March, 2025

FPT UNIVERSITY

Apply Machine Learning Bomberman game using Reinforment Learning

Đặng Thành Tuấn
Hà Quang Hiếu

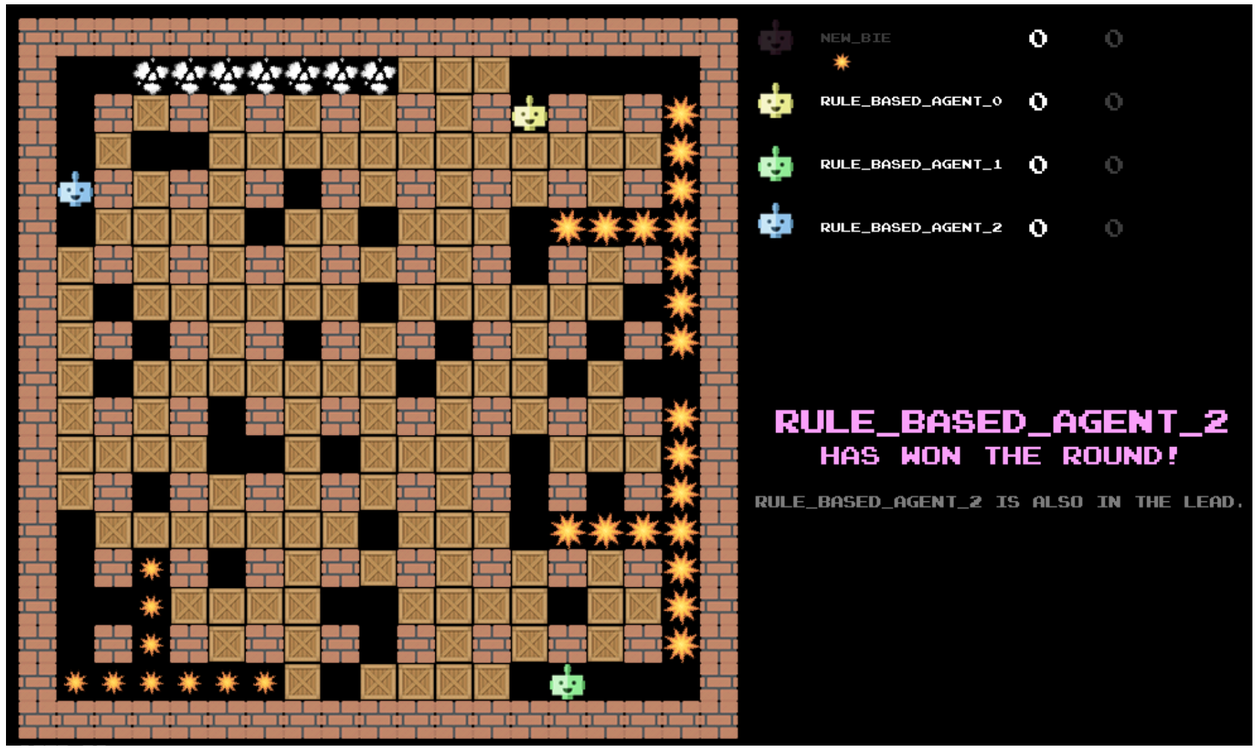
Contents

Introduction

This project uses Reinforcement Learning, specifically Q-learning, to train an agent to play Bomberman by learning behaviors like avoiding bombs, collecting coins, and defeating enemies.

The agent learns through trial and error without labeled data in a dynamic, grid-based environment, and the agent's performance is evaluated reward.

The project highlights the application of RL in developing intelligent game agents in real-time, rule-based settings.

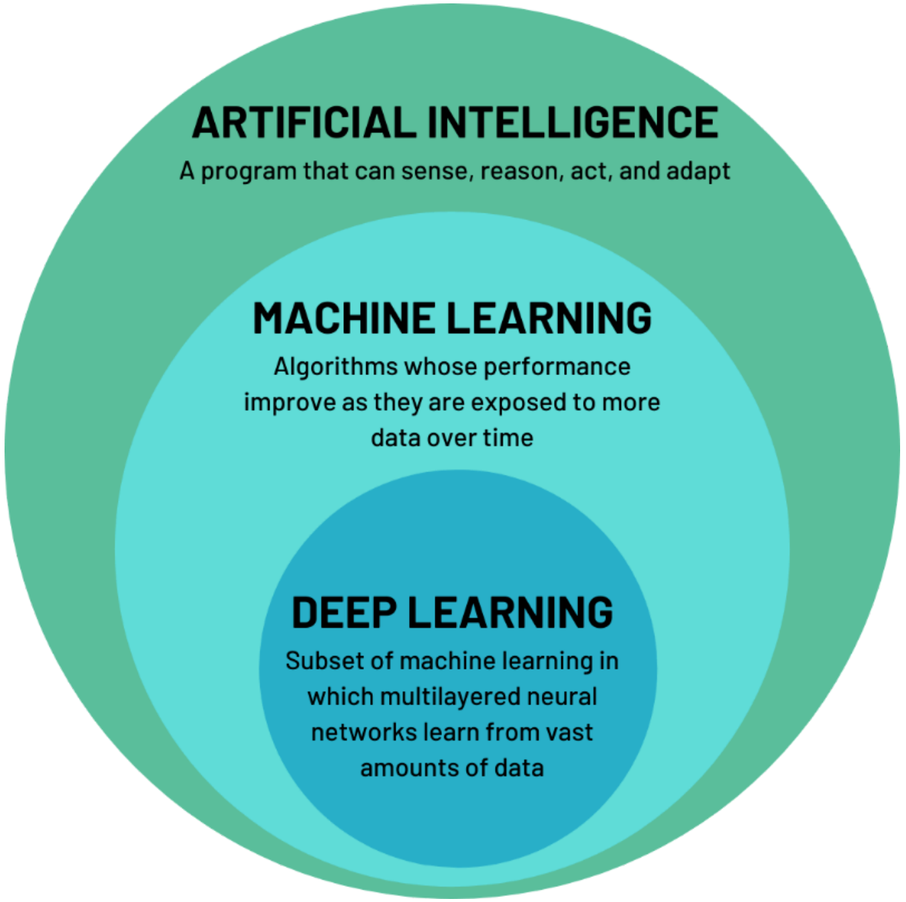


Background

Machine learning

Machine Learning (ML) is a subset of artificial intelligence (AI) that enables computers to learn patterns from data and make decisions or predictions without being explicitly programmed for specific tasks.

At its core, machine learning focuses on building models that can generalize from past experiences (data) to make accurate decisions on new, unseen data.

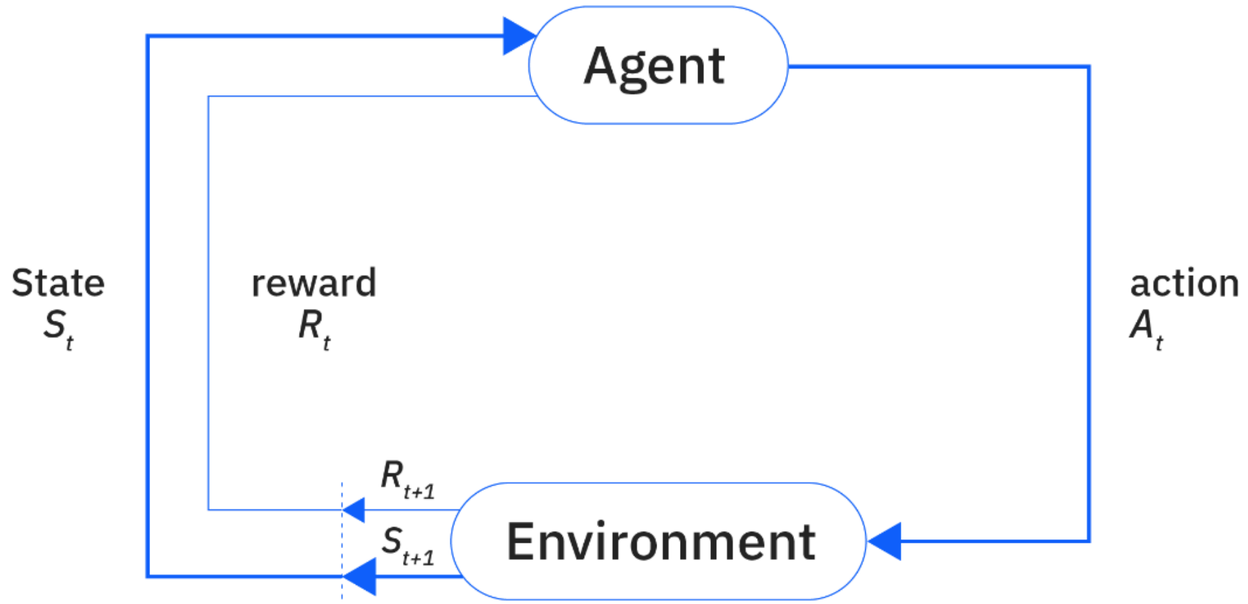


Background

Reinforcement learning

Reinforcement Learning (RL) is a branch of machine learning where an agent learns how to make decisions by interacting with an environment. The agent observes the current state, selects an action, receives a reward, and transitions to a new state.

The objective of the agent is to learn a policy that maximizes the cumulative long-term reward.



Background

Markov Decision Process

A Markov Decision Process (MDP) is a mathematical framework used to model decision-making in environments where outcomes are partly random and partly under the control of a decision-maker (agent). It provides the foundation for most RL algorithms. An MDP is formally defined as a tuple:

$$(S, A, P, R, \gamma)$$

So, what are these notation?

Background

Markov Decision Process

$$(S, A, P, R, \gamma)$$

Where:

- S: A finite set of states the agent can be in.
- A: A finite set of actions the agent can take.
- P(s’ | s, a): The transition probability — the probability of reaching state s’ after taking action a in state s.
- R: The reward function, giving the immediate reward received after performing action a in state s.
- y (gamma): The discount factor, which determines the importance of future rewards.

Background

Markov Decision Process

The Markov property states that the next state depends only on the current state and action, not on the full history. This simplifies the learning and planning process:

$$P(s' \mid s, a) = \Pr(S_{t+1} = s' \mid S_t = s, A_t = a)$$

This equation represents the probability of transitioning to state s′ given that the agent is currently in state s and takes action a. It captures the stochastic nature of the environment, where the outcome of an action may not be deterministic.

Background

Q learning

Q-learning is a widely used model-free reinforcement learning algorithm that enables an agent to learn the optimal action-selection policy in a MDP. It does not require prior knowledge of the environment's dynamics and is capable of learning through direct interaction.

The core idea of Q-learning is to learn a Q-value function, denoted by Q(s,a):

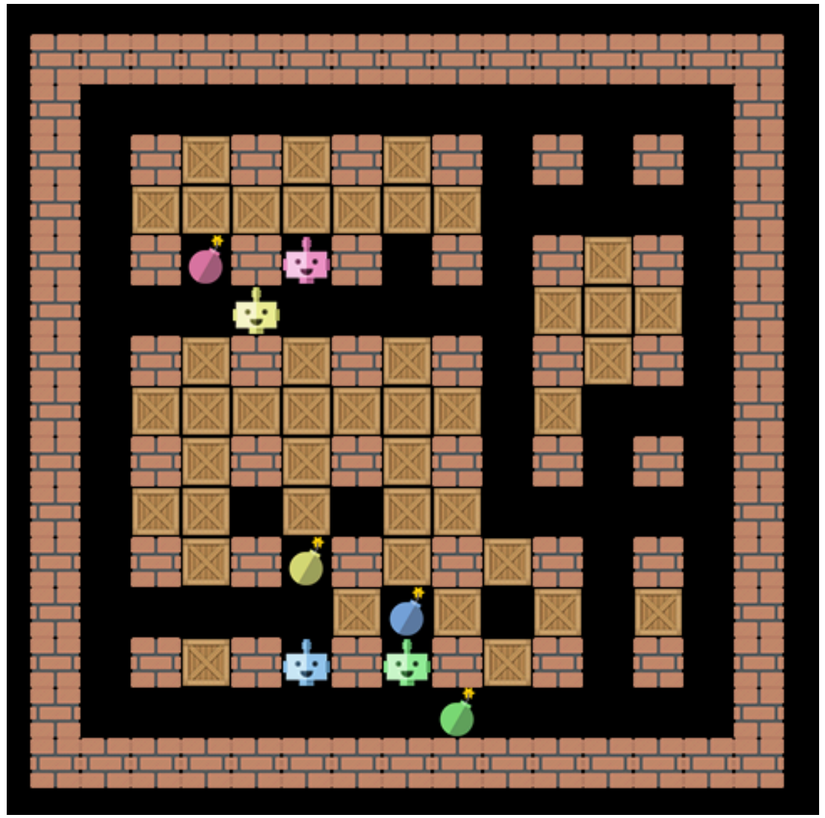
$$Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right]$$

Implement

Initial

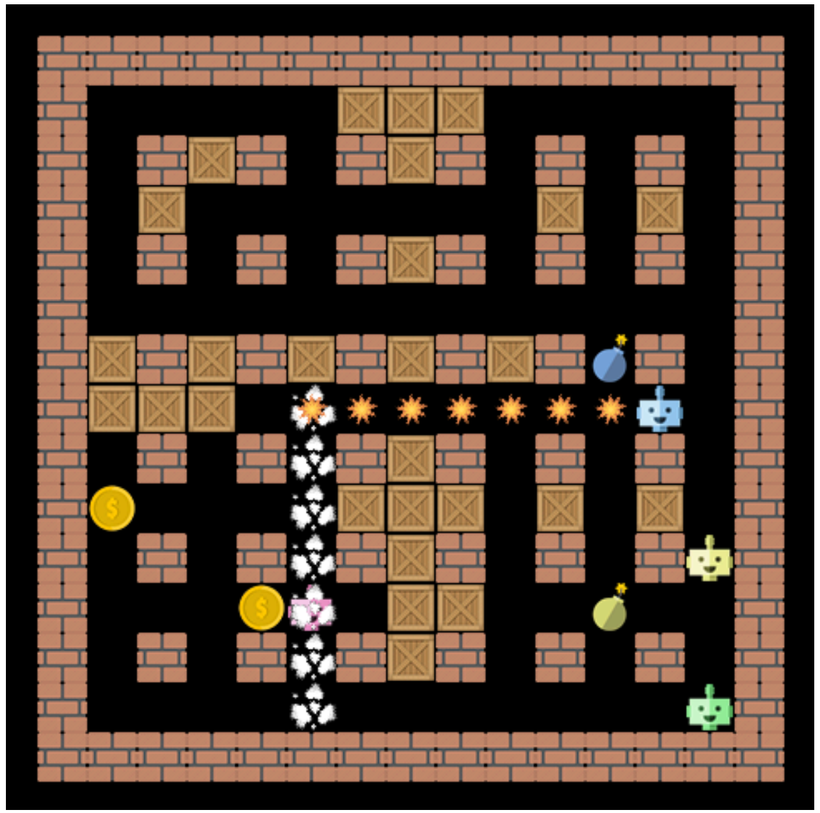
This section details the implement methodologies, including Bomberman game environment, traditional RL and ML, employed to develop and train an AI agent to play the Bomberman game.

The objective is to create an agent capable of autonomously learning to make optimal decisions within the complex game environment, without relying on Deep Learning models.



Implement

Initial

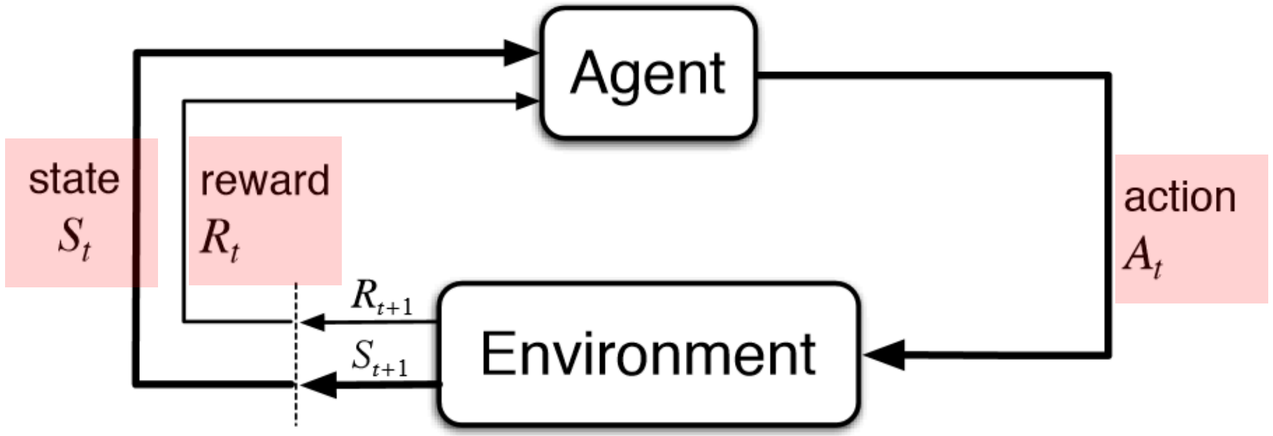


- Game had been created based on grid (n * n).
- There are three types of tile in game:
  - WALL
  - CRATE (can be exploded)
  - FREE (can drop bomb here)
- Exploding crate can produce coins.
- Collecting a coin gains the agent one point.
- Each agent has one bomb.
- Blowing up an opponent is worth five points.
- Every episode ends after 400 steps.
- A agent is consider to win when it is the **last agent standing** or keep the **highest score** when the episode reach max step.

Implement

Initial / *environment*

The Bomberman game environment is defined by its **State** (s), represented as a vector of crucial extracted features (e.g., relative positions, bomb and item information) due to the full state space's immense size. **Actions** (a) include movement and bomb placement. **Rewards** (r) are designed to encourage desired behaviors such as destroying obstacles, eliminating opponents, and survival.



Implement

Initial / *state*

STATE	TYPE	DESCRIPTION
'round'	int	The number of rounds since launching the environment, starting at 1.
'step'	int	The number of steps in the episode so far, starting at 1.
'field'	(width, height)	A 2D array describing the tiles: **1**:crates; **–1**:walls; **0**:free tiles.
'bomb'	[(int, int), int]	A list of tuples ((x, y), t) of coordinates and countdowns for all active bombs.
'explosion_map'	(width, height)	A 2D array, for each tile, for how many more steps an explosion will be present.
'coin'	[(x, y)]	A list of coordinates (x, y) for all currently collectable coins.
'self'	(str, int, bool, (int, int))	Information of agent (name, scores, has bomb? , (x, y)).
'other'	[(str, int, bool, (int,int))]	Like 'self' but for others agent.

Implement

Initial / *action - reward*

Each agents can make 6 action. Based all algorithm and policy, they can make decision what action to do in state ‘s’.

```
self.action = ["UP",
               "RIGHT",
               "DOWN",
               "LEFT",
               "WAIT",
               "BOMB"]
```

```
e.COIN_FOUND : 5,
e.COIN_COLLECTED : 12,
e.KILLED_OPPONENT : 100,
e.KILLED_SELF : -100,
e.GOT_KILLED : -50,
e.SURVIVED_ROUND : 100,
e.OPPONENT_ELIMINATED : 20,
```

Based on state ‘s’, apply policy to calculate reward for agent in that step.

Policy using negative reward to present a bad move, in order word, using positive reward to present a good move.

Implement

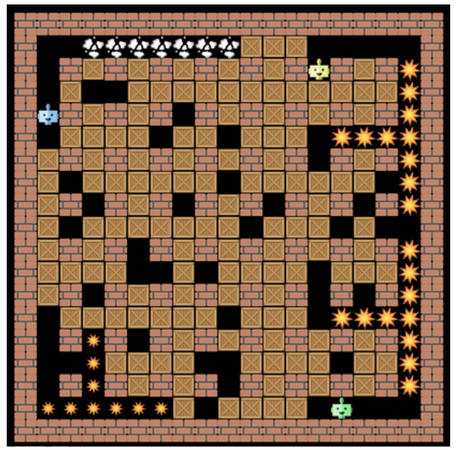
Initial / *rule_based_agent*

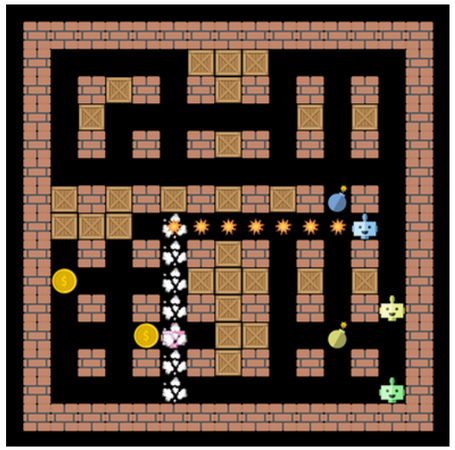
Instead of requiring a human to play 100, 1000, ... rounds—which is impractical—we implement a rule-based agent using predefined algorithms and policies. This agent serves as the opponent to train our learning agents.

To create an Agent with logical principles, it is necessary to determine 3 factors, in the following order of priority:

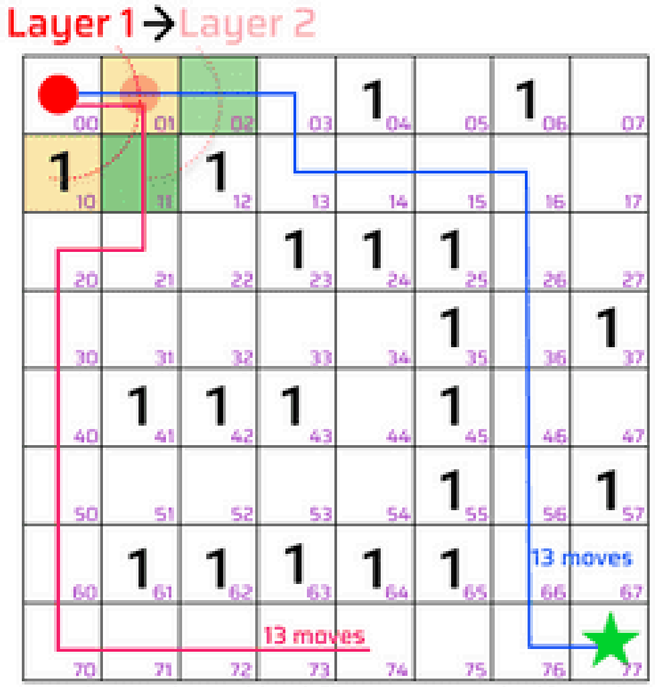
- Dodge bomb → Plant bomb on target → Picked coin

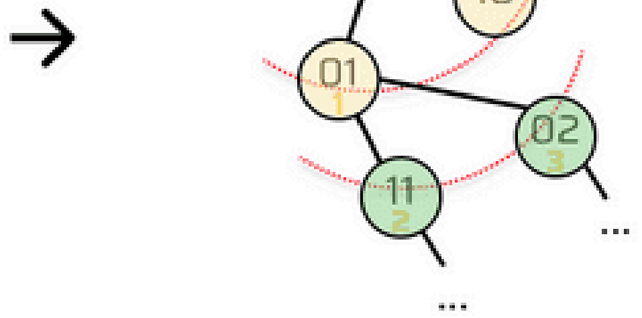
Note: Target can be (component, crates, coin)

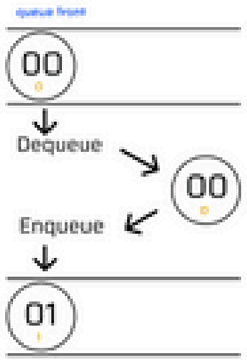


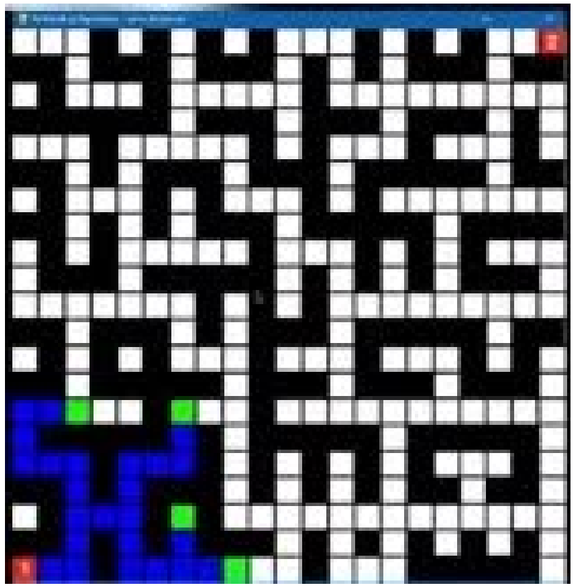


Breadth First Search for matrix









Use BFS to **Plant bomb on target** and **picked coin**
Go random square nearby to **Dodge bombs**

Implement

Q-table

A Q-table is a table that stores Q-values, which represent the expected reward of taking a specific action in a given state. It's used in Q-learning to help the agent learn an optimal policy through experience.

Each **row** corresponds to a **state**, and each **column** to an **action**. The agent updates the Q-table as it interacts with the environment, gradually learning the best actions to take.

	a1	**a2**
s1	2	1.5
s2	0.8	1
s...	...	...

Q value

Implement

Q-table

Bomberman has an extremely large state space, using a single state to perform an action seems impractical.

Instead, we only use small pieces of the state to represent that state for the agent to make decisions, called **features**. Now, each ‘s’ of Q-table is feature of the state returned by environment.

feature s1

	a1	**a2**
s1	2	1.5
s2	0.8	1
s...	...	...

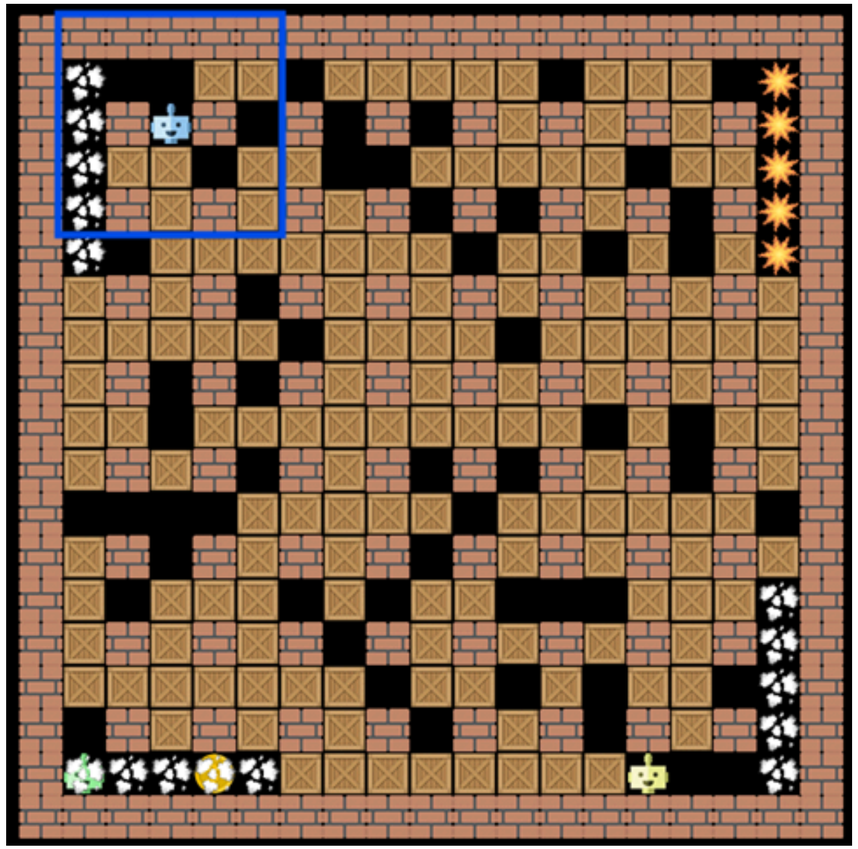
Implement

Q-table / *feature type 1*

The first feature we used is a square around the agent with **radius (offset) = 2**, coordinate of others.

In this type, we use a **dict** to store the Q-table with the key being the state and the value being the Q-value. This is a trade-off between code simplicity and data accessibility.

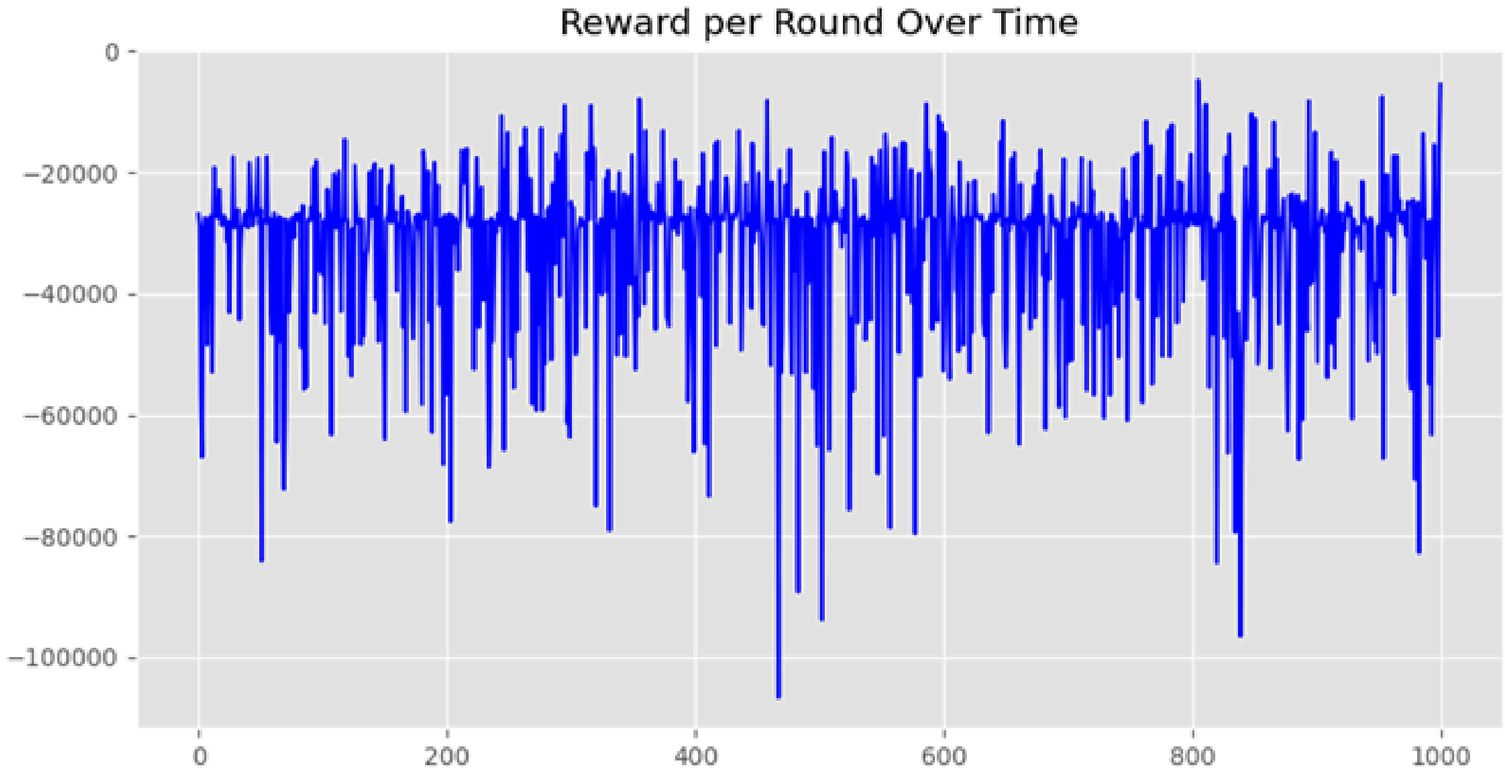
Although the implementation would be quite easy, but with the nature of dictionaries in python, the data access time will be O(logn) and model was larger.



Implement

Q-table / *feature type 1*

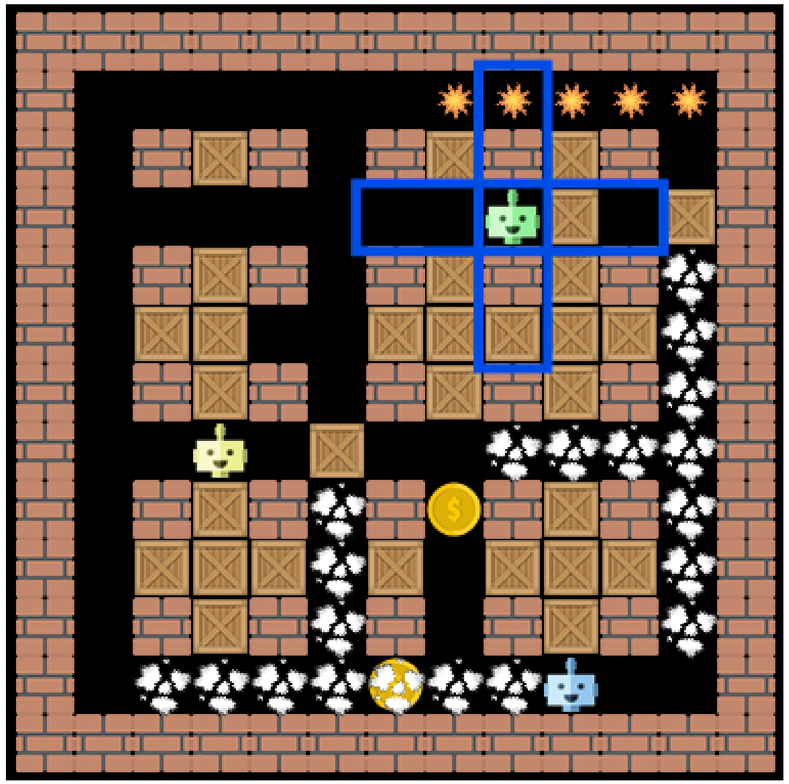
After train the agent (1000 round), model seem like not tend to study how to win:



Implement

Q-table / *feature type 2*

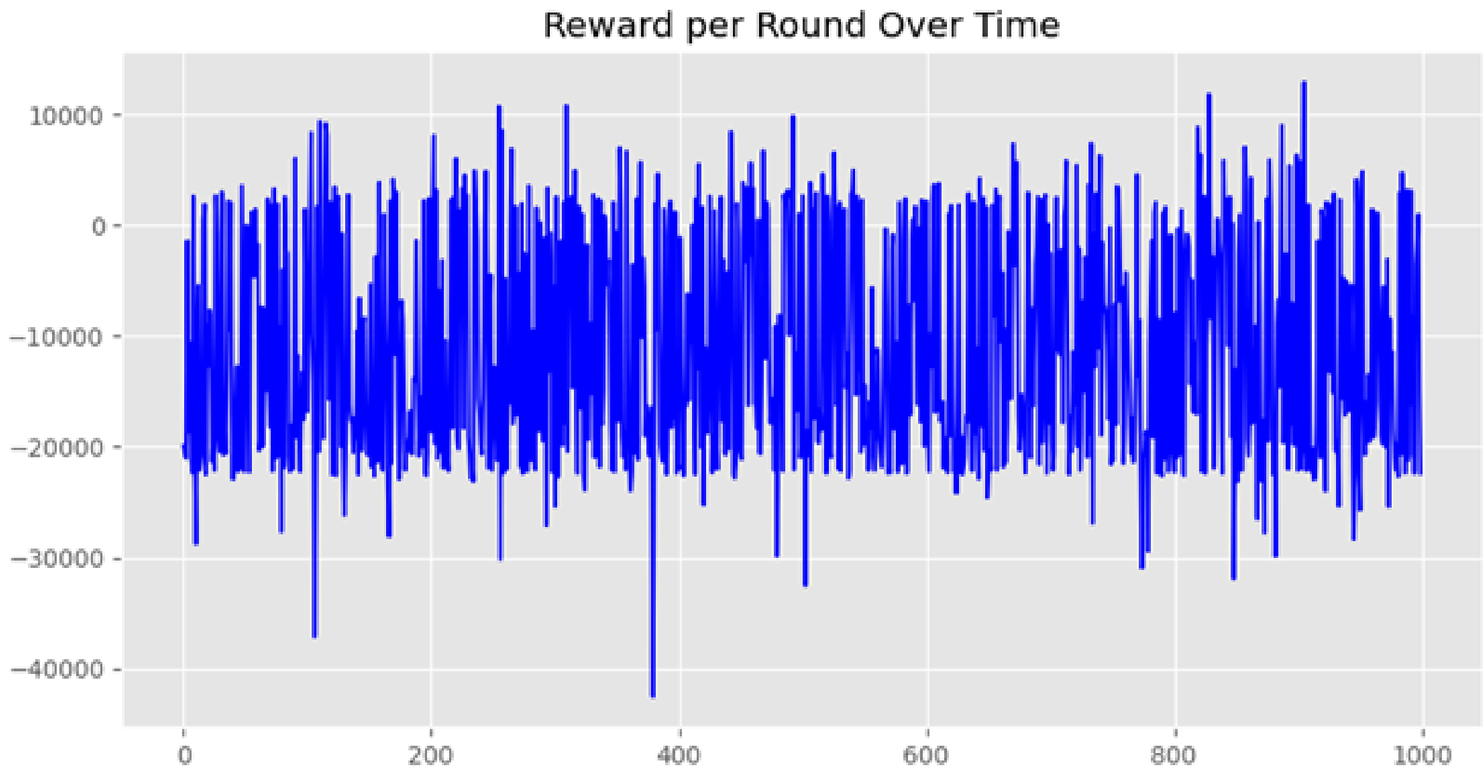
The next feature we used a cross with each side length being 5. The dimension of data was decrease. In this type, we also using dict for storing.



Implement

Q-table / *feature type 2*

After train within 1000 round, it seems that the model did not learn very well, but the results were slightly better than type 1:

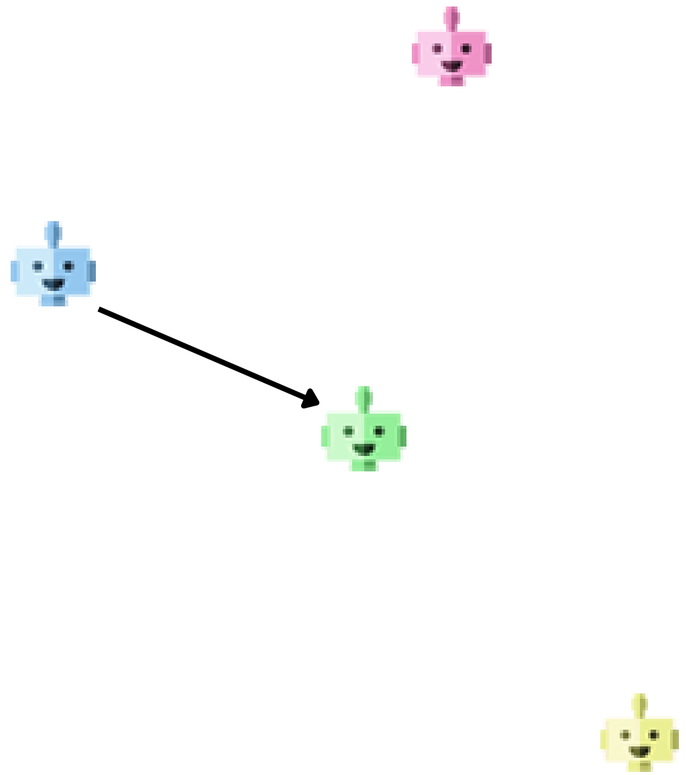


Implement

Q-table / *feature type 3*

At type 3, we use the same feature as type 2 but with some improvements. That is adding **vector to nearest opponent** and some representation of bomb placement ability.

Dictionary is replaced by 2D array, it makes model increase accessibility, reduce time complexity.

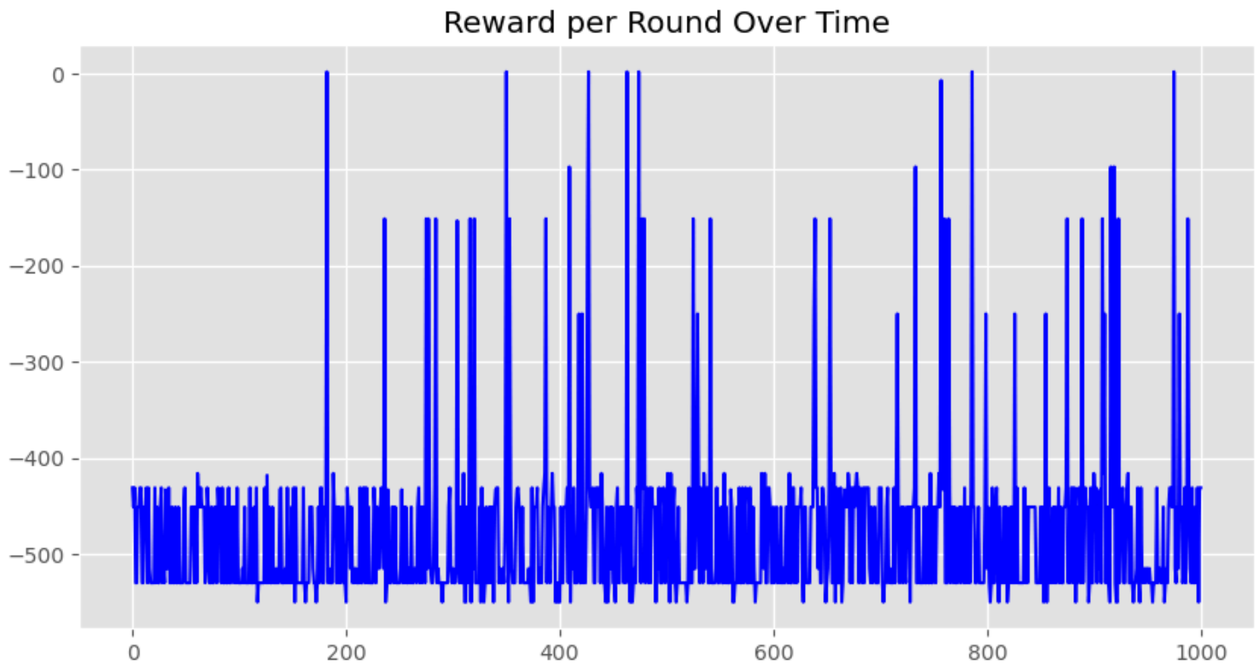


Implement

Q-table / *feature type 3*

Train within 1000 round. Score is low, the agent seems to learn nothing.

→ State space of game is too large for Q-table → agent can not learn anything.

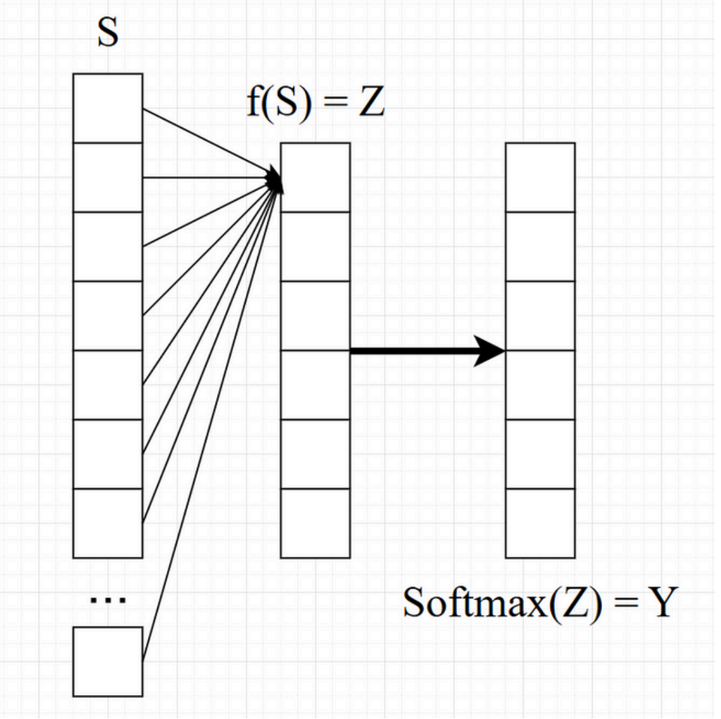


Implement

Neural network / *theory*

Next step, we using neural network (NN) to train the agent. This neural network just contains one **fully connected layer**.

Input is a vector of the feature (state), output of this layer will be a **vector** (6x1 | 6 present number of action agent can make), the vector is forwarded into softmax function and return **probability** of each action.

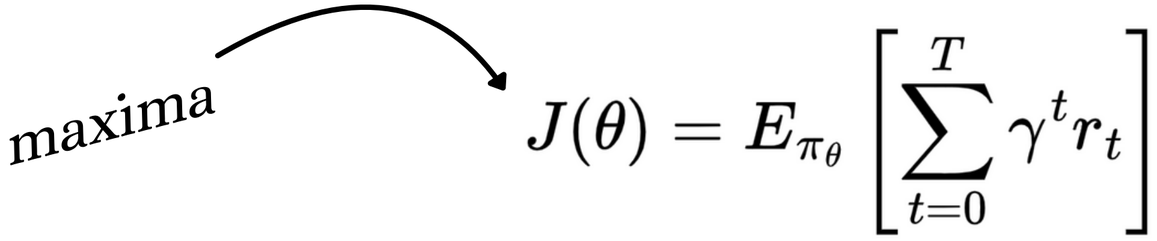


$$\pi(a|s) : y = softmax(Ws + B)$$

Implement

Neural network / *theory*

To update weight (W) and bias (B), we use **policy gradient**. It finds the optimal policy ‘y’ - probability distribution of choosing action a when in state s, with parameter θ: {W, B} for entire the episode.



Using gradient ascent to update θ for reaching max(J):

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$$

Implement

Neural network / *theory*

To calculate the gradient of **J(θ)**, we use derivative trick:

$$\nabla_\theta J(\theta) = E_{\tau \sim \pi_\theta} \left[\sum_{t=0}^{T} \nabla_\theta log \pi_\theta(a_t \mid s_t) \cdot R(\tau) \right]$$

$\nabla_\theta J(\theta)$: gradient of expected return w.r.t parameters θ

$\mathbb{E}_{\tau \sim \pi_\theta}$: expectation over trajectories sampled from π_θ

$\sum_{t=0}^{T}$: sum over time steps from $t = 0$ to T

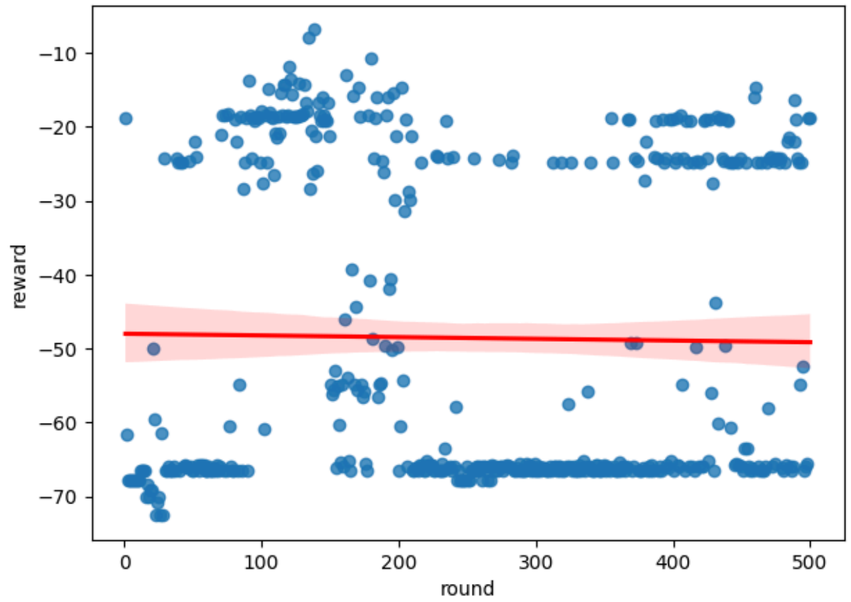
$\nabla_\theta \log \pi_\theta(a_t \mid s_t)$: gradient of log-probability of action a_t at state s_t

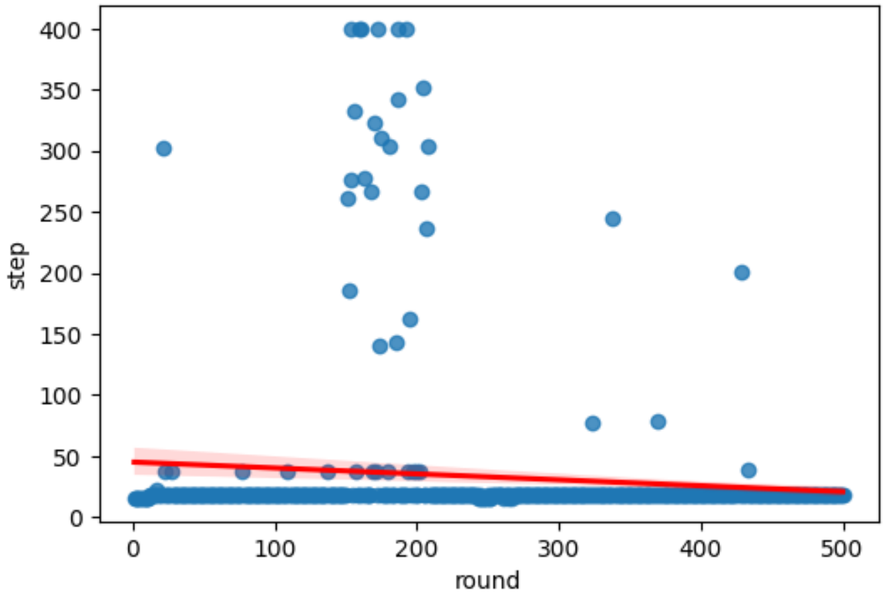
$R(\tau)$: total return of trajectory τ

Implement

Neural network / *first type*

First neural network model using all features of the state which is returned by environment.





Although apply neural network, agent also get hard to learn how to survive.

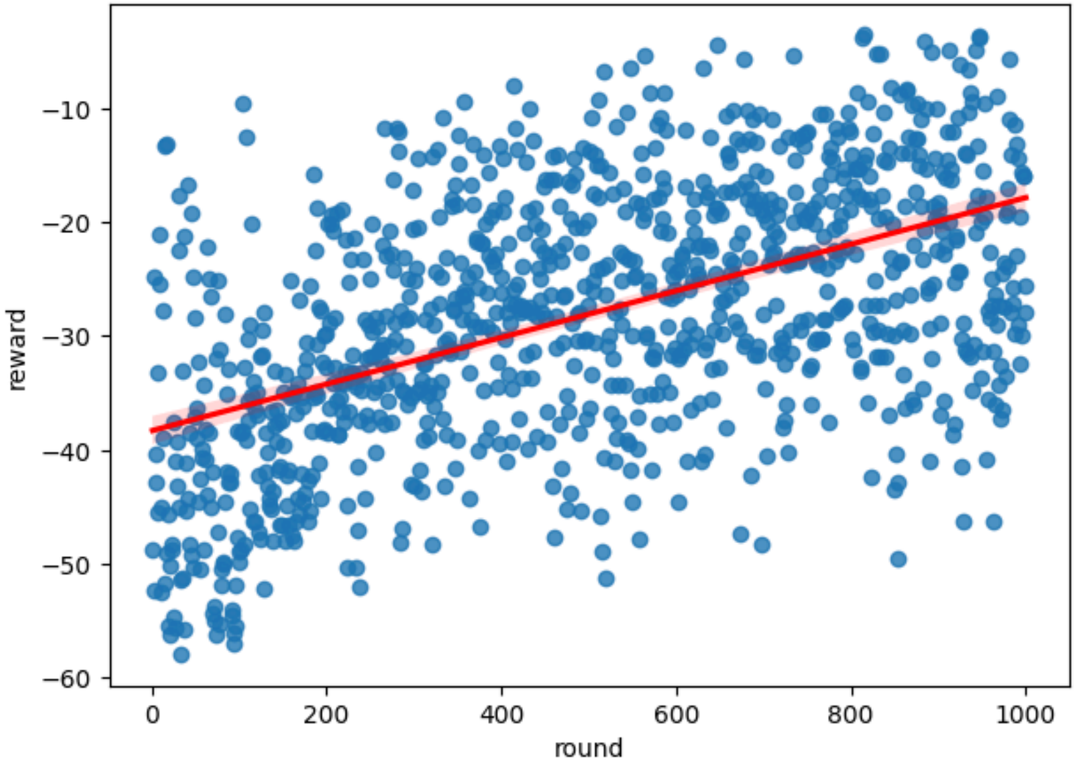
Implement

Neural network

After **normalize** features:

Training agent which is using neural network with three rule_based_agent, 1000 round.

Deeping in the plot, it seems like agent turn to learn something. Although the score is still negative, the agents improve with each epoch.

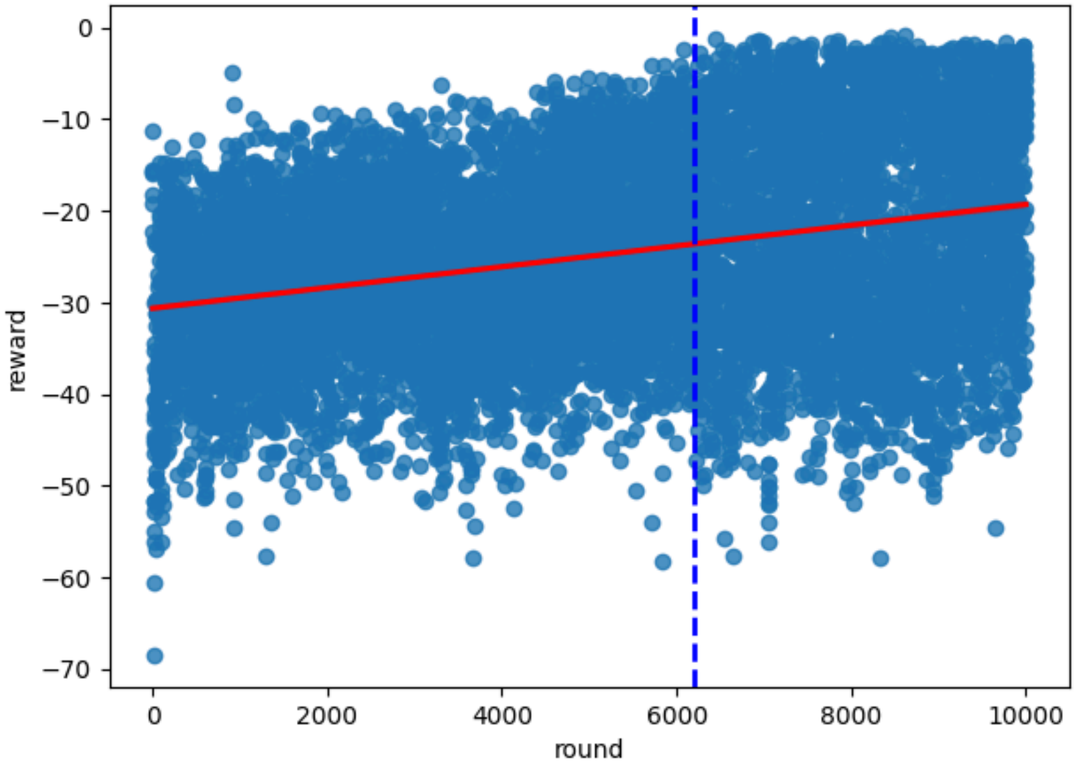


Implement

Neural network

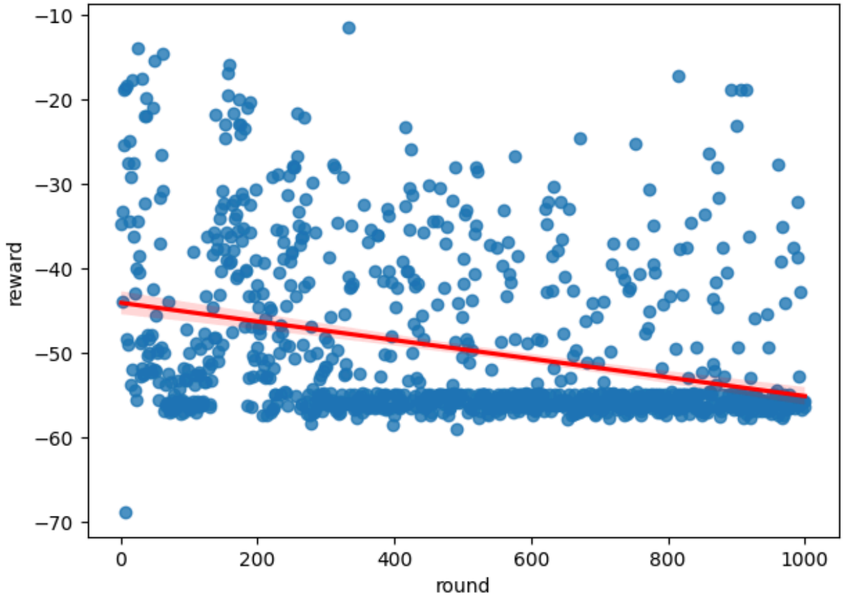
- Up to 50.000 rounds, the agent is improving, but not by much.
- The reward variation is still quite large.
- From around round 6000 onwards, the agent doesn't seem to learn anything.

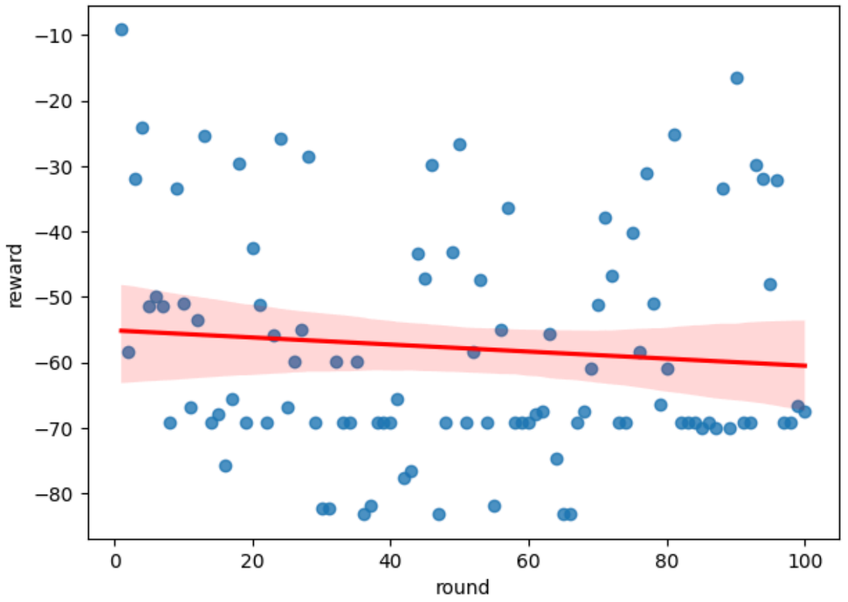
→ One layer of neural network is effective, it make agent learn some thing but not enought.



Summary

- Instead of training the agent by letting it play with a human, I implemented a rule-based agent to serve as an opponent during training.
- Due to the extremely large state space of the Bomberman game, using a Q-table was infeasible, so I applied a simple neural network to approximate the action policy.
- However, despite some improvements, the trained agent only showed limited progress, likely due to the simplicity of the network and training strategy.





Thanks For Your Listening

Group 2 - AI1917