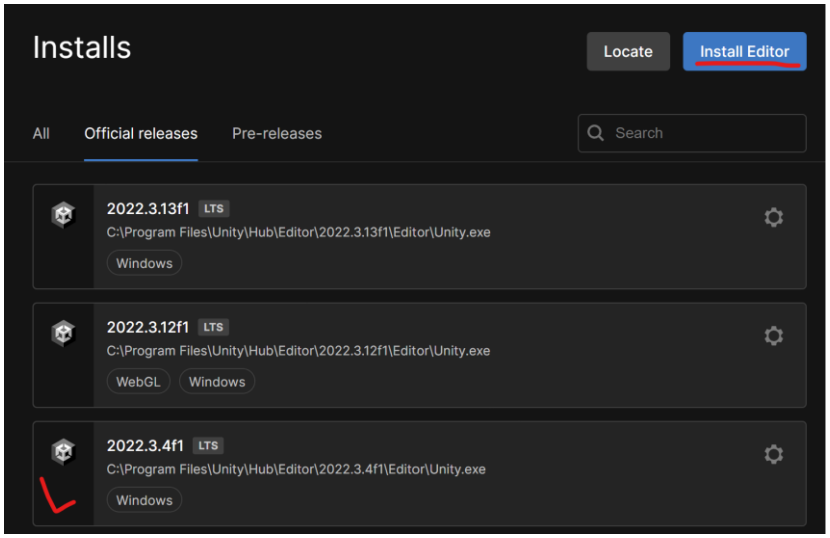
Follow the on-screen instructions, selecting the default options throughout the installation. After installation, launch "**Unity Hub**" to proceed.

3) Accessing or Creating a Unity Account

Sign into Unity Hub with your existing Unity account credentials. If you don't have an account, click on "**Create Account**" in Unity Hub and provide the required information.

4) Installing Unity Editor

Launch Unity Hub and navigate to the "**Installs**" tab. Click "**Add**", select the **2022.3.4f1** Unity Editor version, and initiate the installation process by clicking "**Install**".



***2. Figure**: The installed 2022.3.4f1 version*

Obtaining ML-Agents

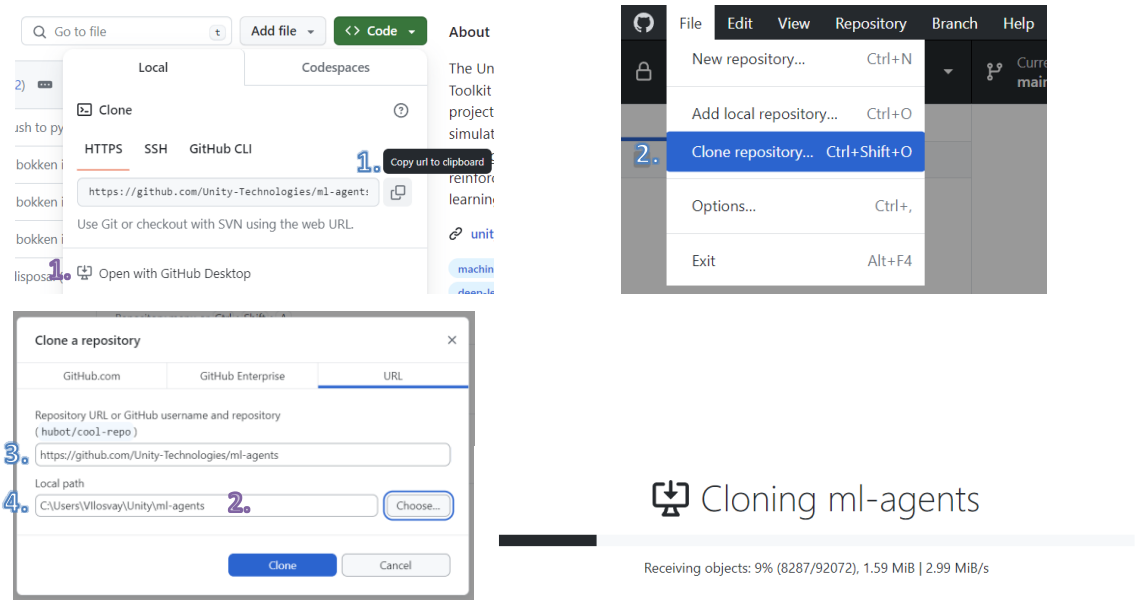
A. Download Unity ML-Agents from GitHub via zip file

Visit the Unity ML-Agents GitHub **Releases page**. Download the latest release's source code (.zip file). Unzip the directory to a convenient location on your computer.

B. Cloning repository from GitHub

You can also clone the repository from Git Hub. For this you need to open a **terminal** window, navigate to the desired directory and finally clone the Unity ML-Agents repository using the following **command**: `git clone https://github.com/Unity-Technologies/ml-agents.git`

For the cloning you can also use the 'GitHub Desktop" application, which can be downloaded from its **webpage**. This app offers two new cloning options. One is to **copy the URL** of the repository from GitHub, then click on the "**Clone repository…**" under the "**File**" menu item in the app, where you switch to the URL tab and insert the link from the clipboard. Once this is all done, you just need to give the location where you want to save the project. (Blue on 3. Figure) The other option is to open the app from GitHub under the "**Code**" button with the "**Open with GitHub Desktop**" option, and then just designate the location for it. (Purple on 3. Figure)



3. Figure: *Process of cloning the repository in two ways*

Deusto
Universidad de Deusto

Installing Anaconda and its dependencies

1) Installing Anaconda

Visit the **Anaconda download page** and **download** the latest version. (e.g., Anaconda3-2023.09-0-Windows-x86_64) After the download, run the **Anaconda Installer**, choosing the default options.

2) Creating a Conda Environment

Open the **Anaconda Prompt** (miniconda3) after installation. Execute the following **commands** for the creation and activation of a new environment with the "**mlagents**" name:

```
conda create -n mlagents python=3.10.12
conda activate mlagents
```

3) Installing PyTorch

Run the following **command** in the Anaconda Prompt:

```
pip3 install torch~=1.13.1 -f https://download.pytorch.org/whl/torch_stable.html
```

4) Installing ML-Agents

In the terminal **navigate** to the directory where the repository is:

```
cd /path/to/ml-agents
```

Install the necessary **dependencies**:

```
python -m pip install ./ml-agents-envs
```

If encountering the following issue, manually install **NumPy**:

```
      [end of output]

  note: This error originates from a subprocess, and is likely not a problem with pip.
  ERROR: Failed building wheel for numpy
Successfully built mlagents-envs
Failed to build numpy
ERROR: Could not build wheels for numpy, which is required to install pyproject.toml-based projects
```

```
pip install numpy==1.23.1
```

*(Originally, we downloaded the 1.21.1 version, but later we had problem with this – check this at **Troubleshooting**)*

Repeat the ML-Agents installation:

```
python -m pip install ./ml-agents-envs
python -m pip install ./ml-agents
```

Without this problem just run the following command:

```
python -m pip install ./ml-agents
```

Troubleshooting

A. Incompatibility between the versions of PyTorch and NumPy

Originally, we had an error after installation when we tried to train a new model due to the incompatibility between the latest version of PyTorch and the downloaded version of NumPy. We downloaded the 1.21.1 version of the NumPy by the following command:

```
pip install numpy==1.21.1
```

Because of this we encountered the following NumPy initialization warning:

```
[W ..\torch\csrc\utils\tensor_numpy.cpp:77] Warning: Failed to initialize NumPy: module compiled
against API version 0x10 but this version of numpy is 0xe (function operator ())
```

For this problem we found the solution **here**, which said we need another version of the NumPy, so we uninstalled the 1.21.1 version and then we ran the following command for installing a newer one:

```
pip install numpy==1.23.1
```

Once we resolved this and then tried to train a new model everything worked correctly.

B. TensorBoard logs

According to some documentations (Using TensorBoard to Observe Training, Monitoring training with TensorBoard) the TensorBoard logs supposed to be saved in the "**summarise**" folder, however for us it was saved into "**results**". It was a little bit confusing, but these documents are very old ones, so we came to the conclusion, that in the past it was like this, and Unity changed the default directory for the logs from summarise to results.

To see the plot via TensorBoard in localhost the following command must be given out:

```
tensorboard --logdir=results
```

Resolution and Testing in the Environment

Introduction to the used reinforcement learning algorithms

Imitation Learning

The idea of imitation learning is that an agent is learning from the experiences of other agents. Instead of taking experience from the world and making its own decision it might watch another agent (can be either a person or an already trained RL agent) and use that experience to figure out how it is supposed to act. This learning process is useful when it's easier to demonstrate the desired behaviour than to specify a reward function for the agents.

Behavioural Cloning (BC)

Behavioural Cloning is the simplest **form of imitation learning**. This learning process focuses on learning the expert's policy using supervised learning. The experts' demonstrations are divided into state-action pairs which can be considered as independent and identically distributed examples. After this, supervised learning is applied.

Proximal Policy Optimization (PPO)

This is a RL algorithm which optimizes a policy by using a stochastic ascent to optimize a surrogate objective function. PPO is based on TRPO; instead of maximizing the objective function with a hard constraint PPO penalizes changes on the policy that move $rt(\theta)$ away from 1 encouraging more cautious updates to the policy, where $rt(\theta) = \frac{\pi_\theta(a|s)}{\pi_{\theta_{\text{old}}}(a|s)}$ that compares the probability of an action under the current policy $\pi\theta$ against its probability under the previous policy $\pi\theta_{old}$. This is done by the formula: $L^{\text{CLIP}}(\theta) = \hat{\mathbb{E}}_t[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)]$

Where ε is the exploration hyper parameter, the first term is the minimum of the unclipped objective and the second term is a clip function that modifies the surrogate objective by clipping the probability ratio, thus, removing the incentive of moving outside of the range [1 − ε , 1 + ε].

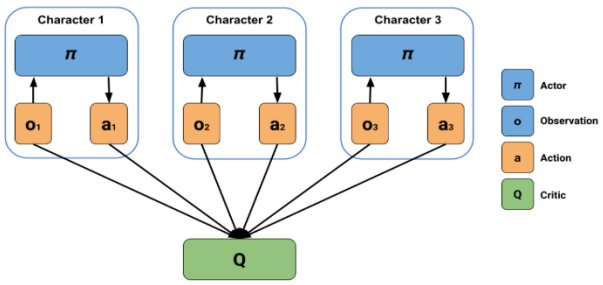
Soft Actor-Critic (SAC)

This is a RL algorithm which maximizes the expected return and entropy needed to get an optimal policy: $J(\pi) = \sum_{t=0}^{T} \mathbb{E}_{(\mathbf{s}_t, \mathbf{a}_t) \sim \rho_\pi} \left[r(\mathbf{s}_t, \mathbf{a}_t) + \alpha \mathcal{H}(\pi(\cdot \,|\mathbf{s}_t)) \right]$

This is achieved by parameterizing a Gaussian policy that act as the Actor (allowing for a natural exploration mechanism in continuous action spaces) and a soft Q-function (that works as a Critic that evaluates these actions, estimating the expected return + the entropy of the policy) and a replay pool which store experience gathered during interaction with the environment and used to update both the Actor and Critic networks: this sampling method allows SAC to break correlation between consecutive experience, reducing variance in training and stability in the learning process.

Policy Optimization by Continuous Approximations (POCA)

The POCA algorithm was developed by the ML-Agents research team in 2020 and represent a huge discover in the domain of multi-agent systems: The high-level explanation is that POCA trains a group of agents to maximize a shared common reward. It also supports the removal/addition of agents during runtime, which is not handled by other multi-agent trainers. The POCA trainer trains policies utilizing a centralized training approach, but each policy acts independently during inference. This means that while the policies of the agents are trained collectively, each agent operates independently based on its policy during actual deployment. This allows the agents to learn strategies and complex interaction not only by an individual perspective but also as a collective.



One of the defining features of POCA is its ability to dynamically adjust to the addition or removal of agents during runtime. This flexibility makes POCA particularly suitable for real-world applications where the number of participants can vary, such as in traffic systems. Moreover, POCA's framework supports the training of heterogeneous agents, that is, agents with different capabilities or roles within the same environment.

Understanding Unity ML-Agents

The already mentioned GitHub repository also contains a lot of documentation about the ML-Agents. Between these there is a Getting Started Guide document as well, so the absolute beginners (like we were that time) can easily get to know how the environments with ML-Agents work. In this guide the **3D Balance Ball** environment is used to introduce the most important things.

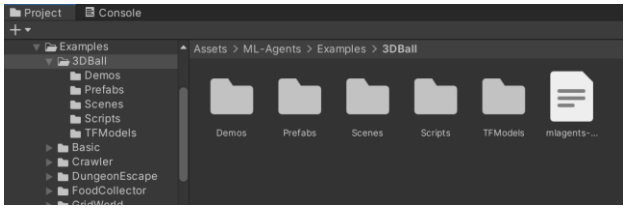
In the 3D Balance Ball environment, an agent gets rewarded for each step it takes to balance the ball. When an agent drops the ball, it gets penalized with a negative reward. The purpose of the training process is to teach the agents how to balance the ball on their heads. In the environment there are 12 agents. They have the exact same behavior, but each and every is an independent agent. These agents contribute to the training in parallel, speeding up the training process.

"An agent is an autonomous actor that observes and interacts with an environment. In the context of Unity, an environment is a scene containing one or more Agent objects, and, of course, the other entities that an agent interacts with." – *Getting Started Guide*

An agent has a few properties that affect its behavior. The **Behavior Parameters script** is responsible for the way of an agent's decision. These parameters contain the observation about the agent's state in the world and its actions. Furthermore, the Max Step property is responsible for the maximum steps that can occur during one training episode.

Running pre-trained models from the Unity Editor

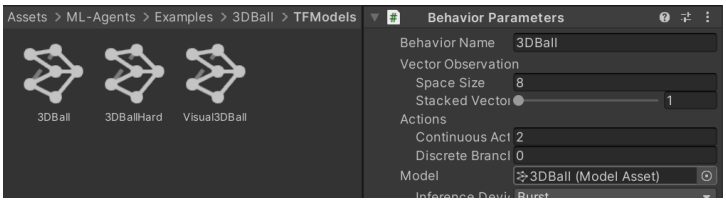
The ML-Agents GitHub repository contains a lot of pre-trained models (*.onnx files*) for the agents. In the *Assets/ML-Agents/Examples/* folder there are several examples. In each and every example folder there is a **Prefabs**, a **Scenes**, a **Scripts** and a **TFModels** folder.



4. Figure: *The contained folders for the 3DBall environment*

The **Prefabs** folder contains the created prefabs. If you are using the same prefab for every agent and you want to change the behavior for every agent, then you can only make the changes only for the prefab and all the agents will be updated.

The model of an agent can be changed due to the **Behavior Parameters** script. In this script there is a **Model** property which can be modified with the models which are in the **TFModels** folder for the corresponding example.

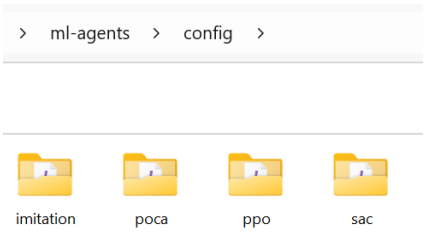


*5. **Figure**: Behavior Parameters script with the Model property*

For testing an environment with the chosen model, the **Play** button must be clicked in the Unity Editor.

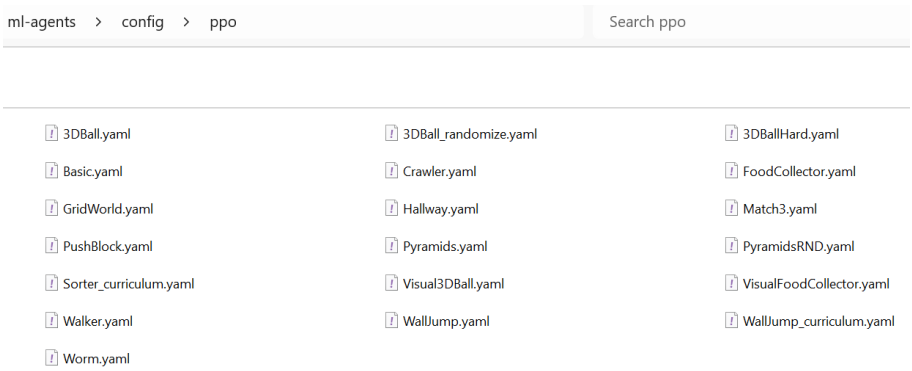
Training the environment

When an agent needs to be trained, then a command or a terminal window must be opened, and the user must navigate to the folder where the ML-Agents repository has been cloned. In the **config** folder by default four reinforcement learning algorithms can be found. Namely the Imitation, the POCA, the PPO and the SAC. (**6. Figure**)



*6. **Figure**: Available reinforcement learning algorithms*

In these folders there are some **.yaml** files, which are training configurations for the corresponding environments. In the name of the files the user can easily see which .yaml can be used for which environment. The **7. Figure** shows the available .yaml files for the PPO algorithm.



*7. **Figure**: .yaml files for the PPO algorithm*

Based on the Getting Started Guide, firstly we tried out the 3DBall environment with the PPO algorithm. The hyperparameters employed for this endeavor align with the standard configurations accessible in "*config/ppo/3DBall.yaml*". The training procedure was executed using the following command:

```
mlagents-learn config/ppo/3DBall.yaml --run-id=first3DBallRun
```

Below, we present a summary of this training outcomes:

```
[INFO] 3DBall. Step: 12000. Time Elapsed: 75.020 s. Mean Reward: 1.198. Std of Reward: 0.726. Training.
[INFO] 3DBall. Step: 24000. Time Elapsed: 89.222 s. Mean Reward: 1.412. Std of Reward: 0.828. Training.
[INFO] 3DBall. Step: 36000. Time Elapsed: 103.030 s. Mean Reward: 1.782. Std of Reward: 1.206. Training.
[INFO] 3DBall. Step: 48000. Time Elapsed: 116.593 s. Mean Reward: 3.114. Std of Reward: 2.237. Training.
[INFO] 3DBall. Step: 60000. Time Elapsed: 130.682 s. Mean Reward: 6.209. Std of Reward: 5.688. Training.
[INFO] 3DBall. Step: 72000. Time Elapsed: 144.833 s. Mean Reward: 16.429. Std of Reward: 14.940. Training.
[INFO] 3DBall. Step: 84000. Time Elapsed: 161.560 s. Mean Reward: 48.529. Std of Reward: 37.915. Training.
[INFO] 3DBall. Step: 96000. Time Elapsed: 173.751 s. Mean Reward: 66.628. Std of Reward: 40.191. Training.
[INFO] 3DBall. Step: 108000. Time Elapsed: 185.797 s. Mean Reward: 97.700. Std of Reward: 7.628. Training.
[INFO] 3DBall. Step: 120000. Time Elapsed: 198.294 s. Mean Reward: 100.000. Std of Reward: 0.000. Training.
[INFO] 3DBall. Step: 132000. Time Elapsed: 210.600 s. Mean Reward: 94.115. Std of Reward: 20.385. Training.
[INFO] 3DBall. Step: 144000. Time Elapsed: 223.015 s. Mean Reward: 100.000. Std of Reward: 0.000. Training.
[INFO] 3DBall. Step: 156000. Time Elapsed: 236.033 s. Mean Reward: 100.000. Std of Reward: 0.000. Training.
[INFO] 3DBall. Step: 168000. Time Elapsed: 248.503 s. Mean Reward: 90.185. Std of Reward: 23.096. Training.
[INFO] 3DBall. Step: 180000. Time Elapsed: 261.294 s. Mean Reward: 100.000. Std of Reward: 0.000. Training.
[INFO] 3DBall. Step: 192000. Time Elapsed: 274.076 s. Mean Reward: 100.000. Std of Reward: 0.000. Training.
[INFO] 3DBall. Step: 204000. Time Elapsed: 286.501 s. Mean Reward: 92.585. Std of Reward: 25.688. Training.
[INFO] 3DBall. Step: 216000. Time Elapsed: 298.777 s. Mean Reward: 100.000. Std of Reward: 0.000. Training.
[INFO] 3DBall. Step: 228000. Time Elapsed: 311.686 s. Mean Reward: 100.000. Std of Reward: 0.000. Training.
[INFO] 3DBall. Step: 240000. Time Elapsed: 324.862 s. Mean Reward: 100.000. Std of Reward: 0.000. Training.
[INFO] 3DBall. Step: 252000. Time Elapsed: 338.458 s. Mean Reward: 94.300. Std of Reward: 16.567. Training.
[INFO] 3DBall. Step: 264000. Time Elapsed: 351.395 s. Mean Reward: 100.000. Std of Reward: 0.000. Training.
[INFO] 3DBall. Step: 276000. Time Elapsed: 364.988 s. Mean Reward: 100.000. Std of Reward: 0.000. Training.
[INFO] 3DBall. Step: 288000. Time Elapsed: 377.312 s. Mean Reward: 100.000. Std of Reward: 0.000. Training.
[INFO] 3DBall. Step: 300000. Time Elapsed: 384.997 s. Mean Reward: 100.000. Std of Reward: 0.000. Training.
[INFO] 3DBall. Step: 312000. Time Elapsed: 397.528 s. Mean Reward: 100.000. Std of Reward: 0.000. Training.
[INFO] 3DBall. Step: 324000. Time Elapsed: 410.459 s. Mean Reward: 100.000. Std of Reward: 0.000. Training.
[INFO] 3DBall. Step: 336000. Time Elapsed: 422.973 s. Mean Reward: 100.000. Std of Reward: 0.000. Training.
[INFO] 3DBall. Step: 348000. Time Elapsed: 435.070 s. Mean Reward: 96.925. Std of Reward: 10.199. Training.
[INFO] 3DBall. Step: 360000. Time Elapsed: 447.311 s. Mean Reward: 100.000. Std of Reward: 0.000. Training.
[INFO] 3DBall. Step: 372000. Time Elapsed: 459.764 s. Mean Reward: 98.592. Std of Reward: 4.671. Training.
[INFO] 3DBall. Step: 384000. Time Elapsed: 472.211 s. Mean Reward: 100.000. Std of Reward: 0.000. Training.
[INFO] 3DBall. Step: 396000. Time Elapsed: 484.995 s. Mean Reward: 100.000. Std of Reward: 0.000. Training.
[INFO] 3DBall. Step: 408000. Time Elapsed: 497.261 s. Mean Reward: 100.000. Std of Reward: 0.000. Training.
[INFO] 3DBall. Step: 420000. Time Elapsed: 509.521 s. Mean Reward: 100.000. Std of Reward: 0.000. Training.
[INFO] 3DBall. Step: 432000. Time Elapsed: 522.117 s. Mean Reward: 100.000. Std of Reward: 0.000. Training.
[INFO] 3DBall. Step: 444000. Time Elapsed: 534.508 s. Mean Reward: 100.000. Std of Reward: 0.000. Training.
[INFO] 3DBall. Step: 456000. Time Elapsed: 542.105 s. Mean Reward: 100.000. Std of Reward: 0.000. Training.
[INFO] 3DBall. Step: 468000. Time Elapsed: 554.677 s. Mean Reward: 100.000. Std of Reward: 0.000. Training.
[INFO] 3DBall. Step: 480000. Time Elapsed: 567.263 s. Mean Reward: 100.000. Std of Reward: 0.000. Training.
[INFO] 3DBall. Step: 492000. Time Elapsed: 580.480 s. Mean Reward: 100.000. Std of Reward: 0.000. Training.
```

From a training you can easily quit with the CTRL+C command, and if you want to continue, you can do it with the following command:

```
mlagents-learn config/ppo/3DBall.yaml --run-id=first3DBallRun --resume
```

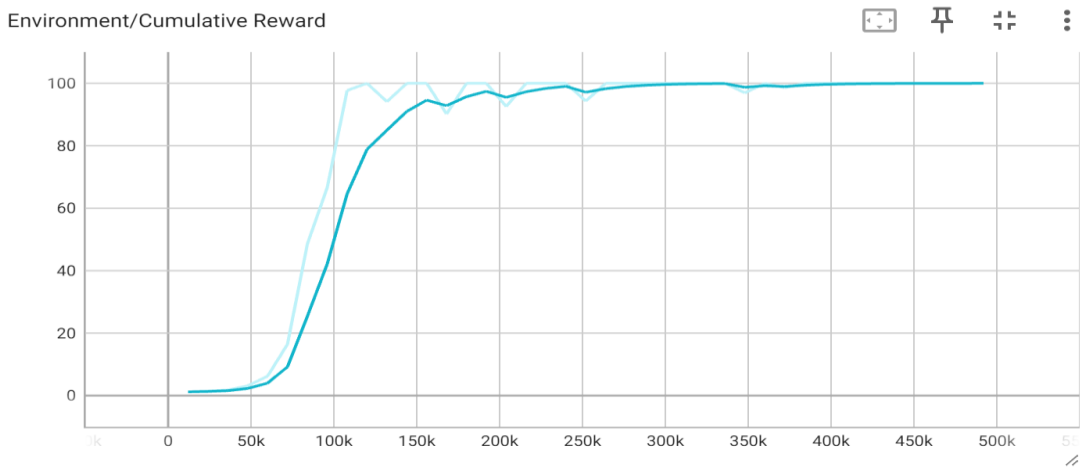
The trained model can be found at the *results/<run-identifier>/<behavior_name>.onnx* file. This model file optionally can be moved to the TFModels folder because it can be used for the corresponding environment.

Observing Training Progress

To monitor the progresses of the trainings and gain insights into an agent's performance, we utilized TensorBoard, which can be accessed through the **localhost:6006**.

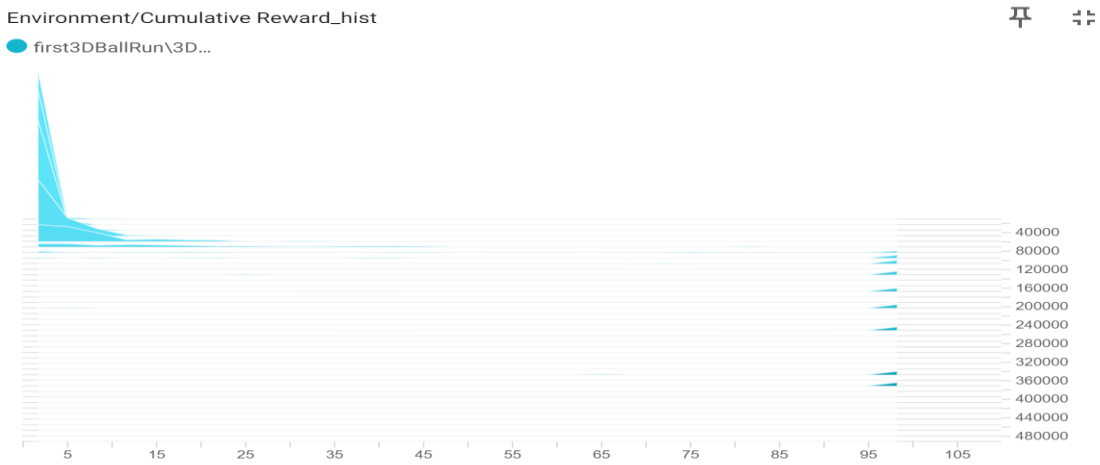
```
tensorboard --logdir results
```

Cumulative Reward



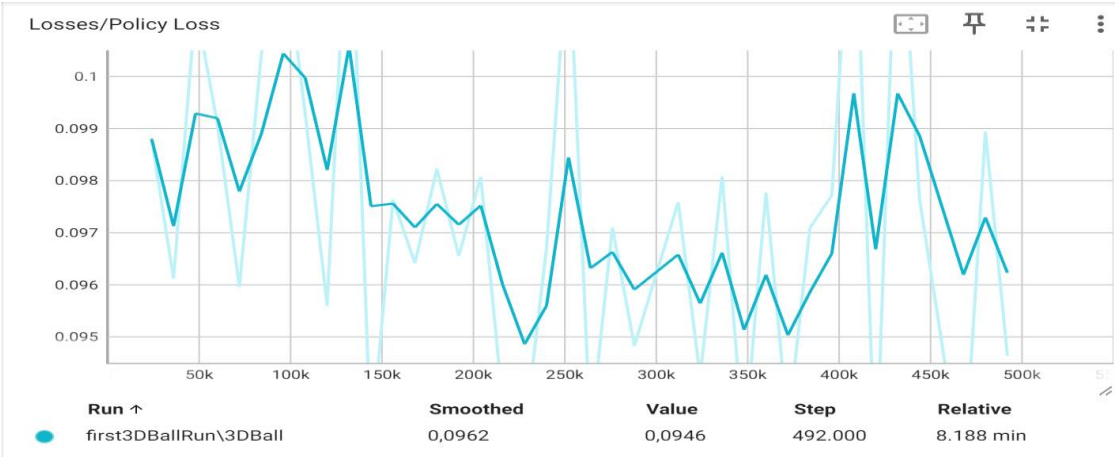
The **Environment/Cumulative Reward** graph represents the total reward accumulated by the agent through the training. The increasing trend indicates successful learning. Plateaus or fluctuations may suggest challenges or changes in the environment. For instance, a plateau around 150k steps could indicate a potential local optimum or increased task difficulty.

Histogram Plot (Reward Distribution)



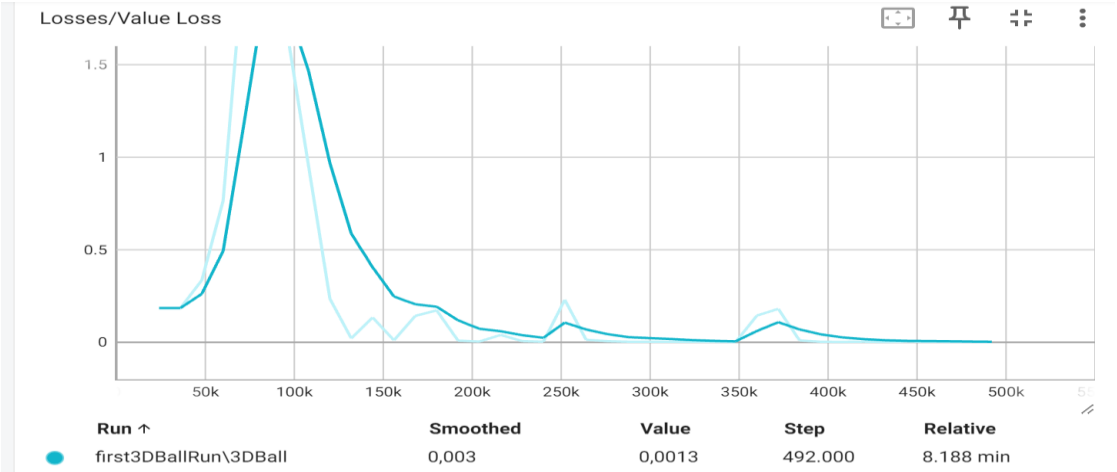
This plot visualizes the distribution of reward values, offering insights into how frequently different reward levels were achieved. Key features to observe include the center (common reward range), spread (variability), and outliers (exceptional high or low reward events).

Policy Loss



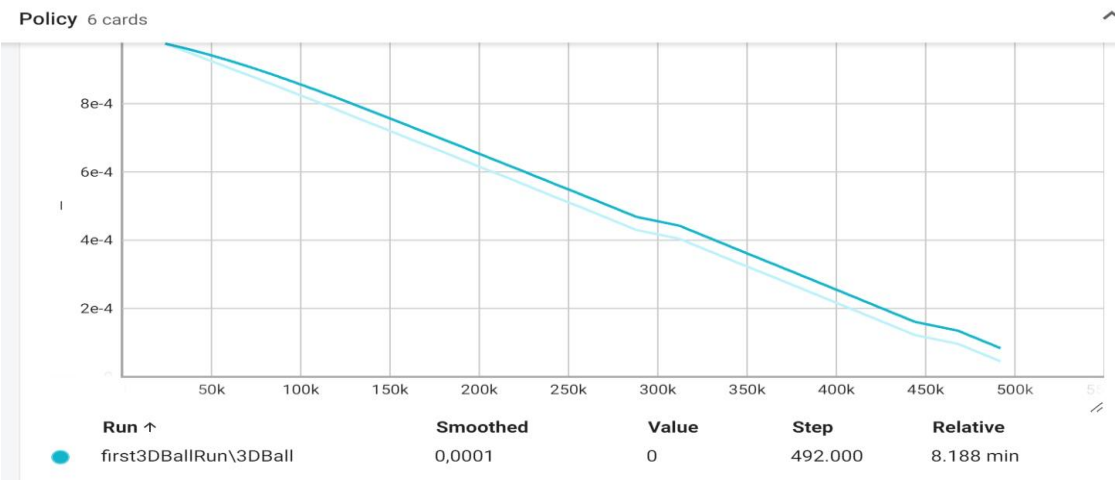
Reflects how well the agent's policy predicts actions leading to high rewards. A decreasing trend signifies improved task performance over time. Occasional fluctuations may indicate exploration or room for further improvement.

Value Loss



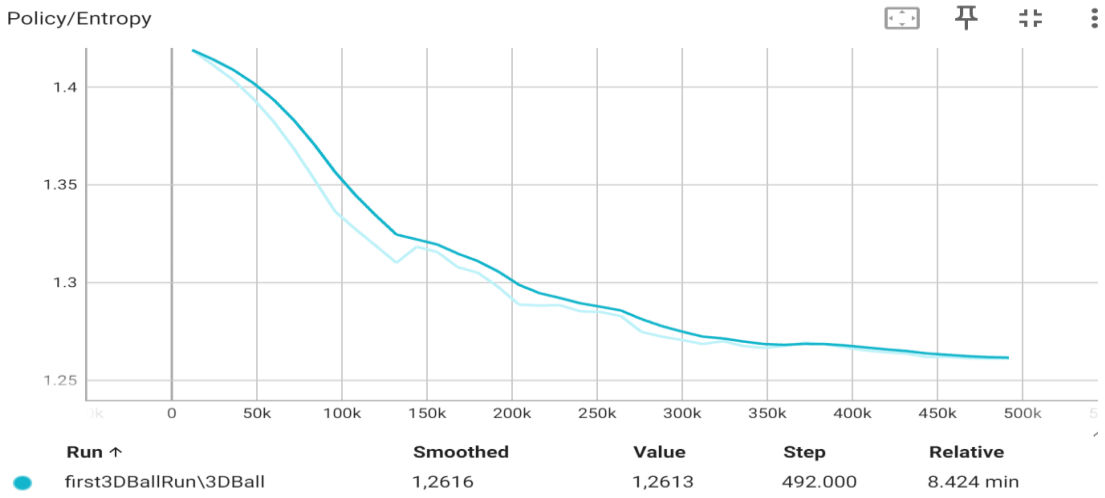
Measures how accurately the agent's value function estimates state values. Decreasing values over time suggest enhanced estimation accuracy. Similar to Policy loss, fluctuations may be present due to exploration-exploitation dynamics.

Policy Value



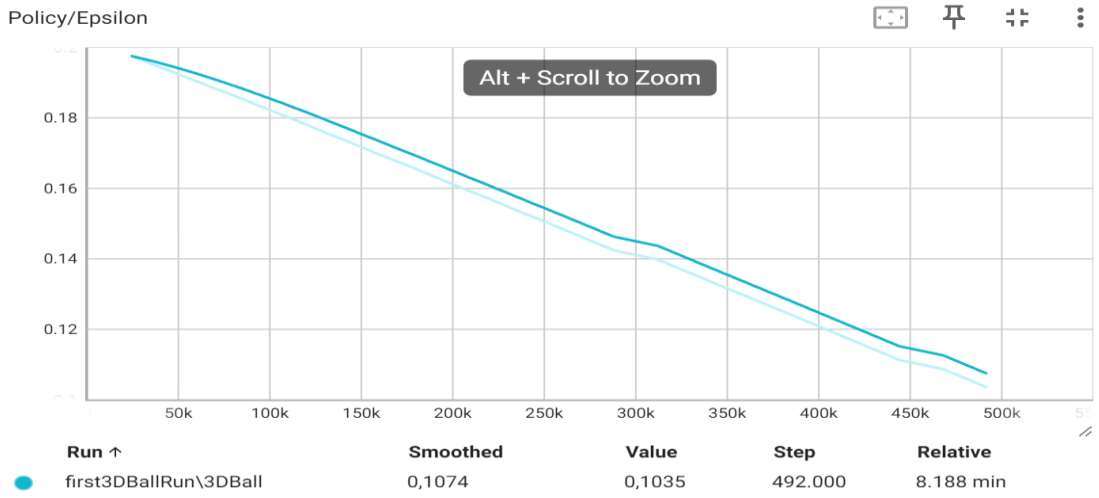
Displays the value of the policy, indicating the policy's effectiveness in predicting high-reward actions. Fluctuations may occur due to the exploration-exploitation trade-off, with the value rebounding as the agent refines its policy over time.

Training Loss



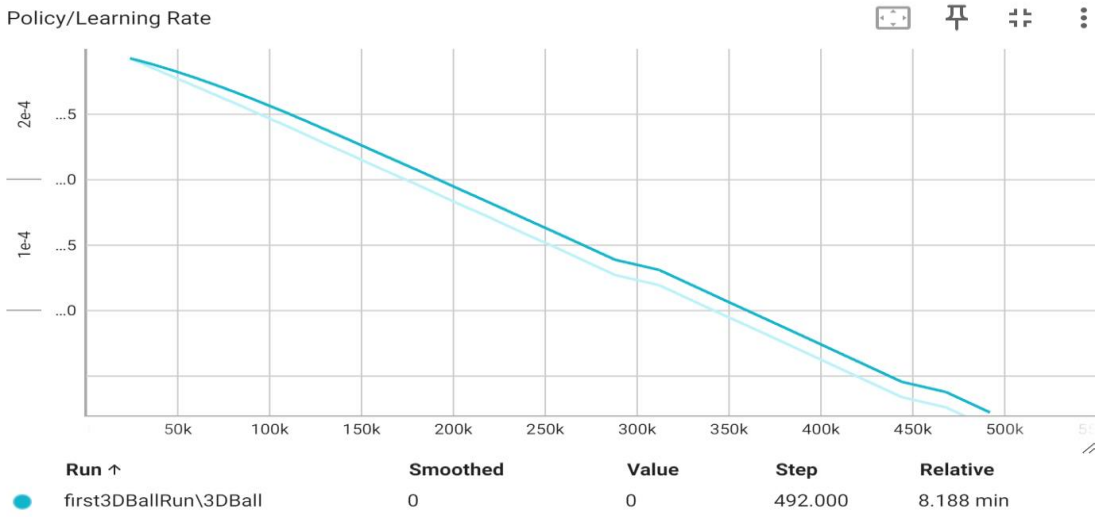
This graph encompasses the entire training process, reflecting the agent's policy and value function predicting optimal behavior. Initial high loss indicates uninformed policies, fluctuation during exploration, and low loss during exploitation phase indicates consistent good decisions.

Epsilon (Exploration-Exploitation Trade-off)



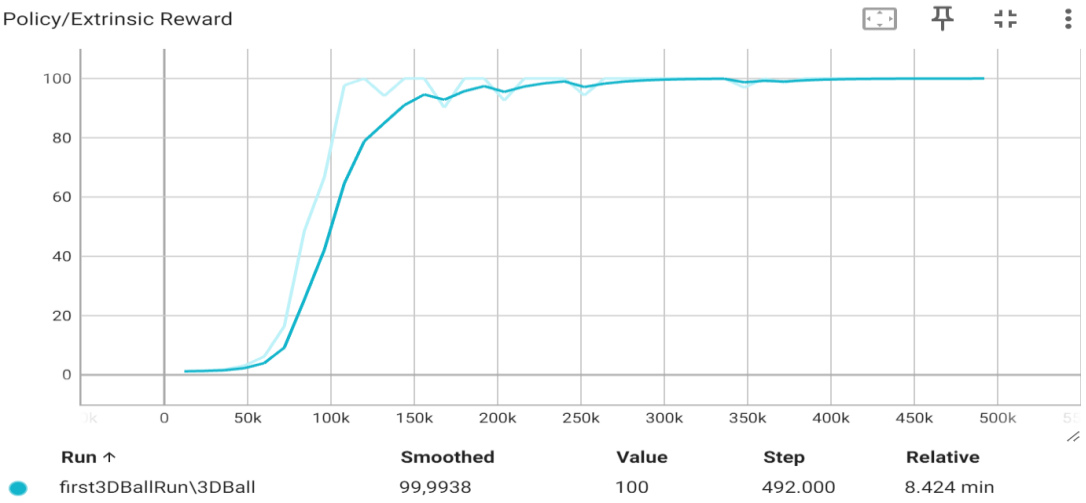
Epsilon controls the balance between exploration and exploitation. Starting high and gradually decreasing allows early exploration. Fluctuations ensure occasional exploration even after learning a good policy.

Learning Rate and Policy



The learning rate controls the pace of policy updates. High initially for exploration, gradually reduced as the agent learns. Fluctuations may occur during ongoing learning or environmental changes.

Policy/Extrinsic Reward



This metric represents the average reward per episode using the current policy. It tends to be higher than cumulative reward, showcasing ongoing learning and policy refinement.

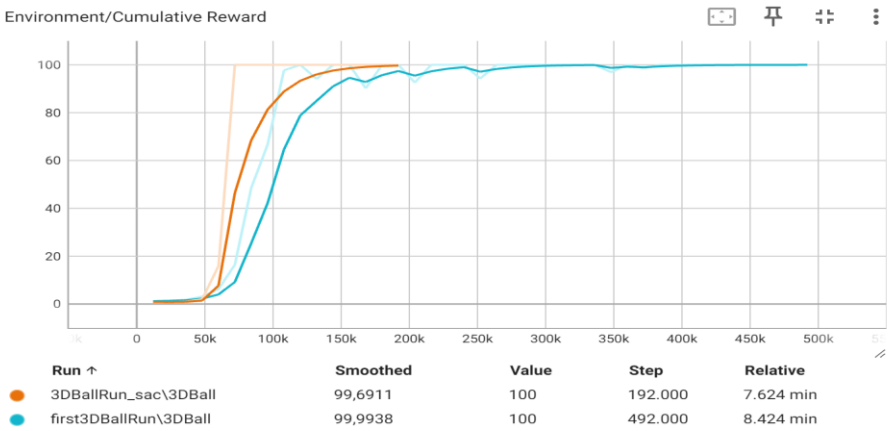
Comparing SAC and PPO via the 3DBall environment:

We decided to experiment with the Soft Actor-Critic (SAC) algorithm as an alternative to PPO.

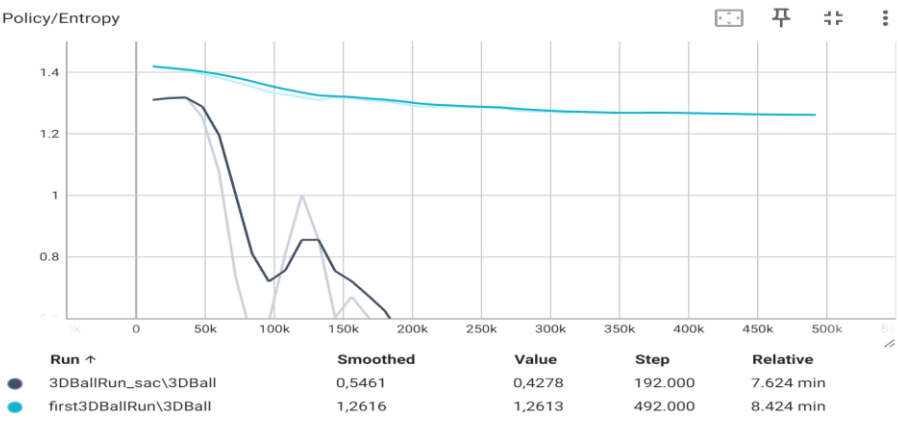
```
[INFO] 3DBall. Step: 12000. Time Elapsed: 32.212 s. Mean Reward: 0.905. Std of Reward: 0.558. Training.
[INFO] 3DBall. Step: 24000. Time Elapsed: 63.365 s. Mean Reward: 0.784. Std of Reward: 0.418. Training.
[INFO] 3DBall. Step: 36000. Time Elapsed: 99.450 s. Mean Reward: 0.998. Std of Reward: 0.587. Training.
[INFO] 3DBall. Step: 48000. Time Elapsed: 133.367 s. Mean Reward: 2.004. Std of Reward: 1.986. Training.
[INFO] 3DBall. Step: 60000. Time Elapsed: 163.621 s. Mean Reward: 15.912. Std of Reward: 24.321. Training.
[INFO] 3DBall. Step: 72000. Time Elapsed: 192.326 s. Mean Reward: 100.000. Std of Reward: 0.000. Training.
[INFO] 3DBall. Step: 84000. Time Elapsed: 222.980 s. Mean Reward: 100.000. Std of Reward: 0.000. Training.
[INFO] 3DBall. Step: 96000. Time Elapsed: 252.367 s. Mean Reward: 100.000. Std of Reward: 0.000. Training.
[INFO] 3DBall. Step: 108000. Time Elapsed: 283.834 s. Mean Reward: 100.000. Std of Reward: 0.000. Training.
[INFO] 3DBall. Step: 120000. Time Elapsed: 315.586 s. Mean Reward: 100.000. Std of Reward: 0.000. Training.
[INFO] 3DBall. Step: 132000. Time Elapsed: 344.921 s. Mean Reward: 100.000. Std of Reward: 0.000. Training.
[INFO] 3DBall. Step: 144000. Time Elapsed: 373.724 s. Mean Reward: 100.000. Std of Reward: 0.000. Training.
[INFO] 3DBall. Step: 156000. Time Elapsed: 403.122 s. Mean Reward: 100.000. Std of Reward: 0.000. Training.
[INFO] 3DBall. Step: 168000. Time Elapsed: 432.663 s. Mean Reward: 100.000. Std of Reward: 0.000. Training.
[INFO] 3DBall. Step: 180000. Time Elapsed: 460.896 s. Mean Reward: 100.000. Std of Reward: 0.000. Training.
[INFO] 3DBall. Step: 192000. Time Elapsed: 489.643 s. Mean Reward: 100.000. Std of Reward: 0.000. Training.
```

It is evident from our results that SAC outperforms PPO in this specific environment. Let's see the results plotted into TensorBoard to have a better understanding:

→ **Cumulative Reward:**



→ **Policy/Entropy:**

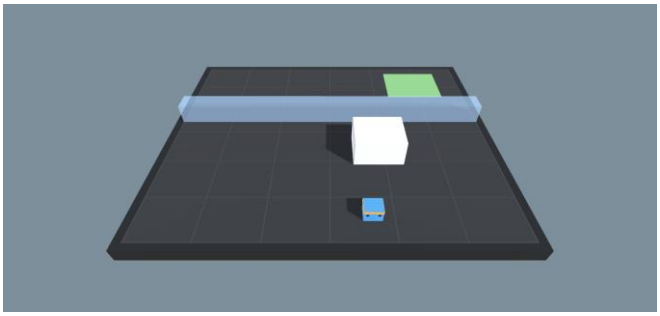


SAC prioritizes optimizing for entropy as it aids in preventing the agent from getting stuck in local optima – policies that are good but not the best possible.

Unlike SAC, PPO doesn't explicitly optimize for entropy. Instead, PPO utilizes a trust region mechanism, restricting policy changes within certain limits to prevent large alterations that could decrease performance significantly.

Curriculum Learning for WallJump Environment

In this section of the paper, we investigate the application of curriculum learning in the WallJump environment provided by Unity. The primary objective is to enable an RL agent to navigate the environment successfully, which includes reaching a green target point separated by a variable-height wall. The wall may be of relatively short height, permitting the agent to execute a direct jump to quickly reach the target, or it may assume a greater height, necessitating a strategic approach where the agent first moves a block to create a platform so that it can jump over the wall. On the below you can see how the environment looks like:



***8. Figure**: WallJump environment*

Methodology and Configuration

The configuration details for our experiments can be found at:

config\ppo\WallJump_curriculum.yaml

Our chosen environment facilitates a clearer understanding of curriculum learning by adjusting the wall's height incrementally. Initially, the agent begins training without any wall, allowing it to learn the basics of reaching the goal. Subsequently, the wall height gradually increases, introducing a curriculum-based training schedule.

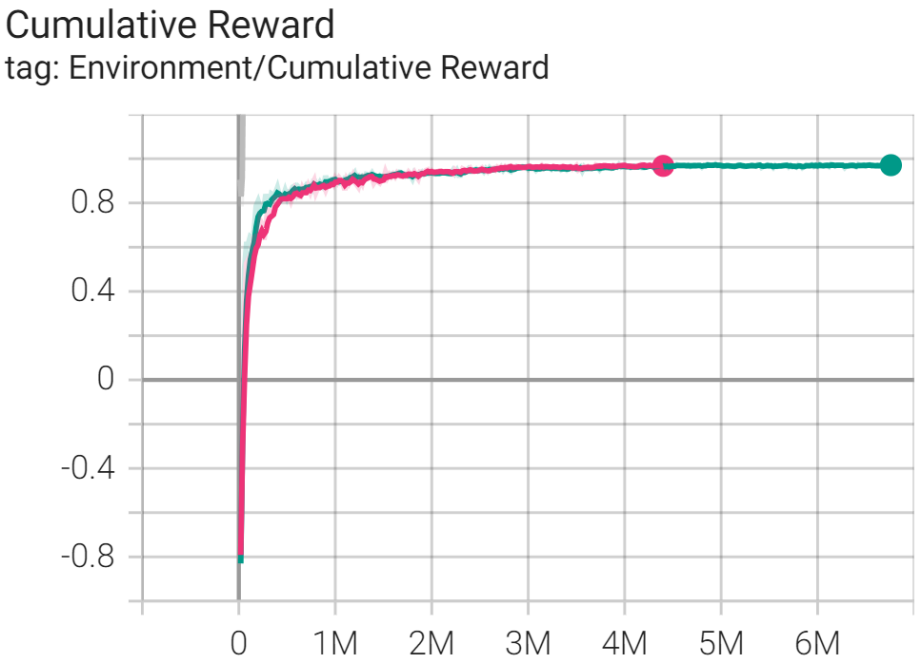
Training Duration

On our machine, the overall training duration for the WallJump environment with Curriculum Learning was approximately 14 hours, aligning with Unity's training standards. Without was not able to finish training after 20 hours.

Small Wall Environment

With curriculum learning, it took only 9.17 hours to achieve satisfactory training results, while without it, training required 19.49 hours.

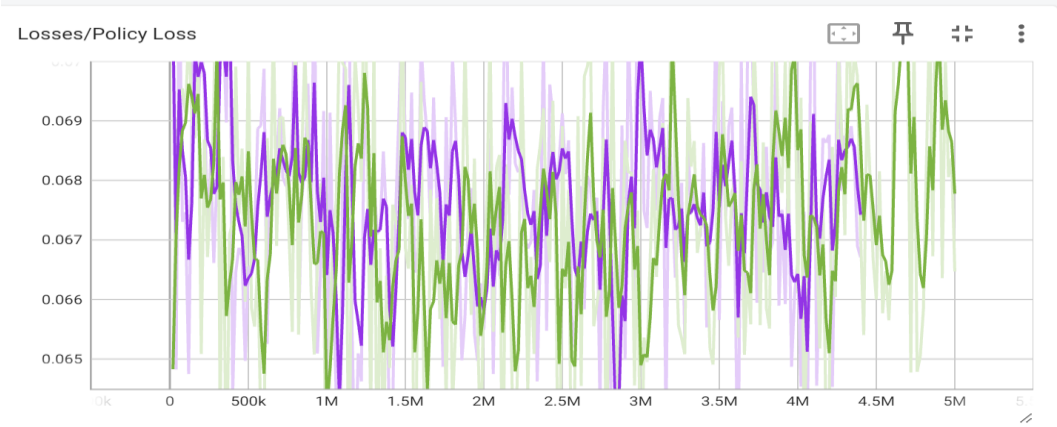
Cumulative Reward



*Where the **pink** line is **without Curriculum learning**, the **green** is **with Curriculum learning**.*

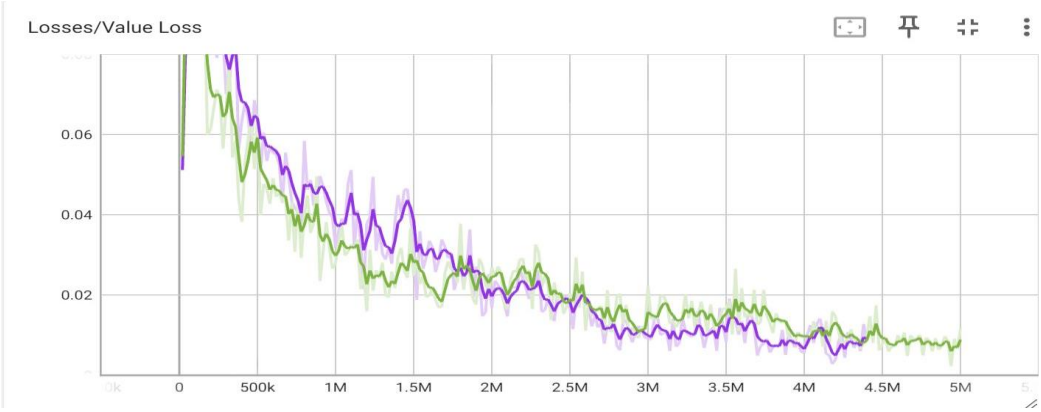
The agent trained **with curriculum learning** exhibits a smoother learning curve, indicating a more stable training progression than the one **without Curriculum Learning**, but in the end both models eventually achieve similar results.

Policy Loss



Policy loss reflects the accuracy of the agent's predictions regarding the best action to take. **Curriculum learning** allows the agent to focus on mastering fundamental skills early in training, resulting in a declining policy loss. In contrast, the agent **without curriculum learning** faces challenges in understanding the environment's underlying mechanics, leading to fluctuating policy losses.

Value Loss

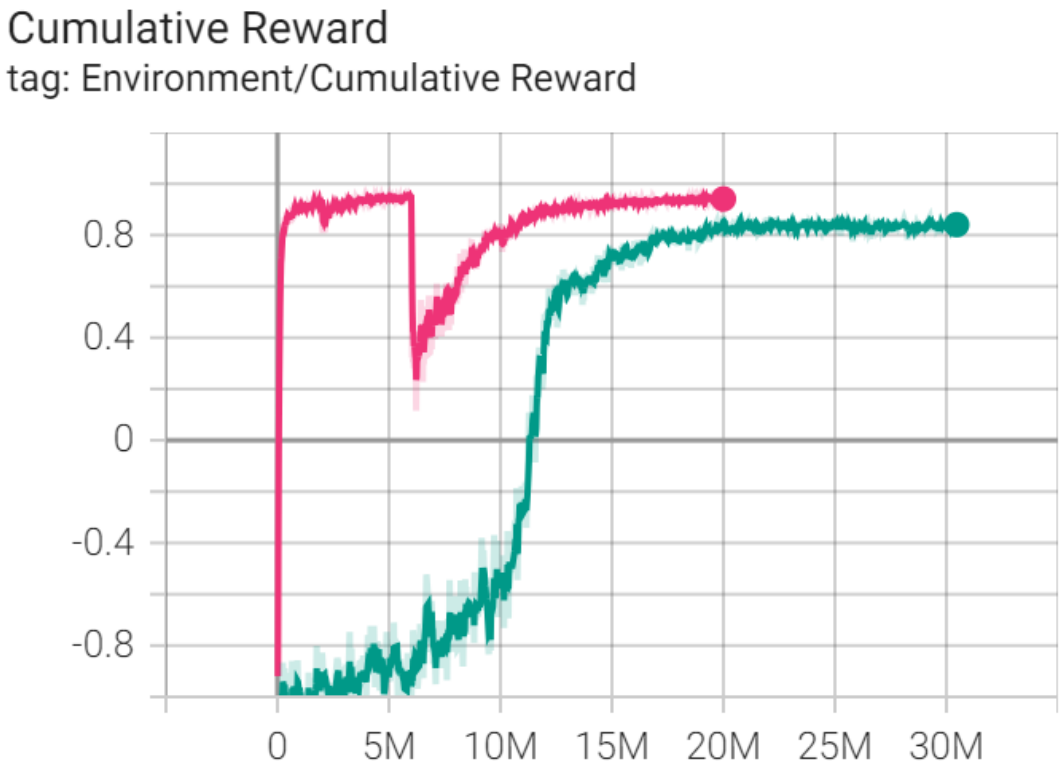


Value loss measures the disparity between the agent's expected value for a state and the observed value. **Curriculum learning** aids in building a more accurate representation of the environment, resulting in a lower value loss initially (green line). However, as the training transitions to more complex tasks, the trend may invert, but the agent eventually stabilizes and performs well.

Big Wall Environment

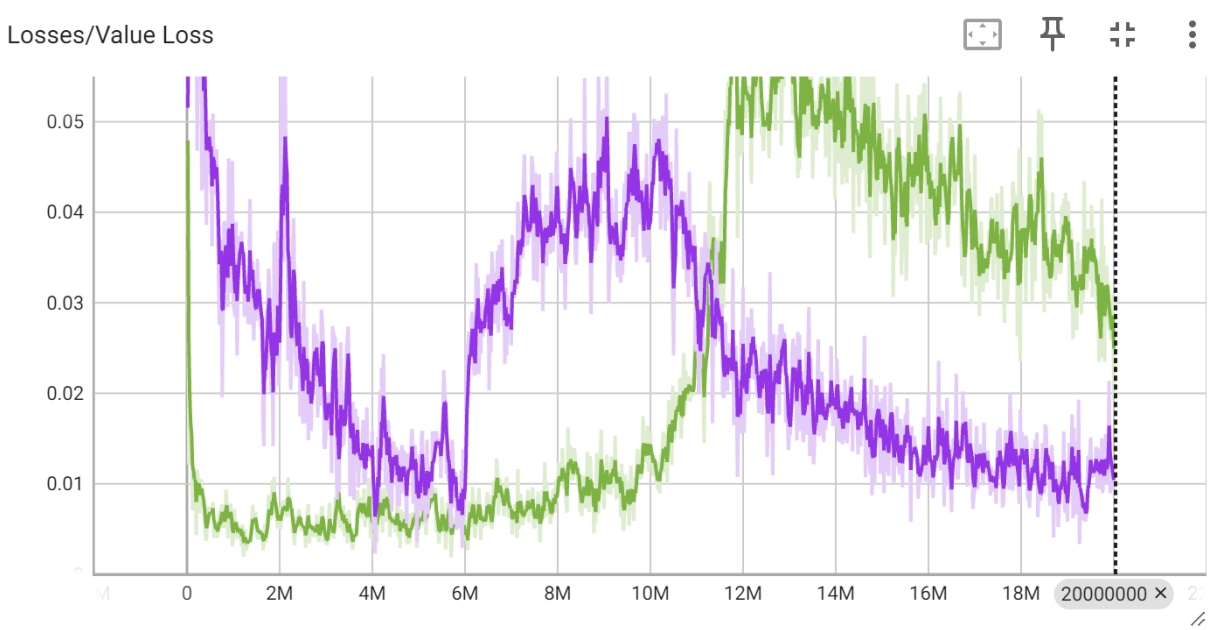
In the case of the big WallJump environment, curriculum learning significantly accelerates training. With curriculum learning, training stopped after 14 hours, while without it, training had to be halted after approximately 20 hours.

Cumulative Reward



The agent trained with **Curriculum learning** achieves a more rapid increase in cumulative reward during the initial stages of training compared to the agent trained **without Curriculum learning**, that even after 30kk steps hasn't properly trained yet. Although there is a temporary drop in performance due to the transition to a complex task, the curriculum-learned agent rebounds quickly and ultimately outperform the non-curriculum agent.

Value Loss:



Run ↑	Smoothed	Value	Step	Relative
WallJump_basic\BigWallJump	0,0249	0,0195	20.000.000	13.96 hr
WallJump_curr\BigWallJump	0,0104	0,0093	20.000.000	13.29 hr

Curriculum learning results in a steeper initial decrease in value loss, indicating more efficient learning of the value function. As with policy loss, the trend may temporarily reverse during the transition to a more complex environment (after 6k steps) but eventually stabilizes and improves.

Conclusion

In summary, curriculum learning offers substantial benefits in training RL agents for the WallJump environment. It accelerates initial learning, stabilizes training progression, and helps agents achieve better overall performance. These findings underscore the importance of curriculum learning as an effective strategy for tackling complex tasks in reinforcement learning.

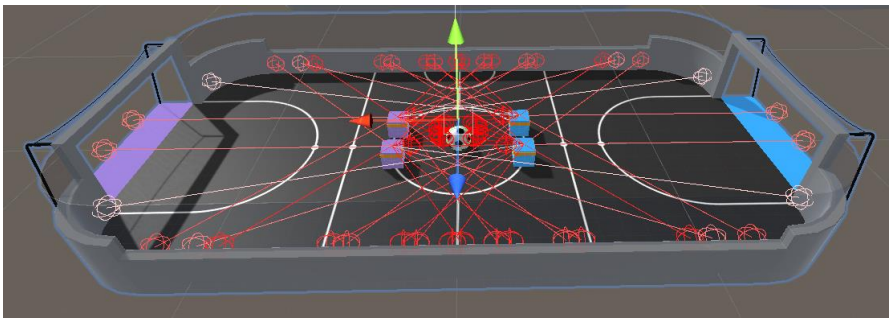
Testing in the SoccerTwos Environment

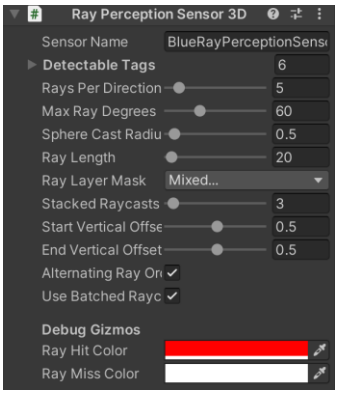
The SoccerTwos environment is a dynamic 2 vs 2 soccer simulation designed for multi-agent competition. In this setup, two teams, each comprising two agents, face off in a soccer match with the objective of scoring goals against the opponent while defending their own goal.

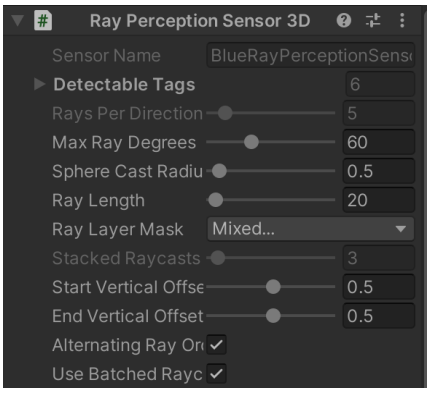
This environment is characterized by its distinct observation and action spaces, along with a tailored agent reward function.

Observation Space

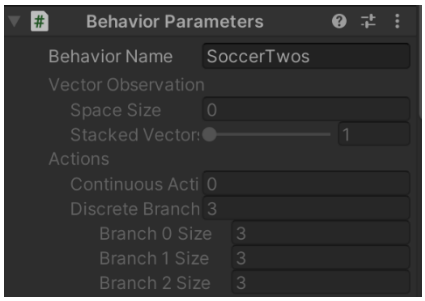
The observation space is notably complex, consisting of a 336-dimensional vector. This vector is derived from a total of 14 ray-casts, with 11 forward casts covering a 120-degree field and 3 backward casts spanning 90 degrees. These ray-casts are capable of detecting six different object types and their respective distances. The forward ray-casts contribute 264 dimensions to the state, while the backward casts add another 72, all across three observation stacks.







Action Space



The action space is defined as 3 discrete branched actions (labeled as MultiDiscrete), encompassing movements such as forward, backward, sideways, and rotational actions, resulting in a total of 27 discrete actions.

Reward Function

In terms of the reward function for the agents, it includes:

- A positive reward (+1) for scoring a goal in the opponent's net.
- A time penalty that accumulates each fixed update cycle, incremented by $1/MaxSteps$ *(with MaxSteps set to 5000 in this build)*

  This penalty is reset to zero at the beginning of each episode.
- A negative reward (-1) if the ball enters the team's own goal.

Competitive-cooperative environment

The SoccerTwos environment exemplifies a competitive-cooperative environment, blending elements of both competitive and cooperative gameplay to create a complex and engaging multi-agent setting.

Competitive Elements: In this environment, agents engage in a competitive dynamic, striving to maximize their individual rewards through strategies that often conflict with their opponents' objectives. Actions taken by one agent can have adverse effects on others, leading to a tactical game of interception, blocking, and tackling. For instance, agents in SoccerTwos will proactively intercept passes, block shots, and challenge opposing players, emphasizing their competitive drive.

Cooperative Elements: Conversely, the environment also fosters cooperation, where agents on the same team collaborate towards a shared goal. This cooperative aspect requires agents to communicate, synchronize their actions, and support each other, which is critical for team success. In SoccerTwos, such cooperation manifests in coordinated movements, strategic passing, and mutual assistance in scoring goals.

Mixed Reward Structure: To encapsulate both competitive and cooperative dynamics, the reward structure in SoccerTwos is intricately designed:

Cooperative Rewards: Teammates earn rewards for executing successful passes, assisting in goal-scoring, and collectively scoring goals. These rewards emphasize teamwork and collaboration.

Competitive Rewards: Agents also gain rewards for individually outmaneuvering opponents, such as effectively dribbling past a defender or stealing the ball, highlighting the competitive aspect of the game.

Self-Play: a classic technique to train competitive agents in adversarial games

Self-Play is a powerful training technique used in adversarial games, particularly effective for developing competitive agents. This method allows agents to learn both offensive and defensive strategies by competing against dynamically adapting opponents, which are essentially previous versions of themselves.

Understanding Self-Play

In adversarial games, matching a training agent against an equally skilled opponent is crucial for effective learning. If the opponent is too strong, the agent may not learn effectively; if too weak, the agent may develop suboptimal strategies. Self-Play addresses this by having the agent compete against its previous iterations. As the agent improves, so does its opponent, ensuring a consistently challenging training environment.

Key Elements of Self-Play in Training

Swap Steps and Team Change: These parameters determine how often the agent's opponent is changed. Regularly updating the opponent helps maintain a challenging environment and prevents the agent from overfitting to a specific opponent's strategy.

Window: This refers to the number of past agent versions retained for training. A larger window stores more versions, leading to a diverse range of opponents. While this can increase training time, it typically results in a more robust and adaptable policy.

Play Against Latest Model Ratio: This parameter dictates the likelihood of the agent playing against its most recent version versus an older one. Balancing this ratio is crucial for ensuring the agent is consistently challenged and learns from a variety of strategies.

Save Steps: Indicates how frequently a new version of the agent is saved and added to the pool of potential opponents. Longer intervals between saves mean the new opponent is generally better trained, offering a more challenging and relevant adversary.

Incorporating these elements in Self-Play ensures that agents progressively improve and adapt, developing strategies that are effective against a range of opponents. It's a delicate balance between ensuring the learning remains stable and avoiding overfitting to a particular style of play. More info can be found here.

ELO Rating System

In adversarial games, evaluating an agent's skill level can be challenging, especially when using cumulative environment rewards as a metric. The effectiveness of these rewards is heavily dependent on the opponent's skill level. An agent might score higher against a weaker opponent or lower against a stronger one, making it difficult to gauge true progress. To address this, the ELO rating system is implemented as a more effective means of assessing an agent's skill.

Understanding the ELO Rating System

The ELO rating system is a widely recognized method for calculating the relative skill levels of players in zero-sum games. It assigns a numerical rating to each player based on their game results against other players. The core idea is to update a player's rating after each game, reflecting their performance relative to expectations.

I. ELO Score in Agent Training:

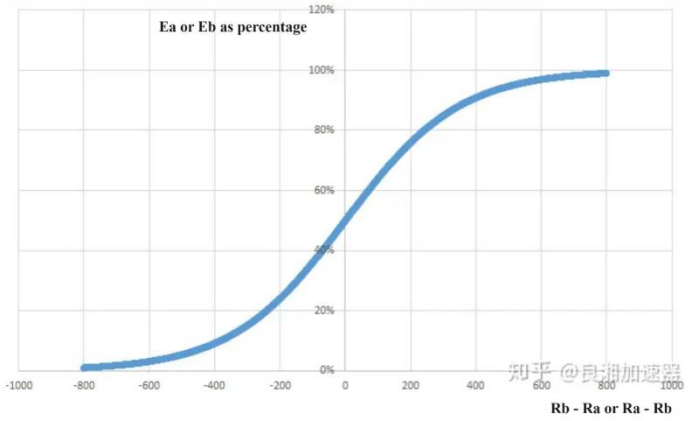
Expected Scores: The ELO system determines the expected score of a player against an opponent, based on their respective ELO ratings before the game. This expected outcome helps in understanding the skill gap between the players.

$$E_A = \frac{1}{1 + 10^{(R_B - R_A)/400}}$$

$$E_B = \frac{1}{1 + 10^{(R_A - R_B)/400}}$$

II. Graphical representation of the expected score formula from the ELO rating system:

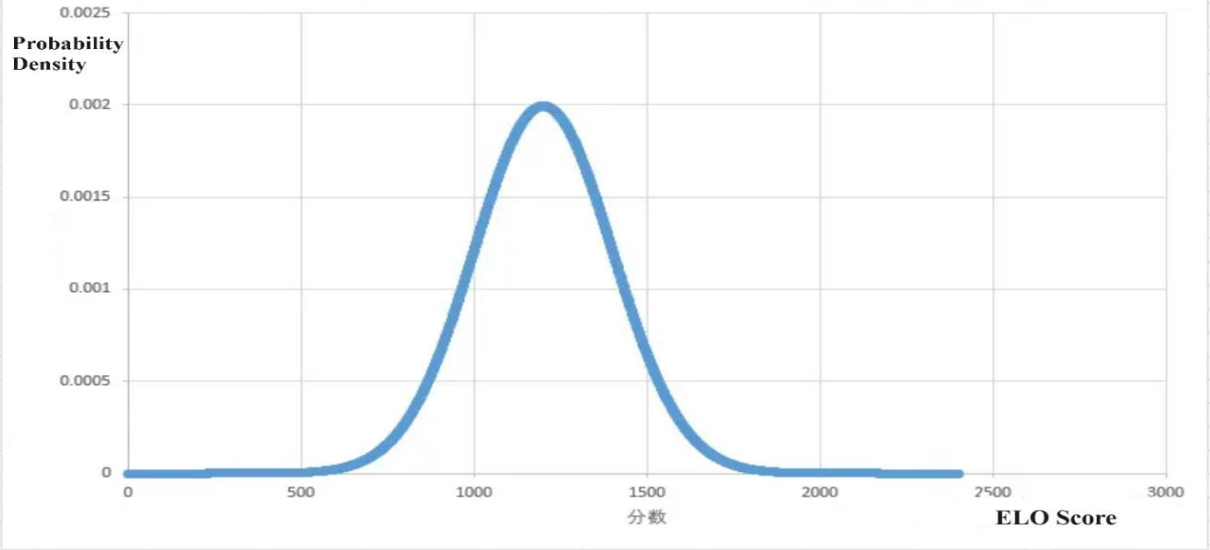
The curve shows the logistic function used in the ELO rating system: it's sigmoidal, which means it has an "S" shape that asymptotically approaches 100% for large positive rating differences and 0% for large negative rating differences. The point where the curve crosses the 50% mark on the y-axis corresponds to a rating difference of 0, which means that when two players have the same rating, they have an equal expected chance of winning (50%).



Integration with TensorBoard

Distribution: the distribution of our players' game levels it's a normal distribution, that is, the game levels of most players are around the average, and there will be a small number of top or very poor players.

If our case where the starting ELO score is 1200 points (that is also the average), then the distribution of points for all players will be roughly as follows:

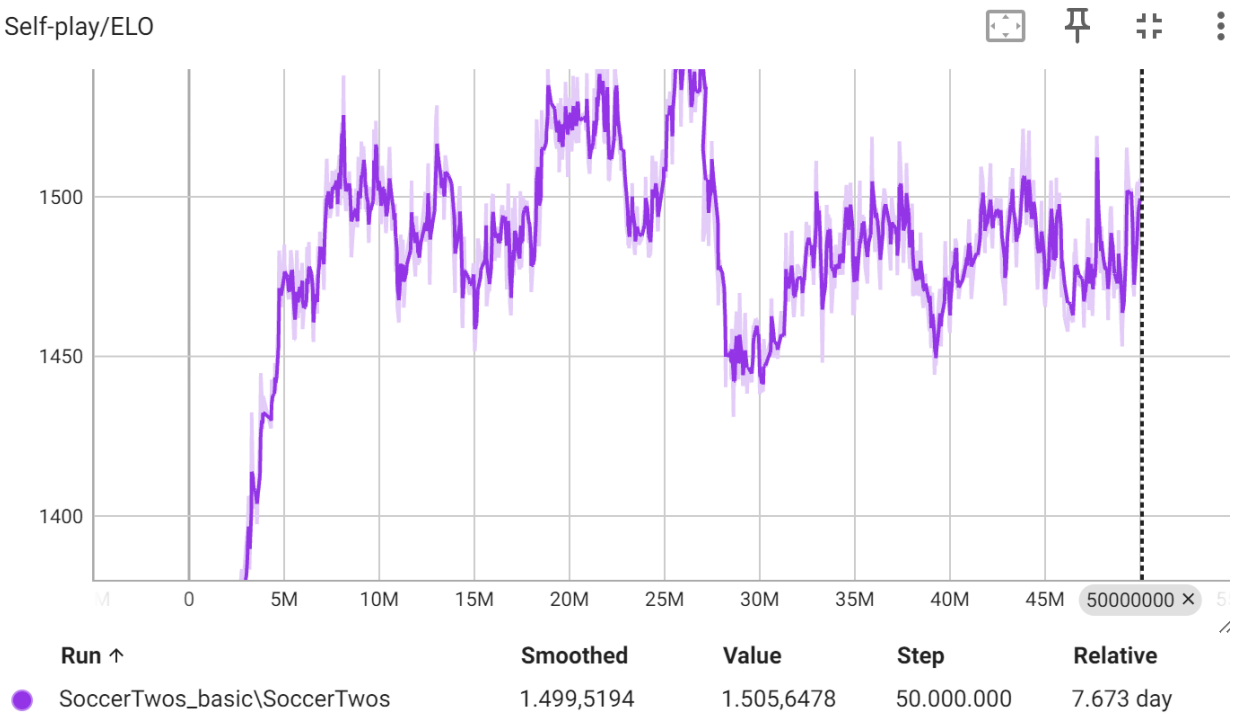


Rating Updates: After each match, players' ratings are updated. If a player performs better than expected (e.g., a lower-rated player wins against a higher-rated one), their ELO rating increases. Instead, underperforming relative to expectations leads to a decrease in rating.

$$R'_A = R_A + K(S_A - E_A)$$

Training Approach

Our process starts with the foundational settings provided by Unity. We trained our model for approximately 183 hours, during which the model underwent over 50 million steps. For evaluation, we utilized the ELO rating system.



The graph incorporates a "Smoothed" ELO value, essentially a rolling average, to offer a clearer view of the performance trend, minimizing the impact of individual fluctuating data points. This graphical representation reveals a notable fluctuation in the ELO scores, a common occurrence in self-play environments as the AI alternates between exploring various strategies and capitalizing on them. The progression pattern in the latter part of the training phase appears relatively stable, hinting at the potential stabilization of the model's performance. A dashed vertical line punctuates the graph's endpoint, marking the cessation of training at the 50 million step milestone. This halt was due to the model reaching a plateau in learning, indicating that it had developed a consistent and stable policy.

Changing Hyperparameters

To improve the efficacy of our reinforcement learning model, we have made important changes to the hyperparameters, which were guided by the insights gained from TensorBoard analysis. These adjustments are detailed in the updated configuration file "config\poca\SoccerTwos2.yml".

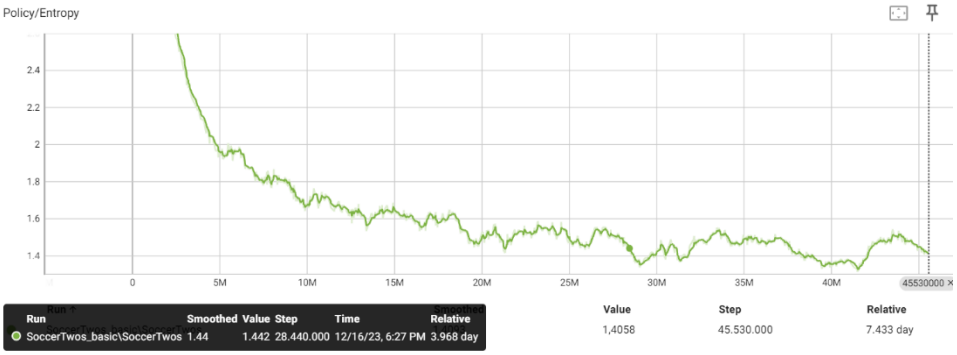
Here's a brief overview of the modifications:

Learning Rate

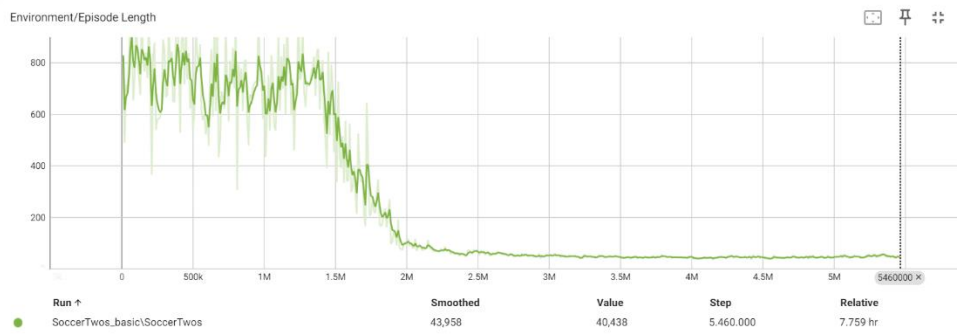
We increased the learning rate from 0.0003 to 0.0007, aiming to speed up the learning. We also transitioned from a fixed to a linear learning rate schedule to better adapt as training progresses.

Exploration Strategy

We can see here that the decreasing trend is generally good as it implies the agent is gaining certainty in its actions. However, if entropy decreases too much, the agent may become too deterministic and not explore enough, which can be detrimental in complex environments.

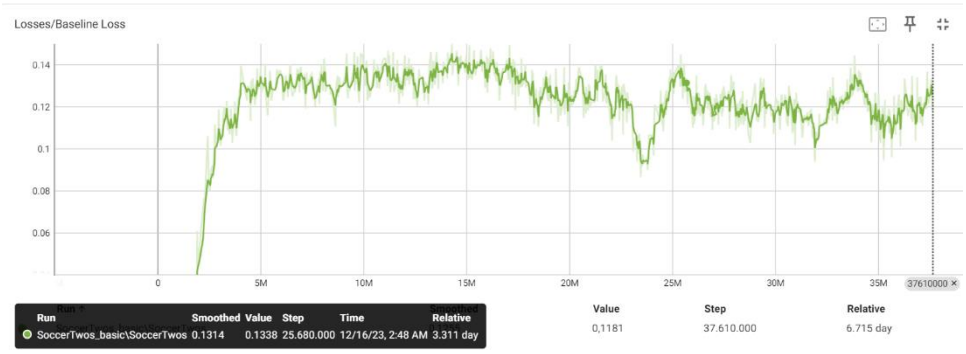


In the following graph the decrease is too rapid, or episodes become too short, it may also mean the agent is exploiting some aspect of the environment that leads to premature termination.

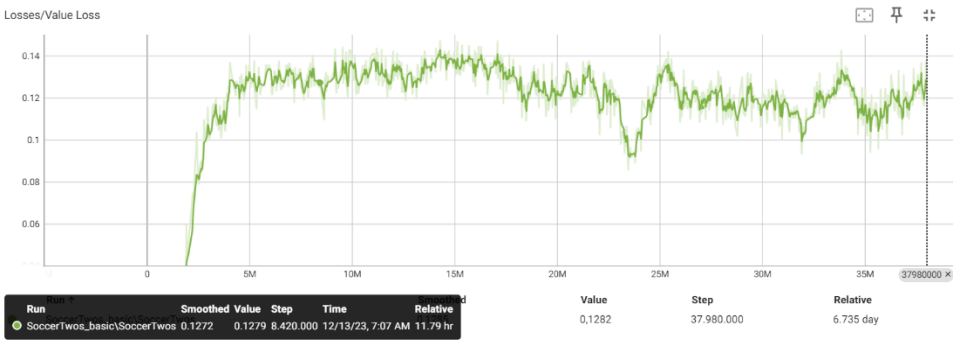


That's why we shifted from a constant exploration rate to a linearly decreasing one. This change allows for more exploration at the beginning and more exploitation as the model's performance improves.

The first graph suggests that the model's ability to predict future rewards improves initially but then stabilizes. The plateau might indicate that the model has learned as much as it can with the current configuration and further learning might require changes in the model's architecture, learning rate, or the environment itself.



Same goes for the value function, which aims to estimate the expected return from each state. This could mean the value network has reached its capacity to learn from the current policy.



Neural Network Layers

We've deepened the neural network from 2 to 5 layers, which is expected to enhance the model's ability to process complex tasks.

Discount Factor

The gamma value has been raised, emphasizing the importance of future rewards and encouraging the model to develop strategies that consider long-term success.

Self-Play Settings

We've refined our self-play strategy by doubling the window size to 20 and increasing the save_steps to 200,000. These adjustments provide a broader dataset for the model to learn from and space out performance evaluations.

Training Duration

The training steps have been significantly extended to 80 million to give the model more time to refine its strategies. This aligns with benchmarks seen in well-known models available on Hugging Face for comparative purposes.

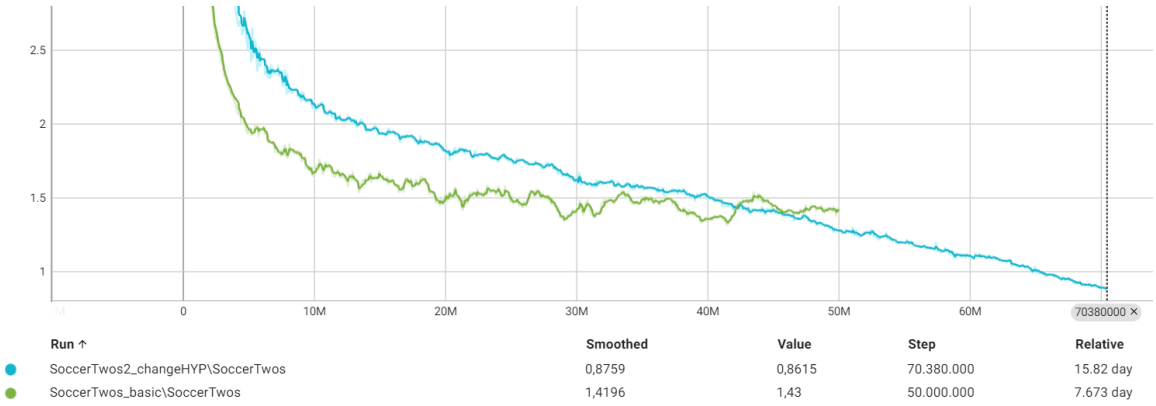
```yaml
default_settings: null
behaviors:
  SoccerTwos:
    trainer_type: poca
    hyperparameters:
      batch_size: 2048
      buffer_size: 20480
      learning_rate: 0.0007
      beta: 0.005
      epsilon: 0.2
      lambd: 0.95
      num_epoch: 3
      learning_rate_schedule: linear
      beta_schedule: linear
      epsilon_schedule: linear
    checkpoint_interval: 500000
```

```yaml
checkpoint_interval: 500000
network_settings:
  hidden_units: 512
  num_layers: 5
reward_signals:
  extrinsic:
    gamma: 0.999
    strength: 1.0
max_steps: 80000000
time_horizon: 1000
summary_freq: 10000
self_play:
  save_steps: 200000
  team_change: 200000
  swap_steps: 2000
  window: 20
  play_against_latest_model_ratio: 0.5
  initial_elo: 1200.0
```

Results from the new Self Play ELO Model

Our **modified hyperparameter** tuning significantly outperformed the **baseline model**, achieving a substantially higher ELO score at the same step count, while also exhibiting reduced variability. However, despite the continued learning and improvement shown by our model, we were forced to cease further training due to limitations in computational resources.

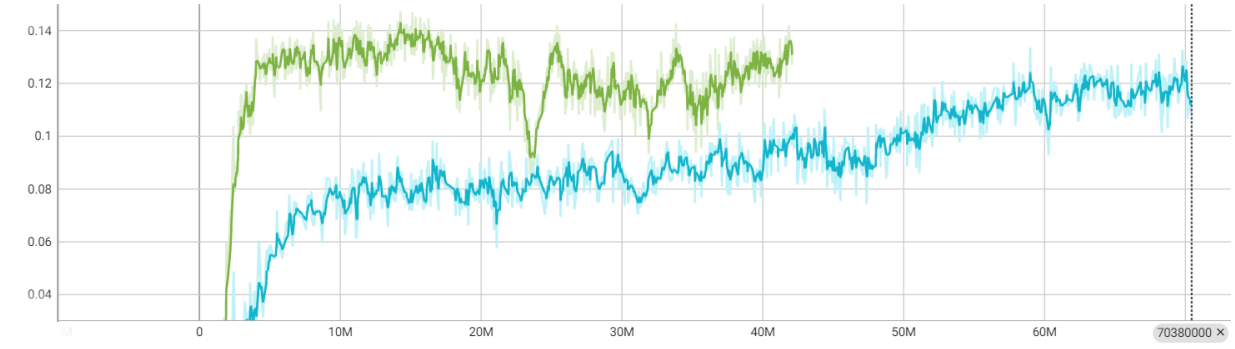
Policy/Entropy



Run	Smoothed	Value	Step	Relative
SoccerTwos2_changeHYP\SoccerTwos	0,8759	0,8615	70.380.000	15.82 day
SoccerTwos_basic\SoccerTwos	1,4196	1,43	50.000.000	7.673 day

It is evident from the graph that, while the entropy reduction plateaued under the basic parameters, indicating a potential convergence to a suboptimal policy, the introduction of new hyperparameters continued to drive the optimization process forward, resulting in a consistent decrease in entropy over time.

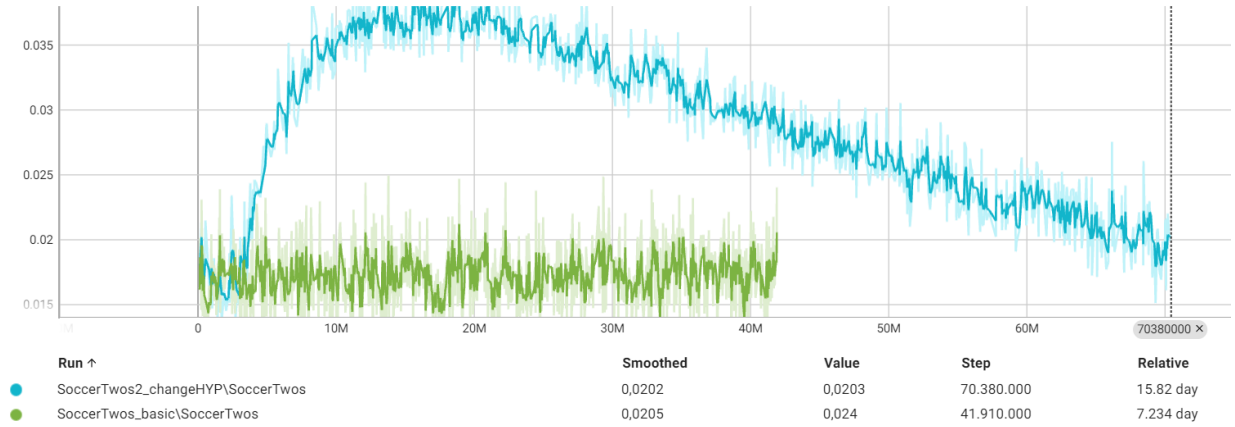
Losses/Value Loss



Run	Smoothed	Value	Step	Relative
SoccerTwos2_changeHYP\SoccerTwos	0,1118	0,1087	70.380.000	15.82 day
SoccerTwos_basic\SoccerTwos	0,1307	0,1271	42.090.000	7.245 day

The graph clearly illustrates that with the baseline hyperparameters, the value loss levels off, suggesting that the model may have reached its learning capacity and could be converging on a suboptimal solution. In contrast, the implementation of revised hyperparameters results in a persistently lower value loss, indicating a superior predictive accuracy and demonstrating that the algorithm continues to refine its value function effectively over time.

Losses/Policy Loss



Upon examination of the policy loss graph, we can observe a clear distinction in the learning patterns of the two models. The model with the **basic hyperparameter settings** quickly stabilizes at a low level of policy loss, indicative of a rapid convergence to a steady policy. In contrast, the model with the **revised hyperparameters** initially experiences a higher loss, likely due to explorative policy adjustments. However, it subsequently demonstrates a steady and promising decrease in policy loss. This trend implies that, with extended training, there is a potential for this model to surpass the performance of the baseline model, as it seems to be on a trajectory toward a more refined policy.

Conclusion

Analyzing a range of metrics through TensorBoard has provided us crucial understanding of the learning behavior and capabilities of our models. By customizing our hyperparameters, we've successfully enhanced model performance beyond the foundational setup.

The adaptability of the revised hyperparameter model is particularly promising. The continuous improvement in its performance metrics suggests that, with additional computational resources and time, it could achieve an even more optimal policy. This adaptability is crucial in complex, dynamic environments where the ability to refine and improve strategies over time can be a significant advantage.

Here we can see our agent play against other agents made by others.

Developed games for testing the created models

In the end, we developed games for testing all the created models. A 1 versus 1, a 2 versus 2 and a 3 versus 3 were made. The **9. Figure** shows a picture about the 2v2Game scene.

***9. Figure**: Developed 2v2Game scene*

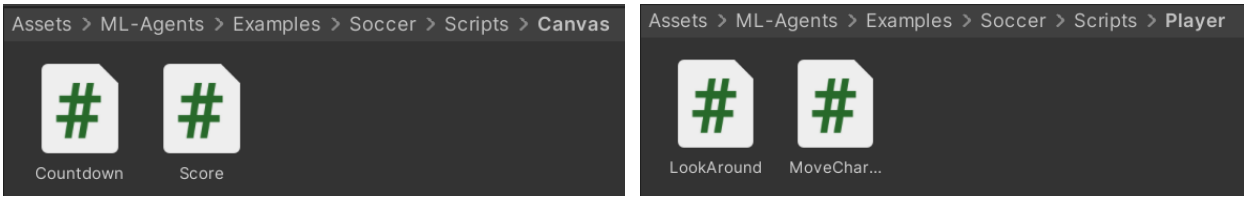
First of all, all the components that made an agent an agent had to be removed from one agent. Then the original camera was deleted and a new one was inserted at the eye level of the modified GameObject. We created a script (*LookAround*) that allowed the camera and the body (GameObject) to be rotated together. After this another script was made (*MoveCharacter*) that allows the player to make the character move using the WASD keys. These two codes result in the ability to easily go around with the character on the playing field.

***10. Figure**: The inserted camera*

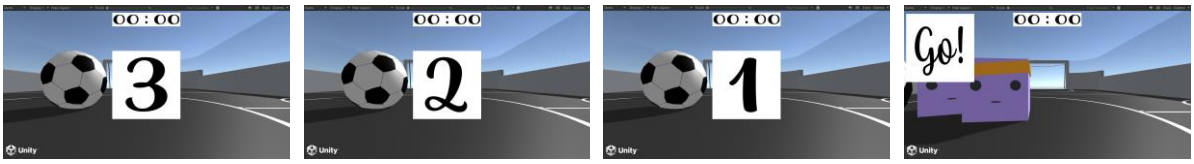
Since one of the agents became a player, the *SoccerEnvController* script had to be modified. We saved information also regarding the player's GameObject and we also made a code snippet for reset the position of the player as well after scoring a goal. The **11. Figure** shows the Scripts folder. As it's noticeable, two folders were created into this.

***11. Figure**: Modified Scripts folder*

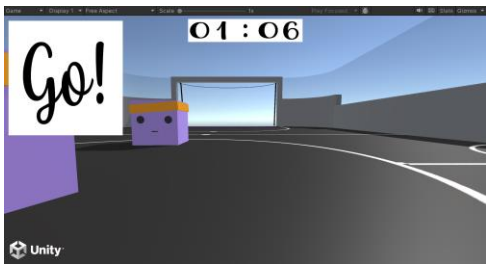
The **Player** folder contains the already mentioned scripts which are responsible for the moving. The **Canvas** folder contains two newly created scripts. The *Countdown* locks all the agents and the player during the countdown at the beginning of the game and then it activates the deactivated components. The *Score* script is used to track the current standings of the match.



***12. Figure**: The content of the two created script folder*



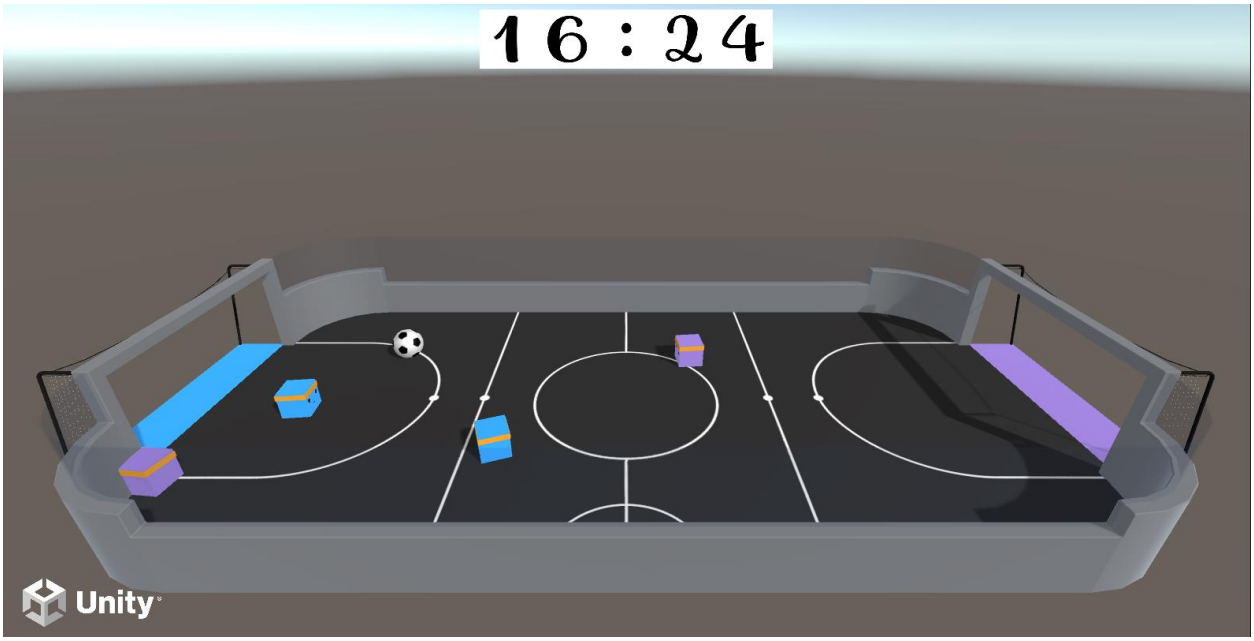
***13. Figure**: The countdown*



***14. Figure**: Scoreboard during a game after a restart*

After manually testing the agents with the game, we decided to create an observation scene with the *ObservationScene* name. We created previously two other models, namely the *LotTrained_SoccerTwos.onnx* and the *ChangedParameters_SoccerTwos.onnx*. We used the game to compare them.

After a lot of examination, the *LotTrained_SoccerTwos.onnx* came out as the best where we just trained the basic agent a lot more. We assigned to the blue agents the *ChangedParameters_SoccerTwos.onnx* model and to the purple ones the *LotTrained_SoccerTwos.onnx* model. The **15. Figure** shows a temporary result of the game.



***15. Figure**: Compare ChangedParameters_SoccerTwos.onnx* and *LotTrained_SoccerTwos.onnx*

The repository for this project can be found here: https://github.com/Ilosviki/SoccerGame

Additional resources used apart from the course material

[1] Unity-Tech-Cn, "PlasticHub," Unity Technologies, 2021. [Online]. Available: https://plastichub.unity.cn/unity-tech-cn/ml-agents/src/commit/dbd21c95-1f52-48ff-b6a8-0ee69780db66...docs/Training-ML-Agents.md.

[2] Unity Technologies, "Unity ML-Agents Toolkit - Release 21," 9. October 2023. [Online]. Available: https://github.com/Unity-Technologies/ml-agents/tree/develop.

[3] Lucas A.E Pineda Metz, "An Evaluation of Unity ML-Agents Toolkit for Learning Boss," September 2020.. [Online]. Available: https://skemman.is/bitstream/1946/37111/1/An%20Evaluation%20of%20Unity%20ML-Agents%20Toolkit%20for%20Learning%20Boss%20-%20MSc%20Thesis.pdf.

[4] "Training Configuration File," Unity Technologies, 14. December 2022. [Online]. Available: https://github.com/Unity-Technologies/ml-agents/blob/main/docs/Training-Configuration-File.md#behavioral-cloning.

[5] Joy Zhang, "Competitive self-play with Unity ML-Agents," Coder One, 22. October 2021. [Online]. Available: https://www.gocoder.one/blog/competitive-self-play-unity-ml-agents/.

[6] Liangxiang Accelerator, "Principle and application of ELO algorithm," zhihu, 02. November 2020. [Online]. Available: https://zhuanlan.zhihu.com/p/57480433.

[7] Immersive Limit, "Unity ML-Agents - Demonstration Recorder for Imitation Learning," 22. January 2020.. [Online]. Available: https://www.youtube.com/watch?v=Dhr4tHY3joE.

[8] Code Monkey, "Teach your AI! Imitation Learning with Unity ML-Agents!," 05. December 2020.. [Online]. Available: https://www.youtube.com/watch?v=supqT7kqpEI.

[9] DContrerasF, "Best parameters for SoccerTwos," Hugging Face, 23. November 2023.. [Online]. Available: https://huggingface.co/DContrerasF/poca-SoccerTwos-alt/commit/c8772ad77b2d890b0bd316a6e89a323d0fdb0414.

[10] SmartLab AI, "A brief overview of Imitation Learning," 19. September 2019.. [Online]. Available: https://smartlabai.medium.com/a-brief-overview-of-imitation-learning-8a8a75c44a9c.

[11] smiling star, "Reinforcement Learning PPO," CSDN, 21. February 2022.. [Online]. Available: https://blog.csdn.net/tianjuewudi/article/details/120212234.

[12] "Proximal Policy Optimization Algorithms," OpenAI, 28. August 2017.. [Online]. Available: https://arxiv.org/pdf/1707.06347.pdf.

[13] "Proximal Policy Optimization," OpenAI, 2018.. [Online]. Available: https://spinningup.openai.com/en/latest/algorithms/ppo.html.

[14] Hugging Face, "Introduction to Hugging Face," [Online]. Available: https://huggingface.co/learn/deep-rl-course/unit7/introduction.

[15] "GAIL vs Behavioral Cloning, what's the difference?," Unity Technologies, [Online]. Available: https://forum.unity.com/threads/gail-vs-behavioral-cloning-whats-the-difference.944463/.

[16] smiling star, "ML-Agents Case: Wall Jumping Game," CSDN, 11. November 2021.. [Online]. Available: https://blog.csdn.net/tianjuewudi/article/details/121269518.

[17] "Off-Policy Maximum Entropy Deep Reinforcement," 08. August 2018.. [Online]. Available: https://arxiv.org/pdf/1801.01290.pdf.

[18] Hugging Face, "Unity ML-Agents Toolkit with Hugging Face," 08. June 2023.. [Online]. Available: https://github.com/huggingface/ml-agents#get-started.

[19] Andrew Cohen, Ervin Teng, Vincent-Pierre Berges, Ruo-Ping Dong, Hunter Henry, Marwan Mattar, Alexander Zook and Sujoy Ganguly, "On the Use and Misuse of Absorbing States in Multi-agent Reinforcement," Unity Technologies, 07. June 2022.. [Online]. Available: https://arxiv.org/pdf/2111.05992.pdf.

[20] Thomas Simonini, "An Introduction to Unity ML-Agents with Hugging Face," 22. June 2022.. [Online]. Available: https://thomassimonini.medium.com/an-introduction-to-unity-ml-agents-with-hugging-face-efbac62c8c80.

[21] Hugging Face, "Getting started with our git and git-lfs interface," [Online]. Available: https://huggingface.co/welcome.

[22] H. Face, "SoccerTwos Challenge Analytics," Hugging Face, 01. March 2023.. [Online]. Available: https://huggingface.co/spaces/cyllum/soccertwos-analytics.

[23] "Models for SoccerTwos," Hugging Face, [Online]. Available: https://huggingface.co/models?search=SOCCERTWOS.

[24] Jari Hanski, Biçak and Kaan Baris, "An Evaluation of the Unity Machine Learning Agents Toolkit in Dense and Sparse Reward Video Game Environments," May 2021.. [Online]. Available: https://www.diva-portal.org/smash/get/diva2:1563588/FULLTEXT01.pdf.

[25] Vincent-Pierre Berges, Ervin Teng, Andrew Cohen and Hunter Henry, "ML-Agents plays DodgeBall," 12. July 2021.. [Online]. Available: https://blog.unity.com/engine-platform/ml-agents-plays-dodgeball.

[26] "On the Use and Misuse of Absorbing States in Multi-agent Reinforcement," [Online]. Available: https://rlg.mlanctot.info/papers/AAAI22-RLG_paper_32.pdf.