PAing

Luca Ghisi

September 3, 2024

This project is the course "Ai for Videogames" assignment at the University of Milan.

1 Assignment text

Goal

The goal of this project is to create an agent to play the PONG game using machine learning

Setup

The field is a flat surface of 10 by 5 meters closed by walls on the long sides (up and down) and open on the short ones (left and right). On the left and right sides there are 2 paddles (one on each side) moving along up and down. Paddles measure 1 meter along the dimension facing the field. Inside the field there is a disk with diameter 0.3 meters bouncing back and forth. The speed of the disk is constant at 15 m/s. At the beginning, the disk starts from the center and is set to a random direction. It is not required to keep track of the score. When the disk goes out of the field a new game is started. The system has no friction and the disk, when bouncing on the walls, will conserve its energy but not the direction. At every bounce on walls, the outgoing direction will change by a random value in the range [-5, 5] degrees. E.g., an incoming angle of 30 degrees will make the disk bounce with an outgoing angle between 25 and 35 degrees. The random change is not applied when the disk bounces on the paddles.

Constraints

You must train, using ML, an agent to move the paddle. Of course, the goal of the paddle is to keep the disk inside the field. Once trained, the same agent must be used on both paddles and play against itself

- **ML-Agents version:** 3.0.0 (git branch release_21)
- **Python version:** 3.10.12

- **Torch version:** 1.13.1+ccu117 (CUDA version 11.7)

2 Assignment analysis

In summary, the assignment's constraints are:

1. The disk needs to have a constant speed with no friction. Whenever it bounces off a wall it conserves its energy but not its direction. At each bounce against the wall, a random value between [-5,5] degrees will be added.
2. The agent applied to the paddles must be trained using the Machine learning provided by Unity technologies to move the paddles up and down to keep the disk inside the game field.
3. After completing the training, the neural network model needs to be applied to two paddles so that they can play against each other.

3 Assignment Solution

Overview

The main key components to solve this assignment are:

- **PaddleAgent**, and **Disk** class: Both scripts are important to set up the environment and the agent behavior.
- Two Unity Scene:

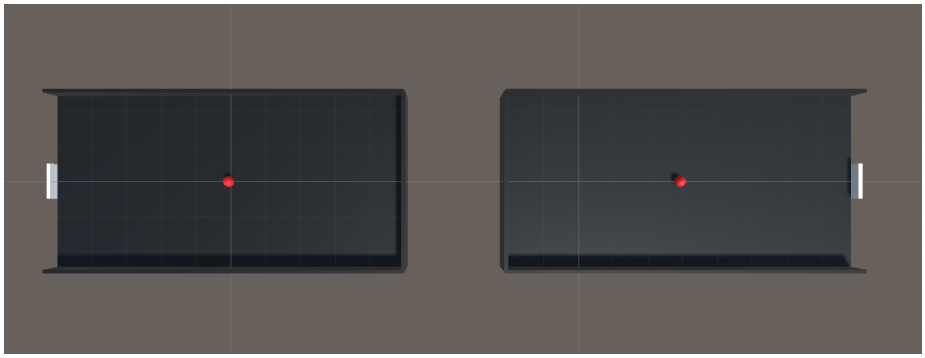

Figure 1: Training scene: Game field setup

  - Training Scene: this scene is the training environment where the agent learns to achieve its objective. It has multiple instances of the same game field, each has one paddle on either the left or right side

and a wall on the opposite side. This setup gives the agent enough observation that help the training process. By training both sides simultaneously, the agent can get a better insight into the disk dynamics. For this particular training, the scene has 64 game instances.

Figure 2: Training scene: Game field setup

  - Paing Scene: this scene contains only one game field instance. It contains two paddles and one disk. By applying the neural network model, developed during the training to both paddles, they will be able to play against each other and keep the disk inside the field.

- The Python API and Python Trainer are essential components for training agents in the Unity environment.
  - Python API: This component connects to the Unity environment and receives the observation through Unity's internal communicator.

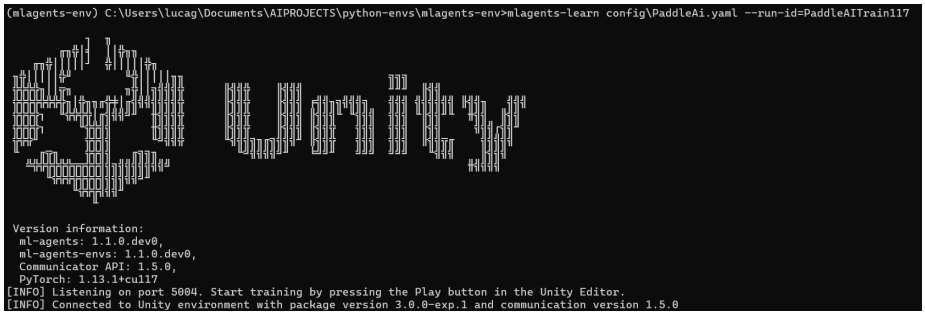

Figure 3: Unity MLagents training environment

  - Python Trainer: This component executes the training through the 'mlagents-learn' command. It is based on the configuration file in YAML format, which has all the hyperparameters needed to set up

the training process. If none are provided when running MLgents-learn, it will create a standard configuration file for that training session. It uses a deep reinforcement learning algorithm, the Proximal Policy Optimization (PPO), to process the observation and decide which action to take and send them back to the Unity environment and receive a positive or negative reward based on that taken action.

These key components work together to generate an effective training environment that will create a neural network model that will fulfill the assignment.

3.1 Disk class documentation

This class is used to simulate a bouncing disk in the game field. The key components of this class are:

- Energy conservation: the disk needs to keep a constant speed of 15 m/s.
- Direction changes: the disk when it bounces off a wall, conserves the energy but not the direction by adding a random amount between [-5,5] degrees. Instead, when it bounces off the paddle it conserves both energy and direction.

The disk has the Rigidbody component to manage the collision with the wall and the paddle. But since it must keep a constant velocity and needs to change direction at each collision, the rigdbody.velocity is used to move the disk to avoid having acceleration or deceleration.
To avoid possible conflict or odd behavior, due to the rigidbody physic, in the rigidbody component:

- the gravity is disabled.
- the linear and angular drag is set to 0
- the x, y, z rotation components are set to freeze

Variables

```
public float speed = 15;
private Vector3 direction;
public Rigidbody rb;
public bool OutOfBounds;
private Vector3 startPosition;
```

The variables are related to the disk object. All of them are self-explanatory.

Methods

- **Start()**: Saves the initial position.

```
private void Start()
{
    startPosition = transform.position;
}
```

- **ResetDiskDirection()**: it calls whenever a new agent episode begins (see PaddleClass). It resets the disk position and calculates a new random direction that is going to be applied to the rigidbody.velocity.

```
public void ResetDiskDirection()
{
   if (OutOfBounds)
         OutOfBounds = false;
    transform.position = startPosition;
    int startRandomDirection = Random.Range(0, 360);
    direction = new Vector3(Mathf.Cos(startRandomDirection * Mathf.Deg2Rad),
    0f, Mathf.Sin(startRandomDirection * Mathf.Deg2Rad));
    rb.velocity = direction * speed;
}
```

- **OnCollisionEnter()**: It extrapolates the collision normal between the disk and the paddle or the wall by accessing their tags set up in the Unity scene. If the disk collides with the wall, the new outgoing direction is processed using the Vector3.reflect, which reflects a vector off the plane defined by a normal. Afterward, a random value between the range of [-5 to 5] degrees is applied to the outgoing direction using the 3D rotation matrix. To calculate the new direction:

$$outgoingDirection = \begin{bmatrix} x \\ y \\ z \end{bmatrix} * \begin{bmatrix} \cos(\theta) & 0 & -\sin(\theta) \\ 0 & 1 & 0 \\ \sin(\theta) & 0 & \cos(\theta) \end{bmatrix}$$

where θ is the random value applied to the outgoing direction. Then the rb.velocity is updated with the new direction while maintaining the same speed.
Instead, if the disk collides with the paddle, the new outgoing direction is first calculated with Vector3.reflect, and then rb.velocity is updated with the new direction.

```
private void OnCollisionEnter(Collision collision)
{
    if (collision.gameObject.tag == "Wall")
    {
        direction = Vector3.Reflect(direction, collision.contacts[0].normal);
        float randomDeflectAngle = Random.Range(-5f, 5f)*Mathf.Deg2Rad;
        float directionX = direction.x * Mathf.Cos(randomDeflectAngle) -
        direction.z * Mathf.Sin(randomDeflectAngle);
        float directionZ = direction.x * Mathf.Sin(randomDeflectAngle) +
        sdirection.z * Mathf.Cos(randomDeflectAngle);
        direction = new Vector3(directionX, 0f, directionZ);
        rb.velocity = direction.normalized* speed;
    }

    if (collision.gameObject.tag == "Paddle")
    {
        direction = Vector3.Reflect(direction, collision.contacts[0].normal);
        rb.velocity = direction.normalized * speed;
    }
```

- **OnTriggerEnter()**: It detects when the disk is out of bounds. When this happens, the agent will reset (see **PaddleAgent** class for more detail).

```
private void OnTriggerEnter(Collider other)
{
    if (other.gameObject.tag == "Bound")
    {
        OutOfBounds = true;
    }
}
```

3.2 PaddleAgent class documentation

This class implements the agent's behavior using the Unity MLagents libraries (MLAgents, Sensors, Actuators). Whenever the agent chooses an action it will receive a positive or negative reward based on that action. A fundamental aspect of this script is the rewards values fine-tuning. These final selected values are good enough to generate, with the help of MLagents Python trainer, a neural network model that learns to keep the disk from going out of bounds.

Variables

```
public Transform diskTransform;
 public float currentSpeed = 0f;
 public float minSpeed = 5f;
 public float maxSpeed = 20f;
 public Vector3 startPosition;
 public float zDistanceThreshold = 0.5f;
 private float optimalSpeed = 2f;
```

The variables are related to the paddle agent object. Some of them are self-explanatory, besides:

- **zDistanceThreshold**: A float that indicates how close the agent's z coordinates position must be to the disk's z coordinate position to receive a reward.
- **optimalSpeed**: value that helps the agent to match the speed of the disk.

Methods

- **Start()**: Save the startPosition as the current transform.position

```
void Start()
{
    startPosition = transform.position;
}
```

- **OnEpisodeBegin()**: The ML-Agents method is used to set up the Agent instance at the beginning of each episode and reset the disk by calling the **ResetDisk()** method.

```
public override void OnEpisodeBegin()
{
    transform.position = startPosition;
    RestDisk();
}
```

- **CollectObservation()**: The MLagents method collects the observation vectors of the agent for the step. This function describes the current environment from the agent's perspective. In this context, the agent needs to keep track of:
  - The z coordinate position, due to how the paddle is placed in the game field in the unity scene.
  - The paddle's speed.
  - The x and z coordinate the position
  - The disk's x and z velocity components

  By keeping track of these six floats the agent has enough information to take action to keep the disk inside the game field. It is important for the training to know how many elements the agent is tracking. This number must match the dimension of the **Behavior Parameters Vector Observation**.

```
public override void CollectObservations(VectorSensor sensor)
{
    sensor.AddObservation(transform.position.z);
    sensor.AddObservation(diskTransform.position.x);
```

```
        sensor.AddObservation(diskTransform.position.z);
        sensor.AddObservation(diskTransform.GetComponent<
        Rigidbody>().velocity.x);
        sensor.AddObservation(diskTransform.GetComponent<
        Rigidbody>().velocity.z);
        sensor.AddObservation(currentSpeed);

    }
```

- **OnActionRecived()**: The MLagents method is used to specify an agent's behavior at every step. The provided actions are passed through the ActionBuffer parameter, which specifies how many actions are needed to control the agent. For this project, the paddle's movement is restricted to the z-axis. This is due to how the paddle is positioned in the game field. This specific agent requires two continuous actions: one to control the movement, and the other to control the speed between a minimum and maximum value. This is done for simplicity because using the discrete actions will add too many elements for the agent to control. For example just for the paddle movement, it needs three discrete actions: one to move up, one to move down, and one to stand still.
  This method also sets positive rewards whenever the agent:

  – Aligns its z-coordinate position with the disk's z-coordinate position under the **zDistanceThreshold**.

  – matches the disk's speed under the **optimalSpeed** threshold.

  This two rewards make the agent able to keep up with the disk dynamics.

```
public override void OnActionReceived(ActionBuffers actions)
{
    float moveZ = actions.ContinuousActions[0];
    currentSpeed = Mathf.Clamp(Mathf.Clamp(actions.ContinuousActions[1],
    0, 1) * maxSpeed, minSpeed, maxSpeed);

    transform.position += new Vector3(0, 0, moveZ) *
    currentSpeed * Time.deltaTime;

    float zDistanceToDisk = Mathf.Abs(diskTransform.position.z
    - transform.position.z);

    float speedVariance = Mathf.Abs(diskTransform.
    GetComponent<Rigidbody>().velocity.magnitude - currentSpeed);


    if (zDistanceToDisk <= zDistanceThreshold
    && Vector3.Distance(diskTransform.position, transform.position)<5)
        SetReward(0.1f);

    if (speedVariance <= optimalSpeed)
```

```
            SetReward(0.5f);

    }
```

- **Update()**: This method returns a negative reward whenever the disk goes out of bounds, ending the current episode to start a new one.

```
private void Update()
{
    if (diskTransform.GetComponent<Disk>().OutOfBounds
    || Vector3.Distance(transform.position, diskTransform.position) > 13f)
    {
        SetReward(-1f);
        EndEpisode();
    }
}
```

- **OnCollisionEnter()**: This method returns a positive reward to the paddle agent each time it successfully hits the disk. This reward was implemented to let the agent know that this action yields a positive outcome, encouraging it to keep the disk inside the field.

```
private void OnCollisionEnter(Collision collision)
{
    if(collision.gameObject.tag == "Disk")
    {
        SetReward(1f);
    }
}
```

- **ResetDisk()**: This method calls the **Disk**'s **ResetDiskDirection()** method.

```
private void RestDisk()
{
    diskTransform.GetComponent<Disk>().ResetDiskDirection();
}
```

3.3 Behaviour Parameters and Request Decision class documentation

For the agent to work and communicate with Python API and Python Trainer, the **Behaviour Parameters** and **Decision Requester** scripts must be attached to the object that has the **PaddleAgent** script.

- **Behaviour Parameters**: It defines the behavior and the brain properties of an agent within a learning environment. The important parameters are:
  - **Behavior name**: The name put in this field needs to be the same as the behavior name of the configuration file yaml for the training to start.(see subsection 3.4)

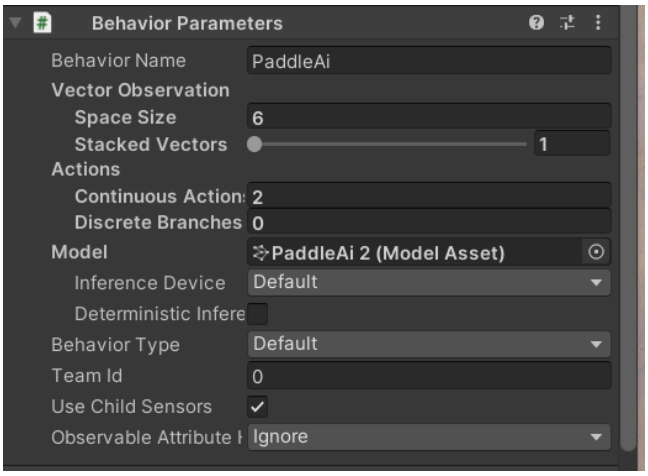

Figure 4: Behaviour Parameter Script

- **Vector Observation**: Contains the 6 float values, added in the **PaddleAgent**'s **CollectObservations()** method.
- **Stacked Vectors**: This parameter represents the amount of stacked observation needed before making a decision.
- **Action**: refers to the agent's decision based on its observation. It sets the type and number of actions used by the agent. In this case, as mentioned in **PaddleAgent**'s **OnActionRecived()**, the paddle agent has two continuous actions.
- **Model**: is the reference where is going to be added the trained neural network model that will control the paddles.

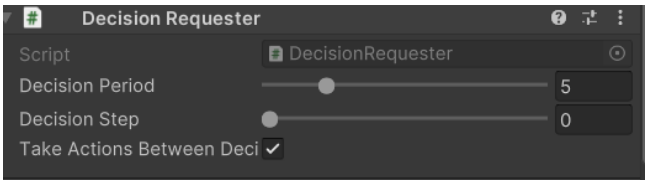

Figure 5: Decision Requester

- **Decision Requester Script**: it requests a decision every certain amount of time and then takes action.
  - **Decision Period**: How frequently the agent requests a decision.
  - **Decision Steps**: Tells when to request an action. By changing this value the timing of the decision can be shifted among agents. In this case all agents request the decision at the same time.

– **Take Action Between Decision**: Highlights if the agent will take an action even when it doesn't request a decision.

3.4 Training Configuration

This is the configuration file used for the agent training **PaddleAI.yaml**

```
behaviors:
  PaddleAi:
    trainer_type: ppo
    hyperparameters:
      batch_size: 512
      buffer_size: 20000
      learning_rate: 0.0003
      beta: 0.001
      epsilon: 0.2
      lambd: 0.99
      num_epoch: 3
      learning_rate_schedule: linear
      beta_schedule: constant
      epsilon_schedule: linear
    network_settings:
      normalize: true
      hidden_units: 256
      num_layers: 2
      vis_encode_type: simple
    reward_signals:
      extrinsic:
        gamma: 0.99
        strength: 1.0
    keep_checkpoints: 5
    max_steps: 4000000
    time_horizon: 1000
    summary_freq: 20000
```

This file contains all the hyperparameters informations used to configure future training sessions. The behavior name in this file needs to be the same as the behavior chosen in the **Behavior parameter** script. MLagents can utilize two algorithms: **Proximal Policy Optimization (PPO)** and **Soft Actor-Critic (SAC)** For this project the PPO algorithm is used since is more general-purpose and stable than many other reinforcement learning algorithms.

3.4.1 Proximal Policy Optimization (PPO) and Hyperparameters configuration

Proximal Policy Optimization (PPO) is a deep reinforcement learning algorithm and a policy gradient method that balances exploration and exploitation

to train a computer agent's decision in solving complex tasks.

Key components:

- **Actor-Critic**: PPO is an actor-critic algorithm that combines aspects of both the policy-based method (Actor) (the behavior of the agent), and value-based Mehtod (Critic) (predicts the expected future reward).
  - **Actor**: Decides which action to take based on the current policy. The objective is to find the optimal policy that maximizes the cumulative reward.
  - **Critic**: Asses the quality of the action done by the current policy (actor). This estimation helps the actor to find the optimal policy.
- **Probability Ratio**:

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$$

  f
  - It estimates the divergence between the current and old policy for taking action in the state. The actor decides by choosing the action based on the current policy
  - If $r_t(\theta) > 1$, the action is more likely based on the current policy; if $0 < r_t(\theta) < 1$, the action is more likely based on the old policy.
- **Advantage Function (Generalized Advantage Estimation - GAE)**: This method evaluates if the specific action taken by the agent under a certain policy is better or worse than any other action in a given state. It reduces substantially the variance of the policy gradient estimate at the cost of some bias, with an exponentially decayed sum of residual terms.

$$A^{GA\hat{E}(\gamma,\lambda)}{}_t = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}^V$$

  - λ: is the parameter that adjusts the* trade-off between bias and variance.
  - γ:is the discount factor that weights the importance of the future reward and also contributes on the bias-variance tradeoff.
  -

$$\delta_t^V = R_t + \gamma V(s_{t+1}) - V(s_t)$$

    This formula represents the temporal difference residual, or Bellman residual, (at a specific time stamp t) where first is estimated the sum of the discounted rewards and then is subtracted the value function estimate of it
    - γ is the discount factor.

    - $V(s_t + 1)$: estimated value of the next state.
    - $V(s_t)$:estimated value of the current state.
    - R_t: Is the reward received after an action in the s_t state.

- **Clipped Surrogate Objective:** Is the policy gradient method used by the algorithm.

$$L^{CLIP}(\theta) = \mathbb{E}_t \left[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon)\hat{A}_t) \right]$$

  - Restricts policy changes during training to maintain stability.
  - Clipping keeps $r_t(\theta)$ within the range $[1 - \epsilon, 1 + \epsilon]$ to prevent large updates that destabilize learning.

Learning Process:

The **Objective function** is the combination of the policy surrogate and the value function error term. To ensure more exploration it is also added the entropy bonus. This function optimizes at the same time the policy (actor) and the value function (critic).

$$L_t^{CLIP+VF+S}(\theta) = \mathbb{E}_t \left[L_t^{CLIP}\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t) \right]$$

- $L_t^{VF}(\theta)$: is the squared error loss $(V_\theta(s_t) - V_t^{targ}t(\theta))^2$.
- S is the entropy bonus.
- c1 and c2 are coefficients.

In the algorithm iteration:

- The agent collects rewards, states, actions, and value estimates from interactions with the environment.
- To the batch is applied the Objective function, by computing all of its components, to optimize the value function and the policy.
- This process is repeated over many episodes to gradually refine the policy until the maximum number of training steps is reached.

Hyperparameters:

- **Batch Size:** Number of experiences used for one iteration of gradient descent, typically a fraction of the buffer size.
- **Buffer Size:** Number of experiences collected before the model updates (makes a decision based on the policy).
- **Learning Rate:** Determines the strength of each gradient descent update step.
- **Beta:** Strength of entropy regularization to encourage exploration.

- **Epsilon:** Divergence threshold between the old and new policies during updates; clipping parameters for PPO gradient descent method.
- **lambd / Lambda**: is the parameter used when calculating the GAE. This represents how much the agent relies on its current value estimate. A low value means a high bias since the agent relies more on the current value estimate. Meanwhile, a high value means more variance since the agent relies more on the rewards collected from the environment. High value also provides more stable training.
- **Number of Epochs:** Passes through the experience buffer during gradient descent.
- **Learning Rate Schedule:** Defines how the learning rate changes over time, typically decreasing linearly.
- **Beta Schedule:** Refers to a constant beta value.
- **Epsilon Schedule:** Gradual adjustment of the epsilon parameter to favor exploration initially and decrease over time.

Network Settings:

- **Normalize:** Normalization applied to vector observation inputs for training stability.
- **Hidden Units:** Number of units in each fully connected layer.
- **Number of Layers:** Number of hidden layers in the network.

Reward Signals:

- **Gamma:** Discount factor for future rewards, indicating how much future rewards are valued compared to immediate ones. This parameter is used for GAE.
- **Strength:** Multiplier factor for raw rewards to influence learning.

The hyperparameters need to be fine-tuned to have a better training configuration that can lead to better results for the agent, similar to rewards

3.4.2 Analysis of the training process with this specific configuration

```
[INFO] PaddleAi. Step: 20000. Time Elapsed: 42.663 s. Mean Reward: 1.094. Std of Reward: 2.086. Training.
[INFO] PaddleAi. Step: 40000. Time Elapsed: 52.635 s. Mean Reward: 1.720. Std of Reward: 3.283. Training.
[INFO] PaddleAi. Step: 60000. Time Elapsed: 64.231 s. Mean Reward: 1.801. Std of Reward: 3.358. Training.
[INFO] PaddleAi. Step: 80000. Time Elapsed: 75.648 s. Mean Reward: 2.196. Std of Reward: 4.387. Training.
[INFO] PaddleAi. Step: 100000. Time Elapsed: 85.926 s. Mean Reward: 2.587. Std of Reward: 4.865. Training.
[INFO] PaddleAi. Step: 120000. Time Elapsed: 95.810 s. Mean Reward: 2.883. Std of Reward: 5.070. Training.
[INFO] PaddleAi. Step: 140000. Time Elapsed: 103.232 s. Mean Reward: 2.352. Std of Reward: 4.383. Training.
```

Figure 6: Training first steps

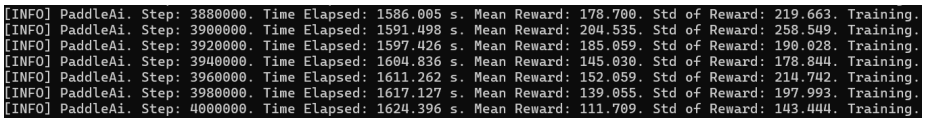

Figure 7: Training final steps

For this project, the training session took more or less 26 minutes to complete using the 64 game field instances in the Unity Training scene.
This is the part of the assignment that was the most time-consuming because it requires monitoring the changes in the mean reward and the standard deviation reward's values. This allows to have a rough idea of the effectiveness of the agent's training.

- **Mean Reward**: is the average reward received by the agent form the most recent episodes.
- **Standard Deviation of the reward (std)**: is the statistical measure of the variability of the rewards received by the agent.

Figure 6 and Figure 7 illustrate the mean reward and standard deviation reward (str) at the beginning and end of the training, between specified intervals defined by the **summary-freq** hyperparameter.
By analyzing the mean reward at each step summary, it highlights a steady mean reward growth, despite some fluctuation. This shows that the agent is steadily learning to take action that yields higher rewards over time.
Also, the std value keeps increasing as well over time and it's having some high fluctuations. They describes an agent that is actively searching for different effective strategies to keep the disk inside the game field.
However, this is not enough information to comprehend the effectiveness of the training. For example, the high value of the Standard deviation reward can also mean instability of strategies learned by the agent. To gain substantial information about the agent performance, is important to also analyze the graphs information provided by the Tensorboard application.

3.4.3 Tensorboard results analysis for PPO

Tensobaord gives a complete insight into different training aspects undergone by the agent in the form of visual graphs. These graphs allow us to analyze and evaluate if the trained model had effective training. The graphs here presented describe the training of the neural network model tuned with the parameters of the configuration used for this agent.

- **Environment Statistics**
  - **Mean Cumulative Reward:** Graph that represents the mean cumulative reward (Figure 8) over all the agents and is the same information that is shown in the training process but in a visual format.

Figure 8: Cumulative Reward and Episode Length Graphs

The graph's trend shows consistent growth of the cumulative reward, despite some fluctuation mainly between the 3 and 4 million steps. This upward trend, as mentioned in the previous subsubsection 3.4.2, shows that the agent is applying strategies that return higher rewards during the training session.

– **Episode Length:** The graph (Figure 8) represents the mean length of each episode for all the agents. This graph shows that over time the length of the episodes increases, meaning the agent is getting better at keeping the disk from going out of bounds.

- **Loss Statistics**

Figure 9: Policy and Value loss Graphs

– **Policy Loss Graph:** The mean magnitude of the PPO loss policy function. It correlates with how much the policy is changing. As shown in the figure Figure 9, its magnitude decreases during a successful training.

– **Value Loss Graph:** The mean loss of the value function update. This represents how well the model can predict the value of each state. Initially, (Figure 9), it increases when the agent is learning and steadily decreases once it stabilizes. In this case, there is a steady increase until the 3 million steps and steadily decreases meaning that the agent started to stabilize around those steps.

- **Policy Statistics**

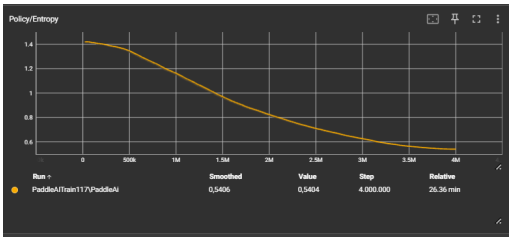

Figure 10: Entropy Graphs

– **Entropy:** Represents the unpredictability of the action taken by the model. A decrease in entropy during a successful training process (Figure 10 shows that agent is exploiting it's strategies effectively.

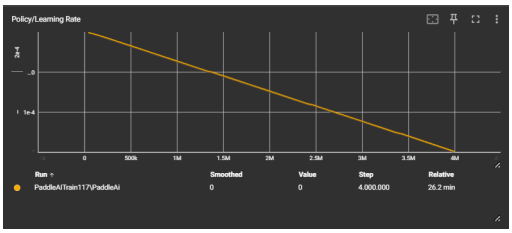

Figure 11: Learning Rate Graphs

– **Learning Rate:** This parameter determines how large a step the training algorithm takes as it searches for the optimal policy. Since it is set as linear in the hyperparameters, the graph is a straight line transitioning from the maximum learning rate to the minimum value when training stops at step 4,000,000 meaning a stabilization on the agent learning process while choosing optimal strategies.

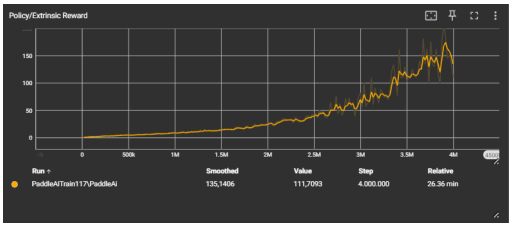

Figure 12: Extrinsic Reward Graph

– **Extrinsic Reward:** This statistic corresponds to the mean cumulative reward (Figure 8). This is because this agent only considers the extrinsic reward.

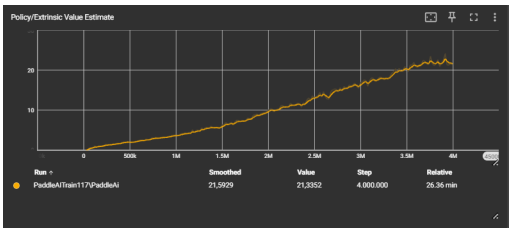

Figure 13: Extrinsic Value Estimate Graphs

– **Extrinsic Value Estimate:** The mean value estimate for all states visited by the agent. An increase in the value estimate means that the agent is evaluating correctly the different states and getting a high value in return.

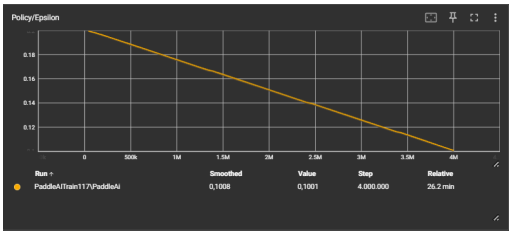

Figure 14: Epsilon

– **Epsilon:** This parameter (Figure 14), set in the hyperparameters of the configuration YAML, is the PPO parameter that represents a balance between exploration and exploitation. It decreases with a decay function set in the configuration yaml, which in this case is linear.
– **Beta:** This parameter is the strength of the entropy regularization and this parameter is used to slowly decrease the entropy. Set in the hyperparameters of the configuration YAML, which is a constant value.

By analyzing the environment, loss, and policy statistics, it can be stated that the model has learned to exploit strategies that lead to effectively reaching the set goal of keeping the disk inside the field.

3.5 Result of the training

The resulting neural network model (.oox file) is placed in the model variable of the Behavior parameters script to both paddles.
When running the scene, the final result fulfills the assignment's requirement: two paddles playing against each other, with the main objective of keeping the

disk inside the game field. Occasionally, one of the paddles may fail to hit the disk causing it to go out of bounds and resetting the game. This is because the agent in those specific situations during the training session didn't fully develop effective strategies. This can be solved by letting the agent have a longer training session.
It is important to note that the scoring system was not implemented, since is not required in the assignment and would have meant more consideration over rewards and some changes in the agent's behavior.

4 Summary

After a deep dive into all the classes, methods, parameters, and training information, it's useful to recap the project, summing up how the requirements have been fulfilled.

Requirement	Solution
Game field setup	Paing prefab built accordingly.
The disk must maintain a constant speed with no friction	`Start()` and `OnCollisionEnter()` to calculate the random direction and speed that are going to be applied to the rigidbody, without using forces.
When the disk bounces off a wall, it conserves the energy but not the direction by adding a value in degrees between -5 and 5 to the outgoing direction.	`OnCollisionEnter()` method to manage the collision with the wall and calculate the new outgoing direction.
When the disk bounces off the paddle it conserves the energy and direction	`OnCollisionEnter()` method handles the collision between disk and paddle.
A new game starts when the disk goes out of the game field.	The `PaddleAgent` and `Disk` handle the moment when the disk goes out of bounds.
Agent must be trained using the ML-Agents libraries	Download the necessary packages needed for creating the training environment in both Python and Unity Scene. In the Unity scene, this is handled by the `PaddleAgent, DecisionRequester, BehaviorParameters` in order to collect observations, make decisions and take actions. In the Python environment, through the configuration YAML and the `mlagents-learn`, starts the training session connected to the Unity scene, using the PPO reinforcement learning algorithm.
Once trained, the same agent must be used on both paddles and play against itself	The training generated the neural network model (.onnx) that will be applied to both paddles to play against each other.

Table 1: Requirements and Solutions for pAing

5 Reference

- The **pyEnvinstalledpackages.txt** of the project contains the information regarding the packages used in the python environment used for the training inside the project folder the file
- MLagents manual: `https://docs.unity3d.com/Packages/com.unity.ml-agents@3.0/manual/index.html`
- MLagents Unity training/hyperparameters: `https://github.com/miyamotok0105/unity-ml-agents/blob/master/docs/Training-PPO.md`

- Proximal Policy Optimization PPO:https://arxiv.org/abs/1707.06347
- Genarl Advantage Estimation GAE: https://danieltakeshi.github.io/2017/04/02/notes-on-the-generalized-advantage-estimation-paper/ and https://arxiv.org/abs/1506.02438
- Tensorboard: https://unity-technologies.github.io/ml-agents/Using-Tensorboard/
-