Reinforcement Learning - Project report
RL agent for Bomberman

Robin Charleuf
Hugo Thevenet

April 4, 2024

1 Introduction

This project aims to develop a reinforcement learning (RL) agent capable of playing the classic game Bomberman at a competitive level. Bomberman is a known game where the agents are in a maze and they can drop bombs. The goal is to kill the other agents in the exxplosion and to be the last survivor.
A game like Bomberman offer a well-defined environment with clear objectives and immediate feedback, making it ideal for exploring and showcasing RL techniques.

In the real Bomberman game, some blocks in the maze are destroyable. To simplying, we did not include this feature.

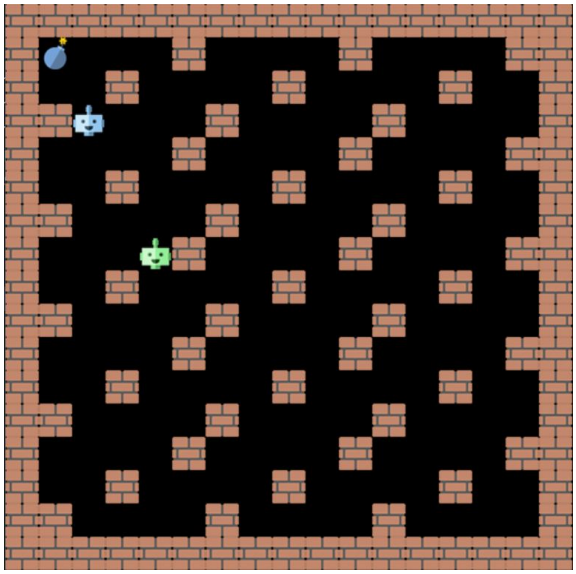

Figure 1: Our implementation of the game with two agents

2 Description of the environment

2.1 Introduction

We created the whole environment by ourselves. We did not copy any repository. We just took the *.png icon for displaying the game from the gym library. There are two modes in our game : displaying mode and classical mode. The display mode is using pygame to display. To run the code faster, we added a version without any display.

Basically, all of the state information are stored in a STATE dictionary. We use this dictionary to get an observation for each agent and they use this observation to take a decision of an action.

Then, the state is updated according to the actions. If the display is on, we use the dictionary to display everything on a screen.

Here is the pseudo code of the main loop of our code

Algorithm 1 Game Loop

```
while running do
    if display then
        use displays functions to display the state
        time.sleep(0.1)
    end if
    actions ← []
    for i ← 0 to len(AGENTS) do
        agent ← AGENTS[i]
        obs_agent ← get_observation(i, state = STATE)
        action ← agent.act(obs_agent)
        actions.append(action)
    end for
    (rewards, next_state, done) ← perform_actions(actions, display)
    for i ← 0 to len(AGENTS) do
        all_rewards[i].append(rewards[i])
        obs_agent ← get_observation(i, state = STATE)
        next_obs_agent ← get_observation(i, state = next_state)
        agent.update_policy(obs_agent, actions[i], rewards[i], next_obs_agent, done)
    end for
    STATE ← deepcopy(next_state)
    if done then
        winner ← get_winner()
        running ← False
    end if
end while

function MAIN(** args)
    world ← Bomberman(**args)
    _, rewards, _ ← world.run()
end function
```

2.2 State Space

The state space is a dictionary containing all of these elements. It represents exhaustively everything useful in the game.

The state space contains:

- **Data Agents**: This contains information about the agents present in the environment. Each agent has attributes such as its position (`agent_x`, `agent_y`), whether it's alive (`alive`), the number of bombs available (`bombs_available`), the type of agent (`type_agent`), and a unique identifier (`agent_id`). For example:
  - Agent 1 (Blue):
    - Position: (2, 1)
    - Alive: True
    - Bombs Available: 0
    - Type: Q-learning
  - Agent 2 (Green):
    - Position: (3, 6)

    - Alive: True
    - Bombs Available: 0
    - Type: Random

- **Data Bombs**: This represents the bombs present in the environment. Each bomb is associated with the agent that placed it (`agent_id`), the time until explosion (`bomb_time`), and its position (`bomb_x`, `bomb_y`). For example:
  - Bomb 1 (Blue):
    - Time until explosion: 10
    - Position: (2, 3)

- **Data Maze**: This represents the layout of the maze environment. It consists of a grid where '#' represents a wall and ' ' represents an empty space.

- **Data Explosions**: This represents explosions caused by bombs. Each explosion is associated with the agent that caused it (`agent_id`), the time until the explosion ends (`explosion_time`), and the zone affected by the explosion (`explosion_zone`). For example:
  - Explosion 1 (Blue):
    - Time until end: 3
    - Affected zone: [(1, 2), (2, 2), (2, 3)]

2.3 Observation space

To make a decision, the agent has to observe the state. We encode the STATE dictionary into a matrix.

So we created a function used by the agent. This function takes the observation dictionary as input and encodes the maze environment into a numerical representation, in the personal point of view of the agent.

- If the tile represents a wall ('#'), it assigns a value of 1 to the corresponding position in the encoded maze.
- If the tile is empty space, it assigns a value of 0.
- If there is a bomb present, it assigns a value of 4.
- If there is an explosion present, it assigns a value of 5.
- If there is an enemy present, it assigns a value of 3.
- If the tile represents the agent's current position, it assigns a value of 2.
- If the agent is on the bomb tile, it assigns a value of 6.

For example, for a state representing this moment of the game:

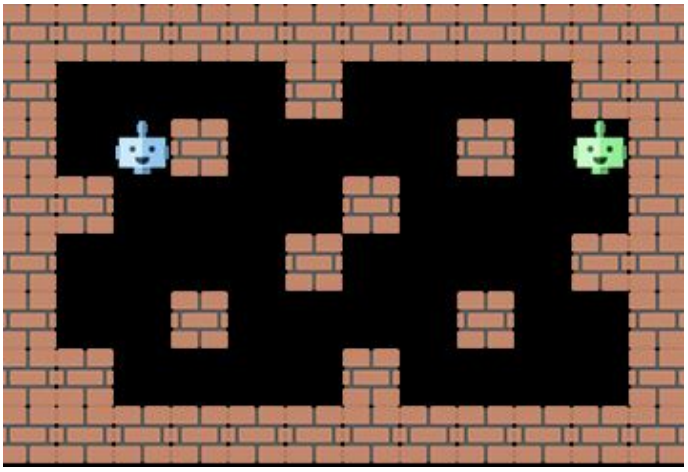

Figure 2: Maze State example

The encoded state for the blue agent will be like this:

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 2 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 3 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Then, we give this flattened matrix to the agent as a tuple, and it gives us an output representing the action taken. We did not include any other information, like memory, to the agent, because this state observation is fully exhaustive and the agent can make the best decision with this knowledge.

We add to this observation state the one-hot encoding of the possible actions for our agent given the current state of the game, which consists of 6 possible actions: the 4 directions, the "planting a bomb" action and the "do nothing" action. For example, if the agent has a wall to its left but can go anywhere else and also drop a bomb, the encoding will be [0,1,1,1,1,1]. This has been added as our agent had tendencies to perform actions that he wasn't allowed to do, especially dropping bombs in series.

With this section, we understand that the observed state can be very high dimensional for the agent. So, we'll have to create complex agent which could handle this high dimensional state as an input.

2.4 Action space

Our action space has a length of 6:

- Move Left (Action 0)
- Move Right (Action 1)
- Move Up (Action 2)
- Move Down (Action 3)
- Drop Bomb (Action 4)
- No Operation (Action 5)

2.5 Rewards

To give the rewards, we need to get some information about the agent and its self awareness. These are the metrics we compute :

- **Action Taken** : If the agent has decided to move, do nothing, or drop a bomb.
- **Closest Enemy Distance** : It calculates the Manhattan distance to the closest enemy agent that is alive and not itself.
- **Closest Bomb Distance** : It computes the Manhattan distance to the closest bomb.
- **Closest Explosion Distance** : It determines the Manhattan distance to the closest point of explosion.
- **Time** : It determines the time since the beginning of the game. If the game becomes too long, we can decide to stop it.

Our reward system is based on the following ideas:

- We want to drop a bomb close to the enemy
- We want to dodge bombs and explosions
- We want to prevent the agent to perform actions he's not allowed to do

This is achieved through the following reward system

Action wanted	Reward	Points awarded
Drop bomb close to the enemy	Positive reward when **approaching the enemy**, based on the difference of distance between two steps	**20 * (new closest distance to enemy – previous closest distance to enemy)**
	Positive reward when **dropping the bomb close to the enemy**	**Max(0, 5 – bomb distance to enemy) * 100**
Dodge bombs & explosions	• Positive reward when **distance between the agent and the bomb increases** and the agent is too close to the bomb, negative if the distance decreases. • Same idea for the explosions	If distance to bomb < 5 / explosion < 3 : **(new closest distance – previous closest distance) * 100**
Perform actions that he's allowed to do	Positive reward if the agent can perform the action, negative otherwise	**+/- 100**

Figure 3: Reward system

3 Agents implemented and results

For the experiments, we trained each of our agents like this :
- One agent is the learning agent. It's the implemented agent.
- The competitor agent isn't a learning agent but an randomly moving agent. It is the target.

3.1 1st step : the tracking agent

In this section, we present our first implemented version. We struggled to tune the rewards policy, so as a first try, we implemented a agent that follows the other agent. The closer he is to the other ones, the highest rewards he gets. We also made the maze smaller so even a Q-learning agent could learn faster.

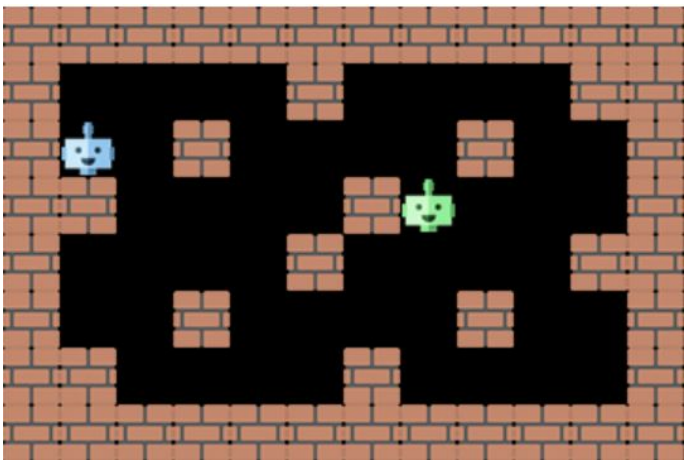

Figure 4: The tracking agent environment

3.1.1 Q-Learning

We implemented a simple Q-Learning agent with an exploration factor of 10%. Even with a smaller maze, the Q-Learning agent takes a large number of environments to learn given the complexity of the game, but exhibits a clear learning pattern, with total rewards increasing throughout the learning.

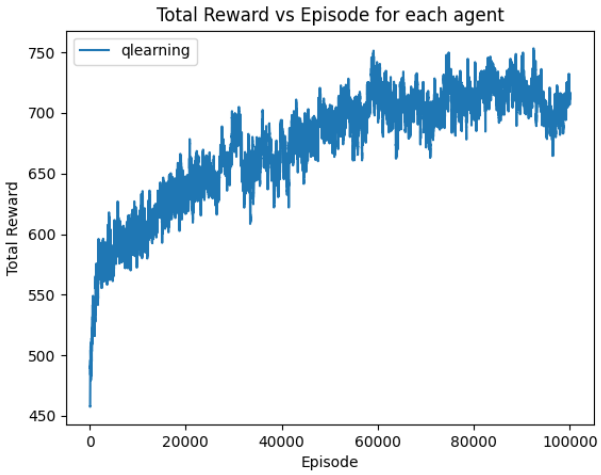

Figure 5: Follower Q-Learning agent rewards by episodes

3.1.2 DQN

We implemented a DQN agent with the following parameters:

- 4 layers of sizes (102,96),(96,64),(64,32),(32,6) and ReLU activation layers in between.
- 5% exploration

- Adam Optimizer with $\gamma = 0.1$ and a 0.005 learning rate.

Figure 6: Follower DQN agent rewards by episodes

This agent learns much faster than the Q-Learning agent, which was expected as the DQN agent learns to approximate the Q

3.2 2nd step : the bomber agent

3.2.1 Q-Learning

Given the complexity added by adding the possibility to drop a bomb, our Q-Learning agent wasn't successful, which could be expected given the very high number of episodes needed only for the follower Q-Learning Agent. This is one more proof that Q-Learning reaches limits when complexity increases.

3.2.2 DQN

We implemented a DQN agent with the following parameters:

- 4 layers of sizes (102,96),(96,64),(64,32),(32,6) and ReLU activation layers in between.
- 5% exploration
- Adam Optimizer with $\gamma = 0.1$ and a 0.005 learning rate.

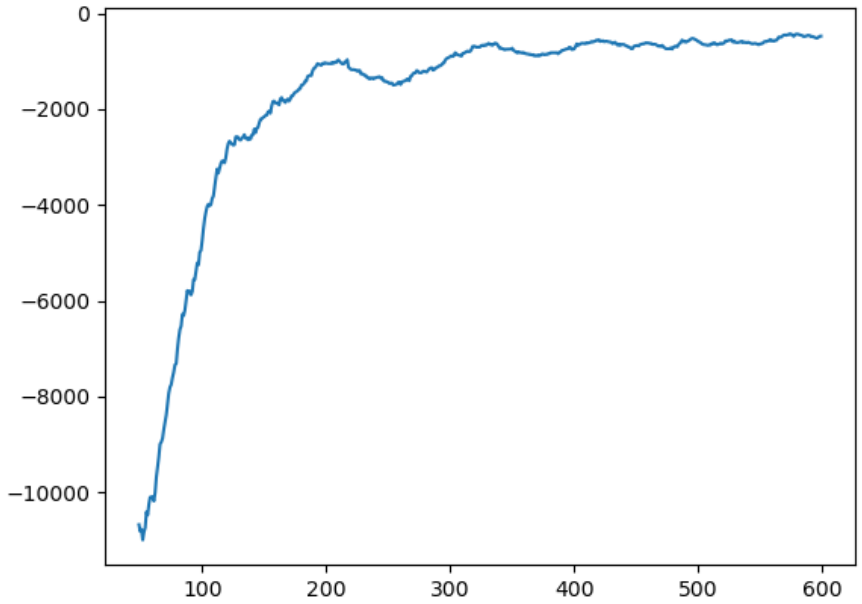

Figure 7: Bomber DQN agent rewards by episodes

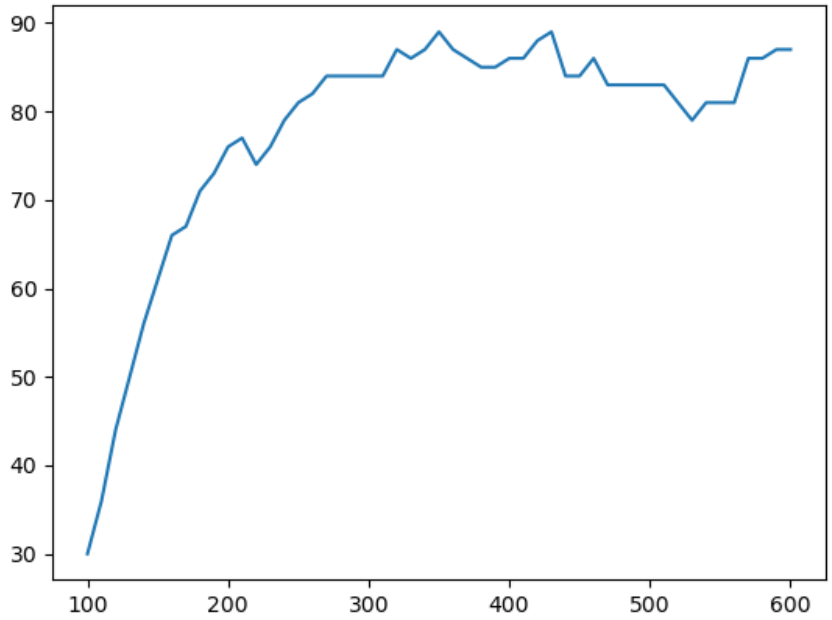

Figure 8: Win rate over the last 100 games by episode

Our agent exhibits a clear learning pattern, showing capacity to learn how to bomb a random agent in the maze, with increasing win rate over the episodes. However, it appears to reach a plateau, at around 85% win rate. We believe it to be due to the fact that our agent lacks of a true "understanding" that bombing the random agent will give him rewards, and rather has incentives to do things that will ultimately lead to this goal.

4 Conclusion

In this project, we created a scratch-built play environment. The game is based on a dictionary, which is updated every turn. This dictionary serves as a database for the agents' display and decision-making. We then created a simplified game environment with one agent simply chasing the other. This simpler system enabled us to code our agents efficiently and develop a first version of the reward system. Next, we added the ability for an agent to drop a bomb. We trained our agents to kill a random agent. To do this, we had to adapt the reward system. We obtained very good results with our agents. If we'd had to continue with the project, we would have been able to pit several types of intelligent agent against each other, and not just one agent against a random agent.

Our code and environment is completely original and was created by us from scratch. It is available here