# Introduction to Reinforcement Learning through Mistborn.

Authors:

- Adrián Estoquera
- Liang Hao Ramón:

# Table of contents

# Introduction:

Mistborn is a series of books written by Brandon Sanderson. It tells a fantasy story starring people with magical abilities. This reinforcement learning environment is used to learn the basics of this branch of machine learning through agents that mimic some of the fantasy skills of the characters of this book saga. Specifically, our virtual character can not only jump, run and move like a normal person, but can also "throw" and "pull" metal. These abilities, which we describe in detail below, can be understood as the character being able to pull metal objects to approach them as if he were Batman with a hook and pull metal as if he were Mr. Fantastic by extending his legs. The difference is that our agent does not need neither a hook nor to lengthen his legs: he uses magic to propel or attract himself.

Our project has been successful thanks to the curriculum learning technique. If we wanted to train our agent directly with the most complicated levels, he would fail regardless of the time we gave him to train. Instead, what we do is to train him for a short time in the simplest levels and gradually train him in increasingly more complicated levels once he is fluent in the simple ones. In our experience, if we train the agent in a complicated level without having taught him the simple ones first, the agent can spend hours training without success. On the other hand, by applying curriculum learning, the agent spends about 10 minutes on each level and completes all levels in less than an hour.

# Installation and Setup

## Prerequisites:

This installation guide assumes that you already have Anaconda installed in your machine. An Internet connection is required for downloading the packages, environment and programs.

It is required that you download and install **Unity Hub.** Once installed, please download and install at least the version **2022.3.4f1.** That is the version the project is built on, so we recommend using that exact version.

## Adding the project:

The project we have developed already has the scenes and agents to run the agent or create a new training loop. To import it, decompress the submitted file in any directory and click in the *Add>>Add project from disk* button. Then select the directory in which you have saved the project.

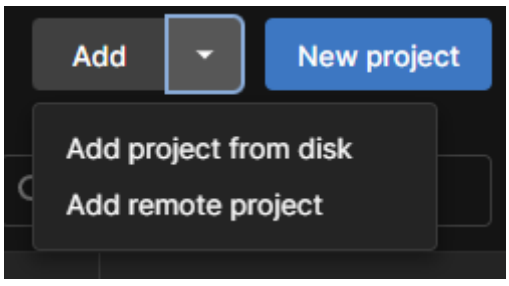


*Figure 1: Add project from disk button*

## Creation of the Anaconda Environment

It is required that the Anaconda environment used for the execution of this project contains the following packages:

- mlAgents
- Protobuf 3.20.3 (for executing the training loop)
- Onnx (for storing the agents)
- Packages (for executing the training loop)

Additionally, we will need to install pyTorch and cuda

Bear in mind that the version of **Python** we encourage to build the anaconda environment on, is **3.9**

### Installation of pyTorch

Execute the next command to install the required pyTorch version:

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

*Code 1: How to install torch*

### Installation of Cuda

First of all check wether cuda is ready in the current environment using the following command:

```
nvcc --version
```

*Code 2: How to check if cuda is installed*

If the description of the current cuda version is not provided, go to this link and select the download that best suits your machine's configuration.

### Summary

In a nutshell, you can execute the following commands to make the process we have just described.

```
conda create -n nameOfTheEnv python=3.9

conda activate nameOfTheEnv

pip3 install torch -f
https://download.pytorch.org/whl/torch_stable.html

nvcc --version

pip install protobuf==3.20.3

pip install mlAgents

pip install packages

pip install onnx
```

*Code 3: Summary of the Anaconda configuration*

### Additional troubleshooting help

We have added to the bibliography a link to a video we have found useful; However, the video is incomplete and does not tell you all the packages to install. That is why in case of following only the instructions in the video you may encounter runtime errors when launching the training loop. However these errors are easy to fix as they state which package needs to be installed.

# Resolution and testing in the environment

## First environment developed

We first developed a 3D environment for this project, and we had it working, ready to implement ML-Agents. However, after reflecting about the difficulty of the task and if the agent would be capable of learning the more complex actions of pushing and pulling metal, we decided that 3 dimensions were maybe too much to handle on this group work, so we recreated the environment in 2 dimensions. Here is an image of the first environment we had prepared.

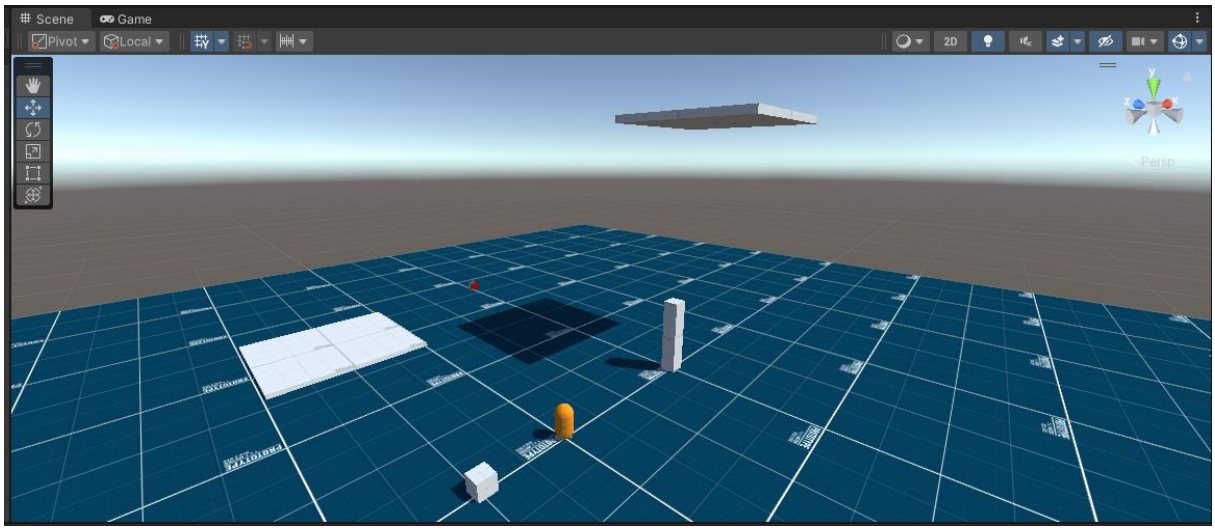



*Figure 2: Prototype Environment in 3D*

# Definition of the problem

In the end our problem consists of an agent that must reach the goal of a two-dimensional map in the shortest possible time. Our agent has the following skills: He can jump, run, walk, and "pull" and "push" metal. That is, the agent can "look" (With an element called Head) at any angle between 0º and 360º and if the object he is looking at has the metal label he can either apply force against the object and be thrown in the opposite direction ("push") or apply force towards the object and "pull" it. It is worth mentioning that these powers -and the environment in general- are taken from the Mistborn book saga, by Brando Sanderson. For more information about these powers, visit the link in the bibliography.

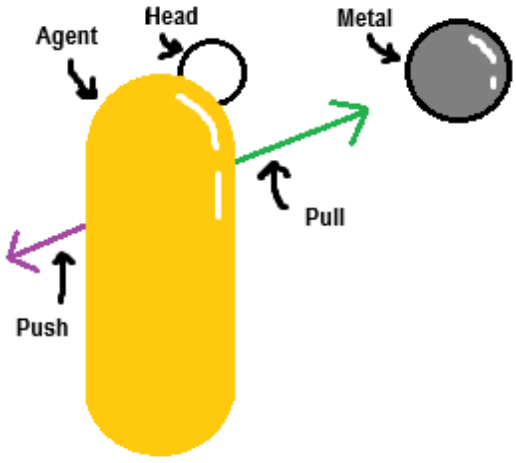



*Figure 3: The Agent visual scheme*

# Agent

Once we have described the problem we move on to describe the agent:

**<u>The observation Space of our agent is:</u>**

- Observations of the game state (5 stacked vector, for time continuity):
    - Velocity
    - If the agent is touching the ground
    - If the Agent is moving towards a metal
    - If the Agent is moving away from a metal
    - Absolute position of the Agent
    - Absolute position of the Goal
    - Absolute position of the Head of the Agent
    - Rotation of the Head of the Agent
- Ray Perception Sensor 3D object, for Agent sense:
    - Can detect Ground, Metal, Goal and Poso tags
    - Has a total of 70 rays in a 360º circle around it
    - 70 units of ray length

**The actions of our agent are:**

- Head rotation
    - Do not rotate head
    - Rotate clockwise
    - Rotate counterclockwise
- Movement
    - Do not move
    - Walk towards right
    - Walk towards left
    - Run towards right
    - Run towards left
- Jump
    - Do not jump
    - Jump
- Pull-push
    - Do not pull or push
    - Pull
    - Push

Note that pulling and pushing are actions that will only take effect if the head is pointing towards a metal object. When this is the case, the head pointer ball will change its colour to blue. If the agent were pulling towards metal it would change its colour to green and if pushing away from metal it would change to purple.

## Different Rewards

During the development of this project we have tried different reward strategies, which we have tested and each with its consequences. This approach of constantly changing reward was absolutely essential in the course of our training, as we relied a lot on reward updates and curriculum learning for the agent to act the desired way.

### Dense rewards

Taking into account that our goal is for the agent to cross the finish line as quickly as possible, we provided the agent with a very large positive reward if it reaches the goal (2000 base points). Additionally, we also gave it a positive reward inversely proportional to the distance from the agent to the goal, this way the closer it gets the more points it gains. On the other hand, we gave

a negative reward for every second the agent remained in the episode without solving it. This strategy, although initially seeming correct, posed significant problems for us: Firstly, even if the agent had discovered the goal and crossed it for the first time, in subsequent iterations, it did not go directly to the goal but instead stayed close to it, attempting to accumulate the maximum possible points (given for the distance) before the end of the training. In short, this type of training caused the agent to gather all possible points before entering the goal. This is the opposite of what we want, and although we could have tried to improve these reward sets, we thought it would be more convenient to completely reformulate a different way of rewarding the agent.

### Sparse rewards

One of the problems with the previous reward system was that rewards became uncontrolled and had extremely high or low values. In addition to this, the previous reward system failed to make the agent enter the goal as quickly as possible. That's why we decided to implement the following changes. Firstly, we would only provide rewards at the end of the episode, either because the agent ran out of time (it has 60 seconds to reach the end) or because it managed to cross the goal.

Furthermore, the way the reward is calculated depending on how the agent finished the episode is also different. If the agent runs out of time, the reward would be the sum of 1500 divided by the distance between the agent and the goal (this value has an associated multiplier that we tuned for the task we had in hand at each moment of the training), plus the reward for the use of movements. To calculate the reward for the use of movement, the agent starts with a value of 500. For each special action it takes (jumping, throwing, and pushing), one unit will be subtracted. If the agent runs out of time, the value of this variable will always be a minimum of 0.

On the other hand, if the agent manages to reach the goal, it will initially receive 2000 reward points, but we will deduct the seconds it took to reach the goal and add the reward for the use of movements. In this case, yes, the last value can be negative. The reason we have decided to calculate a different reward depending on whether the agent has reached the end of the episode or not is to avoid giving too large a negative reward in case the agent has not reached the goal but has simply discovered where it is and how to get there (exploratory phase). Once it has discovered the goal, we want it to try to reach it as quickly and as simply as possible (exploitative phase).

```
if (timerIsRunning)

{

    if (timeRemaining > 0)

    {

        // Resta el tiempo del temporizador en cada frame

        rewardTiempo = (timeReset - timeRemaining);

        timeRemaining -= Time.deltaTime;

    }

    else

    {
```

```
        // Recompensa exploratoria: si la reward por movimientos es
negativa, vamos a hacer que sea 0; a pesar de que eso vaya a hacer que
sea lo mismo hacer 500 acciones o 5000

        // durante la fase exploratoria del aprendizaje, nos interesa que
el agente llegue a la meta lo antes posible y que una vez descubra que
llegar a la meta es lo que tiene que hacer,

        // entonces empiece a pulir la forma en la que llega

        if (rewardMovimiento < 0)

        {

            rewardMovimiento = 0;

        }

        // Si el episodio se acaba por falta de tiempo, damos las rewards
que hemos mencionado anteriormente

        float distManhattanAlFinal = 1500 /
(Mathf.Abs(goal.transform.position.x - obj.transform.position.x) +
Mathf.Abs(goal.transform.position.y - obj.transform.position.y));

        float totalReward = rewardMovimiento + (distManhattanAlFinal);

        GiveReward(totalReward);

        Debug.Log($"Sin tiempo: dist:{distManhattanAlFinal}");

        Debug.Log($"Sin tiempo: movementPoints:{rewardMovimiento}");

        Debug.Log($"Total:{totalReward}");

        EndEpisode();

    }

}
```

*Code 4: Reward if time is ended*

```
    public void ReachedGoal()

    {

        if (rewardMovimiento < -500)

        {
```

```
            rewardMovimiento = -500;

        }

        float totalReward = rewardMovimiento - rewardTiempo;



        Debug.Log($"GOAL REACHED...");

        Debug.Log($"Basic reward: {2000f}");

        Debug.Log($"Movement points: {rewardMovimiento}");

        Debug.Log($"Time reward: {-rewardTiempo}");

        Debug.Log($"Total: {2000f + totalReward}");

        GiveReward(2000f + totalReward);



        EndEpisode();

    }
```

*Code 5: Reward goal reached*

Final Hybrid Version

We identified two behaviours that made us reconsider the way we provide rewards, especially negative ones. These behaviours were: a) the agent failed to understand that it shouldn't jump so much and was wasting the positive reward it could achieve by minimising the use of jumping, pushing, or pulling (we have the suspicion that this is because with an sparse reward at the end, the agent doesn't know the reason of this reward is the movements it has made), and b) the agent could end up in catastrophic forgetting when transitioning from one scenario to another, continuously getting stuck in a corner, as we will see in the level_0. To address these issues, we decided to change the way we rewarded the agent by providing some rewards in the middle of the episode.

The first change we implemented was giving a negative reward each time the agent jumped at the moment it did so. This way, the agent quickly learned to stop jumping when it was unnecessary. The second change we made was related to how we implemented the mechanism of pulling and pushing metal, which had some code errors we hadn't identified. Fixing them resulted in a positive domino effect, enabling the agent to better control its pulling and pushing, avoid wasting time interacting unnecessarily with metal balls, and improve interaction with new scenarios and levels, ultimately solving the catastrophic forgetting problem.

Once we addressed these issues, our agent increased its adaptation speed and reduced the time needed to adapt to new levels, resulting in significant rewards. However, this had a side effect: although our agent didn't have a controlled way to move in the depth direction, the Z-axis, we observed that it appeared to be doing so.

Additionally, we noticed that the agent was jumping much higher than it should. Initially, we thought this might be caused by its abilities to pull or push metal, but upon inspecting that in detail, we realised it wasn't pulling or pushing at all. We deduced that our agent had managed to exploit the physics of our project to achieve higher rewards. As this was undesirable since the agent needed to "follow the rules," we implemented the following code, providing a negative reward and ending the episode if the agent escaped from its plane. Consequently, during training, the agent not only adapted to the level normally but also learned not to cheat in order to maximise its gains.

```
// Corregir la posición en el eje Z si es diferente de cero

if (obj.transform.position.z != 0)

{

            Vector3 correctedPosition = new
Vector3(obj.transform.position.x, obj.transform.position.y, 0);

            obj.transform.position = correctedPosition;

            Debug.Log("Se ha movido en Z");

            // Dar una recompensa negativa de -1500

            GiveReward(-1500f);

            EndEpisode();

        }
```

*Code 6: Preventing the agent from cheating*

# Resolution of the Environment

## Curriculum Learning

To ensure that our agent achieves its objective, we arrived at the conclusion that the curriculum learning technique is the best way to teach our agent what it needs to do. Therefore, we have constructed four levels of increasing difficulty that teach the model different skills. Below is a description of the map, the difficulties, and the agent's final performance before moving on to the next map.

## Level 0

Level 0 aims for the agent to become familiar with its actions, the environment, and its components, such as the goal, the floor, and metal anchors, etc. As the most simple level possible, it consists of a floor, a wall for the agent not to fall, the goal and a metal anchor. Although the

agent takes no more than 10 minutes to find the goal and plan optimal routes to it, this is not the case when we introduce the metal anchor from the beginning. We have observed that the agent tends to inspect areas with solid objects that raycasts collide with before exploring open areas. However, especially with metal anchors, the agent spends a significant amount of time pulling and pushing them, even though it doesn't bring him closer to the goal.

Here is where we realised the importance of curriculum learning and going little by little teaching the agent how it should behave.

Once the agent mastered this scenario, we introduced the anchor and hope it learnt to use it. However, the agent was taking too much of a long time for this to happen. In fact, even though the agent had learnt to reach the goal optimally, adding just a single metal anchor almost discarded all the previously acquired knowledge. The agent spent too much time and actions attracting and pulling the metal, as it can be seen in this video. This was really because of the bad implementation of the pull & push mechanics on the agent script. In view of this, we decided to keep training the agent without the metal anchor, and later we tackled the problem by adding **Level 3**, a level to test and debug the implementation of pulling and pushing.

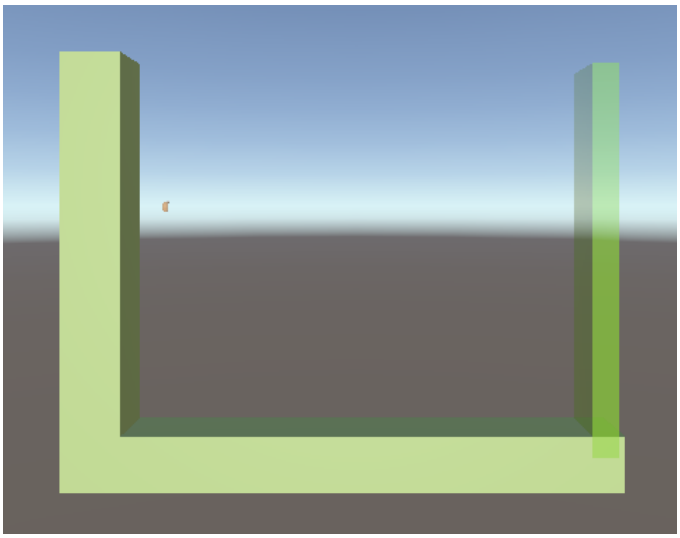
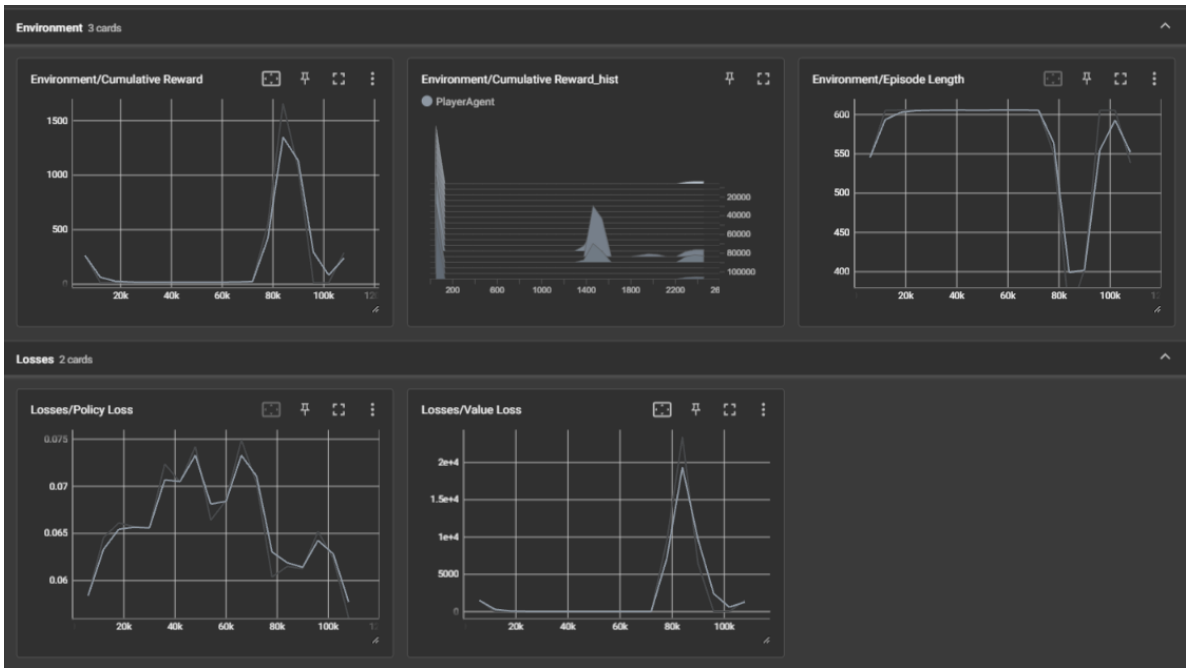


*Figure 4: Level 0.0*

This was also the level where we faced the catastrophic forgetting problem. We had trained the agent to achieve the goal, but after a bit it just forgot everything it had learnt and started going left against the wall. As we can see on the TensorBoard, the reward had increased because of reaching the goal, and suddenly it just drops again, due to the catastrophic forgetting. The reverse case can be observed in the episode length, where it was improving its time by reaching the goal and suddenly it didn’t even reach.




At this moment our agent just jumped all the time around as we can see on this video, but we needed it to understand what jumping is, thus we added a little step so it needed to jump over it.

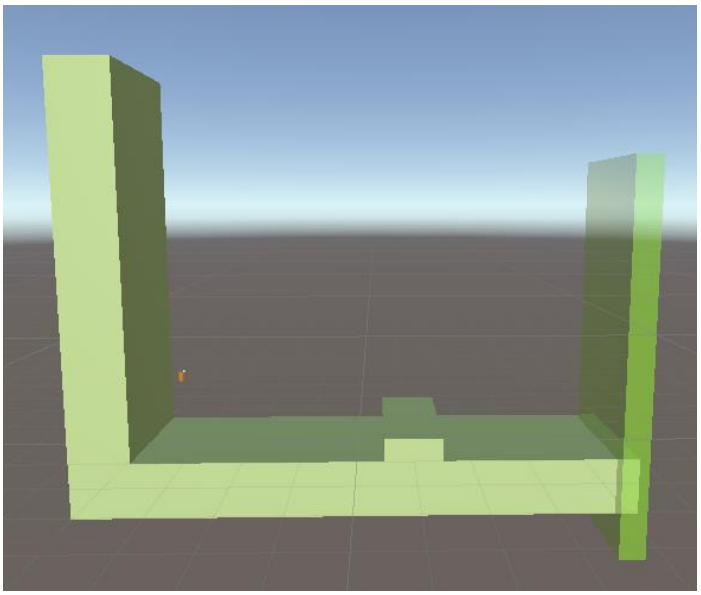
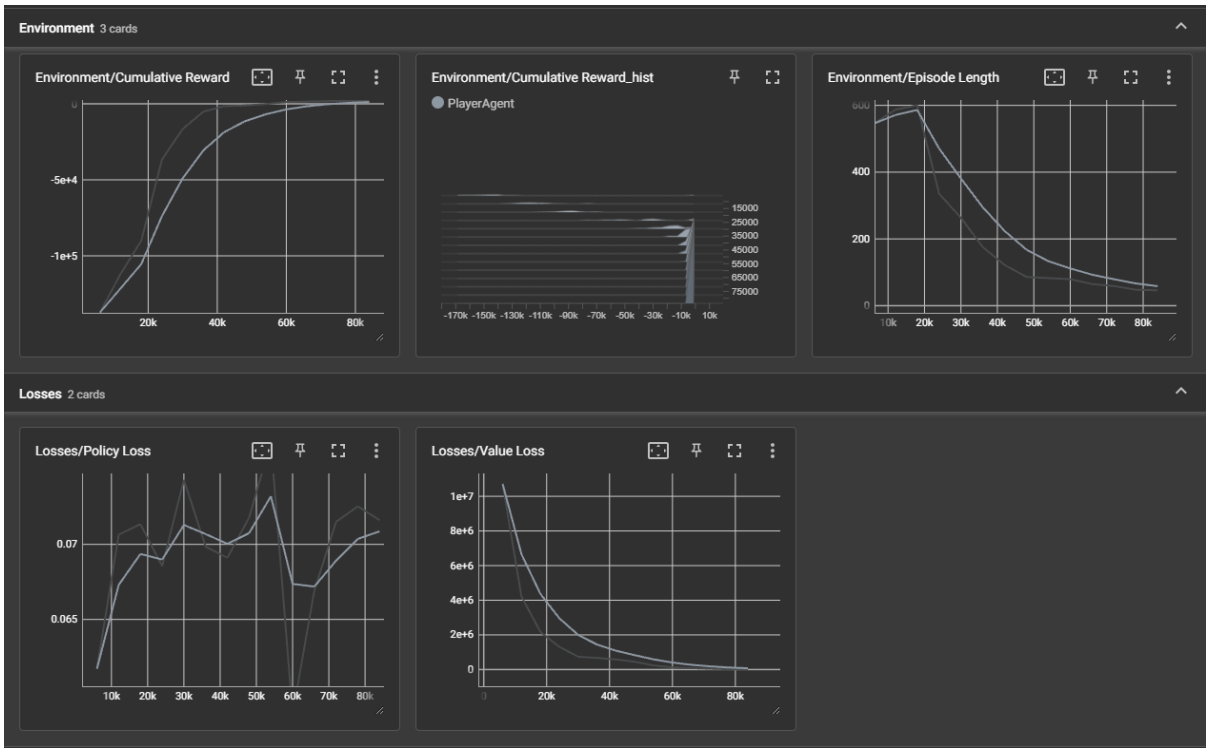


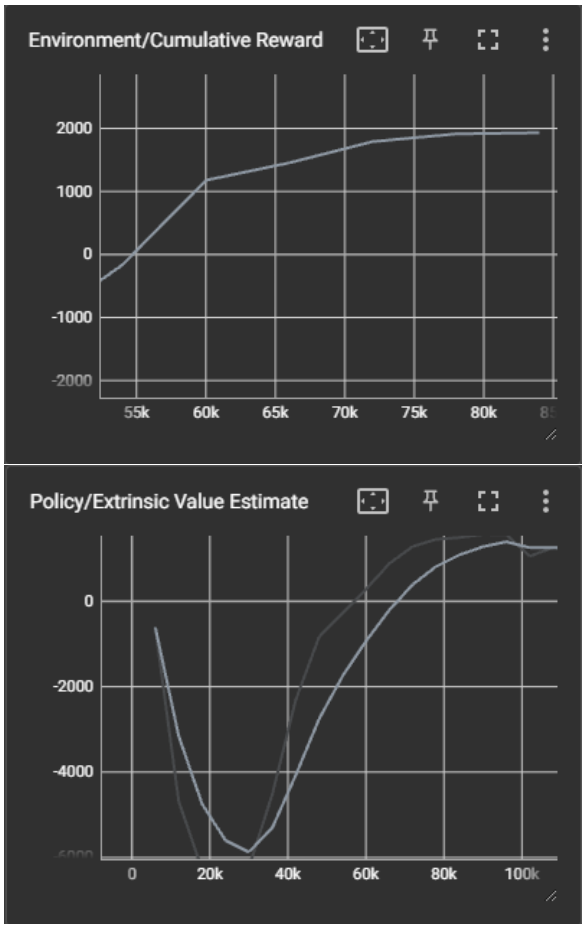
*Figure 5: Level 0.2*

At first it just jumped all the time and this was certainly a valid solution, however for it to understand that jumping is a certain action with consequences and value we penalised it whenever it jumped. Thanks to this methodology it learnt to jump only when necessary, and understood that jumping is an action it should take to overcome an obstacle, as seen in this video.

We will now analyse the TensorBoard information at this moment of the agent's training process. In this step of the evolution it won't be necessary, but later on we will see that the huge scale of the negative rewards (jumping at level 0.2) doesn't let us see with clarity the results, so we will also put a close up to the last part of the training.




Here we can see how after learning not to jump all the time the model finds the way to the goal and settles on a near optimum reward behaviour, thus reducing its episode length by a lot. We can have a closer look as well, at the reward and the value estimate, which both reflect this change. The reward increases as the agent learns not to jump, and in consequence its value estimate (the reward the agent expects to obtain) increases as well, recovering from the hole it had fallen into due to the implementation of the negative reward.


Environment/Cumulative Reward
2000
1000
0
-1000
-2000
55k
60k
65k
70k
75k
80k
Policy/Extrinsic Value Estimate
0
-2000
-4000
0
20k
40k
60k
80k
100k

Level 1

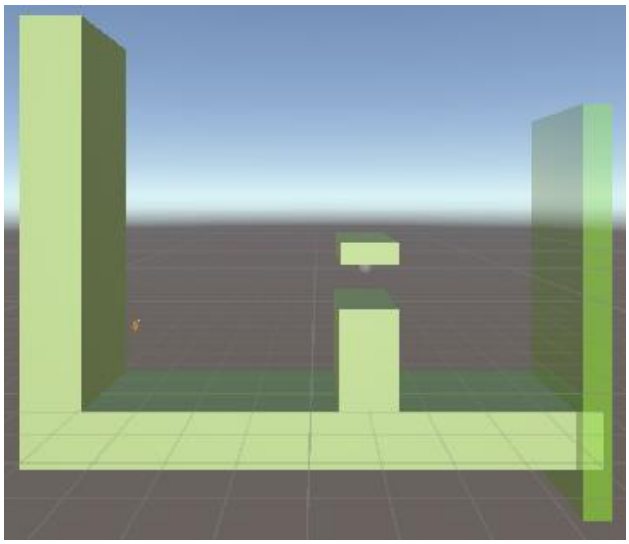


*Figure 6: Level 1*

Once the agent has become familiar with the environment at its most basic form, we make it face this new level. In this level, the agent must traverse the space between two blocks that it cannot jump over. This level aims to test the agent's ability to generalise to new spaces and demonstrates the effectiveness of curriculum learning. In other words, the agent should be able to adapt quickly to slightly modified new levels. Additionally, this new level focuses more on the actions related to the movement of the agent rather than its ability to pull and push.

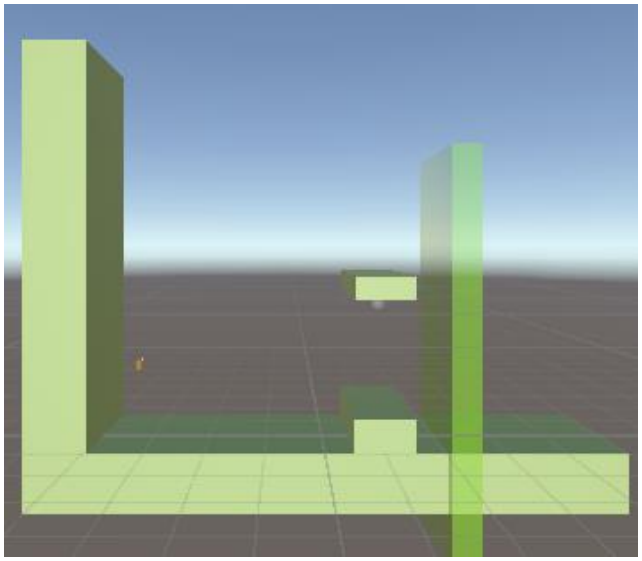


*Figure 7: The goal is nearer to the agent*

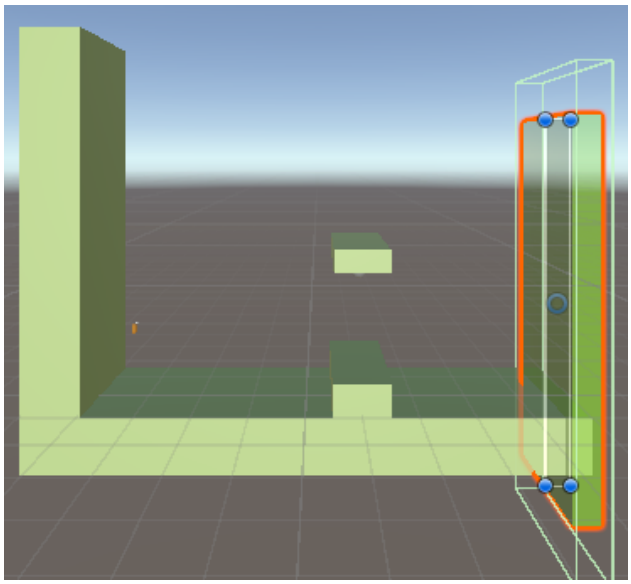
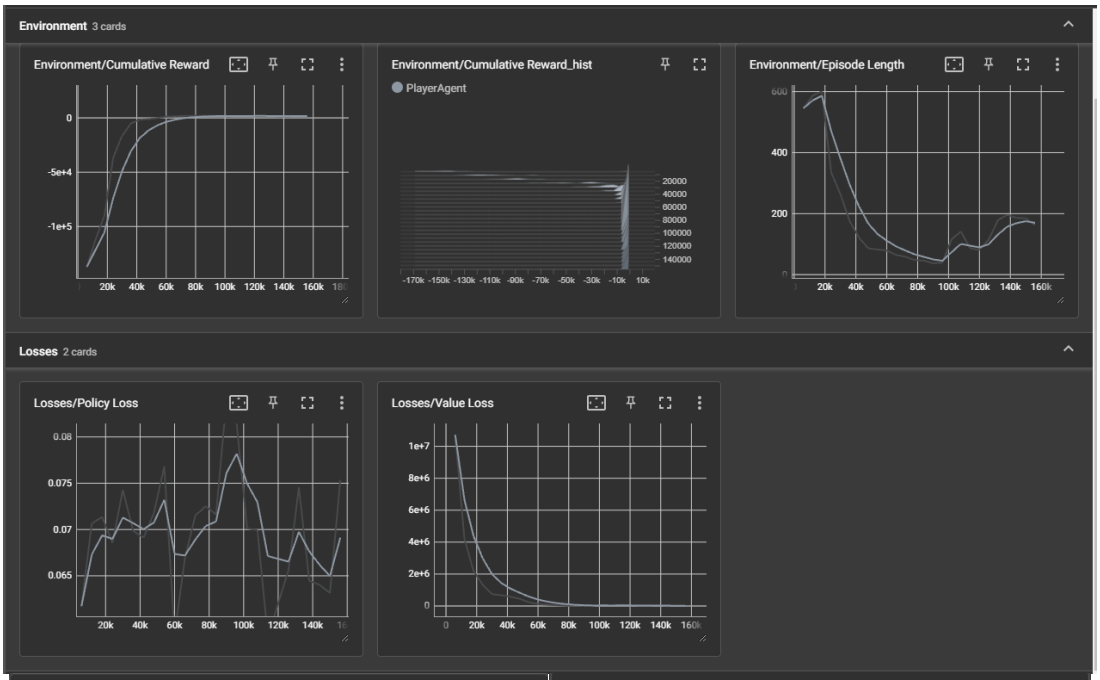
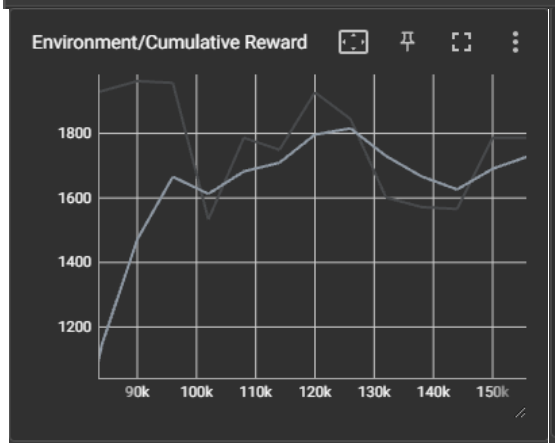


*Figure 8: At its original position*

This map was a turning point in our model's training. Initially, we attempted to leave it on its own for an extended period to see if it could figure out a solution, but it couldn't progress beyond the tower with the ball. After discarding that approach, we conceived a more nuanced curriculum learning strategy and decided to lower the height of the tower so that the agent could jump over it and place the goal next to it. While the agent had already learned what the goal was and how it functioned in the previous map, it seemed evident that the acquired knowledge was not sufficient to pass this level. Now, right after jumping over the tower, the agent could see the goal and indeed managed to reach it without any issues. Once the agent mastered this situation, we placed the goal back in its initial position, where the bot could no longer see it. After a few episodes, it successfully solved the level. Finally, we restored the tower to its original height, and after a few attempts, it completely solved the level. After this experience, it became apparent that fine-tuning curriculum learning is more effective than investing a significant amount of time training it in a scenario it clearly cannot resolve. This is because the total time spent resolving each stage of this level is not even half the time we invested in having it solve it for the first time. We can see how the agent finally (after correctly implementing the pull and push actions) managed to complete this level in this video.

We can see the tensorboard results, where we see a little increase in the episode length, this is just because the level itself takes longer to finish than level 0, and taking a close look at the reward we can see it remains very close to the optimum score. We can also see the tallest peak in the policy loss graph (and it will keep being the tallest point until the end of the training). We think this might be the product of learning how to pull from an object correctly, as it is a newly introduced mechanic of complex nature and really defining in our environment.


Environment 3 cards
Environment/Cumulative Reward
0
-5e+4
-1e+5
20k 40k 60k 80k 100k 120k 140k 160k
Environment/Cumulative Reward_hist
PlayerAgent
20000
40000
60000
80000
100000
120000
140000
-170k -150k -130k -110k -90k -70k -50k -30k -10k 10k
Environment/Episode Length
600
400
200
0
20k 40k 60k 80k 100k 120k 140k 160k
Losses 2 cards
Losses/Policy Loss
0.08
0.075
0.07
0.065
20k 40k 60k 80k 100k 120k 140k
Losses/Value Loss
1e+7
8e+6
6e+6
4e+6
2e+6
0
0 20k 40k 60k 80k 100k 120k 140k 160k


Environment/Cumulative Reward
1800
1600
1400
1200
90k 100k 110k 120k 130k 140k 150k


Losses/Policy Loss
0.08
0.075
0.07
0.065
0.06
20k 40k 60k 80k 100k 120k 140k 160k

## Level 2

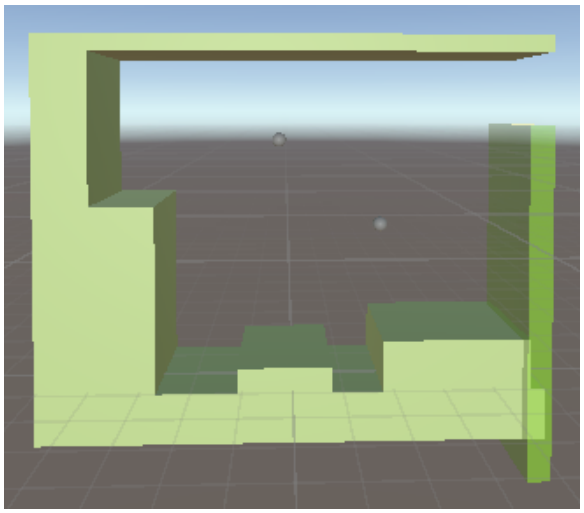


*Figure 9: Level 2*

Once the agent has learned to move and gained fluency in the "learning" levels, it would progress to this new level where we expect it to take more time due to its increased difficulty. We anticipate interesting results stemming from its design. Firstly, there are two metal anchors strategically positioned to serve as the optimal solution. If the agent has the required ability, it could pull one and then the other (or pull one and subsequently push) to quickly reach the goal without taking the longer path. However, especially at the beginning, we expect the agent to drop and have to climb the two boxes in front of it.

The agent can jump to climb the first box, but to ascend the second one, it must pull the second metal ball. This map aims to test both the agent's "normal" and "magical" abilities, as well as its perception and its capacity to find the optimal route independently of what it has learned so far. In other words, we don't expect the agent to replicate exactly what it learned in the previous scenarios but rather to generate a global vision and deductive ability to devise the optimal route and how to navigate it.

Therefore, this level is crucial in the model's learning trajectory, and we anticipate that it will require much more time than the previous levels. However, we hope that this extended learning time will result in an unprecedented increase in its skills.

Finally after some learning time, the agent managed to solve the level, most of the times it ends taking the purple path shown below, which is the slowest one, however sometimes it takes a path really similar to the green one, which demonstrates it know that path is faster, but the agent isn’t skilled enough in this part of the training to accomplish a 100% accuracy when using metal anchors. This video demonstrates the mentioned behaviour.

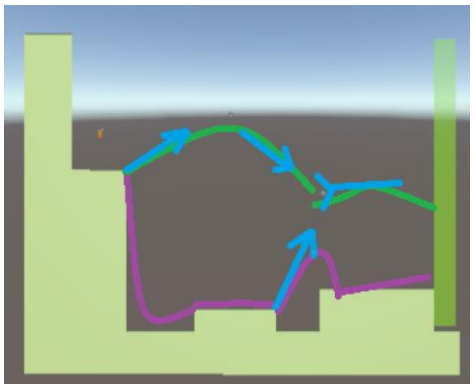
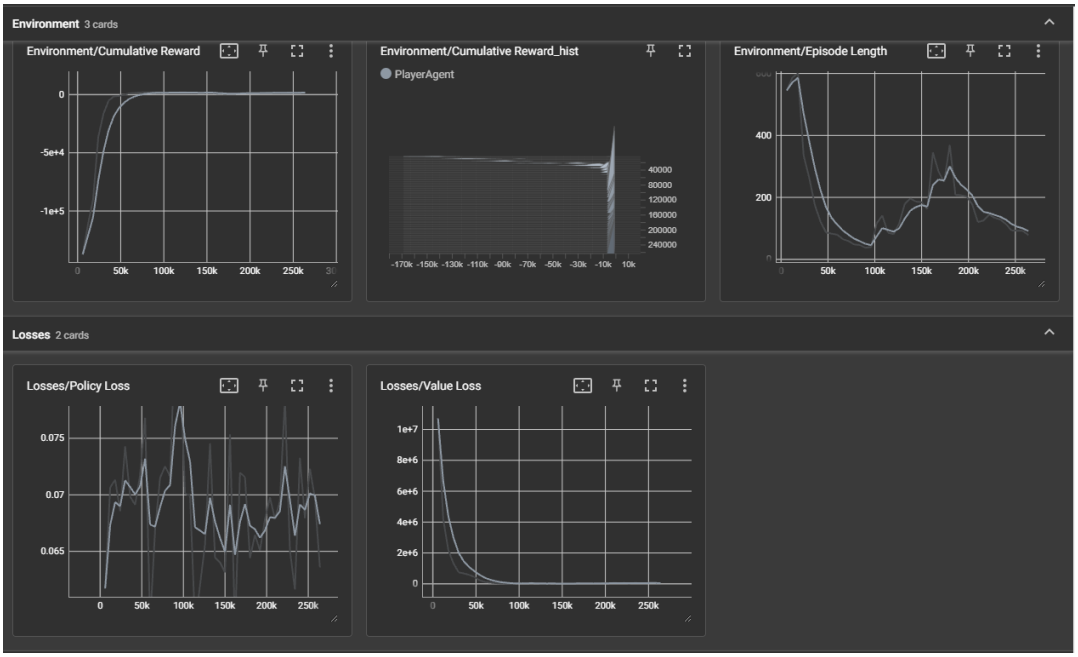
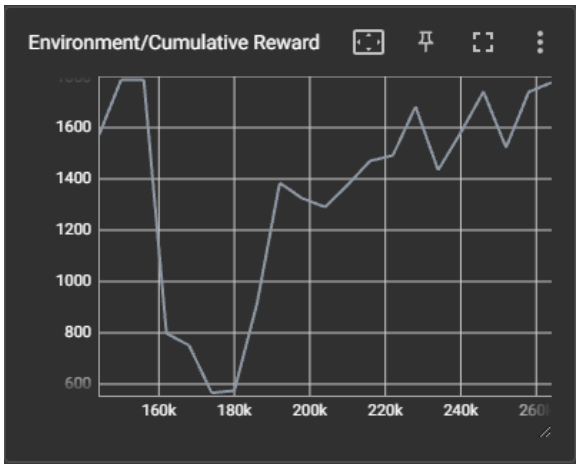


*Figure 10: Possible solutions to level 2 scheme*

*Figure 4: Ideal representation of the two main trajectories to pass this level. In green, the fastest one: the agent would have to pull the first anchor, pull to move to the second one, and push the last one to reach the goal. In purple, the slower trajectory. Notably, it would have to pull the second anchor to reach the top of the second box.*

When analysing the tensorboard for this level's training process, after the level 1 it is clear there is a drop in the reward and a rise in the episode length, both of these are due to the curriculum learning process, the agent when facing level 2 with its knowledge doesn't finish the level, but after some episodes it learns how to overcome the complexities of this challenge, resulting in a recovery of the reward value and a descent on the episode length.




Environment/Cumulative Reward
1600
1400
1200
1000
800
600
160k
180k
200k
220k
240k

## Level 3

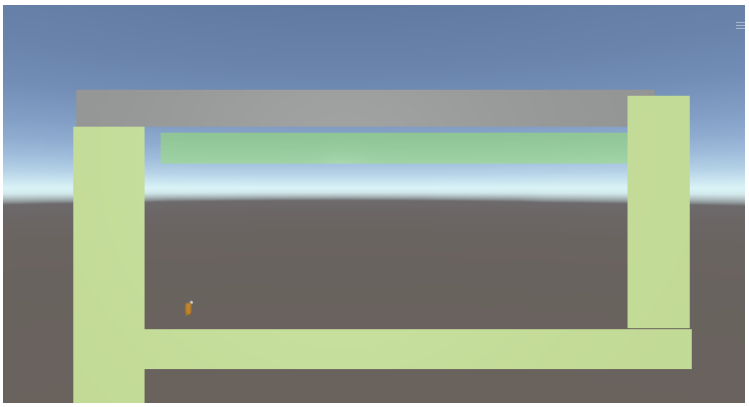


*Figure 11: Level 3.1*

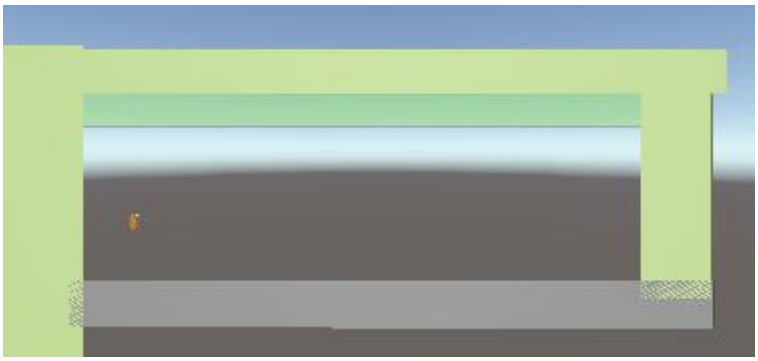
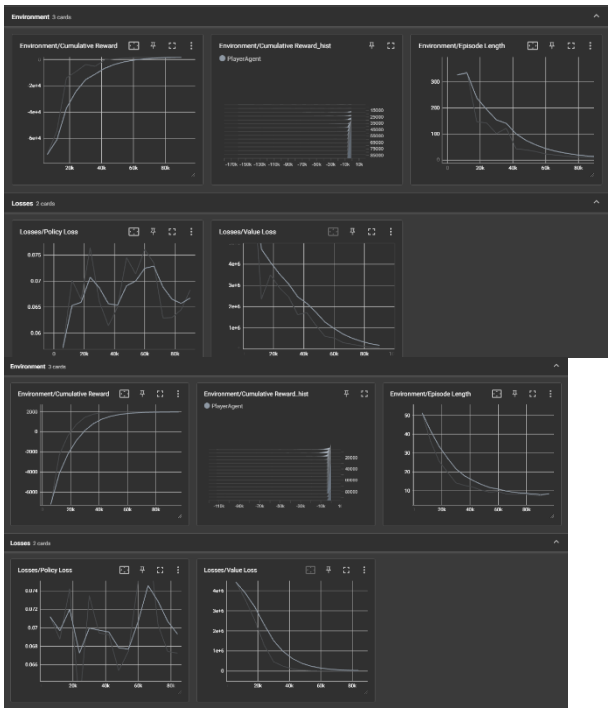


*Figure 12: Level 3.2*

This level is a standalone one, and was not part of our curriculum learning process, it can be considered as a debugging and proof of concept level. It aims to test and train our model's understanding of its pulling and pushing abilities, we designed it due to the constant failing of the agent to correctly use the metal anchors, which in the end turned out to be an implementation issue. For the first time, the goal is positioned at the top of the map, not on the right. The agent has no pull distance limit, but the goal is situated at a height where the agent can only reach if it pushes from the metal floor at almost a perpendicular angle (placed near the maximum distance the agent can push). However, it is positioned low enough for the agent's raycasts to see the goal. This means that if the agent has understood correctly, it should be able to see the goal and quickly deduce what it needs to do. At the beginning, due to implementation issues it didn't work, but after fixing the code and training for a bit it came clear that the agent was perfectly capable of pulling and pushing metal while understanding its consequences, and thanks to it we were able to train the agent correctly in the difficult levels, where metal pulling and pushing are involved.

Here we can see how it pulls towards metal and pushes away from it to get to the goal. And the TensorBoard shows us the training process, which, when well implemented, wasn't difficult at all.




This is the final implementation of the agent's pull and push actions, where we can see they are not trivial, since their effect changes depending on some variables, but we believe that the 5 stacked vectors allowing for time continuity made it possible for the agent to learn their behaviour.

```
        //Tirar y Empujar Metal: Numero 3
        int pull_push = (int)vectorAction[3];
        if (pull_push == 0)
        {
            //Debug.Log("NOT Metal PULL or PUSH");
            if (_playerController.movingTowardsMetal)
            {
                _playerController.MoveTowardsMetal(1);
            } else if
(_playerController.movingTowardsMetalInverse)
            {
                _playerController.MoveTowardsMetalInverse(1);
            }
```

```
            _playerController.CastRay(0); // No Tirar ni empujar
del Metal


        }
        else if (pull_push == 1)
        {
            //Debug.Log(" ---> PULL");
            //GiveReward(-1000f);
            pulled++;
            if (_playerController.movingTowardsMetal)
            {
                _playerController.MoveTowardsMetal(0);
            } else if(_playerController.movingTowardsMetalInverse)
            {
                _playerController.MoveTowardsMetalInverse(1);


            }
            _playerController.CastRay(1); // Tirar del Metal


        }
        else if (pull_push == 2)
        {
            //Debug.Log(" <--- PUSH");
            //GiveReward(-1000f);
            pushed++;
            if (_playerController.movingTowardsMetalInverse)
            {
```

```
                _playerController.MoveTowardsMetalInverse(0);
            }
            else if(_playerController.movingTowardsMetal)
            {
                _playerController.MoveTowardsMetal(1);


            }
            _playerController.CastRay(-1); // Empujar del Metal
        }
```

*Code 7: Implementation of pull and push actions*

## Level 4

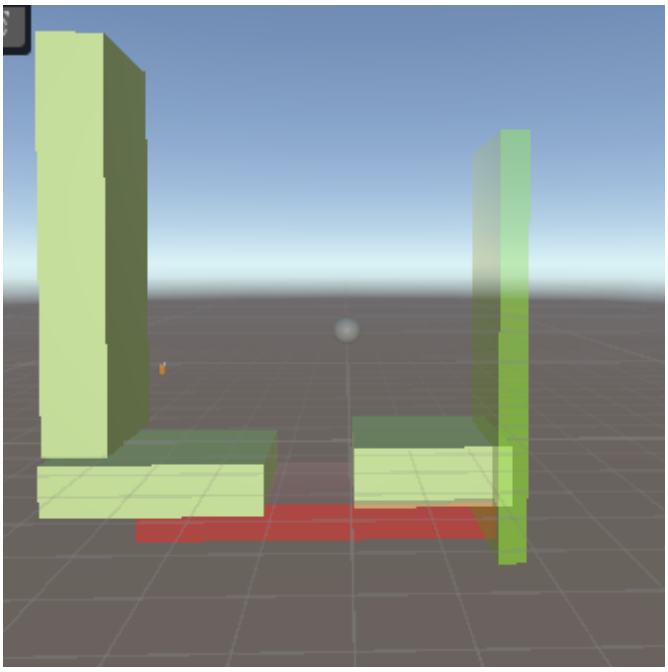


*Figure 13: Level 4.1*

We first designed this level to use only a metal ball, but after fixing the pull/push action's code, it was just too simple, so we decided to add a second ball, and see if the agent could learn to transition between two balls.

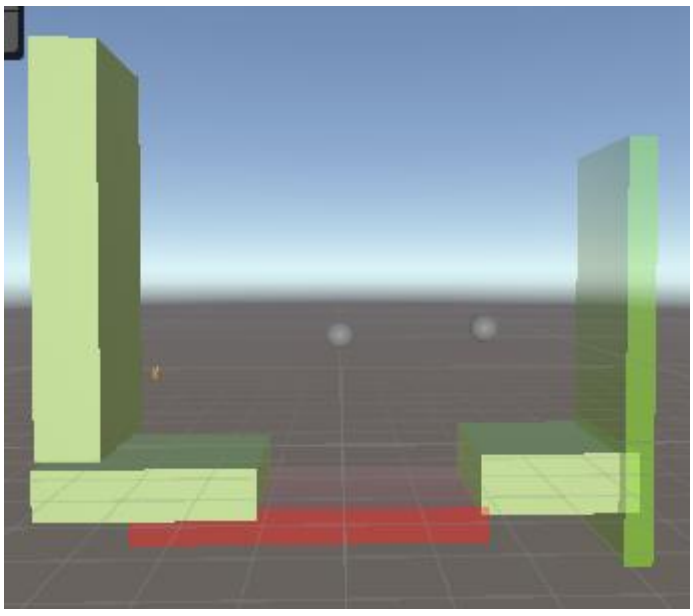


*Figure 14: Level 4.2*

As you can see, this level also introduces a new mechanic that the agent must confront: the pit. If the agent collides with the object with the reddish material, it will "die". In other words, we will apply a set of rewards different from those applied if the agent runs out of time or reaches the goal. Additionally, a new episode begins after this event. As we do not want to instil "fear" in the agent, our negative rewards will not be too severe. "Fear" is what we call the consequence of the

agent deciding to stop exploring near the location where it received a very low negative reward, preventing it from ever finding the goal. This is something we discovered in the first iteration and want to avoid in this one. Hence, we give it a negative reward in the following way:

```
public void died()
{
    Debug.Log("x_x");
    Debug.Log($"Total reward: {0f}");
    GiveReward(0f);
    EndEpisode();

}
```

*Code 8: Reward if dead*

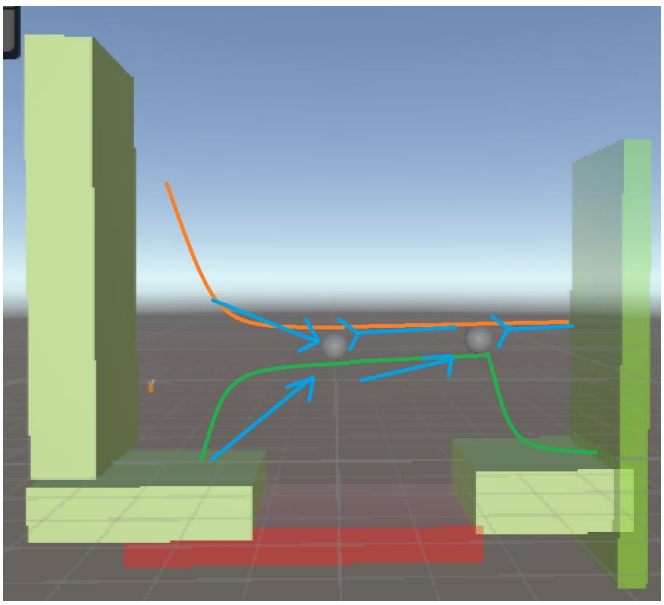


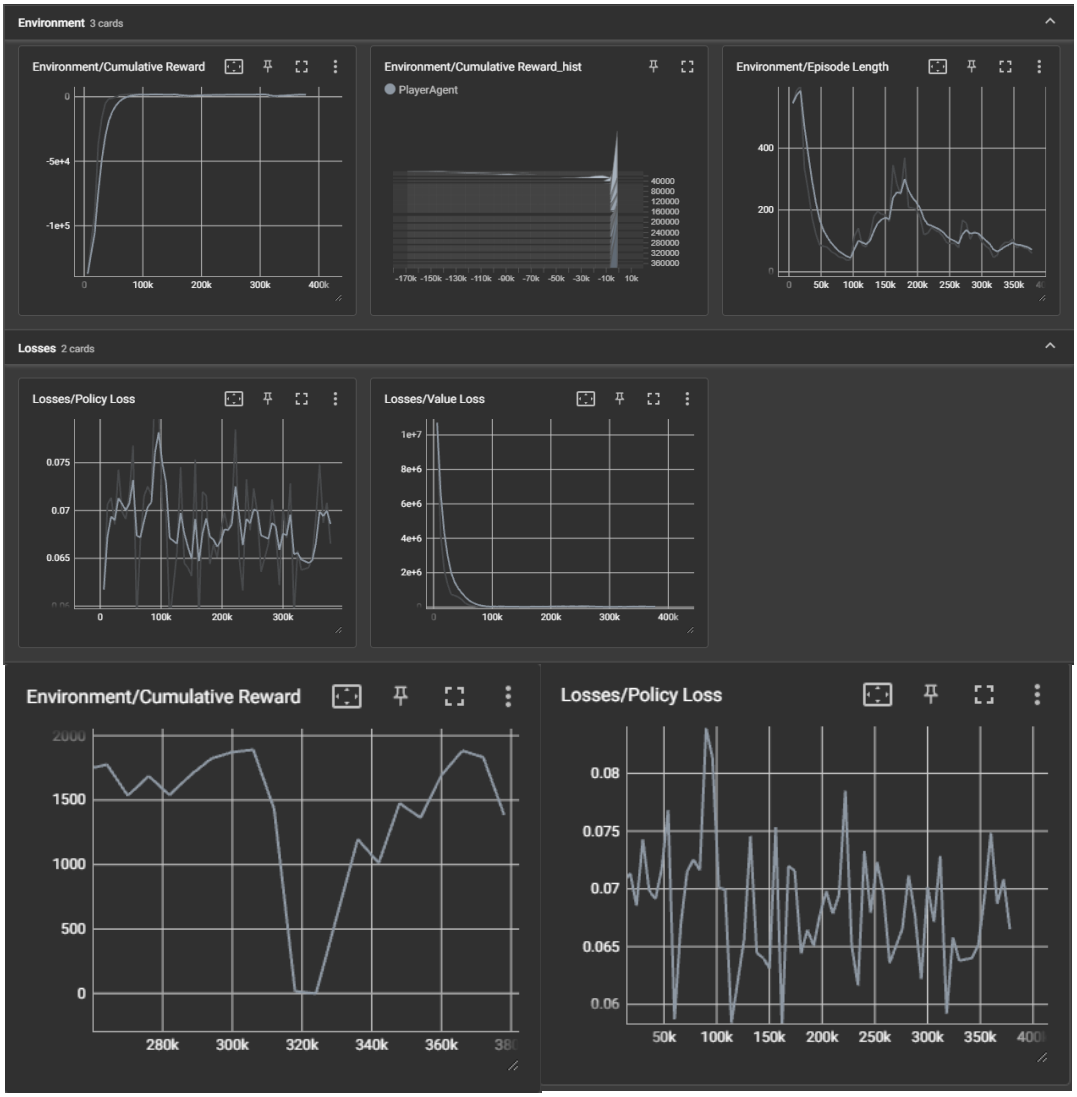
*Figure 15: Expected (green) vs agent's trajectory (orange)*

Before training the agent on this level, we expected the agent to take the green path, just fall to the ground and cross the pit by using the balls to pull itself upwards, leaving him at the other side. However we were surprised to see the agent had found a better path.

After failing many of its attempts to cross the pit, the agent successfully learnt, not one, but two paths to the other side. If it was skilled enough to turn its head in time (which we didn't expect it to be in the beginning) it would go through the upper part of the metal ballas, by pulling and pushing them, which is the fastest path possible. However, if, in contrast, it ended up falling to the

ground, it didn't give up, it learnt to pull the balls from beneath consecutively or by pulling directly the second one, and managed to cross the pit, even if this was a little worse of a path and the one we had in mind when designing the level. Both of these paths are shown on this video, which is the final version of our agent.

If we look closely at the final TensorBoard of level 4, we can see how it follows the pattern of level 2, the pattern of curriculum learning. However this time the decrease in the reward is bigger, as it reaches a reward of zero (falling to the pit), while the episode length increase is lower, probably because dying takes little time in opposition to not being able to finish a level. We can also see how Policy Loss follows a downwards trend, as the agent's behaviour is more and more defined and suffers less impactful updates.




# Conclusion

In conclusion, we think this project has shown interesting results, and the agent was able to complete all the levels proposed in a correct way. We have realised the importance of step by step learning and the power of curriculum learning. We have also learnt to interpret some tensorboard graphs we didn't know before, such as Policy Loss and Value Estimate. Finally we have addressed the importance of having a well configured environment, with correctly implemented features, and how a balance between dense and sparse rewards effectively helps the agent learn.

# Bibliography

How to install Unity ML Agents Release 20 in 2023