Gliwice, 07.09.2021

Politechnika Śląska w Gliwicach Wydział Automatyki,
Elektroniki i Informatyki

Biologically inspired artificial intelligence

Report

„Bomberman AI opponents”

Skład sekcji:

- Tomasz Knura
- Dawid Herc
- Wojciech Francke

I. Introduction

The goal of the project was to create a fully functional AI opponent for simple unity bomberman game, which includes smooth moving, laying down bombs, running away from explosions, collecting power ups etc.
With the last version AI should be capable of being a challenging opponent for a human player
Learning process was planned to be conducted using Imitation Learning and Self-Play, starting with IL to teach an AI basics of the game
The base for this project was the aforementioned simple unity bomberman game that we created for an earlier project.

II. Task analysis

1. Problem solving approach

Since our goal is making human -like real time decision making bot we had 3 options of algorithm classes for the learning process: unsupervised learning, supervised learning, imitation learning and reinforcement learning. What ultimately separates them is the type of data available to learn from. In unsupervised learning our data set was a collection of attributes, in supervised learning our data set was a collection of attribute-label pairs, and, lastly, in reinforcement learning our data set was a collection of observation-action-reward tuples. We decided to use reinforcement learning because it is the best way to simulate the human decision process.

The goal of reinforcement learning is to learn a policy, which is essentially a mapping from observations to actions. An observation is what the training subject can measure from its environment and an action which it will perform depending on the circumstances. The subject is trained to learn a policy that maximizes its overall rewards. We provide rewards (positive and negative) indicating how well it is doing on completing the task.

II.1a. Reinforcement learning cycle

The main disadvantage of such an approach is the learning time. At the very start our training subject will try random things to figure out what gives him the most reward points which with

a complex environment can take ages. To improve that we decided to combine bare reinforcement learning with GAIL (Generative Adversarial Imitation Learning) and self-play.

GAIL uses an adversarial approach to reward your Agent for behaving similar to a set of demonstrations. Its techniques aim to mimic human behavior in a given task. An agent (in this case - AI opponent) is trained to perform a task from demonstrations by learning a mapping between observations and actions from a real human.

Self-play is where the algorithm learns by playing by itself without requiring any direct supervision while the performance is still controlled by a system of rewards (Reinforcement learning).

2. Tools and frameworks

As this is a Unity game, so as a library we have chosen Unity Machine Learning Agents Toolkit (ML-Agents), which is an open-source project that enables games and simulations to serve as environments for training intelligent agents. The toolkit is built upon another open-source library called PyTorch. It benefits in capability of using TensorBoard for collecting and analyzing all the data from the learning process. It allows the visualization of certain agent attributes (e.g. reward) throughout training which can be helpful in both building intuitions for the different hyperparameters and setting the optimal values for the Unity environment.

It provides Python API for training using reinforcement learning and imitation learning which we planned to use. Due to its official support from Unity engine contributors we decided to use it as it has the best compatibility with our base project and relatively easy to use interfaces.

3. Datasets

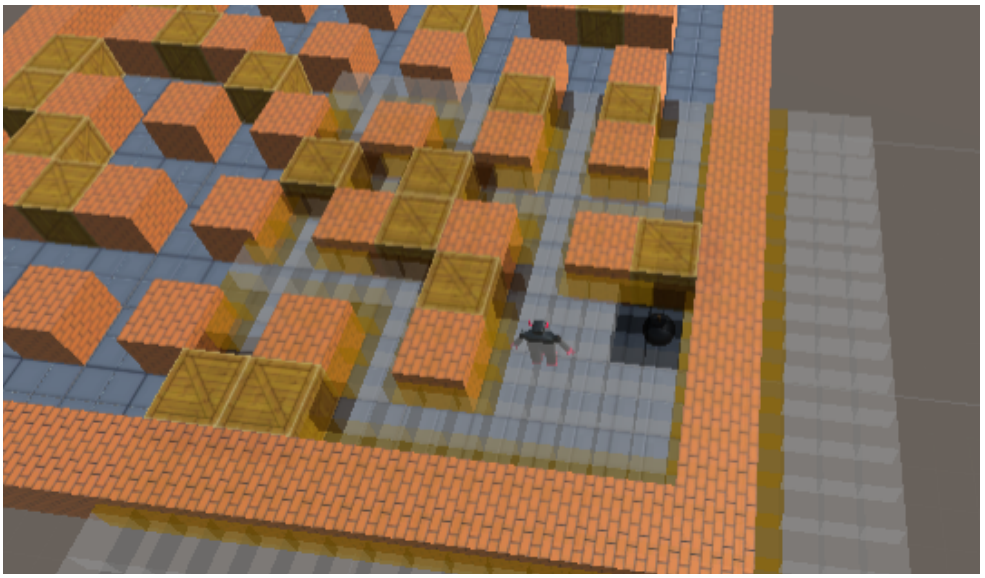
Our goal environment for the bot is a game made with Unity3D game engine, so the best solution is to collect data from the game at runtime. Bot should have access to similar data as a real human player does. We can separate the data that should be collected into two subsections:

- character data: amount of possible bombs to drop, amount of health points, current speed, information about distance to nearby boxes, relative distance and direction towards an opponents
- game data: player starting position, amount of competitors, layout of the game map indicating which boxes are breakable and which are not

All the data (observations) were provided by the agent sensors which are a part of ML-Agent library. More information about the sensors was described in the Internal specification section.

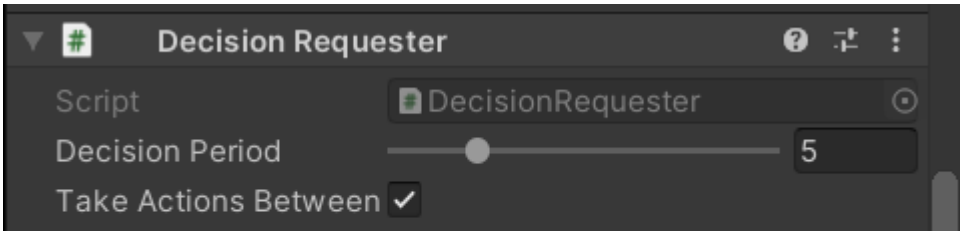
III. Internal specification

The ML-Agent library comes with a variety of different sensors. One of the sensors we used was Grid Sensor from which bot is provided with data about nearest objects and their type (bomb, box, player or wall). The sensor provides a set of observations on base which the Agent will be taking decisions about what to do next. We configured the sensor to scan the environment in the range similar to the range of real human player's field of sight.



IIIa. Preview of grid sensor

Every trained agent has its own script attached called DecisionRequester by which we are able to control the frequency of making decisions and enable/disable taking actions in between them.



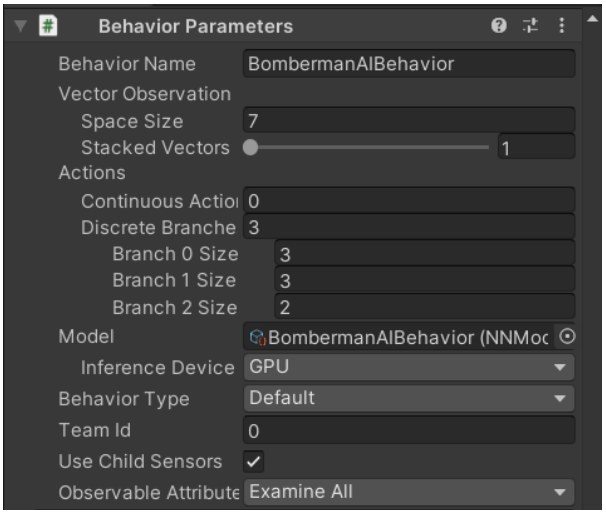
IIIb. Decision Requester script in Unity3D Inspector

To process the decision an observation vector is provided into another script called BomberAgent. Method called *CollectObservations* let us pass additional values to the observations, we added there amount of bombs left, agent current speed(velocity) and its current position:

```
public override void CollectObservations(VectorSensor sensor) {
  // add current position as an observation
  sensor.AddObservation(
    gameObject.transform.InverseTransformPoint(target.transform.position)
  );
```

```
  // add speed as an observation
  sensor.AddObservation(gameObject.GetComponent<Rigidbody>().velocity);
  // add amount of bot bombs as an observation
  sensor.AddObservation(gameObject.GetComponent<Character>().bombs);
}
```

Afterwards, all collected observations are passed to python script which parse them and create action data then pass it to C# method. Amount od action buffers, their size and type (continuous or discrete) can be set in in Discrete Branches section fo Behavior Parameters script (figure IIIc). With this action data we can use discrete values to convert numeric decisions into some in game actions, for example moving bot or placing a bomb.



IIIc. Behavior Parameters script in Unity3D Inspector

For movement we are using two discrete action branches, one for movement in X axis, second for Z axis. The branches are declared to contain values ranging from 0 to 2. It allows us to convert these values into force value causing the bot to move. Example for X axis values:
0 - move left,
1 - don't move,
2 - move right.
Similarly the values are used for bomb placing: 0 - don't place a bomb, 1 - place a bomb.

The calculation and parsing action buffers is performed in OnActionReceived method. After performing an action we are also checking if any reward could be given. For example we are giving a big negative reward to the bot if it dies, or a very little negative reward for every decision it take to accomplish the goal.

```
public float timeReward = -0.0001f;
public float deathReward = -1.0f;
public float finishReward = 1.0f;

...
```

```
public override void OnActionReceived(ActionBuffers actionBuffers) {
  var moveX = actionBuffers.DiscreteActions[0] - 1;
  var moveZ = actionBuffers.DiscreteActions[1] - 1;
  var placeBomb = actionBuffers.DiscreteActions[2];

  ...

  if (character.isDead()) {
    SetReward(deathReward);
    EndEpisode();
  }
  if (targetFound) {
    AddReward(finishReward);
    EndEpisode();
    targetFound = false;
  }
  AddReward(timeReward);
}
```

The training is separated into small episodes. An episode ends every time bot dies, maximum time passes or bot accomplishes current goal. Every time the episode begins whole environment is being reset to the start state: bot is moved to its starting position, new destroyable boxes are generated, player statistics are being reset.

```
public override void OnEpisodeBegin() {
  gameController.FillMap();
  // move bot to starting position
  Vector3Int offset = Vector3Int.FloorToInt(
    this.gameObject.transform.parent.gameObject.transform.position
  );
  gameObject.transform.position = new Vector3Int(positionX, 1, positionY)+offset;
  // reset player statistics
  gameController.ResetPlayer(gameObject.GetComponent<Character>());
}
```

Next important part of the project is training configuration yaml file, where we can provide set of configurations (including neural network parameters) for the behavior to be trained.

```yaml
BombermanAIBehavior:
    trainer_type: ppo

    hyperparameters:
      batch_size: 512
      buffer_size: 4096
      learning_rate: 3.0e-4
      learning_rate_schedule: linear
      # PPO-specific hyperparameters
      beta: 5.0e-4
      epsilon: 0.2
      lambd: 0.95
      num_epoch: 5

    # Configuration of the neural network
    network_settings:
      normalize: false
      hidden_units: 256
      num_layers: 2

    reward_signals:
      # environment reward
      extrinsic:
        gamma: 0.99
        strength: 1.0

      # curiosity module
      curiosity:
        strength: 0.02
        gamma: 0.99
        encoding_size: 256
        learning_rate: 3.0e-4

    max_steps: 3000000
    time_horizon: 128
    summary_freq: 5000
```

We decided to use ppo as a training algorithm as we observed that it provides a better convergence and performance rate than other techniques. Its main disadvantage is sensitiveness to changes and getting all the parameters right. It is configured for faster learning in opposite of bots stability (causing bot to do more “random” things)

Neural network hidden units corresponds to how many units are in each fully connected layer of the neural network. As the correct actions is a relatively easy combination of the observation inputs we set it to value of 256 (as default 128). For similar reasons we are only using 2 hidden layers of the neural network.

We are also using a mechanism of curiosity which should encourage bots to explore map at the begging of training, although we its strengths is very small the results were visible.

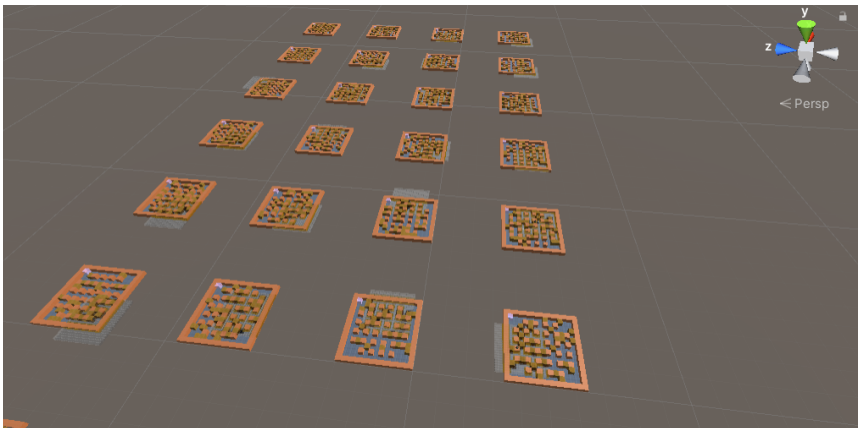
Detailed description of every property of the configuration can be found here.

At the end of every training session a NNModel as .onnx file is generated, which is something we can call bots “brain”. It can be used as a brain in standalone application or it can be used for further training. Model can be loaded to the bot(agent) through Model section of Behavior Parameters script (figure IIIc).

IV. External specification

In this project we can define two different user environments, one, used for bot training and optimization, and second, the game itself.

Bot training system was created using the ML-Agents toolkit. First and biggest part of it is the game environment specifically redesigned, so it provides the most suitable and optimal learning conditions for the agents (bots/AI). That means, removing all unnecessary parts designed for user interaction (control schemes, user interface) and other parts unsuitable for early stages of learning (advanced game mechanics). In return, this much more lightweight version of the game allows us to implement multiple pairs of agents plus learning environments into a single Unity project, which considerably speeds up the learning process.

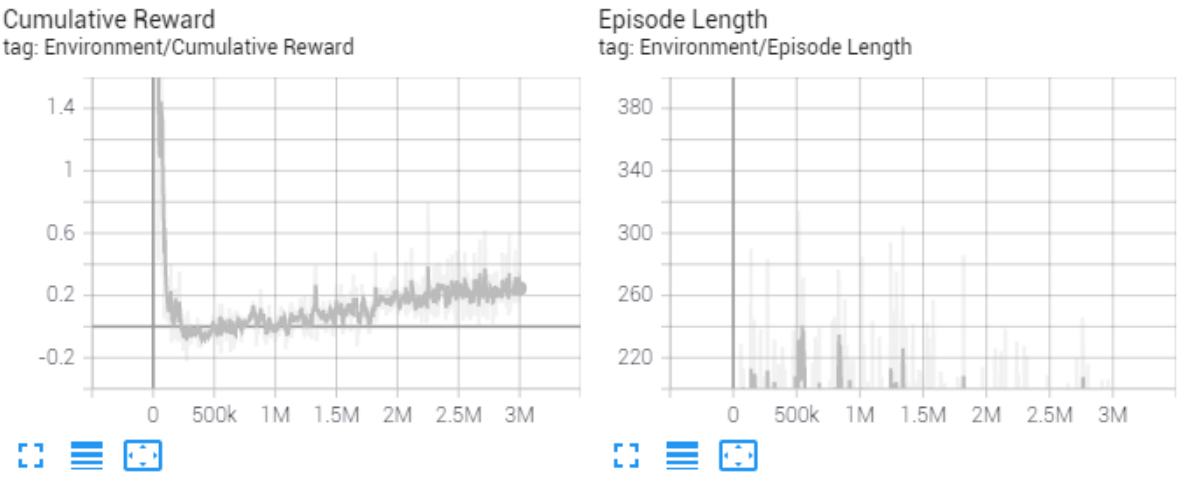


IVa. Example of multiple learning environments

When the scripts and rewards are ready to test, we use the console to run a script that creates a neural network and begin recording learning episodes. Basic reward points results are displayed in the console after each episode, but the ML-Agents also provides the ability to generate TensorBoard in which more specific data and trends can be viewed.

```
            learning_rate:      0.0003
            encoding_size:      256
        init_path:      results\finalV3\BombermanAIBehavior
        keep_checkpoints:       5
        checkpoint_interval:    500000
        max_steps:      3000000
        time_horizon:   128
        summary_freq:   5000
        threaded:       True
        self_play:      None
        behavioral_cloning:     None
[INFO] Initializing from results\finalV3\BombermanAIBehavior.
[INFO] Starting training from step 0 and saving to results\finalV4\BombermanAIBehavior.
[INFO] BombermanAIBehavior. Step: 5000. Time Elapsed: 51.940 s. Mean Reward: 2.626. Std of Reward: 1.448. Training.
c:\users\dawid\appdata\local\programs\python\python39\lib\site-packages\mlagents\trainers\torch\utils.py:309: UserWarnin
g: This overload of nonzero is deprecated:
        nonzero()
Consider using one of the following signatures instead:
        nonzero(*, bool as_tuple) (Triggered internally at  ..\torch\csrc\utils\python_arg_parser.cpp:882.)
  res += [data[(partitions == i).nonzero().squeeze(1)]]
[INFO] BombermanAIBehavior. Step: 10000. Time Elapsed: 83.150 s. Mean Reward: 4.332. Std of Reward: 3.300. Training.
[INFO] BombermanAIBehavior. Step: 15000. Time Elapsed: 94.317 s. Mean Reward: 2.659. Std of Reward: 1.848. Training.
[INFO] BombermanAIBehavior. Step: 20000. Time Elapsed: 126.560 s. Mean Reward: 3.198. Std of Reward: 4.193. Training.
[INFO] BombermanAIBehavior. Step: 25000. Time Elapsed: 146.932 s. Mean Reward: 3.767. Std of Reward: 3.412. Training.
[INFO] BombermanAIBehavior. Step: 30000. Time Elapsed: 173.660 s. Mean Reward: 2.661. Std of Reward: 4.161. Training.
[INFO] BombermanAIBehavior. Step: 35000. Time Elapsed: 188.940 s. Mean Reward: 4.678. Std of Reward: 5.833. Training.
[INFO] BombermanAIBehavior. Step: 40000. Time Elapsed: 214.950 s. Mean Reward: 2.107. Std of Reward: 5.226. Training.
[INFO] BombermanAIBehavior. Step: 45000. Time Elapsed: 239.666 s. Mean Reward: 1.332. Std of Reward: 2.227. Training.
[INFO] BombermanAIBehavior. Step: 50000. Time Elapsed: 263.304 s. Mean Reward: 2.413. Std of Reward: 3.258. Training.
```

IVb. Example results of a script run

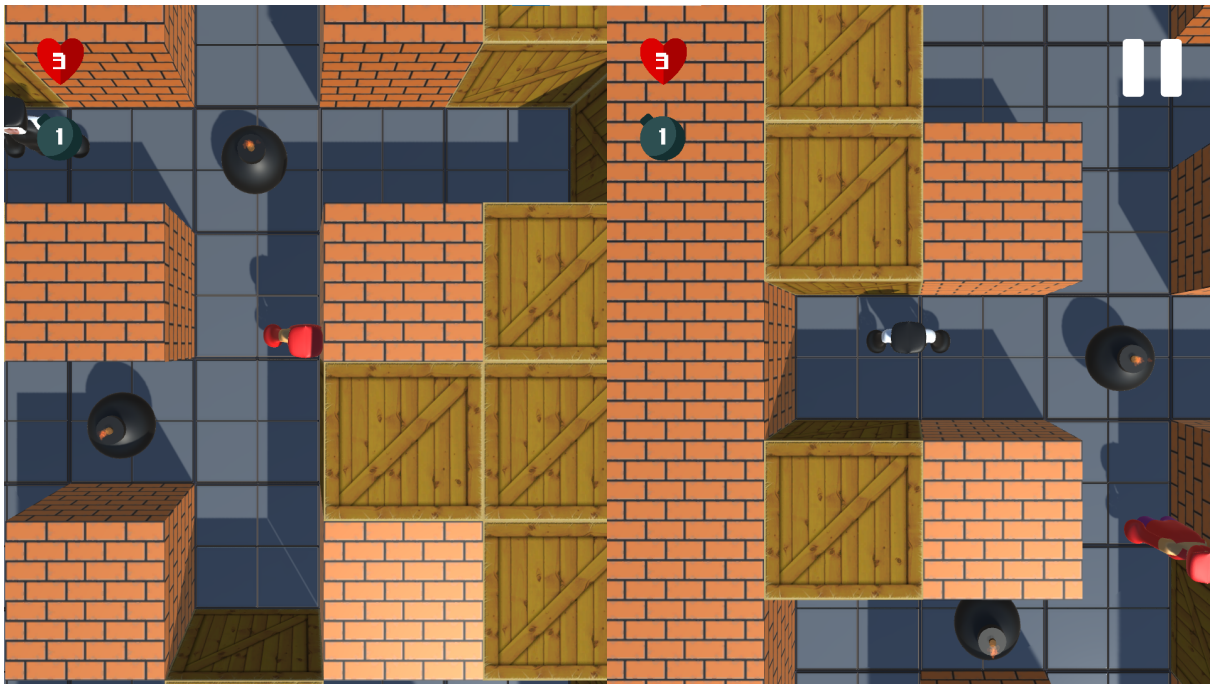


IVc. TensorBoard Graph example

As for the game environment. It was almost entirely done beforehand, and only needed to have the new bots implemented. Game plays like a typical bomberman, with multiple players trying to defeat the opponent by placing bombs. Game supports different control schemes (controller and different keyboard layouts), and different play modes (player vs player and player vs AI). Gameplay looks like this:

IVd. Main menu

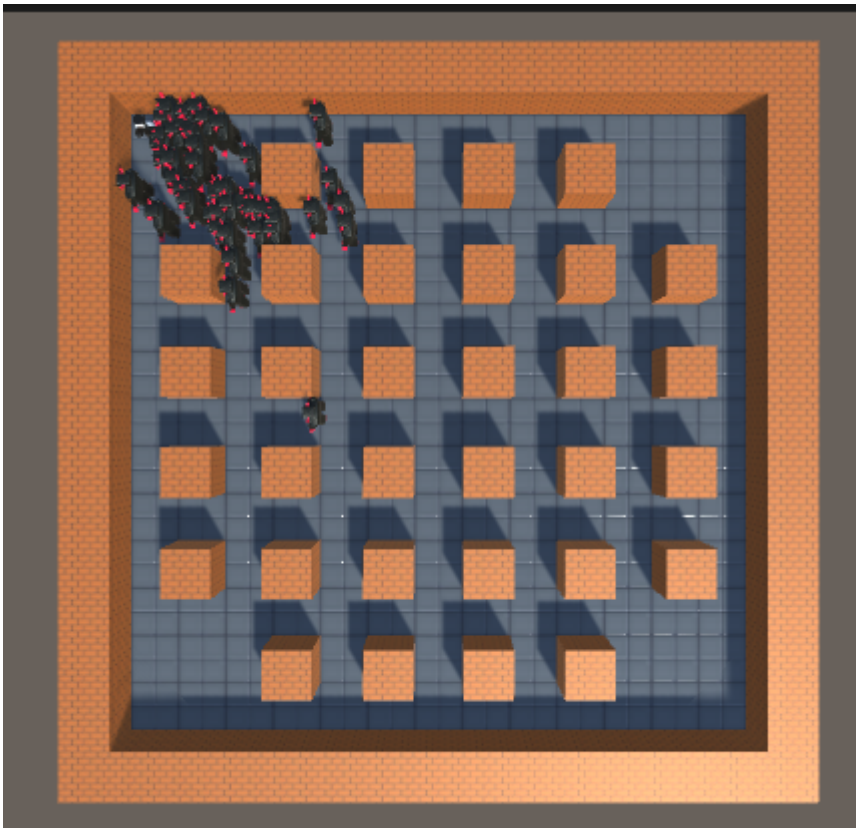
IVe. Classic single player (PvAI)



IVf. Classic multiplayer (PvP)

V. Experiments

Our very first environment was a simple game map with around 30 agents on it. All agents' goal was following the moving player. After the environment was prepared we could begin training. Before we could start imitation learning or self-play we mostly spent time making him move properly or stop him from breaking the game. The rewards were very simple: every second of not following the player (not moving towards him) caused a slight negative reward (-0.1). The training was surprisingly good - satisfying results were achieved in only 6 generations.

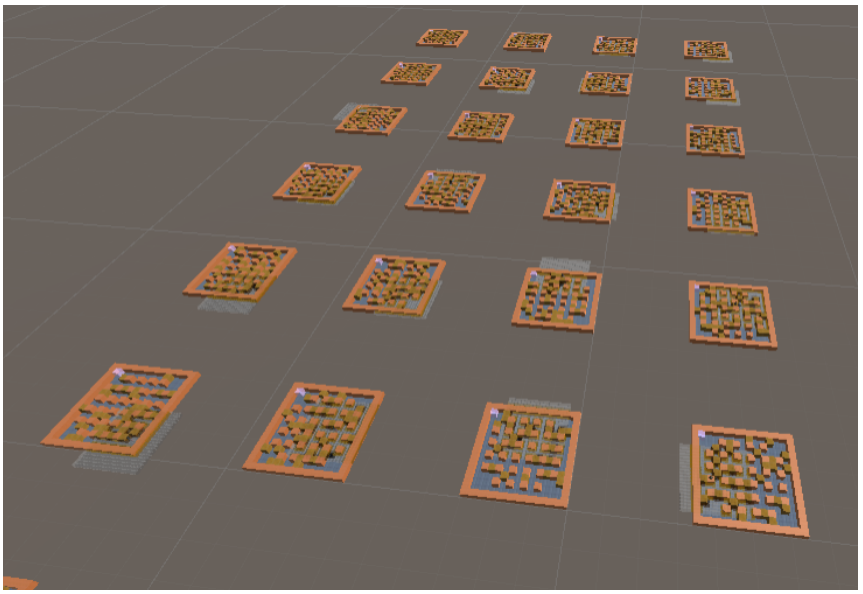


Va. First training environment

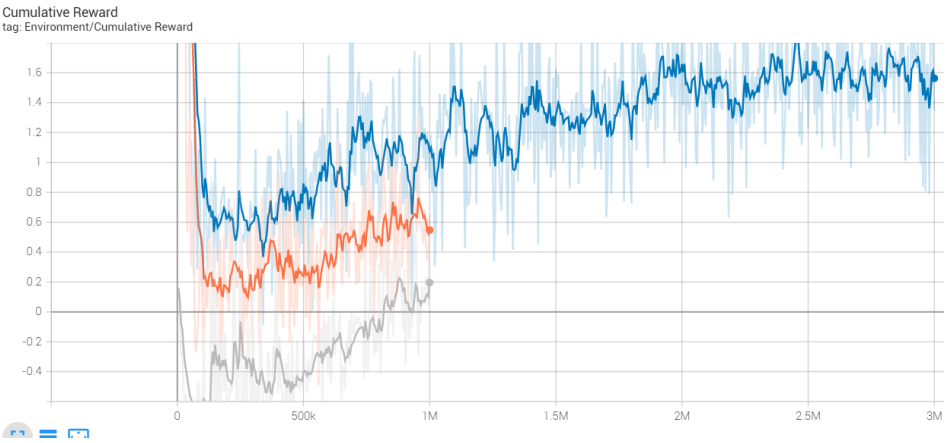
After that we decided to teach them how to actually play the game by giving them the ability to plant bombs to move through a blocked map. More focus went towards reinforcement learning. Agents were taught through optimization of the reward system.

Every agent had his own environment and its goal was finding a way and moving to the opposite corner of the map. Every episode we randomly spawned boxes on its way to teach them how to use bombs to optimize its movement towards the target. The idea of imitation learning was abandoned due to its inefficiency. We also extended the time of learning (up to 12 hours) which gave us the best results. We think we did not face the issue of overtraining, so maybe the time could be extended more for even better results. We also added curiosity to agents' behavior in order to help them not be afraid of exploring the map. We were experimenting a lot with reward values, but the best results we came up with were made using the following ones:

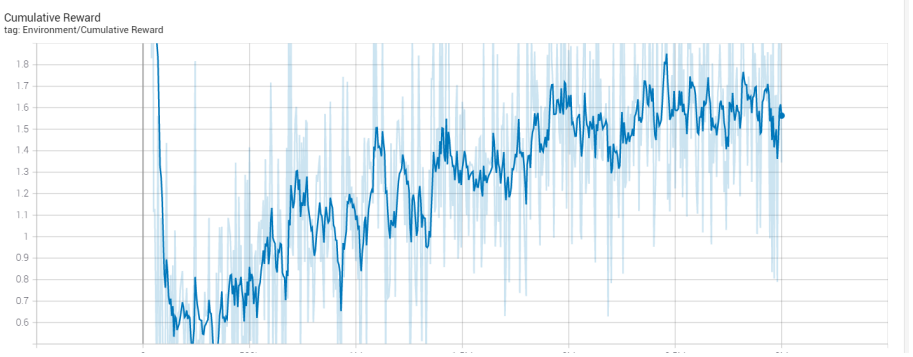
- Time: -0.001
- Moving forward: 0.1
- Putting bomb properly: 0.1
- Being near bomb: -0.05
- Death: -1
- Reaching target: 1



Vb. Multiple training environments

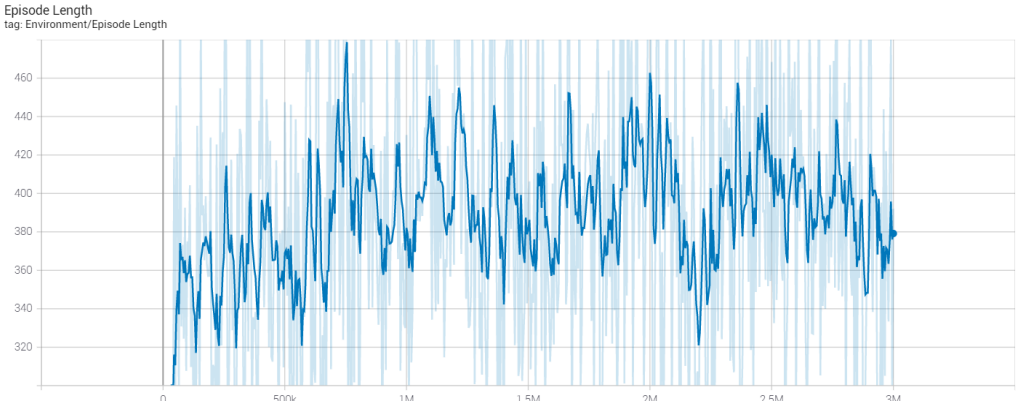


Vc. Cumulative rewards visualizations comparison

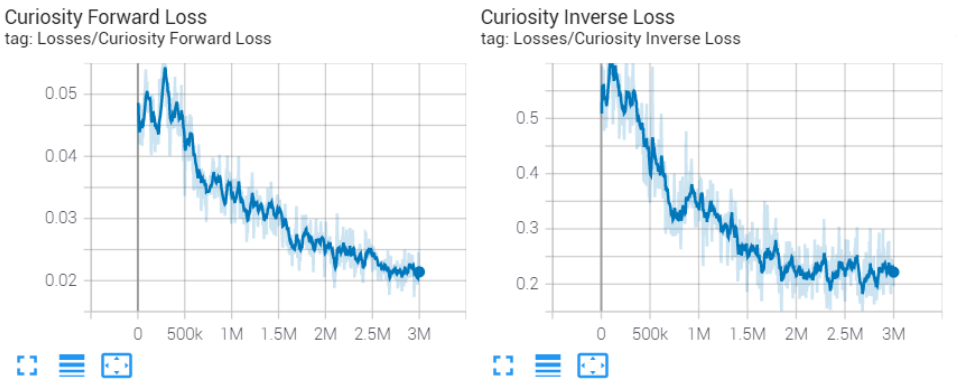


Vd. Cumulative rewards best results

As you can see by the rewards graph, with each episode the score was steadily increasing, although it can be seen to be nearing a plateau towards the end, we didn't really feel like this was the case and we would definitely try to maximize training time to its limit. Side note: the big spike seen at the beginning is due to miscalibration of rewards and is not relevant to the results.



Vd. Episode length on best run



Ve. Curiosity visualisations on best run

Episode length here being mostly static in our opinion is mainly thanks to the curiosity, which job was to increase agents' survivability, because they would be more inclined to explore the map rather than just going straight and blindly planting bombs causing them to blow themselves up. Curiosity loss on the other hand would be proof that it was indeed working and agents were getting familiar with the map and finding more optimal routes rather quickly.

For our last method of training we wanted to use self play training. We changed the environment a little to simulate a real game environment, so two agents were spawned in opposite corners at the beginning of every episode. The randomly spawned boxes on the way between them remain the same, but we added functionality of power ups. The agents could have picked them up increasing its health, amount of available bombs, speed or bomb explosion range. We decided not to change rewards much, we just changed the main goal from reaching the target to dealing damage to the opponent, so the agents had a positive reward every time their opponents health was decreasing (but only if the agent caused the decrease). The results were not that satisfying as the previous ones. Agents learnt pretty well how to avoid each other's bombs, but in attacking their opponents they were doing just fine.

VII. Summary

When it comes to results, it's hard to present anything concrete given the nature of our project. Environment of our AI is a video game, so its "effectiveness" is valued by how much of a challenging opponent it is for the player. This is extremely subjective, so we can't present it as something "scientific". The best we can offer is the result of us playing against it, which is a win rate of around 20-30%.

This result is far from perfect, although we feel our goal has been mostly met and because of project size, improving it would mostly require very small optimizations of the reward system and a lot of hours dedicated to running training, which we believe is not really concerning the ideas of this project that much.

This project has been a great opportunity for us to expand our knowledge on subjects regarding artificial intelligence, such as neural networks or reinforced learning. We also discovered a lot of beginner-friendly tools and a large community, dedicated to the expanding field of machine learning.

VI. References

ML-Agents Toolkit documentation:
https://github.com/Unity-Technologies/ml-agents/tree/release_18_docs/docs
TensorBoard documentation: https://www.tensorflow.org/guide
Proximal Policy Optimization: https://openai.com/blog/openai-baselines-ppo/
https://medium.datadriveninvestor.com/which-reinforcement-learning-rl-algorithm-to-use-where-when-and-in-what-scenario-e3e7617fb0b1

VII. Project

https://github.com/dHerc/Bomberman/tree/release0.3