Reinforcement Learning Agent for Bomberman

Paulo Pereira
DETI
University of Aveiro
Aveiro, Portugal
paulogspereira@ua.pt
98430

João Reis
DETI
University of Aveiro
Aveiro, Portugal
joaoreis16@ua.pt
98474

Artur Romão
DETI
University of Aveiro
Aveiro, Portugal
artur.romao@ua.pt
98470

***Abstract*—This report relates to the development of a reinforcement learning agent with the purpose of playing the game Bomberman. This project was developed within the scope of the Intelligent Systems II course at the University of Aveiro, 2022/2023.**

***Index Terms*—artificial intelligence, DQN, Bomberman, Reinforcement learning**

I. Introduction

The goal of this project was to develop an autonomous agent that played the game Bomberman using the material taught during the Intelligent Systems II course, namely Python programming, agent architectures, and search techniques for automated problem-solving. Our implementation relies on a reinforcement learning algorithm, using the library of Tensorflow.

Reinforcement learning is a machine learning approach where an agent learns to make decisions and take actions in an environment to maximize a reward signal. The agent interacts with the environment, receiving feedback in the form of rewards or penalties based on its actions. Through trial and error, the agent adjusts its behavior to learn the optimal actions that lead to the highest rewards. This is achieved through the use of algorithms, such as DQN learning, as we implemented in this work, which enables the agent to learn from past experiences and make informed decisions based on the expected rewards associated with different actions.

In this report, we will document the implemented solution and explain the failure of this implementation.

II. Bomberman

A. Historical background and how to play

Bomberman is a beloved video game franchise that originated in the 1980s. Created by Shinichi Nakamoto, the first Bomberman game was released for the MSX computer system. Players control Bomberman, a robot-like character, in a maze-like grid where the objective is to strategically place bombs to defeat enemies and clear obstacles. The game's addictive gameplay and multiplayer modes have made it a staple in the gaming industry. With its charming characters and intense battles, Bomberman continues to captivate players and provide an exciting gaming experience.

To play Bomberman, players navigate Bomberman through the maze using directional controls. The key element of the game is bomb placement, where players strategically plant bombs to trap enemies and destroy obstacles. Once a bomb is placed, it has a timed fuse before it detonates, causing an explosion in four directions. Players must carefully avoid the blast, enemy encounters, and maze structures. Power-ups can be collected to enhance Bomberman's abilities, and multiplayer modes allow players to compete against each other, adding a thrilling competitive aspect to the game. Bomberman's timeless gameplay, combining strategy and action, has made it a classic in the world of video games.

B. Our game setup

As to the setup used for this project, we were provided with a PyGame implementation starting out with a 51x31 grid dimensions, as visible in Fig. 1. Since there is a border around the grid and there are rows that have walls every two squares, we end up having 15 playable rows and 25 playable columns. There are also soft walls (can be destroyed by the agent) that are generated randomly across the map, with one of them containing the passage to the next level. The type and number of enemies are dependant on the level and its correspondent difficulty. The agent loses a life whenever he's hit by an enemy or by a bomb.

In order to move the character, the WASD keys were used and to detonate a bomb, the B key was used. The game runs on a loop, where in each iteration, a key press is sent to the server and processed. For the loop cycles where there is no user/agent input, an empty string is sent and nothing in done in the game. The agent advances to the next level whenever he manages to eliminate every enemy and get to the passage hidden in one of the soft walls.

The information regarding the grid state is communicated to the agent in the form of a dictionary that contains the:

- level
- current step
- timeout
- player's name
- score
- lives still left
- Bomberman's position
- bombs' positions (empty if none)
- enemies' name, id and position
- soft walls' positions

- passage to the next level position.

III. Agent Implementation

In the following section, we are going to explain our approach to implementing the agent. This implementation is found in the following files:

A. student script

Our student_training script is the script responsible for implementing the game loop in a way that our other classes can interact with it.

It implements a web-socket connection to the server, the communication dynamic between client and server can be simply defined with the flow of the server sending a state, the client replying with an action and the agent replying with the new state, and this keeps the game loop alive until the agent dies. Our script every time the agent dies should be prepared to reset the game and start another iteration, this is done in order to train. We also intended to implement the ability to save the training data to a file in order to create a deep enough model that when ran in a non-training environment our agent could hold his own.

Inside our loop, the first iteration is responsible for modifying the state in a way that our other classes can understand it better, as well as adding new information such as adding a tag to the enemies, that later will be used to calculate the reward for killing an enemy. Every loop will make a call with the **state** from the server to a class we developed called StateTuple that will simply convert the state data to tensors in order for TensorFlow to actually use it. That converted state is then sent to the helper function that works as the interface between the reinforcement learning implementation and the client. This function will return an integer between 0 and 5 that will be mapped to a key that will be the reply to the server.

This class also implements some game logic as it is a direct copy of the student.py script provided by the teacher.

B. RLagent script

This code represents an RLAgent class implementation for the game of Bomberman using TensorFlow Agents. The RLAgent class is responsible for creating and training a Deep Q-Network (DQN) agent to play the game.

The RLAgent constructor initializes the environment by converting a custom Bomberman environment into a TensorFlow environment. It defines the QNetwork architecture, which consists of a preprocessing layer and a fully connected layer. The agent is created using the DqnAgent class from tf_agents, specifying the time step and action specifications, the QNetwork, and an optimizer. A replay buffer is set up to store the agent's experiences.

The get_time_step method takes a state_tuple as input and creates a TimeStep object with the necessary components such as observations, rewards, discounts, and step types. It returns the constructed TimeStep. This function is used to create the structure that the policy will use in order to generate an action.

The get_next_time_step method takes an action as input and calls the environment's step function to obtain the next_time_step. The resulting next-time step is returned and will be used in the helper function to generate the Trajectories. The helper method is a utility function that helps in the interaction between the agent, environment, and replay buffer. It constructs the time step and receives from the policy the action using the constructed time step. It then calculates the next-time step and generates the trajectories and adds them to the replay buffer, it then returns the action the policy generated to the client.

IV. Environment

In this section, we will explain the implementation of the custom Bomberman game environment using TensorFlow and TF-Agents. The Bomberman_Environment class represents the environment in which the game is played. It inherits from the PyEnvironment class provided by TF-Agents.

The environment is initialized with the first state we receive from the served. It then defines the action and observation specifications.

The environment has a _reset() method to reset the environment to its initial state after each iteration from the main loop, as well as, the _step() method to take an action and calculate the next state. The method also defines the consequence of the action, in case it was negative action. We decided that if the agent hits a wall, tries to spam bombs, and the death of the agent would be the only negative rewards we would give. In terms of positive rewards, we decided to reward the agent for not dying in an episode, finding the goal, picking up powerups, and killing enemies, as well as, a time-based reward that would give points to the agent for each step he lived with decreasing rewards, in order to incentivize the agent to finish the level as fast as possible.

It also is responsible for ending the episode (either by reaching the exit, losing all lives, or reaching the timeout), and doing a soft reset each time the agent loses a life. This is done according to the server logic which always returns the agent to the [1,1] position and only moves enemies if they are within the vital space of the agent.

V. Problems

Our implementation is flawed, in the sense that, our agent can't run certain commands and when a loop finishes the training isn't able to actually start due to the fact our structures inside the replay buffer and the Trajectories that we add to the replay buffer aren't compatible.

These problems are mostly due to the fact that we only realized too late that the way the code that was being implemented would be incompatible with the native methods of Tensorflow. Other contributing factors such as the overall lack of time to dedicate to the project and our below-recommended number of group members were mainly to blame for this lack of awareness. We strongly believe that these problems could be mitigated with a total refactoring of the code structures, but unfortunately, that would take more time than is available.

Conclusion

In conclusion, we presented our attempt of developing a Reinforcement Learning agent for playing Bomberman. We explained our implementation, using the DQN algorithm to train the agent, and debated about the implementation issues we encountered due to the incompatibility of our code structures with the native TensorFlow methods. Despite these challenges, we think that we could overpass this problem by refactoring our code structures.

References

[1] Course material for Intelligent Systems II, University of Aveiro.