Controlling Video Game using Reinforcement Learning

Christos Yiannakou

June 1, 2021

University Of Cyprus

COMPUTER SCIENCE DEPARTMENT

UNIVERSITY OF CYPRUS COMPUTER SCIENCE DEPARTMENT

Abstract

On this thesis we Control virtual characters by using Reinforcement learning. For this project we recreate a retro game the Bomber Man using Unity game engine and the machine learning library of unity ML-Agents to train the agent of the game and observe the strategies that use to complied the game tasks

Content

6 Chapter 6 : Design And Implementation

6.1 Stage

6.1.1 Prefabs

6.1.2 Stage Implementation

6.1.3 Stage Generator

6.1.4 Player And Enemy Placement

6.2 Bomb

6.2.1 Bomb Asset

6.2.2 Explosion Asset

6.2.3 Explosion Representation

6.3 Agent

6.3.1 Agent Asset

7 Chapter 7 : Training And Results

8 Chapter 8 : Conclusion

9 Chapter 9: Bibliography

Chapter 1 : Introduction

The topic of Machine Learning and Artificial Intelligence in Computer Science was always something that pique interest to the fans of technology and the thought that one day computers would be better than humans is something that scares people but is also exciting as an idea.

The reason I choose ”Controlling Virtual Characters using Reinforcement Learning and Examples” as a theme for my thesis is that I find very interesting the fact that a machine can develop by its ”own” intelligence and decide what action is best in every situation according to its training and its in-time observations from the environment in which it is located.

The first time that humanity created such an artificial intelligence that can beat a human in a game was the Deep Blue chess computer developed by IBM to challenge the then world champion at chess Garry Kasparov and in one thousand nine hundred ninety-six in a six-game of chess, the Deep Blue won the match.

It's fascinating that computers actually can be better than humans and that's inspired me to deal with reinforcement learning and create a game that the computer can train with and observe its actions and strategies that are used to complete the tasks of the game.

To accomplish the dissertation I use Unity 3D to create the game Bomber-Man and the library of Unity ML-Agents to attach Reinforcement Learning in the game so the agent can train and learn behaviors to accomplish the game tasks. I choose Unity because is a user friendly game engine and is easy to use and learn it. Al-sow has lots of free assets in the assets store and lots of packages and library's that helps me to develop the game and to run the training on.

The challenges that I have to go through are the observations that parse in the neural network and the reward's the agent receives which is the most challenging part of reinforcement learning because the training and the agent's next actions depend on rewards.

Reinforcement Learning

Reinforcement Learning is a type of machine learning technique that enables an agent to learn in an interactive environment by trial and error using feedback from its actions and experiences.
According to Sutton and Bardon (1998) reinforcement learning is a merge of trial and error law-of-effect tradition in psychology.
The agent isn't told which actions to take or the optimal way of performing a task. Reinforcement Learning uses rewards and penalties to signal whether a taken action is good or bad.

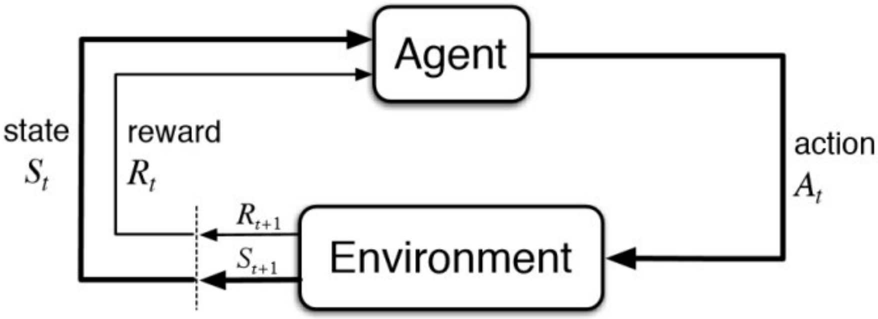


Figure 1: Reinforcement Learning explanation image

The image above show's the interaction between the agent and the environment. The agent takes an action that parses into the environment and the environment sent to the agent as input the state S_{t+1} that the agent is in time t+1 according to its previous action and a reward R_{t+1} for the action that it takes.

2.1 Markov Decision Processes

A reinforcement learning task that satisfies Markov properly is a Markov Decision Processes. If the state and the action spaces are finite then its called a finite Markov Decision Processes.

Finite Markov Decision Processes are practically significant in the theory of reinforcement learning.

A particular finite Markov Decision Processes is determined by its state and action sets and by the one-step dynamics of the environment. Given any state s and action a the probability of each possible next state s' is

$$P^{a}_{ss'} = Er_t + 1|st = s, at = a, st + 1 = s'$$

These quantities are called transition probability's. Similarly, given any current state and action s and a together with any next state s', the expected value of the next reward is

$$R^{a}_{ss'} = Er_t + 1|st = s, at = a, st + 1 = s'$$

These quantities specify the most important aspects of the dynamics of a finite Markov Decision Processes.

2.2 Taxonomy Of Reinforcement Learning Algorithms

Model-Free vs Model-Based

One of the most important branching points in Reinforcement Learning of the question of whether the agent has access to a model of the environment. By a model of the environment, mean a function that predicts state transitions and rewards. The upside to having a model is that it allows the agent to plan by thinking ahead seeing what would happen for a range of possible choices, and explicitly deicing between options. Agents can distill the results from planning

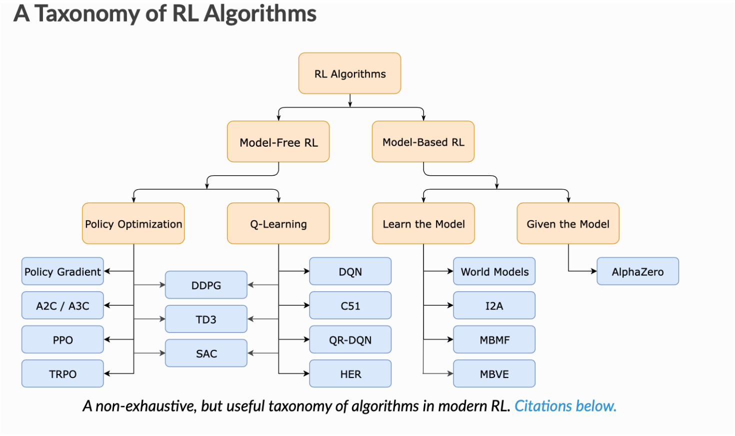


Figure 2: Reinforcement Learning Taxonomy

ahead into a learned policy. a particular example is Alpha zero. The main downside is that a ground-truth model of the environment is usually not available to the agent. If the agent wants to use a model in this case, it has to learn the model purely from experience, which creates several changes. The biggest challenge is that bias in the model can be exploited by the agent, resulting in an agent which performs well with respect to the learned model, but behaves sub-optimally in the real environment willing to throw lots of time to compute at it can fail to pay off. Algorithms that use a model are called model-based methods. Those that don't use a model are called model-free methods. While model-free methods forego the potential gains in sample efficiency from using a model, they tend to be easier to implement and tune.

2.3 Q-Learning

The main idea in Q-Learning is to acknowledge or learn the optimal action in every state visited by the system (optimal policy) via trial and error.
Trial and error mechanism can be implemented within the real-world system or a simulator.
The agent chooses an action, acquire feedback for that action, and uses the feedback to update its database.

In its database, the agent stores a Q-factor for every state-action pair. When the feedback for selecting an action in a state is positive, the associated Q-factor's value increases, while the feedback is negative the value is decrease.

$$Q(s_t, a_t) = max(rewardst + 1)$$

The best possible score at the end of a game after performing a in state s. Bellman's Equation

$$Q(s, a) = r + ymax(Q(s', a'))$$

The maximum feature reward for this state and actions is the immediate reward plus the maximum feature reward for the next state.

2.4 PPO

There are two primary variants of PPO, the PPO-Penalty and the PPO clip. The PPO-Penalty approximately solves a KL-constrained update like TRPO but penalties the KL-divergence in the objective function instead of making it a hard constrain, and automatically adjust the penalty coefficient over the cause of training so that is scaled appropriately. The PPO-clip doesn't have a KL-divergence term in the objective function and doesn't have a constraint at all. Instead relies on specialized clipping in the objective function to remove incentives for the new policy to get far from the old policy. PPO is an on-policy algorithm that can be used for environments with either discrete or continuous action spaces. PPO updates via

$$\theta_{k+1} = \arg\max_{\theta} s, a \sim \pi_{\theta_k} \mathrm{E}\left[L(s, a, \theta_k, \theta)\right],$$

typically taking multiple steps SGD to maximize the objective. The L on the above equation is given by

$$L(s, a, \theta_k, \theta) = \min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s, a), \;\; clip\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, 1 - \epsilon, 1 + \epsilon\right) A^{\pi_{\theta_k}}(s, a)\right)$$

in which is a hyperparameter which roughly says how far the new policy is allowed to go from old. There is a more simplify equation that is used to implement in codes

$$L(s, a, \theta_k, \theta) = \min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s, a), \quad g(\epsilon, A^{\pi_{\theta_k}}(s, a))\right)$$

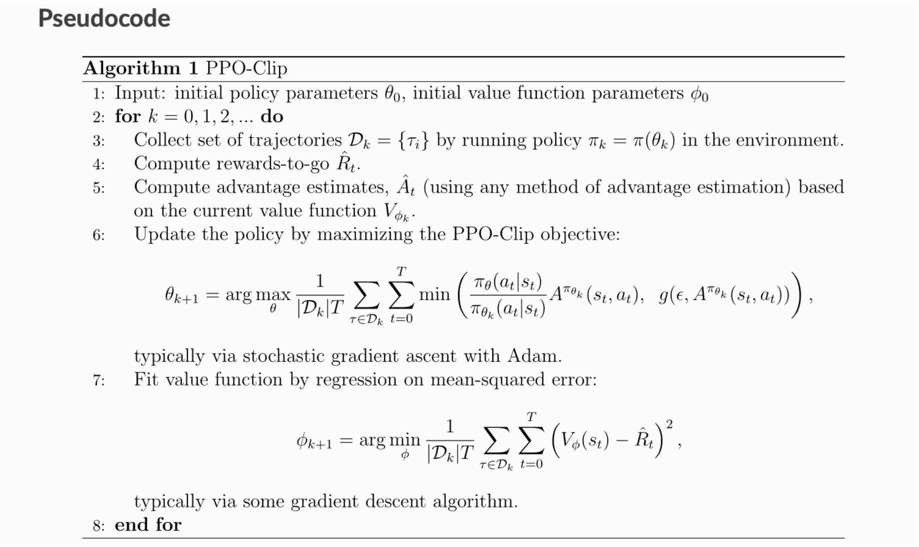
where

$$g(\epsilon, A) = \begin{cases} (1+\epsilon)A & A \geq 0 \\ (1-\epsilon)A & A < 0. \end{cases}$$

To figure out what intuition to take away from this, let's look at a single state-action pair (s,a), and think of cases.

2.5 Exploration vs Exploitation

PPO trains a stochastic policy in an on-policy way. This means that it explores by sampling actions according to the latest version of its stochastic policy. The amount of randomness in action selection depends on both initial conditions and the training procedure. Over the course of training, the policy typically becomes progressively less random, as the update rule encourages it to exploit rewards that it has already found. This may cause the policy to get trapped in local optima.



Pseudocode

```
Algorithm 1 PPO-Clip
Input: initial policy parameters $\theta_0$, initial value function parameters $\phi_0$
for $k = 0, 1, 2, ...$ do
    Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
    Compute rewards-to-go $\hat{R}_t$.
    Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based
    on the current value function $V_{\phi_k}$.
    Update the policy by maximizing the PPO-Clip objective:
        $\theta_{k+1} = \arg\max_{\theta} \frac{1}{|\mathcal{D}_k| T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min\left( \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t), \;\; g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t)) \right),$
    typically via stochastic gradient ascent with Adam.
    Fit value function by regression on mean-squared error:
        $\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k| T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left( V_\phi(s_t) - \hat{R}_t \right)^2,$
    typically via some gradient descent algorithm.
end for
```

Figure 3: PPO-clip Algorithm

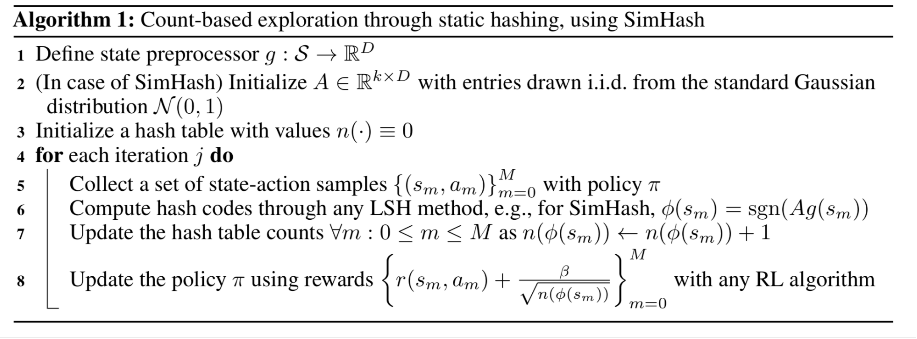
Chapter3: Related Work For Reinforcement Learning

Games such as chess, Go, and Atari has become testbeds of testing deep reinforcement learning algorithms. Companies like Deep Mind and OpenAI have done a tremendous amount of research in this field and have set up gyms that can be used to train reinforcement learning. Some examples f this reinforcement learning approach is the AlphaGo Zero, Count-Based Exploration, SATNet. The AlphaGo Zero has as a task to play the game of GO. Go was invented in China and is 2500 years old game where the players make strategies to lock each other moves. Those who fail to make a move, lose. After a millennia Deep Mind which is now owned by Alphabet, create a policy-based deep neural network called AlphaGo that competed against legendary GO player Mr. Lee Sedel, the winner of 18 world titles, and beat him 4-1 in the world championship back in 2016. The authors of the AlphaZero algorithm generalize this approach into a single Alpha zero algorithm that can archive superhuman performance in many challenging domains. Alpha zero

Figure 4: Alpha Go Logo

with no prior domain knowledge has manege to achieve an expert level play of chess and within 24hrs defeated a world-champion program in chess.

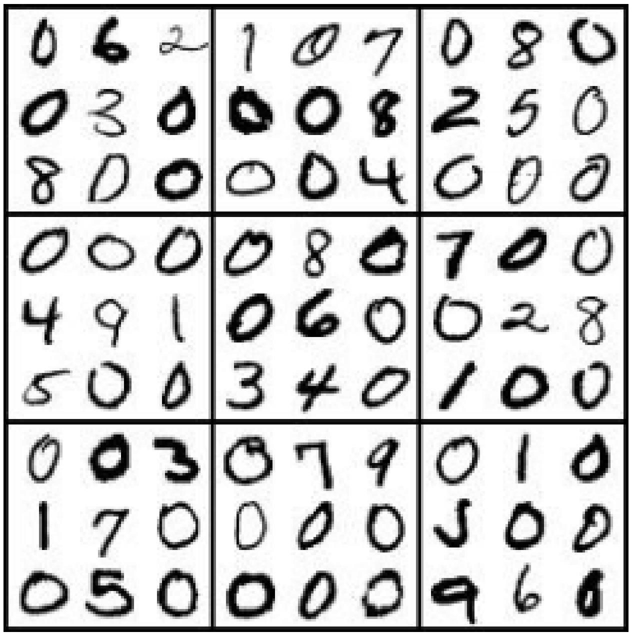
Count-Based Exploration for Deep Reinforcement Learning algorithm has as task the Atari games. This work describes a simple generalization of the classic count-based approach that can reach near state-of-the-art approaches on various high-dimensional and continuous deep reinforcement learning benchmarks. This goes against the thought process that count-based methods cannot be applied in high-dimensional state spaces since most of the state spaces will occur once. The authors use hash functions in their method, where state spaces are mapped to hash codes. This mapping allows counting their occurrences with a hash table. These counts are then used to compute a reward bonus according to the classic count-based exploration theory.



```
Algorithm 1: Count-based exploration through static hashing, using SimHash
Define state preprocessor $g : \mathcal{S} \rightarrow \mathbb{R}^D$
(In case of SimHash) Initialize $A \in \mathbb{R}^{k \times D}$ with entries drawn i.i.d. from the standard Gaussian
  distribution $\mathcal{N}(0, 1)$
Initialize a hash table with values $n(\cdot) \equiv 0$
for each iteration $j$ do
    Collect a set of state-action samples $\{(s_m, a_m)\}_{m=0}^{M}$ with policy $\pi$
    Compute hash codes through any LSH method, e.g., for SimHash, $\phi(s_m) = \mathrm{sgn}(Ag(s_m))$
    Update the hash table counts $\forall m : 0 \leq m \leq M$ as $n(\phi(s_m)) \leftarrow n(\phi(s_m)) + 1$
    Update the policy $\pi$ using rewards $\left\{ r(s_m, a_m) + \frac{\beta}{\sqrt{n(\phi(s_m))}} \right\}_{m=0}^{M}$ with any RL algorithm
```

Figure 5: Count Based Algorithm

SATNet has as the task to solve sudoku puzzles by integrating logical reasoning within deep learning architectures which has been a major goal of modern AI systems. In this paper, the authors propose a new direction towards this goal by introducing a solver that can be integrated into the loop of larger deep learning systems. This work showed how to analytically differentiate through the solution and efficiently solve the associated backward pass. The authors demonstrate that by integrating this solver into end-to-end learning systems can learn the logical structure of challenging problems in a minimally supervised fashion. In particular, with this method, they show that Sudoku learned solely from examples. By combining MAX-SAT solver with traditional convolutional architecture, they have also solved a “visual Sudoku” problem that maps images of Sudoku puzzles to their associated logical solutions



0	6	2	1	0	7	0	8	0
0	3	0	0	0	8	2	5	0
8	0	0	0	0	4	0	0	0
0	0	0	0	8	0	7	0	0
4	9	1	0	6	0	0	2	8
5	0	0	3	4	0	1	0	0
0	0	3	0	7	9	0	1	0
1	7	0	0	0	0	5	0	0
0	5	0	0	0	0	9	6	0

Figure 6: Sudoku

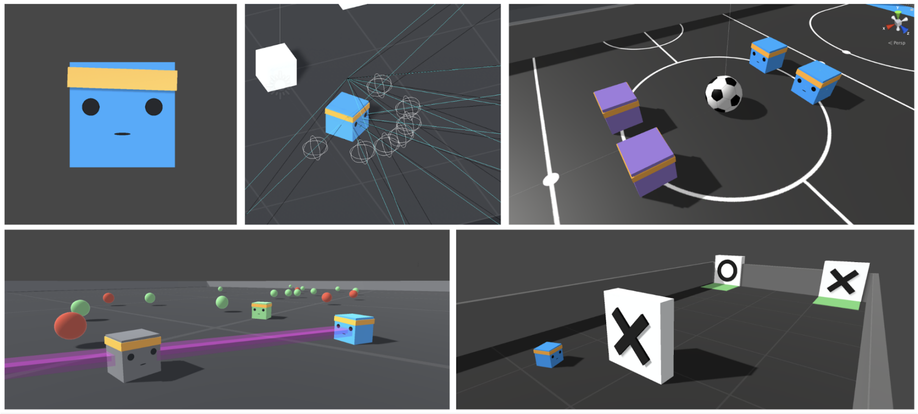


Figure 7: ML-Agents Examples

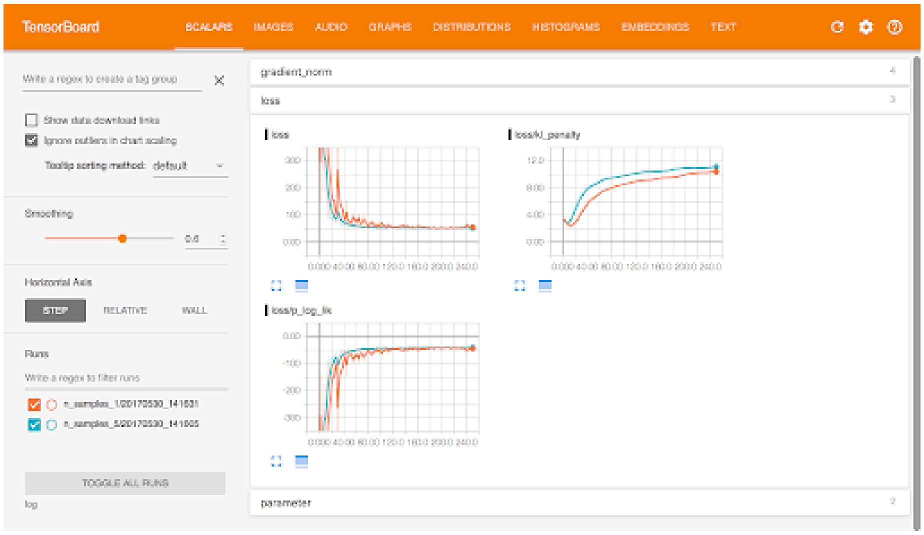
Chapter4: ML-Agents

The Unity Machine Learning Agents (ML-Agents) is an open-source project that gives to the user the opportunity to use its games and simulations as environments for training intelligent agents. It provides user applications based on futuristic PyTorch algorithms to give the power to game developers to easily train intelligent agents for 3d games.

ML-Agents can also use the provided simple-to-use Python API to train agents using reinforcement learning, imitation learning, neuroevolution, and many other methods.

4.1 Proximal Policy Optimization

ML-Agents uses a reinforcement learning technique called Proximal Policy Optimization (PPO). PPO uses a neural network to approximate the idea function that maps an agent's observations to the best action an agent can take in a given state. The ML-Agents PPO algorithm is implemented in Tensor flow and runs in a separate python process that communicates with the running unity application via socket.



□ Show data download links

☑ Ignore outliers in chart scaling

Tooltip sorting method: default

Smoothing 0.6

Horizontal Axis: STEP RELATIVE WALL

Runs

☑ ○ n_samples_1/20170530_141531

☑ ○ n_samples_5/20170530_141505

Figure 8: TensorBoard Graphs

4.2 Tensor Flow

Tensor Flow is an end-to-end open-source platform for machine learning. It has an inclusive and flexible ecosystem of tools and community resources that lets the user push the state-of-the-art in machine learning, build and deploy machine learning-powered applications.

4.3 Tensor Board

Tensor Board provides the visualization and tooling needed for machine learning experimentation.

It can track and visualize metrics such as loss and accuracy, it can provide a visualising model graph, viewing histograms of weights and biases or other tensors as they change over time.

Chapter5: Bomber-Man

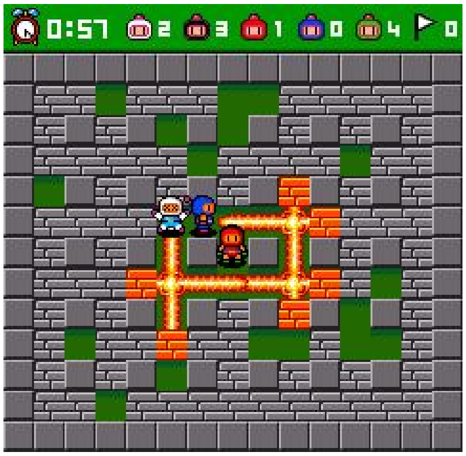
5.1 History

Bomber-Man is a retro strategy maze-based video game witch is released first time in Japan on December 20 1985.

The Bomber-Man game play has two modes, the single mode and the multiplayer mode. In the first mode the player try's to defeat a sires of enemy's and exit the maze to move on the next stage. On the multiplayer mode the player try's to eliminate the other players in the arena.

5.2 Game Play

The game play involves strategically placing down bombs, witch explode after a specific amount of time, in order to destroy obstacles and eliminate enemy's and other players with the goal to be the last-man standing in the arena.



5.3 The Project

The mode that is chosen for this project is the multiplayer mode that will be two agents and try to eliminate each other in the arena with the purpose to observe the strategical bomb placing and the arena exploitation from the agents to achieve their goal.
The two agents will have the same trained brain and the same actions and observations, the characteristics of the two agents will be the same.

Design And Implementation

The design of the project was inspired by the original game of Bomber-Man with similar mechanisms and implementation. The way that the player moves and the concept of the game are an adaption of the classic game.

6.1 Stage

The environment that the game takes place in the arena. The arena includes a parametric wall that encloses the area with unbreakable blocks that are not affected by the bomb explosions. Also, those blocks will found in the arena as obstacles and a way to make the players take more strategic actions to win the game.

Also in the arena, there are bricks that are vulnerable to the bomb explosions and also force the players to break them to move forward or to open a path that leads to each other.

6.1.1 Prefabs

The first prefab is the arena blocks that is the first thing tats made on the arena creation is the green and white blocks that combine to create a floor that looks like chess.

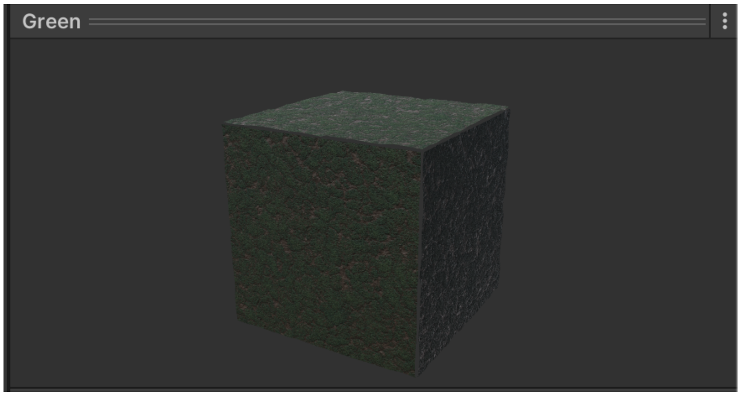


Figure:Green Block Prefab

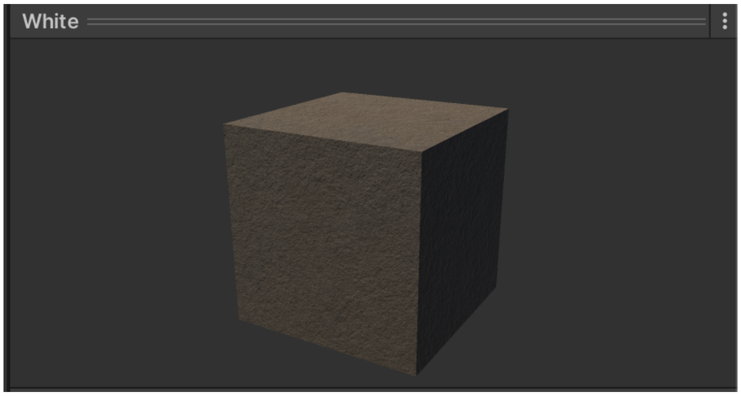


Figure:White Block Prefab

The stage has some boundary’s that the players cant cross, and cant break. Those boundaries are created with unbreakable blocks to keep the players in the game.

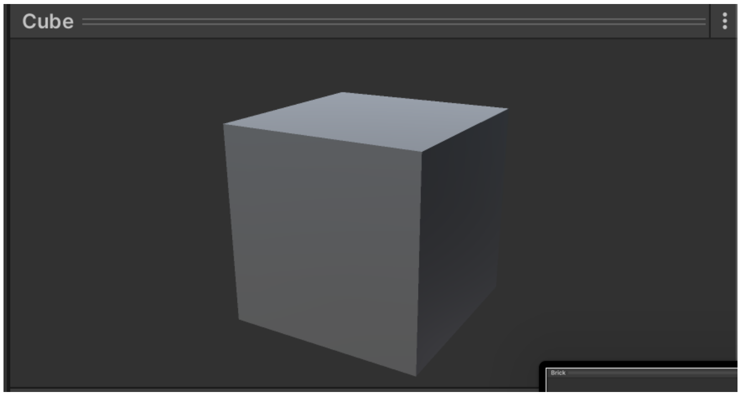


Figure: Unbreakable Block

The first part of the stage is always the same so when the game starts is the first thing that’s created and looks like this:

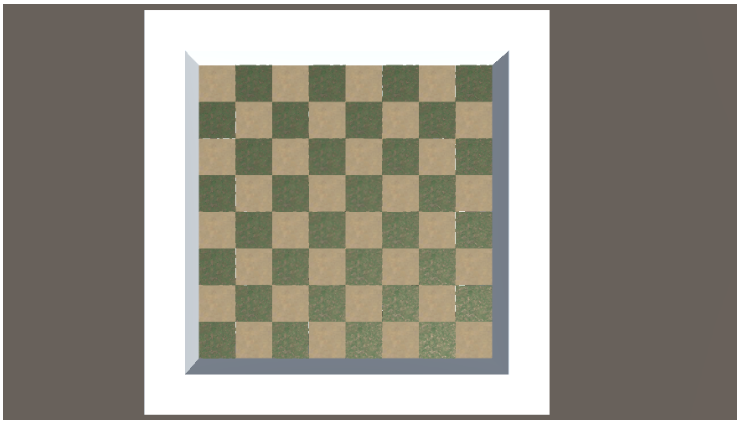
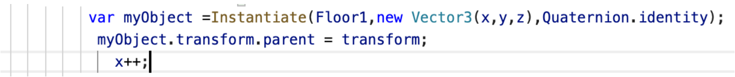


Figure: Stage Initial.

The reason that the stage should look like chess board is because the game use discrete actions and when the player takes an move action it change block so its easier for the observer to watch the change of the player placement.

For the stage, creation makes use of the function of Unity Instantiate that is useful to place objects on the scene on specific coordinates and with specific rotation.



```
var myObject =Instantiate(Floor1,new Vector3(x,y,z),Quaternion.identity);
 myObject.transform.parent = transform;
   x++;
```

6.1.2 Stage Implementation

In machine learning, there is a term that calls ”overfeeding” the neural network and that leads to the agent to take the same actions all over again because it has learned the environment by rote memory or in other words, the agent learns the stage by repetition so the agent doesn’t try to find the other way to complied the task and it has no need to explore the arena because it has already known the stage.
For that reason, the stage at the end of each episode must change randomly to avoid that overfeeding of the neural network so we can take the best results from the training.

6.1.3 Stage Generator

To accomplish that in that project, the design of the arena makes use of a Stage-Generator function that use randomly choices of what object will be instantiated in the arena and where.
Also, this function makes use of some probabilities such as the percentage of empty space in the arena or the percentage of the obstacles or the bricks to avoid the creation of stages with dead ends and to be the stage more playable and easier for the agent to learn some behaviors.

6.1.4 Player And Enemy Placement

For the project, the percentages that have been used for the stage creation are 15 percent of bricks, 15 percent of blocks(obstacles), and 70 percent of empty space.

For the last step of the stage creation is the placement of the player and the enemy in the arena.

The player makes of use of a function called ValidPlayerPlacement() where in that function decided the x and y coordinates for the player in the stage randomly and then their is a check of this place if its valid.

For the placement of the player to be valid it should first be an empty space in the arena, second, the squares of the diagonals of the chosen square to be empty and to be a way out of this position, if those criteria are not met then the function choose randomly other x and y coordinates until the placement of the player would be valid, the same is for the enemy placement.

And the last check is the distance between the player and the enemy to be bigger than the half-length of the arena size.

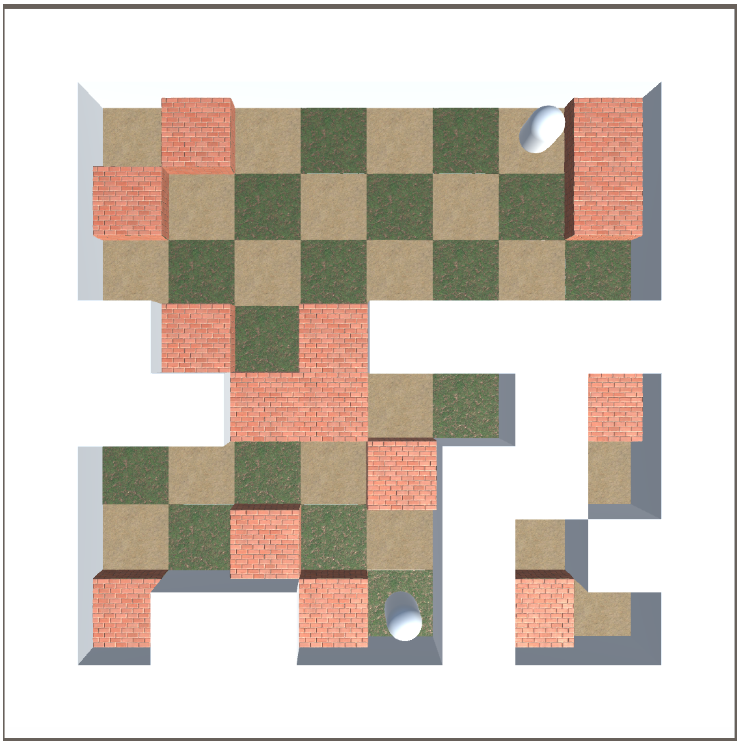


Figure:Final Stage.

6.2 Bomb

The bomb in the game is the only weapon such as the players can use and is the most important part of the game because is the only asset that the players can use to break bricks and open roads, eliminate their enemies, and set up strategic traps for the other players.

The bomb in the original game can explode in four directions (up, down, left, right) and the explosion cannot affect any bricks or players in a diagonal direction.

6.2.1 Bomb Asset

The bomb asset that is chosen for the bomb is a sphere item with size in 3D (.07,.07.07).

Those dimensions take place because the floor blocks are size (1,1,1) and if the bomb prefab was the same size as the floor block there will be a problem with the normal execution of the game.

Also, the color of the bomb is silver and shine so it can be observable by the observer in a real-time game.

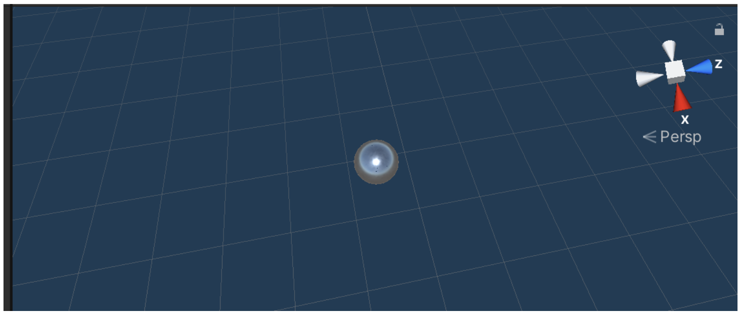


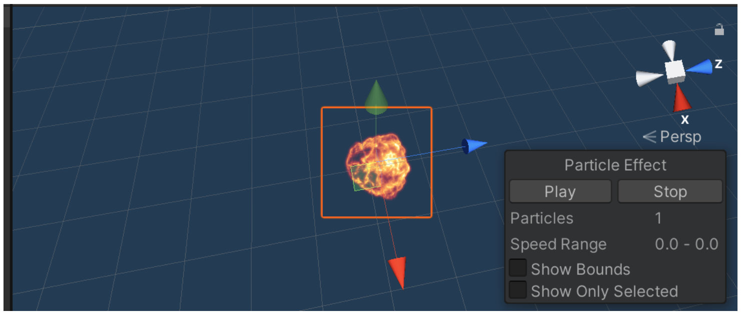
Figure:Bomb Asset.

6.2.2 Explosion Asset

For the explosion asset for this project, its been used a free particle system asset from the asset store.

The explosion prefab takes place in the scene when the bomb explodes after three second's delay after the placement in the arena to conjugate the explosion feeling in the game.

The explosion prefab also has a rigid body and box collider and the reason is when the player collides with the explosion particle system it dies and if a brick gets hit by the explosion asset the break breaks.



Particle Effect
Play Stop
Particles 1
Speed Range 0.0 - 0.0
□ Show Bounds
□ Show Only Selected

Figure:Explosion Asset.

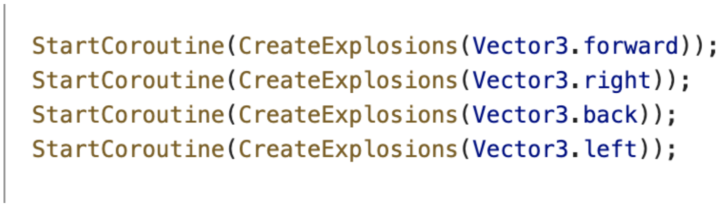
6.2.3 Explosion Representation

As mentioned above when the bomb explodes the explosion prefab would be instantiated in the scene in four directions and not diagonally.

To accomplish that is been useful the use of another function of Unity the coroutine.

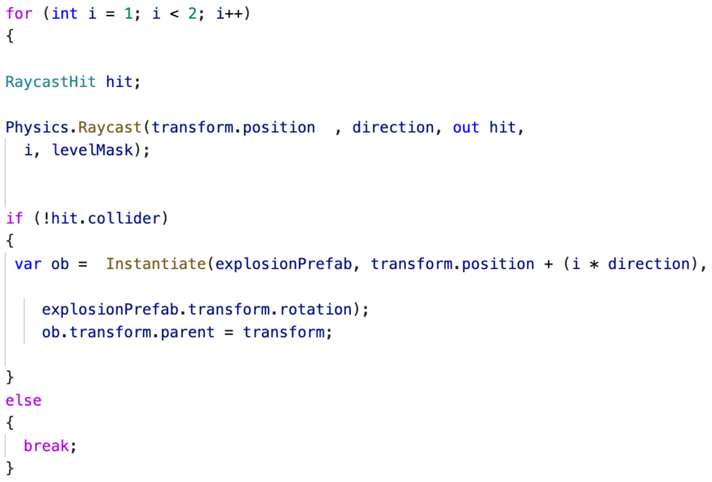
A coroutine is a function that allows pausing its execution and resuming from the same point after a condition is met.

For the purpose of the project, this special function is used to represent the explosion when the bomb explodes, it starts a series of coroutine execution one for each direction.



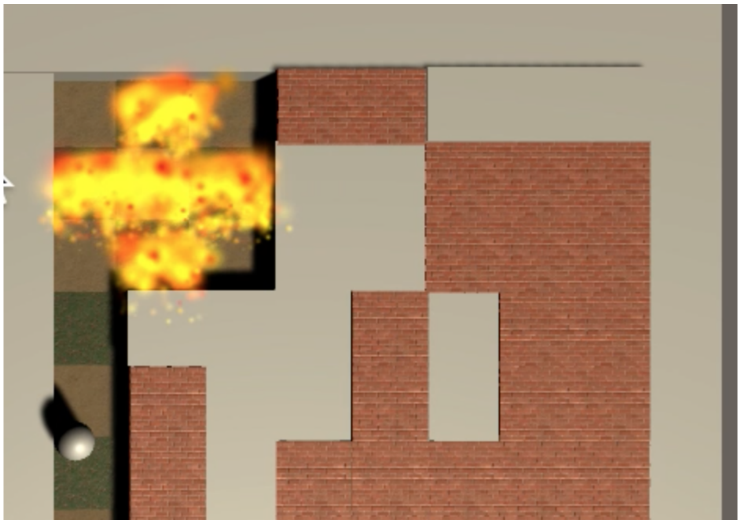
```
StartCoroutine(CreateExplosions(Vector3.forward));
StartCoroutine(CreateExplosions(Vector3.right));
StartCoroutine(CreateExplosions(Vector3.back));
StartCoroutine(CreateExplosions(Vector3.left));
```

For each coroutine, it starts a reycast where when the ray collides on an object the instantiate doesn't execute, and the coroutine intercept so the particles don't appear inside the blocks and or the player and give the feeling of the collision of the explosions and the objects in the environment.



```
for (int i = 1; i < 2; i++)
{

RaycastHit hit;

Physics.Raycast(transform.position  , direction, out hit,
  i, levelMask);


if (!hit.collider)
{
 var ob =  Instantiate(explosionPrefab, transform.position + (i * direction),

    explosionPrefab.transform.rotation);
    ob.transform.parent = transform;

}
else
{
  break;
}
```

Also if the bomb collides with an explosion particle the three-second delay intercept and the Explode() function execute to make possible the chain reaction of bombs in the game.



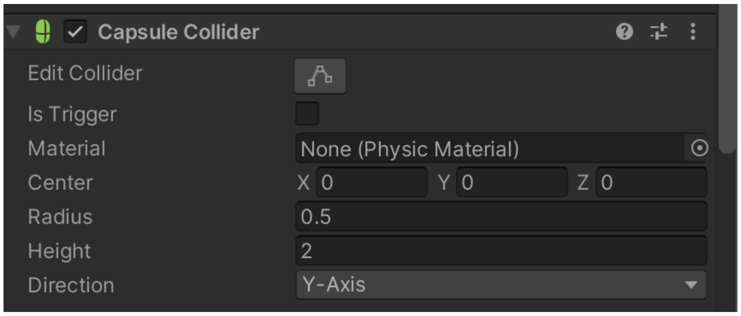
6.3 Agent

An agent in reinforcement learning is an actor that can observe its environment, deiced on the best course of action using those observations, and execute those actions within the environment.
The agent's in this project they observe the arena by using sensor and special functionalities and try to take the wright action due to the state that they are in a specific time.

6.3.1 Agent Asset

The agent asset that is decided to used for this project is a capsule because is more close to a human figure.

The agent characteristics are the capsule collider where is useful in the game because with the collider the agent can observe collisions in the environment.

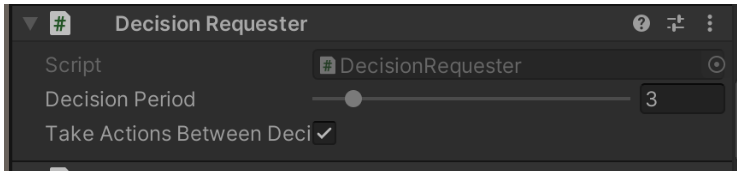


☑ Capsule Collider

Edit Collider	
Is Trigger	☐
Material	None (Physic Material)
Center	X 0 Y 0 Z 0
Radius	0.5
Height	2
Direction	Y-Axis

The capsule collider has no physics material, its radius value is 0.5 and the height value is 2.

The agent has attached a script of ml-agents library the Decision Requester script. The Decision Requester component automatically requests decisions for an agent instance at regular intervals. It provides a convenient and flexible way to trigger the agent decision-making process. The Decision Requester script has two fields. The field with the name Decision Period is used to set a period of when the agent will take an action, lets say if the Decision Period gets the value 5 that means that the agent will request a decision every 5 Academy steps. The second field of the Decision Requester script is the Take Actions Between check box that indicates whenever or not the agent will take an action during the academy steps where it does not request a decision.



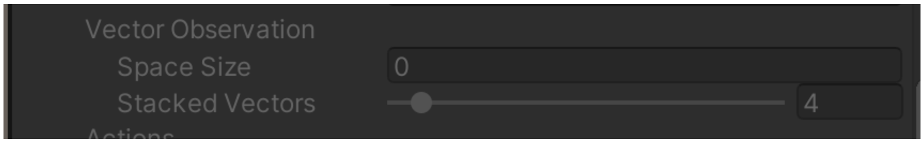
Decision Requester

Script DecisionRequester

Decision Period 3

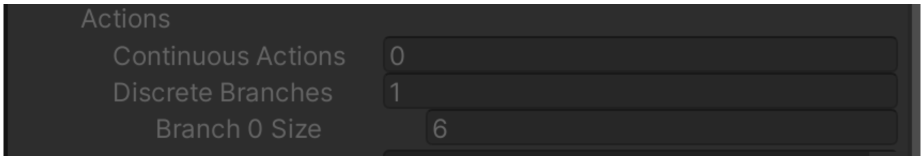
Take Actions Between Deci ✓

An other important script that is attached on the agent to determinate its actions and policy is the Behavior Parameters script. Behavior Parameters script is a component for setting an Agent instance's behavior brain parameters. At run time, this component generates the agent's policy objects according to the settings the developer specified in the editor. The behavior parameters script has some fields that specify the behavior of the agent. The first field is the Behavior Name where is the name of this behavior, which is used as a base name. The vector observation section determines a vector of observations that the agents collect during their interaction with the environment. The vector observations have two sections, the first section is the space size where it takes an integer value to declare the size of the vector that is equal to the number of elements that parse in the observation vector during a step in the episode, and the second field is the stacked observation that is use-full to the training if we need the agent to take actions based on the current observations and the previous observations. This field takes as input an integer that determines the value of the stacked observations we need the agent to observed. That space gives the agent the ability to has memory and remember its previous actions and observations so it can calculate more accurately the next policy.



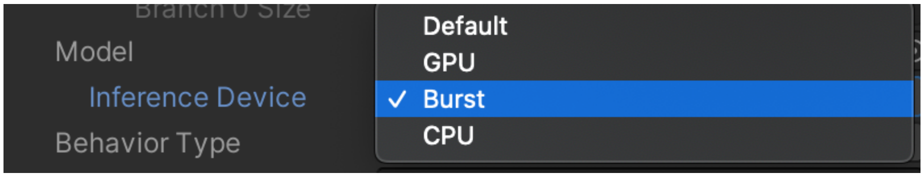
Vector Observation
Space Size 0
Stacked Vectors 4
Actions

The actions section declares the number of actions and the type of actions of the agent. The actions have three fields, the first field and the second field define the type of the actions of the agent. The first option is continuous actions. The continuous actions reference values that are not discrete such as floats and double. An example if the action is to turn certain degrees this action will be continuous. The discrete actions reference values that are integers for example if the player should move a square the action will be described. In the project, the actions are discrete because the agent must move by squares, and also there is no rotation in the game, which makes the training easier and reduce the complexity of the game. The last section of the actions is the Branch Size that defines the number of actions in each branch.



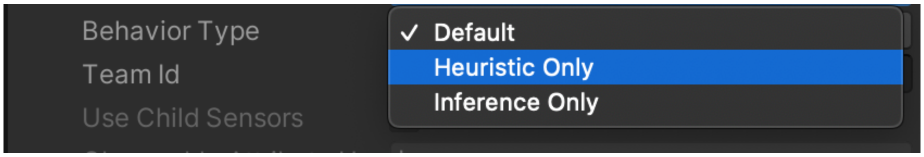
Actions
Continuous Actions 0
Discrete Branches 1
Branch 0 Size 6

The inference Device is a drop-down menu with options on where does the user needs the inference to perform. The first choice is the Burst which is a CPU inference using Burst. The second choice is the CPU but the Burst option is recommended. The CPU choice is kept for legacy compatibility. The third choice is the Default that is currently the same as Burst.

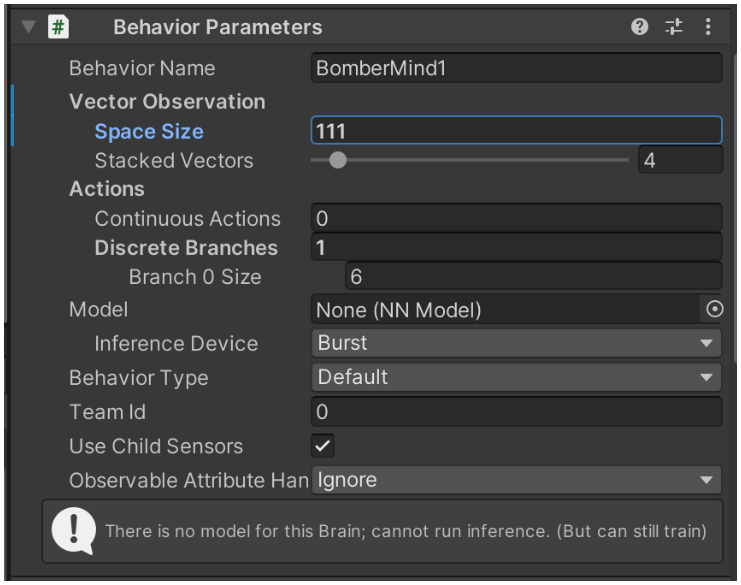


Branch 0 Size
Model
Inference Device ✓ Burst
Behavior Type

The Behavior type field is used to declare the behavior of the agent. This category has also a drop-down menu with three choices. The first choice is the Default option where the engine understands if the mode of the game is training or its gameplay and chooses on its own. The second option is the Heuristic model that gives to the developer the opportunity to test the actions of the agent and the game before training. The third choice is Inference and this mode is only for training.



If the game contains teams, the team id field is used, and the engine clusters the agents by team number in training mode. The last field of the Parameters Behavior script is the Use Child Sensors in the training.

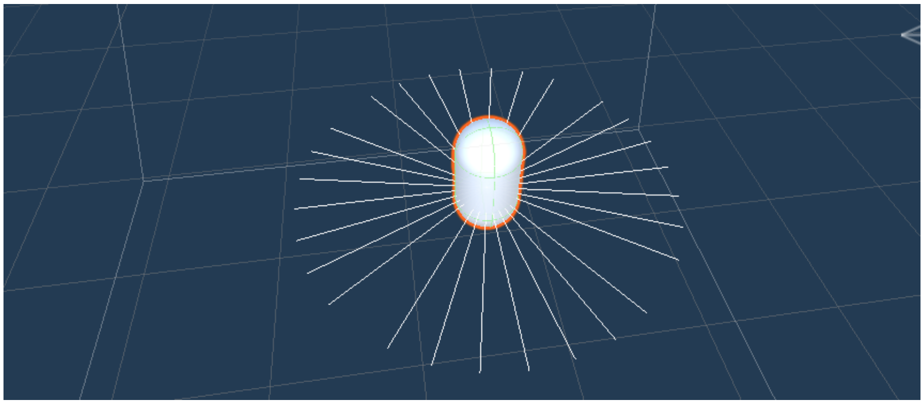


Behavior Parameters

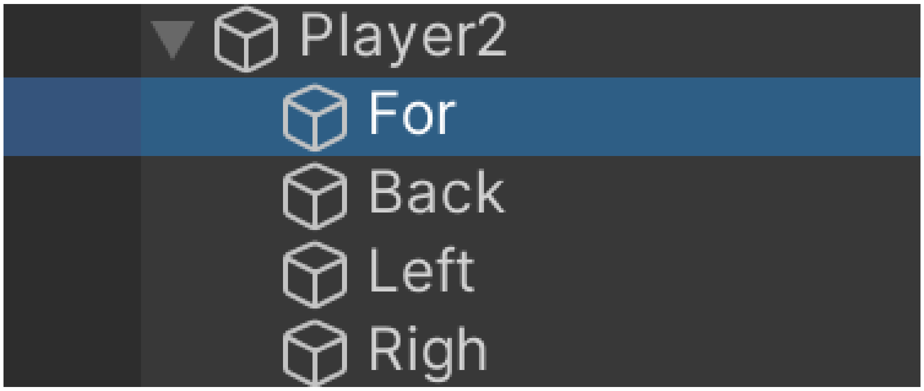
Behavior Name	BomberMind1
Vector Observation	
Space Size	111
Stacked Vectors	4
Actions	
Continuous Actions	0
Discrete Branches	1
Branch 0 Size	6
Model	None (NN Model)
Inference Device	Burst
Behavior Type	Default
Team Id	0
Use Child Sensors	☑
Observable Attribute Han	Ignore

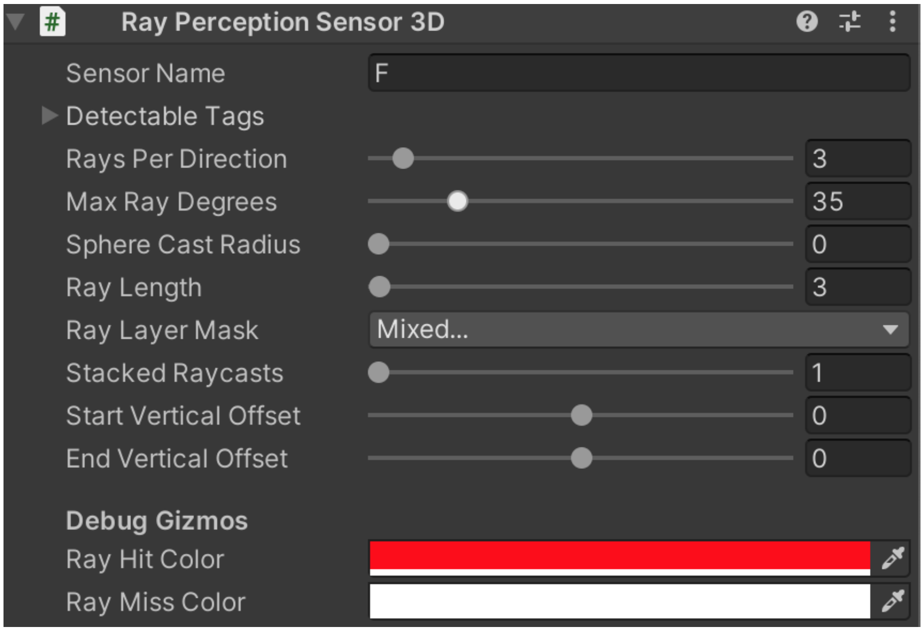
There is no model for this Brain; cannot run inference. (But can still train)

On the agent model there are attached Ray Perception Sensors that used to parse extra observations to the agent neural network and help on the development of a policy for the environment around the agent.



The Ray Perception script has lots of fields and from those fields the developer can control the length of the rays, the number of rays per angle, the start and the end vertical offset of the rays that changes the field of the rays hits, there are ray layer masks for the collision and detectable tags for objects that the agent must observe. On the player the number of rays that are attached are four one for each side of the capsule 3d object so it can take a full 360 degrees of observations.

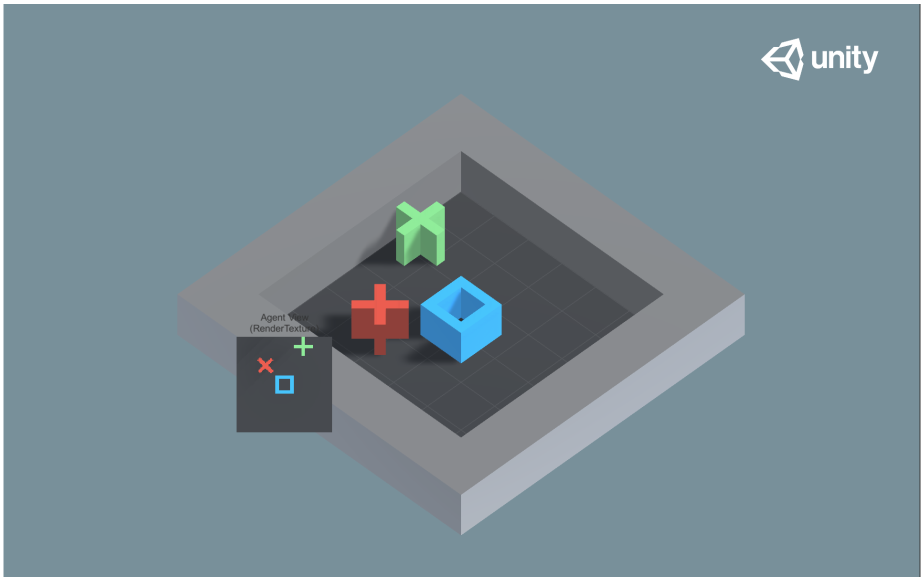




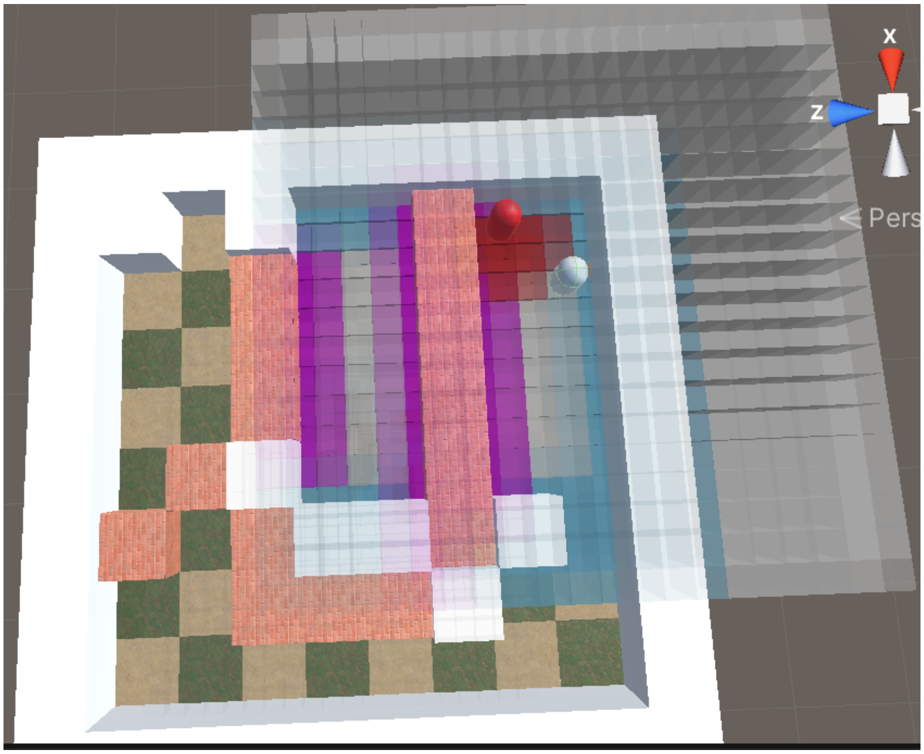
Ray Perception Sensor 3D

Sensor Name	F
▶ Detectable Tags	
Rays Per Direction	3
Max Ray Degrees	35
Sphere Cast Radius	0
Ray Length	3
Ray Layer Mask	Mixed...
Stacked Raycasts	1
Start Vertical Offset	0
End Vertical Offset	0
Debug Gizmos	
Ray Hit Color	🟥
Ray Miss Color	⬜

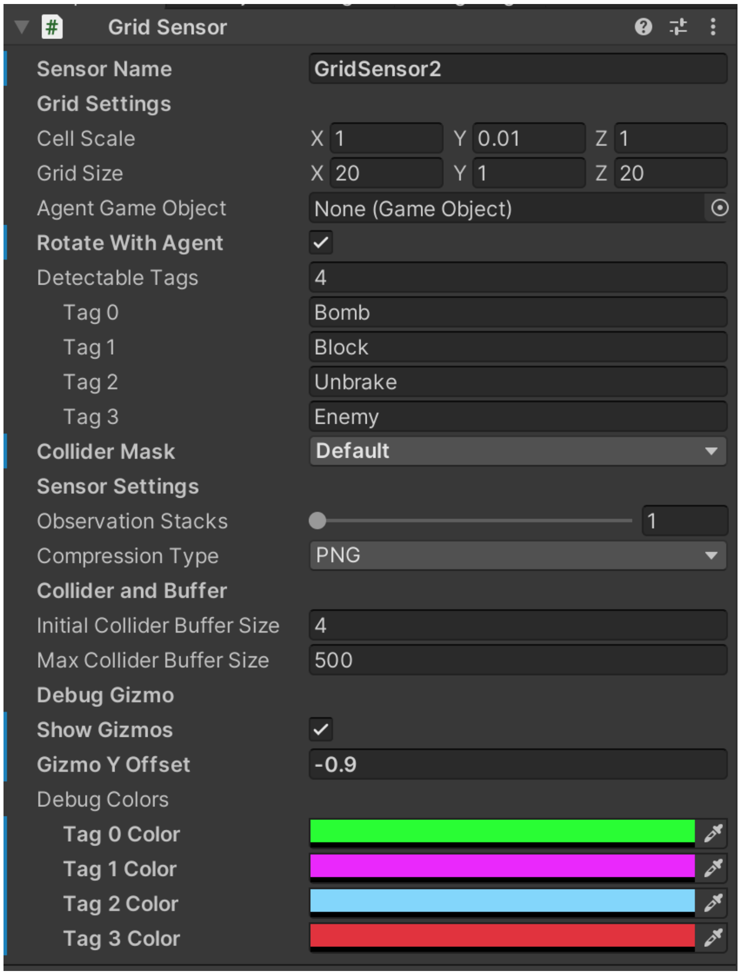
The agent also uses another script for observations collection the GridSensor. The ML-Agents provides two types of sensors to generate observations for the agent that are used to train reinforcement learning models. The first is the raycast that has been explained above, which allows the agents to observe and collect data about a GameObject on line sight. The second type use pixels from the camera that is attached to an agent. The camera provides the agent with either a grayscale or an RGB image of the game environment based on how the camera is rigged and positioned. Those images can be used to train convectional neural networks which learn to understand the nonlinear relationship between pixels. Both of those sensors have limitations. The length of the ray cast is usually limited because the agent does not need to know the objects that are on the other side of the scene. Also, the ray cast is not computational efficient so fewer ray casts are conducted. By using camera rendering of the scene can significantly slow down the number of engine ticks per minute versus other alternatives, such as running heedlessly with only discrete observations.



The Grid Sensor combines the generality of data extraction from ray casts with the computational efficiency of the convolution neural networks. The Grid Sensor collects data from game objects by querying the physics properties and the structures of the data into a height x width x channel matrix. This matrix is analogous to an image from an orthographic camera but rather the representing red, green, and blue color values of objects, the pixels of the grid sensor represent a vector of arbitrary data of objects to the sensor. Another benefit is that the grid sensor can have a lower resolution which improves the training by times. This matrix can then be fed in a convolution neural network and used to train a reinforcement learning model.



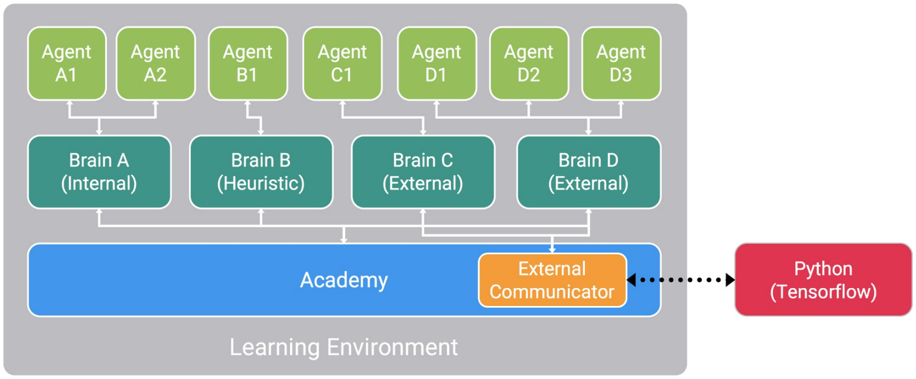
The components of the grid sensor are the grid settings which in that field the developer can modify the scale of the grid and its position in the scene according to the player position. Also on the grid sensor has to be parsed in the detectable tags of the objects that the agent must observe, there is also the option of collider mask so it can ignore some collisions if there are not desirable. On the sensor settings field, there are options for stacking observations and the settings of the collision buffer. Also, there is the Debug color section where it can be chosen the color that appears in the sensor when an object with a specific tag is detected from the sensor.



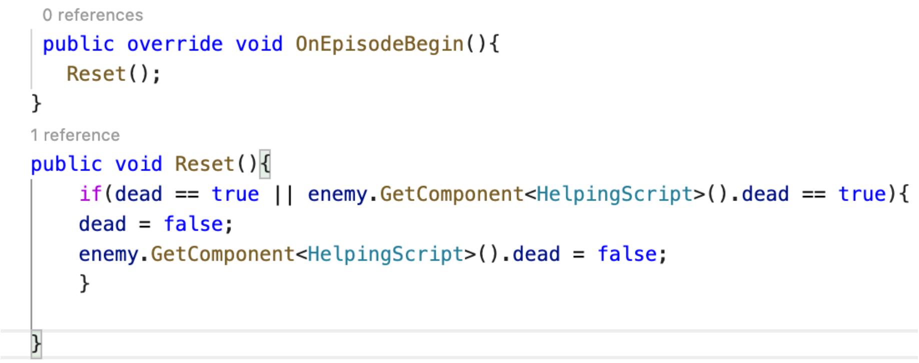
Grid Sensor

Sensor Name	GridSensor2
Grid Settings	
Cell Scale	X 1 Y 0.01 Z 1
Grid Size	X 20 Y 1 Z 20
Agent Game Object	None (Game Object)
Rotate With Agent	✓
Detectable Tags	4
Tag 0	Bomb
Tag 1	Block
Tag 2	Unbrake
Tag 3	Enemy
Collider Mask	Default
Sensor Settings	
Observation Stacks	1
Compression Type	PNG
Collider and Buffer	
Initial Collider Buffer Size	4
Max Collider Buffer Size	500
Debug Gizmo	
Show Gizmos	✓
Gizmo Y Offset	-0.9
Debug Colors	
Tag 0 Color	🟩
Tag 1 Color	🟪
Tag 2 Color	🟦
Tag 3 Color	🟥

The player script that implements in the project inherits the subclass Agent from the ml-agents. Agents in an environment operate in steps. At each step, an agent collects observations, passes them into its decision-making policy, and receives an action vector in response.



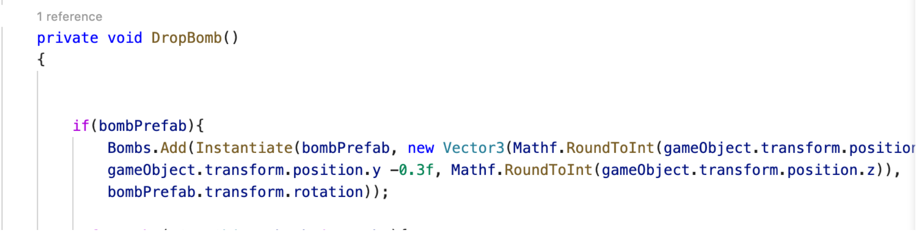
The agent class gives the functionality that is needed to generate an agent that would interact with the environment and build its decision-making policy by overwitting specific functions such as OnActionReceive() function, CollectObservations(), and other functions that help the agent build.
The first function that's bean overwrite in from the agent class is OnEpisodeBegin(). The OnEpisodeBegin function is called when an episode reaches its end and is used to reset the agent to its initial state.



```
 0 references
 public override void OnEpisodeBegin(){
   Reset();
}
1 reference
public void Reset(){
    if(dead == true || enemy.GetComponent<HelpingScript>().dead == true){
    dead = false;
    enemy.GetComponent<HelpingScript>().dead = false;
    }

}
```

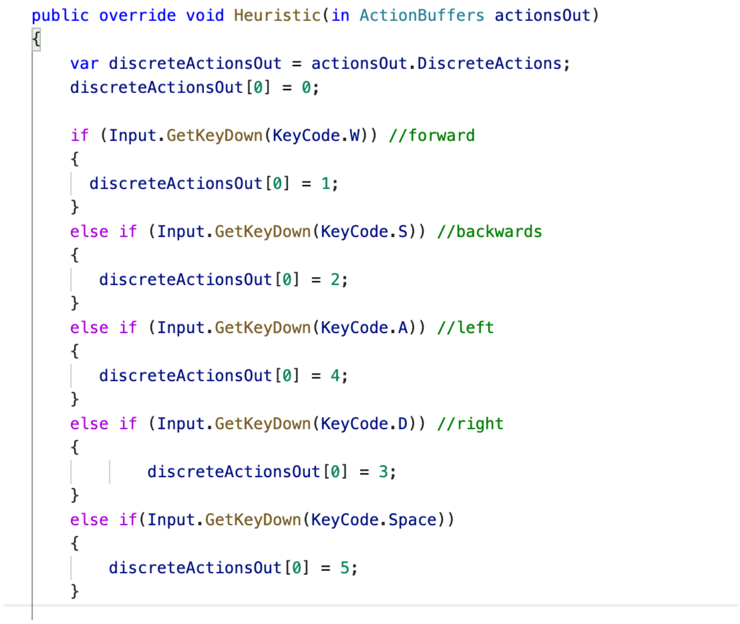
The other function that is useful in the agent building is the OnActionReceive(). This function is used to generate an action vector array that parses into the AgentAct function where all the actions are implemented.

The actions they needed for the agent in this project are six. The agent should be able to move forward, backward, left, right, to not move at all and, to place a bomb in the scene. The agent when the action vector is received in the AgentAct execute the action on the first cell of the array. If the action in the first cell is the forward, backward, left or right the agent check if the cell he wants to move is free and then move to its new position. If the action is drop bomb then the AgentAct executes the function DropBomb(). The Drop Bomb function instantiates a bomb prefab on the coordinates of the player.



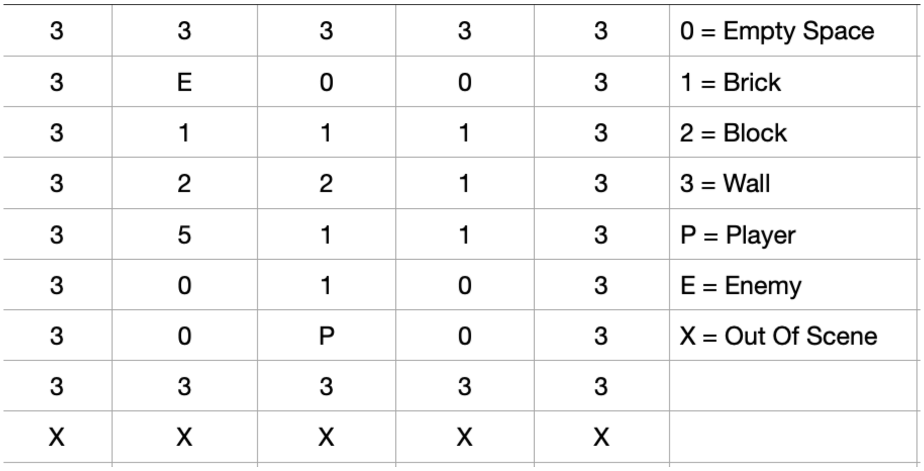
```
1 reference
private void DropBomb()
{


    if(bombPrefab){
        Bombs.Add(Instantiate(bombPrefab, new Vector3(Mathf.RoundToInt(gameObject.transform.position
        gameObject.transform.position.y -0.3f, Mathf.RoundToInt(gameObject.transform.position.z)),
        bombPrefab.transform.rotation));
```

The Heuristic function implemented to specify the agent actions using the developer own heuristic algorithm. Implementing heuristic function can be useful for debugging.



```
public override void Heuristic(in ActionBuffers actionsOut)
{
    var discreteActionsOut = actionsOut.DiscreteActions;
    discreteActionsOut[0] = 0;

    if (Input.GetKeyDown(KeyCode.W)) //forward
    {
      discreteActionsOut[0] = 1;
    }
    else if (Input.GetKeyDown(KeyCode.S)) //backwards
    {
       discreteActionsOut[0] = 2;
    }
    else if (Input.GetKeyDown(KeyCode.A)) //left
    {
       discreteActionsOut[0] = 4;
    }
    else if (Input.GetKeyDown(KeyCode.D)) //right
    {
            discreteActionsOut[0] = 3;
    }
    else if(Input.GetKeyDown(KeyCode.Space))
    {
        discreteActionsOut[0] = 5;
    }
```

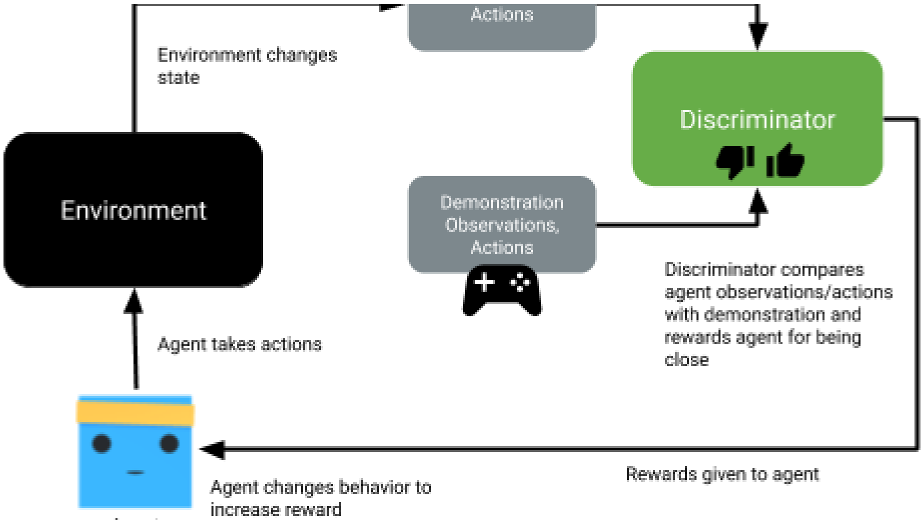
The CollectObservations() function is used to parse the observations into the agent and make a decision policy. The player has a point of view array in the size of the arena. The Point of view array change as the player moves in the environment and has as its center the player. This array contains all the objects in the scene and is used to feed the agents decision making policy with this observations.



3	3	3	3	3	0 = Empty Space
3	E	0	0	3	1 = Brick
3	1	1	1	3	2 = Block
3	2	2	1	3	3 = Wall
3	5	1	1	3	P = Player
3	0	1	0	3	E = Enemy
3	0	P	0	3	X = Out Of Scene
3	3	3	3	3	
X	X	X	X	X	

The Rewards that been used for this project on the training are the following:

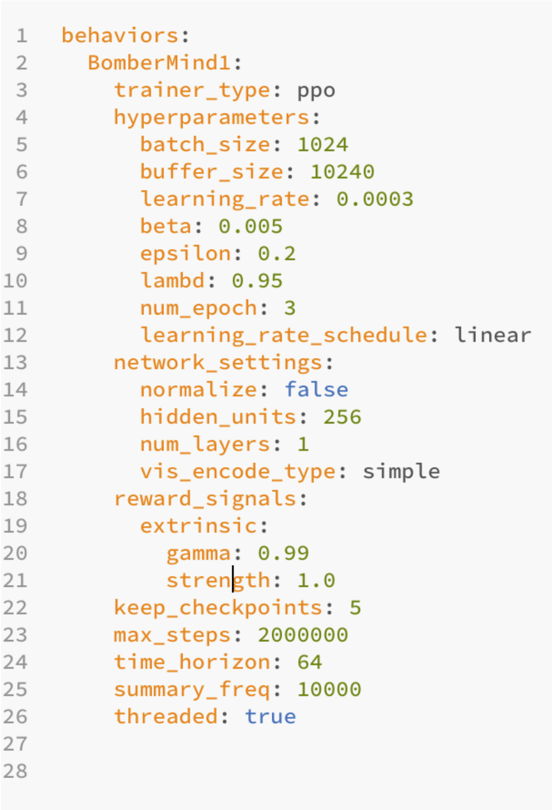
- Player Dead = -1f
- Enemy Dead = 1f
- Bomb Placement = -1/MaxSteps
- Live Penalty = -1/MaxSteps
- Brick Break = 1/MaxSteps


Actions
Environment changes state
Discriminator
Environment
Demonstration Observations, Actions
Discriminator compares agent observations/actions with demonstration and rewards agent for being close
Agent takes actions
Agent changes behavior to increase reward
Rewards given to agent

When the player archive to eliminate the enemy receives a reward of 1 point and if the enemy eliminates the agent or the agent killed by its bomb receives the reward of -1 point. When the agent place a bomb in the scene receives the reward of -1/MaxSteps which is a small penalty so he can learn to not place bombs uncontrolled and fill the arena with bombs because is more possible to be killed by its bombs. The agent receives the penalty of -1/MaxSteps as a living penalty to force the agent to take action and don't sit in the arena if he doesn't know what to do. If the agent breaks a brick with the bomb then it receives the reward of 1/MaxStep to encourage him to continue breaking bricks so it can open roads that lead to the enemy.

Chapter7: Training And Results

The Unity ML-Agents Toolkit provides a wide range of training methods and options. As such, specific training runs may require different training configurations and may generate different artifacts and Tensor Board statistics. It is important to highlight that successfully training Behaviors in the ML-Agents toolkit involves turning the training hyper-parameters and configuration. The trainer's config file determines the features that are used by the developer during the training. the primary section of the trainer config file is a set of configurations for each behavior. These are defined under the section behaviors in the trainer config file. For the project, the config file that's been used is below.



```
behaviors:
  BomberMind1:
    trainer_type: ppo
    hyperparameters:
      batch_size: 1024
      buffer_size: 10240
      learning_rate: 0.0003
      beta: 0.005
      epsilon: 0.2
      lambd: 0.95
      num_epoch: 3
      learning_rate_schedule: linear
    network_settings:
      normalize: false
      hidden_units: 256
      num_layers: 1
      vis_encode_type: simple
    reward_signals:
      extrinsic:
        gamma: 0.99
        strength: 1.0
    keep_checkpoints: 5
    max_steps: 2000000
    time_horizon: 64
    summary_freq: 10000
    threaded: true


```

The drainer type defines the type of the algorithm that is bean used in the training. There are three types of training algorithms in ml-agents the PPO, SAC, and the POCA.

Under the hyper parameters section is the option of batch size where is set to 1024 and this defines the number of experiences in each iteration of gradient descent. This should always be smaller than the buffer range and the number of batch size depends on the type of the actions and the type of the training algorithm.

The buffer size for PPO default is 10240 and for SAC is set to 5000. For the PPO algorithm, the buffer size represents the number of experiences collected before updating the policy model. Typically a larger buffer size corresponds to more stable training updates.

The learning rate corresponds to the strength of each gradient descent update step. This value should typically be decreased if training is unstable, and the reward does not constantly increase. The range of learning rate is between 1e$^-5to1e^-$3.

The beta value is the strength of the entropy regularization which makes the policy more random. this ensures that agents properly explore the action space during training. By increasing the beta value will ensure random actions will be taken.

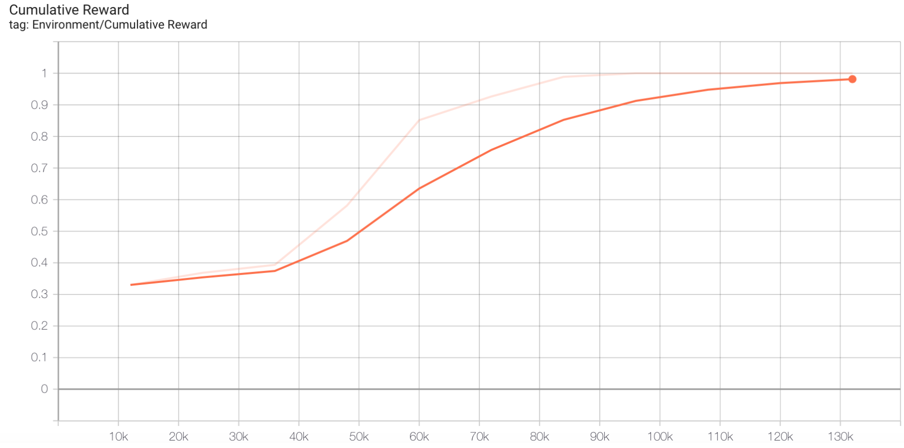
The epsilon parameter influences how rapidly the policy can involve during training. This corresponds to the acceptable threshold of divergence between the old and the new policy.

The lambda parameter used when calculating the generalized advantage estimate. This can be thought of as how much the agent relies on its current value estimate when calculating an updated value estimate.

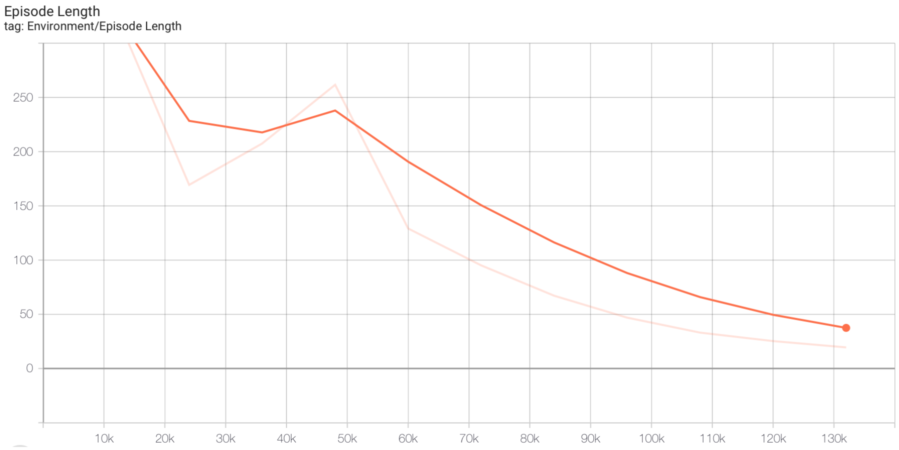
The network settings field is the hyper parameters of the neural network and how it will be build. In the project the neural network has 256 hidden layers and number of layers 1.

The reward signals section enables the specification of settings for both extrinsic and intrinsic reward signals. Each reward signal should define two

parameters, strengths and gamma. The strength parameter is the factor by which to multiply the reward from the environment and the gamma parameter is the discount factor for future rewards coming from the environment.
The goals of this project was to rebuild a retro game from scratch by using Unity 3d game engine and create an agent that he will learn the functionalities of the game by training using reinforcement learning. The first approach to achieve this goal was by using simple arenas and train the agent to walk around to find the enemy in the arena.



As we can see from the above graph the agent achieve that simple task pretty easily in short time period and to do that it needed only 130k steps.



The graph above shows the episode length how decreases over time, this is something that shows that the agent complete the task faster every episode and the episode length drops and the reward rises so the agent has successfully complete this training.
The rewards for this training was:

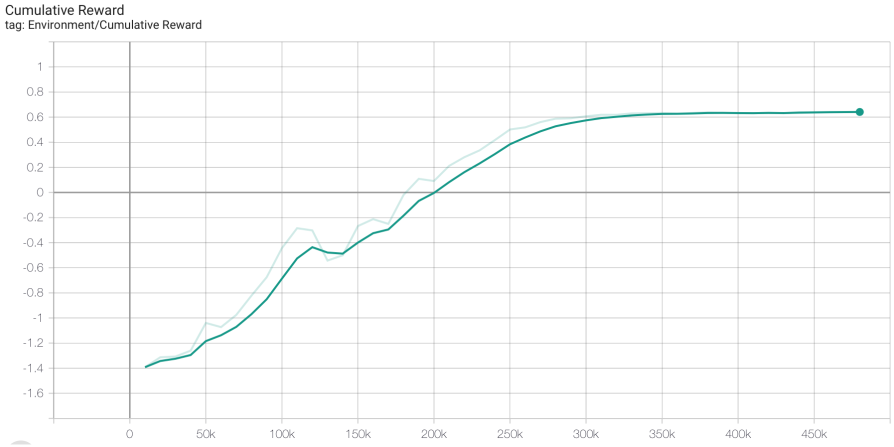
- Reach the enemy by distance 1.5 float units: +1f
- Living penalty : -1/MaxSteps

If the agent finds the enemy receives the reward of 1 point and if he random walk pointless or stay in the spot without taking an action receives the reward of -1/MaxSteps. The living penalty forces the agent to take actions so it can explore the arena and find a way to the positive reward.
The second approach for to complete the task was to involve in the training and the bombs since this is the way to win the game in the real game is to eliminate the enemy by using bombs and be careful to not eliminate your self by the enemy's bombs or your bombs. The scenario for this approach was simple again.The point in this scenario was the agent to use the previous training that walks and finds the enemy position but this time should learn to eliminate him by using bombs. This scenario was more complicated from the

previous training because the agent know must learn to avoid the bombs also and not to place them in a way that would kill him. As mention in the chapter of the implementation in the bomb section, the bombs needs 3s to explodes but if a bomb near explodes and collide with the new bomb the new bomb explodes to overlooking the rule of the 3s. That detail was a part of the agent training and to help the training of the agent in this scenario was added one more reward. The rewards for the second scenario:

- Kill enemy with bomb: +1f
- Killed by bomb : -1f
- Living penalty : -1/MaxSteps
- Placing Bomb : -1/MaxSteps

The first reward is if the agent achieve to eliminate the enemy with a bomb then receives the reward of 1 point. If the agent eliminate its self by using bombs receives the reward of -1 point. The living penalty stays the same as above. If the agent place a bomb in the arena receives the reward of -1/MaxStep. By using the last reward for the bomb placing it achieves that the agent is more careful of bomb placing and not spawn them uncontrollable in the arena and increases the chances of self-elimination.

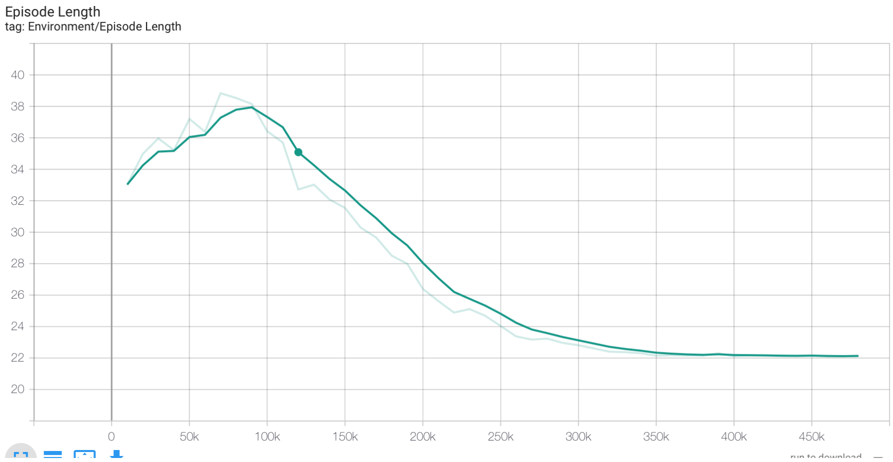


In the second scenario the agent took more steps to learn the expected behavior as seen in the graph above the agent starts with a minus reward because the bomb placement was a new ability that he has to learn and become familiar with.

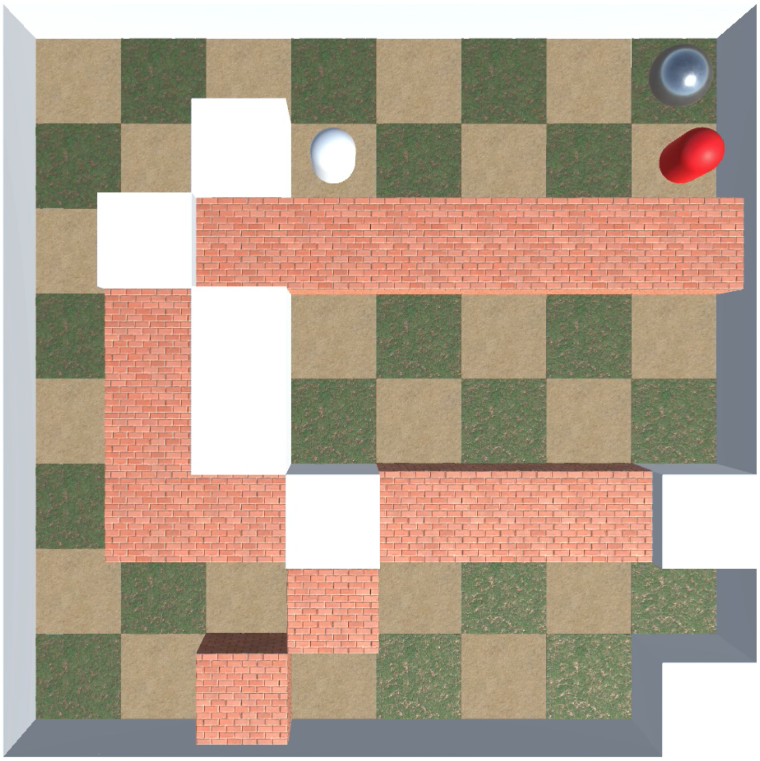
But after 200k steps the agent start to understand the goal in this scenario and learn that he should place the bomb and then try to hide from the bombs so he dont eliminate him self by them and receive a negative reward.

Also he started to place the bomb near the enemy position to eliminate him and receive the positive reward because he achieve the goal of the training.

The agent on the 325k steps has already became very familiar with the idea of the scenario and started to receive constantly positive rewards and stabilize the training rate.



As show in the graph the episode length start to decrease dramatically after the 100k steps which is something that shows that the agent already started to understand the goal of the training and the decision policy of the agent is more accurate and more efficient so he end the goal faster and reduce the time that he needed to complete the episode.



The third scenario is to make the arena more complicated and push the agent to learn how to create paths to achieve its goal. To make that possible in the arena the roads that leads to the enemy are now blocked by breakable blocks or unbreakable blocks that leads the agent to find a new way to reach its goal. Now the agent has to use the bombs not only to eliminate the enemy and receive the positive reward but also to open road that leads to the enemy. For that scenario the reward stay the same except that one reward added on. The rewards for the second scenario:

- Kill enemy with bomb: +1f

- Killed by bomb : -1f
- Living penalty : -1/MaxSteps
- Placing Bomb : -1/MaxSteps
- Block Break : +1/MaxSteps

As in the second scenario the enemy elimination and the dead by bomb has the same values. Also the living penalty and the bomb placement are the same. The new reward that takes place in the training is the block break reward. This reward has a purpose to encourage the agent to place bombs near the breakable blocks and destroy them, with this approach the agent now try's to brake blocks and try to find the enemy behind this blocks.

The final scenario for this project is the scenario of two agents that share the same brain try to eliminate each other by using the above knowledge that they learn from the training's. The two agents has the same behavior parameters, the same observations and receive the same rewards. There are multiple environments for this training and that leads to a competitive game between two equal rivals.

Chapter8: Conclusion

The reinforcement learning is a pioneering way of artificial intelligence in games.

With machine learning in the game industry the companies can create more competitive enemy's in the games and more intelligence than the classic artificial intelligence that has a given structure of actions in any situation.

By training agents to play a game could be usefully also to find bugs in the game or shortcut's that the developers cant see that they exist in the game and the game's can exploit them.

Also by using reinforcement learning already the industry achieve superhuman abilities in classic games such as Go, chess and more other games.

That show the possibility's that we can archive by using more the power of the computers and the power of machine learning.

Chapter9: Bibliography

[1]URL:https://github.com/Unity-Technologies/ml-agents/blob/main/docs/Training-ML-Agents.mdtraining-configurations

[2]URL:https://github.com/Unity-Technologies/ml-agents/blob/main/docs/Training-ML-Agents.md

[3]URL:https://analyticsindiamag.com/reinforcement-learning-top-state-of-the-art-games-alphago/

[4]URL:https://www.kdnuggets.com/2018/03/5-things-reinforcement-learning.html

[5]URL:https://jonathan-hui.medium.com/rl-proximal-policy-optimization-ppo-explained-77f014ec3f12

[6]URL:https://towardsdatascience.com/proximal-policy-optimization-tutorial-part-1-actor-critic-method-d53f9afffbf6

[7]URL:https://spinningup.openai.com/en/latest/algorithms/ppo.html

[8]URL:https://docs.unity3d.com/Packages/com.unity.ml-agents@2.0/api/Unity.MLAgents.Sensors.GridSensorComponent.html

[9]URL:https://blog.unity.com/technology/how-eidos-montreal-created-grid-sensors-to-improve-observations-for-training-agents

[10]URL: https://bigdataanalyticsnews.com/history-of-artificial-intelligence-in-video-games/

[11]URL:
https://www.andreykurenkov.com/writing/ai/a-brief-history-of-game-ai/

[12]URL:https://sitn.hms.harvard.edu/flash/2017/ai-video-games-toward-intelligent-game/