Intelligent Bomberman with Reinforcement Learning

Ngo Hung Minh Triet
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
ngohungminhtriet@student.utwente.nl

ABSTRACT

Bomberman is a strategical, maze-based game where the players defeat their enemies by placing a multi-direction count-down bomb that would explode and destroy obstacles and other players. In this paper, a simplified version of Bomberman is implemented in Java, where different controlled agents are placed. Each agent represents one of five reinforcement learning methods: Q-Learning, Sarsa, Double Q-Learning, and Deep Q Neural Network with two state representations: 5-tiles information and complete information. Then, we investigate whether a specific reinforcement learning method can successfully learn to play Bomberman efficiently by evaluating them with ad-hoc agents and finally against each other. The configuration of 5-tiles information with Sarsa archives the best overall quantitative results.

Keywords

Reinforcement Learning, Bomberman, Computer Game, Q-Learning, Neural network, Deep Reinforcement Learning

1. INTRODUCTION

In the past decades, Reinforcement Learning is gaining more attention. Being inspired by animal learning theories, reinforcement learning (RL) is developed with the idea that an agent can deduce from its experience to decide which actions to take to maximize predefined values. This agent learns by interacting with the environment and receiving feedback indicating the values or rewards of the action. The objective is that after the learning process is that the agent will perceive a specific state of an environment and reason the actions that would give the most rewards. [11, 16].

Nowadays, there are many studies on creating reinforcement learning agents for various genre of games ranging from grid games including traditional games such as shougi, Go, and chess [13, 15, 9, 12, 14] to modern computer games such as Mario, Atari, Bomberman, Ms. Packman [8, 4, 2] in order to improve the efficiency and winrate of the reinforcement learning agents against other reinforcement learning agents or human champions of the corresponding games.

In this study, we show the correlation between reinforcement learning algorithms with different state representations and the ability to learn and play the game Bomberman. We used five reinforcement learning algorithms: Q-Learning, Sarsa, Double Q-Learning, and Deep Q Neural Network and two state representations: 5-tiles information and complete information. Each configuration is tested with three tests: the learnability test, the advance test, and the comparative test. In each of the tests, the agents are evaluated per generation. Each generation consists of 10000 training episodes and 100 evaluation episodes. Then, win rates and average rewards are extracted to make comparisons. The results show that although simple Q-Learning works, but performance deteriorates in scenarios with larger state space. While a preliminary version of the Deep Q network shows superior results and looks promising compare to Q-Learning with the use of the neural network to tackle the ample state space. In the end, 5-tiles information with Sarsa configuration archives the best overall quantitative results.

In Section 2, we introduce the game and describe the implementation of simple Bomberman in Java with different state representations. Then, in section 3, we give an overview of five reinforcement algorithms we used for the agents. After that, we present the experimental setups and results in section 4. In section 5, we discuss the research process and draw reflections from thorough our research. Finally, we end the paper by summarising our paper and discuss some future improvements.

2. BOMBERMAN

In 1983, Hudson Soft published the original variant of Bomberman, and new generations of Bomberman are still being published to this day. Nowadays, there are almost 100 different variants of Bomberman games released on various platforms [7]. In the original variant of Bomberman, which is named Bankudan Otoko in Japan but is called Bomber Man in most releases, the player is trapped in a maze with monsters. In each stage, the objective is to find the exit and eliminate all monsters by moving around the maze, placing bombs, obtains power-ups that empower player's bombs. When the goal is met, the player can proceed to the next stage. While single-player campaigns mode occupied a large portion of Bomberman, we focus on the multiplayer variant in this research, where players are trapped in the same maze, and their only objective is to eliminate each other by placing bombs strategically until the very last man is the winner. Furthermore, since the game can prolong forever, over the iterations, there will be more challenges that help end the game, such as more power-ups or bombs appear randomly to speed up the termination of the game.

2.1 Environment

We implemented an environment that represents a simplified version of Bomberman. The core of the environment is Java and JavaFX as the framework that helps visualize the game. The grid size can be customized, but in our experiments, a board with the size of 7x7 (including the unbreakable border)is used, with its environment configured as in Figure 1. The players in each test type are different but ranging from two to four. There are two types of agents: ad-hoc agents and learning agents. Ad-hoc agents consist of stationary agents and random agents. This type of agent is used to assess the ability to learn and compatibility (the quality of the agents' policies) of a learning agent with Bomberman. Each learning agent is implemented with a combination of a reinforcement learning algorithm and a state representation. Each turn, the agents receive the state representation from the environment (the game) that the agents use to determine their next move. After receiving all the moves from the active agents, the environment executes the moves and sends back the results and rewards of the agents to them, and then the cycle repeats until there is less than one agent on the field.

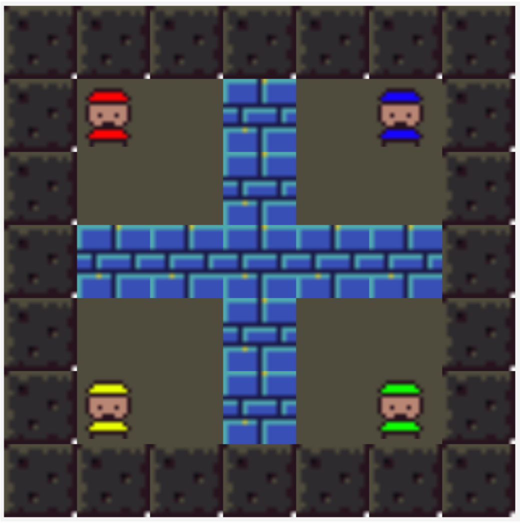

Figure 1. The game configuration uses for all tests in this study. Blue tiles are breakables walls, deep brown tiles are unbreakable walls, and learning agents (in this case, 4) are represent by human-like figures and are distinct by colors. This configuration separates the agents with walls, and each agent is contained in a smaller grid which turns out to be the balance between complexity and fairness. To win the game, the agents need to learn how to break walls and approach then eliminate the opponents.

The rules of the game are as follows, each turn, the agent can move in one of four directions or place a bomb in the same position that the agent is standing. If a bomb is placed, it will explode in 3 turns in four directions within one radius. If the explosion hits a breakable wall or an agent, those objects will be removed from the game, and the credit belongs to the owner of the bomb. Furthermore, if a bomb is inside another bomb's explosion, it will explode as well, which will set off a chain reaction. Therefore, the game is more complicated, and the agent has to consider the nearest bomb countdown and those that can set off that bomb. The agents can be in the same position but are not allowed to move to a tile that is a wall (breakable and unbreakable) or a bomb. As time progress, particularly after 50 turns, we introduce the same probability of a bomb that will spawn at the agent position for all the agents in the game. This probability will increase linearly with turn count to 1.0, which prevents the game from running forever. Therefore, to win the game, the agents must learn how to dodge bombs and finish the opponent as soon as possible.

2.2 State representation

According to Figure 1, as there are 36 changeable tiles (free/breakable/unbreakable), 36 tiles to place a bomb, 36 tiles for the agents (ranging from two to four) to move into, the number of state can be approximated as $3^{36} \cdot 2^{36} \cdot 4^{36} \approx 4.9 \cdot 10^{49}$ different game states in total. In this part, we introduce two ways to represent the game. One of them will show the complete information of the game, which results in the enormous state space mentioned above. The other is an attempt to minimize the state spaces by limiting the agent knowledge (how much the agent can perceive from the game/environment) and only showing the agent's surroundings to them.

2.2.1 Complete information

In this configuration, the game is converted to a flattenable array that contains all the information of the current game state. The array is in 2D and has the same size as the board. Each of the cells in the array contains the information of the respective tile in the game. That information is computed by the following values:

- Tile type: Free, Breakable wall, Obstacles (Unbreakable wall or Bomb).
- Whether the tile contains the agent.
- Whether the tile contains any of the other agents.
- The foreseeable explosion countdown of the tile. In other words, the agent knows whether there will be an explosion in this tile with the precise countdown.

As mentioned in the game rules section, the bombs in this game can be chained to create a bigger explosion. Therefore, The environment uses BFS (Breadth-First Search) to calculate the foreseeable explosion countdown in order to group explosion tiles together, and the group countdown value is equal to the lowest countdown bomb in the group. This way, the agent knows whether the tile is safe according to the nearest bomb countdown but the precise countdown of the exploding bomb even it is chained by another bomb. In this configuration, the number of states is equal to the calculated game states, which is $\approx 4.9 \cdot 10^{49}$.

2.2.2 5-tiles information

Since the main goal of this study is to evaluate how well the simple reinforcement learning algorithm performs in Bomberman, we think that it is not feasible to store or explore the state space with the size of a 50 digits number. Therefore, we developed a game representation that limits the agent's vision to only its adjacent surrounding and the information relating to the nearest opponent. The information that the state provides is as below:

- Tile type: Free, Breakable Wall, (Unbreakable wall or Bomb) and are normalized as 0, 1, -1, respectively in four directions (North, East, South, West) and the agent's current position.
- The foreseeable explosion countdown of the tile and is ranging from 0 to 3 and are normalized by dividing the current countdown to the maximum countdown, which is 3 in four directions (North, East, South, West) and the agent's current position.

- An integer value that represents the relative position of the nearest opponent.

Since in this configuration, the information that the agent perceives are relative to its current position and surrounding, we couldn't store the opponent position as fix number. Therefore, we measured the number of steps which agents require to reach the according position using the Manhattan Distance [3] of the agent as in the equation $d(\mathbf{X}, \mathbf{Y}) = \sum_{i=1}^{2} |x_i - y_i|$ and extract the minimum distance then calculate the displacement vector that represent the current relative position vector from the agent to the nearest opponent as in the equation $\mathbf{X} - \mathbf{Y} = (x_1 - y_1, x_2 - y_2)$ as represented in Figure 2. Last but not least, in this configuration, since there are 36 positions that the agent and its nearest opponent can occupy, the approximation of the number of states can be calculated as follow $3^5 \cdot 2^5 \cdot 4^5 \cdot 36^2 \approx 10^9$. While it is still a relatively big number, it has been significantly reduced from a 49 digits number. Early in the study, we tried to only include the surrounding of the agents without the nearest opponent's information, but the results were not promising. The agent seems to lose its way and couldn't find the opponent and had to rely purely on randomization.

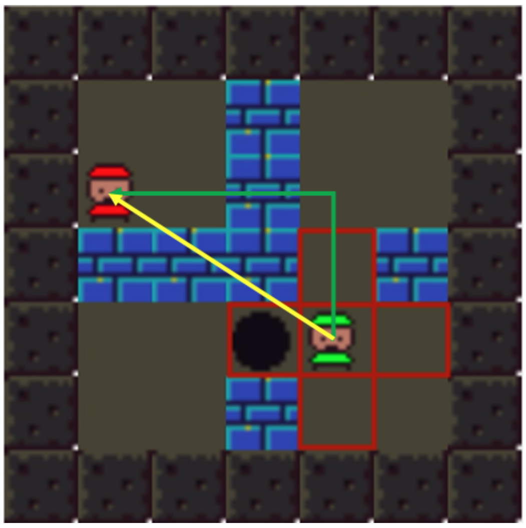

Figure 2. The image illustrates the 5-tiles information state representation is an example of the game. The red cross-shaped represent the agent's surrounding where it takes state information. At the same time, the green arrow represents the Manhattan distance that measures the size of the vector that shows the relative position (direction and distance) between the agent and its nearest opponent, illustrated by an arrow in yellow.

2.3 Ad-hoc agents

In this study, we want to assess whether a configuration of state representation and reinforcement learning algorithm can learn how to play Bomberman. Therefore, we implemented two simple ad-hoc agents that act as a baseline to compare among our configurations. The two agents are a stationary agent and a random agent. The stationary only chooses to stay at its starting position in all games, which helps us evaluate how far the agent gets by evaluating its average rewards after every generation's evaluation phase. The agent can learn the possible knowledge: how to perform correct moves, break walls with bombs, and eliminate the opponent with bombs. On the other hand, the random agent, who can perform six moves, including five movements and a 1% chance of placing a bomb, help us create a more complex environment where the opponent of the agent can move freely. Therefore, this agent not only creates a vastly wider state space that the learning agent needs to anticipate but also has the ability to potentially dodging bombs primitively.

3. REINFORCEMENT LEARNING

Reinforcement Learning was introduced by Sutton and Barto in 2015 [16]. This type of machine learning algorithm helps the agent learn to achieve certain goals in a certain environment by optimizing the agents' behaviors with trial and error (aka making a sequence of actions repeatedly) in order to maximize the expected future reward sum. From each time step, the agent can perceive the current state of the environment and make an action which in turn is mapped into a numerical reward. This reward is dependent on the effect of the agent's action on the environment. If the action helps the agent moves towards the goals, it will be positive and negative otherwise.

To model the discrete-time stochastic control process of the interaction between the agent and the environment, Markov Decision Process (MDP) is introduced with a finite space state S, and a finite set of action A [1]. In this definition, the environment state at time-step t is denoted at s_t while the action that the agent chose time-step t is denoted at a_t. The transition function $Tr(s, a) \to s'$ denotes transition of the state of the environment from state s after executing action a while the reward function $R(s, a) \to \mathbb{R}$ denotes the numerical reward that when executing action a in state s and lastly, the discount factor $\gamma \in [0, 1]$ decide how important the future rewards to the optimal decision making process.

3.1 Reward Function

In our study, it is hard to evaluate each of the agent's actions for numerical rewards since immediate actions are not resulting in an immediate reward. For example, placing a bomb does not always guarantee breaking walls, killing other agents, or suicide. Therefore, we mapped events or rather consequences of those actions into numbers. For the events that improve the agent knowledge and push it towards victory, the rewards are positive, such as eliminating another agent or breaking a breakable wall. On the other hand, if the event obstructs or even halts from reaching the goal, the rewards become negative. Further details regarding specific event-reward value can be found in table 1.

Event	Reward
Make a valid move	-1
Make a invalid move	-5
Kill a player	500
Die	-300
Break a wall	30

Table 1. Event-based reward function. Positive rewards are used for events that nudge the agent towards the goal, and negative rewards are used for either promoting active movement or punishments for events that prevent the agent from archiving the goal.

The final event-reward value is modified throughout our study since reward engineering directly affects the learning process of the agents. For example, choosing -1 instead of 0 in order to encourage the agent to move or a high reward for killing an opponent encourages the agent to eliminate the opponent faster. Although we modified the rewards based on our assumption, engineering the rewards was trickery, and the learning agent often exploited it. After our tests, we arrive at our best performance reward function, which is table 1. Finally, as we experiment with the parameters, we decide to use the discount factor of 0.99, which archives better policies for the learning agents.

3.2 Q-Learning

As one of the breakthrough in reinforcement learning, an off-policy TD control algorithm as known as Q-Learning is introduced by Watkins in 1989 [17]. The action-value function Q in equation 1 directly approximate the q^* (the optimal action-value function) is independent from the the policy that it is following. $Q(s, a)$ denotes the expected sum of rewards after executing action a in state s and represents how significant the action a in state s in gaining future rewards. For each time-step, the quadruple $(s_t, a_t, r_{t+1}, s_{t+1})$ is extracted to update $Q(s, a)$ as follow:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a_t) - Q(s_t, a_t)] \tag{1}$$

Q-Learning uses Temporal Differences (TD) to estimate the value of q^*. TD-error is the latter part of equation 1 that is encapsulated in square brackets and represents the difference between the estimated reward at a time step, and the actual reward received. The larger the TD-error is, the larger the difference between the expected rewards and the actual reward. Therefore, the goal of Q-Learning is to minimize the TD-error. The learning rate α is used to determine how quickly we want to shift the current Q(s,a) to its expected value. Finally, if we allow the agent to execute all possible actions in all states by learning indefinitely, the action-value Q function will converge to the optimal policy q^*.

3.3 Double Q-Learning

While Q-Learning is often powerful, the involvement of repetitive maximization operations to approximate the maximum expected action value introduces a significant positive bias in the process [16]. As example 6.7: Maximization Bias Example (Sutton and Barto, 2015), they introduced a simple episodic MDP that has two states A and B, starting state in A, with the two actions: left or right, where the optimal actions is 5% probability choosing the left action from A. The experiment shows that Q-Learning is affected by maximization bias and takes the left action in a large portion at the start of its learning process, while Double-Q learning is unaffected by maximization bias.

The proposed Double Q-Learning is to maintain two Q-value functions Q_1 and Q_2, and in each time-step, with the probability of 0.5, one Q-value function will update with the value of the other Q-value function. The update, in each time step, in case Q_1 is chosen, consists of finding the maximizing action a^* from Q_1 by using

$$Q_1(s', a^*) = \max_a Q_1(s', a) \tag{2}$$

then use a^* to determine $Q_2(s', a^*)$ and finally use those value to update $Q_1(s, a)$ as follow

$$\begin{aligned} Q_1(s_t, a_t) \leftarrow & Q_1(s_t, a_t) + \\ & \alpha[r_{t+1} + \gamma \max_a Q_2(s_{t+1}, a_t) - Q_1(s_t, a_t)] \end{aligned} \tag{3}$$

In case Q_2 is chosen, the label 1 and 2 is swapped, and the same process applied.

3.4 Sarsa

State–action–reward–state–action (Sarsa) is introduced in 1994 as an on-policy TD control method [10]. Sharing some similarities with Q-Learning, they are both Markov chains with a rewards process and Sarsa's goal is to minimize TD-error but on policy, hence Sarsa aims to estimate $q_\pi(s, a)$ rather than $q(s, a)$. Essentially, Sarsa agent interacts with the environment based on its policy and updates the policy based on actions taken using the equation below:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_t) - Q(s_t, a_t)] \tag{4}$$

As in the equation 4, Sarsa, or rather all on-policy methods, it use behavior policy π to continually estimate q_π and simultaneously modify policy π toward greediness with respect to q_π. A distinction from Sarsa to Q-Learning is the convergence properties since Sarsa convergence properties is dependent on the behavior policy π [16].

3.5 Deep Q-Network

While Q-Learning is a simple but powerful algorithm to derive policy that helps our agents learn how to play the game Bomberman, it is not sufficient when the number of states is a number with 49 digits. The resources that are needed to store the complete Q table would be enormous, and it would also require nearly infinite runs/episodes to train the agent. Therefore, it is found out that applying machine learning models such as neural networks would help tackle the Q-Learning bottleneck. As illustrated in figure 3, in Deep Q-Network the agent will consist of a neural network that receives the state from the environment and then approximate the Q-value of the actions at the output of the neural network, the maximum action is chosen to send to the environment and get the according reward similar to Q-Learning. Unlike Q-Learning, the loss function is the mean squared error of the predicted Q-value and the target Q-value which is calculated with:

$$L = \mathbb{E}[(r + \gamma \max_{a'} Q(s', a') - Q(s, a))^2] \tag{5}$$

Further more, consecutive samples consist of strong correlations, by randomize and utilize experience replay, we can store the agent's experience at each time-step as $e_t = (s_t, a_t, r_t, s_{t+1})$ in a data-set, pooled it up and withdraw mini-batches and updating the batch to the neural-network [8].

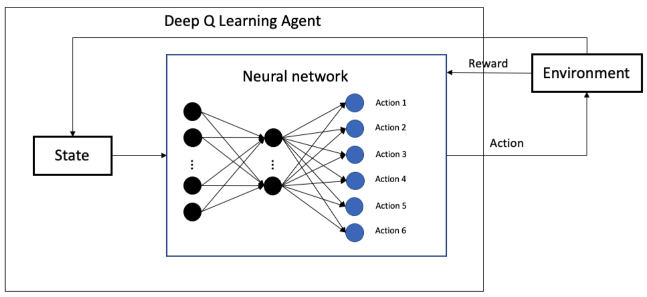

Figure 3. Deep Q-Network agent configuration.

In this study, we implemented a deep Q-Learning agent that has a multi-layer perception. It is a feed-forward neural network that maps the flattened complete information state as an input vector representing the game's current state and output. A vector consists of 6 nodes. Each node is the approximation of the Q-value for one of the possible actions in Bomberman. We used only one hidden layer consists of 256 nodes with the ReLU activation function. Finally, our output function is a linear function, so the network can output value outside range $[0, 1]$. While training, we backpropagating the TD-error using the loss function 5 to update the weights and bias inside the neural network. After the training, the neural network is able to approximate the Q-value of each action in a specific state without the need to store all of the possible states in a table.

4. EXPERIMENTAL RESULTS

In this section, we divide our experiments into three parts: learnability test, advance test, and comparative test. The learnability and advance test is training the learning agents with ad-hoc agents to measure how much the agents can learn from the game. In contrast, the comparative test chooses four distinct learning agents and obtains results so that we can roughly show which configuration of state representation and reinforcement learning method yields the most efficient results.

While having different purposes, all of the tests use the same procedure and parameters and hyper-parameters for the configurations. Each test consists of 100 generations. In each generation, we train the agents for 10000 episodes (games). Then we evaluate the agents with 100 evaluation games. In the training episodes, the agents' starting positions are randomly generated while we used one configuration (Figure 1) for all of the evaluation episodes. Furthermore, we disable all of the exploration methods in evaluating episodes. Therefore, only the results from the evaluation episodes are reliable. We collect win rates, mean rewards and then observe the agents' policies developed by running evaluation on the game GUI with graphical-delay which otherwise is disabled for faster training speed.

4.1 Hyperparameters

Hyperparameters are particularly difficult to decide since each modification affects the configurations differently.
Therefore, we perform many tests to adjust the hyperparameters at the beginning of the tests so that it yields the acceptable policies. All of the configurations use the learning rate of 0.001 and the discount factor of 0.99. Furthermore, the diminishing greedy exploration method is used for training the deep Q-Network agent, while the simple greedy exploration method is used for all of the tabular TD learning, including Q-Learning, Sarsa, and Double Q-Learning. Lastly, all of the hyperparameters can be found in table 2

	QL	Sarsa	DQL	DQN
ϵ	0.1	0.1	0.1	$0.5 \rightarrow 0.01$
α	0.001	0.001	0.001	0.001
γ	0.99	0.99	0.99	0.99
batch size	-	-	-	16
buffer size	-	-	-	100000
epoch	-	-	-	4
hidden layer	-	-	-	1

Table 2. Hyperparameters for reinforcement learning methods. In this table, Q-Learning is denoted as QL, Double Q-Learning is denoted as DQL and Deep Q-Network is denoted as DQN

4.2 Learnability test

In this test, we evaluate whether the agents are able to learn how to play Bomberman and their policies. As a rule of thumb, we would like the agents to learn how to move properly, which means the agents should make almost no invalid moves. Then, the agents should learn how to destroy the wall and eliminate other opponents while dodging the explosion of the bombs. Therefore, in this test, we place all of the agents against a stationary agent. Furthermore, we disable the spawning bomb for this stationary agent, making the game into a puzzle game. The agent needs to learn how to move, break the wall, find the opponent, and then eliminate them before the game spawning bomb rapidly to kill the learning agents.

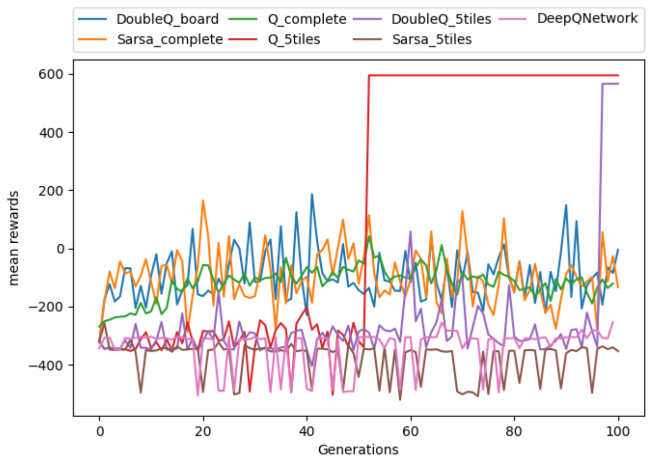

Figure 4. The figure shows the Learnability test average rewards according to the learning agents' configurations. Each configuration is named as a reinforcement learning method and a state representation (e.g., Q_complete is an agent who uses Q-Learning and complete information state representation). Except for the Deep Q-Network agent who always uses the complete information state.

The results of the test are shown in Figure 4. In the graph, we observe that all of the configurations show the progress of the agents in their learning period. Since the opponent is stationary, it is both an advantage and disadvantage for the learning agents. Using a stationary agent, we reduce the state size for the learning agent, but the agent would rely on randomization to reach and eliminate the stationary agents. All of the simple TD learning such as (Double Q-Learning, Q-Learning, or Sarsa) with the complete information state representation can learning how to move properly and destroy walls to get more points. But only the configurations with 5-tile information with Double Q-Learning and Q-Learning are able to learn how to reach the opponent and eliminate them, which resulting large spikes in Q-5tiles' line and DoubleQ-5tiles' line at generation 55 and generation 96, respectively, and since this is a puzzle game with a stationary opponent, the learning agent will stop learning when they found their optimal policy. While 5-tiles information is scalable, it is not generic since further training after the optimal policy is found to degrade rewards. While Deep Q-Network is not showing any remarkable progress, observation showed that it could move properly and then suicide to lessen the negative reward because it couldn't find a better policy. One possibility could be that it loses the spatial features by flattening the state vector (the correlation between neighbors cells).

One potentially approach would be to implement convolutions layers that preserve those spatial features, which help the neural network learn more efficiently, unlike Q-Learning with the same complete information state representation, which shows steady progress over the training period.

4.3 Advance test

In this test, we evaluate how well the learning agent performs against an ad-hoc agent, specifically a random agent. Sharing similarities with the learnability test, the setup and procedure of this test are as same as the learnability test. We place both agents on the same board, but this time the random agent can move. In this test, the optimal or rather desirable policy that we expect from the agent would be to destroy walls, dodging bombs while trying to trap the opponent. Since this is not a puzzle game like in the Learnability test, we don't stop learning when an optimal policy is found. Furthermore, we decided to collect win rates in this test to see the progress of the learning agents. Lastly, this test would pose more challenges for the learning agents since their opponent can move, which in turn generates more states. Therefore a traditional tabular with complete information state representation would suffer due to the vast state space that the random agent generates by moving.

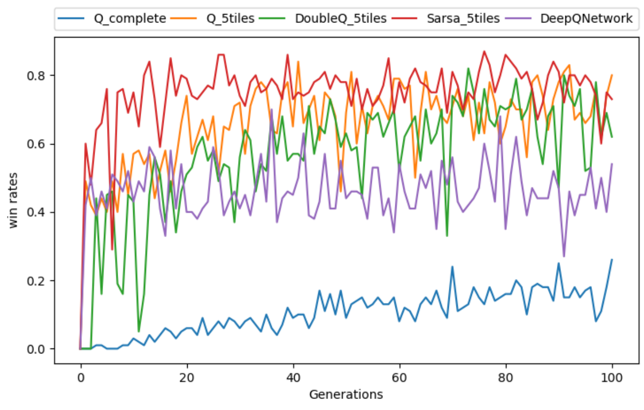

Figure 5. The figure shows Advance test win rates according to the learning agents' configurations. Each configuration consists of a reinforcement learning method and a state representation (e.g. Q_complete is an agent who uses Q-Learning and complete information state representation). Except for the Deep Q-Network agent who always uses the complete information state.

The win rates of the advance test can be found in Figure 5. Since we expect that other TD learning methods such as Sarsa and Double Q-Learning with the complete information state representation would result in some similar patterns, we exclude them to preserve memory storage and training speed. From the result, we can observe that despite the chosen configuration, all of the learning agents showed their progress as their win rates increase linearly with the generation count but with different speeds. As expected, the agent with Q-Learning and complete information state representation perform worst since the state space is much larger now. Nevertheless, it can learn to beat the random agent with a roughly 20% win rate at generation 100. In contrast, as an improvement of the Q-Learning and the use of the neural network from machine learning, the Deep Q-Network agent can approximate the Q-value function, reducing the memory usage significantly. As observed, the win rates of the Deep Q-Network agent, although it fluctuates but can reach 70% at generation 38. Last but not least, the 5-tiles information state representation performs better than the others. The agent who uses Sarsa and 5-tiles information reaches the win rate of 90% in several generations.

4.4 Comparative test

In this test, we want to push the learning agents even more by putting four learning agents on a board and let them learn the game simultaneously. As a representation of their group, we chose

- Q-Learning with 5-tiles information state representation.
- Sarsa with 5-tiles information state representation.
- Q-Learning with complete information state representation.
- Deep Q-Network with complete information state representation.

Like all the above tests, all of the tests consist of 100 generations, and each generation contains 10000 training episodes and 100 evaluation episodes. In this test, we collect win rates in two different scopes, the test, and generations. Similar to the previous assumption, with four movable agents, the state space would be even larger for the complete information state representation users. Therefore, in this test, the expected optimal policy we would like the agent to derive would be to destroy walls, dodging bombs while balancing between eliminating other opponents and staying alive since the game's goal is to become the sole survivor of the game. There are more ways for them to be eliminated (either by game spawning bombs or the other agents' bombs).

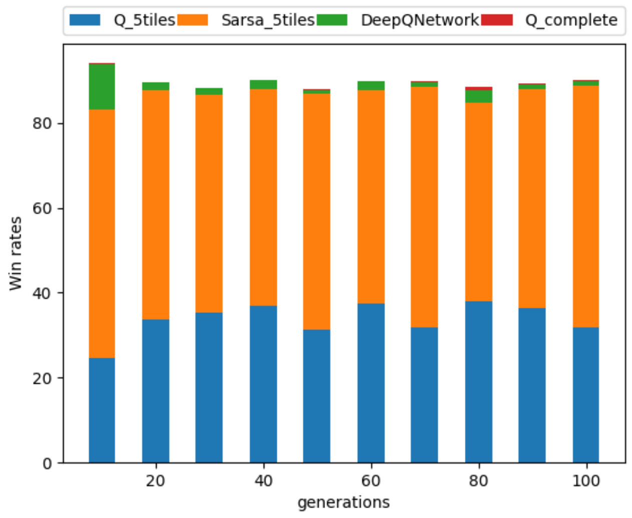

Figure 6. The figure shows Competitive test win rates according to the learning agents' configurations. Each configuration is named as a reinforcement learning method and a state representation (e.g. Q_complete is an agent who uses Q-Learning and complete information state representation). Except for the Deep Q-Network agent who always uses the complete information state.

With the results are shown in Figure 6 and table 3. We can see that the 5-tiles information state representation

Configuration	Win rates
Q_5tiles	33.72%
Sarsa_5tiles	**53.22%**
DeepQNetwork	2.59%
Q_Complete	0.18%

Table 3. The table shows Competitive test overall win rates according to the learning agents' configurations. Each configuration is named as a reinforcement learning method and a state representation (e.g. Q_complete is an agent who uses Q-Learning and complete information state representation). Except for the Deep Q-Network agent who always uses the complete information state

is dominating the complete information state representation. While the Deep Q-Network agent seems to be able to learn how to win a minority of the game, its win rates are often larger than the normal Q-Learning with complete information state representation (Deep Q-Network at 2.59% and Q-Learning at 0.18%). As we suspected, Q-Learning is suffering greatly from the state space with all four moving agents. Lastly, the Sarsa on-policy TD control method seems to be reliably archiving the highest win rates throughout the test and obtain a final approximately 53% win rates, which is more than half of the games in the comparative test. The 5-tiles state representation can only create a significantly smaller state space than the complete state representation since the 5-tiles state representation only concerns the surrounding of the agent and its nearest opponent, while the complete state representation represents all of the information from the game. Therefore Sarsa_5tiles and Q_5tiles learn faster than Q_Complete and DeepQNetwork. While it would take an eternity to fill in the Q table of normal Q-Learning, Deep Q-Network uses a neural network to approximate the Q values by refining its weights and bias in the network, but since it would take longer to archive a good configuration, our preliminary Deep Q-Network agent is falling behind. Nonetheless, the experiment showed that even though its win rate is not as high as the 5-tiles state representation agents, it shows significant improvement compare to normal Q-learning with the complete information state representation.

5. DISCUSSION AND REFLECTION

In the previous section, we discuss how different configurations can archive various win rates or mean rewards. After all the tests, the agent who uses Sarsa as the reinforcement learning method and 5-tiles information state representation performs the best. Sarsa_5Tiles archive approximately 600 points in the learnability test, which means it is able to find a way to eliminate the opponent by combining, moving, breaking walls, and dodging bombs. While in the advance test, it archives approximately 90% win rate against a random agent, which also involves combining its bomb and the environment bombs. Lastly, in the comparative test, it obtains consistency with the highest win rates throughout the generations and an overall win rate of approximately 53%.

From observations, while training the agents throughout the study, we note that reinforcement learning methods are rather easy to exploit to win the game or to maximize its rewards. For example, in the early phase of this study, we place the learning agent against an ad-hoc stationary agent while training it to eliminate the agent. We found out that through randomization, the agent never learned how to place a bomb and dodge it properly or move breaking wall until killing the opponent. Therefore, it either exploits a trick to win the game (such as moving in between two tiles and waiting for the game to spawn bombs and kill the opponents) or exploits a way out of the game (aka suicide). The agent will move back and forth from its position and learn how to dodge the bomb better than the stationary opponent since we did not disable the spawning bomb for the ad-hoc stationary agent at that point of the study. Or, on another board configuration, it decided that it would be better to commit suicide to avoid further punishment. After trial-and-error, we decide to replace the board with another board with less complexity and disable the spawning bomb for the stationary agent.

Not only the complexity of the board and the rules of the game that the learning agent can exploit but also the Engineering the reward function for the environment that can greatly affect the agents. Any small adjustment in each event can cause the agents to exploit the game. But not all the reward functions work with all of our tests. We find out that one of our decent reward function that works decently in the learnability test but suffers greatly in the other test. Therefore, we aim to provide the reward function that produces a decent policy at an acceptable learning rate since the test results are heavily dependent on hyperparameters and the reward function.

As an extension of our study, we have the least time to work with deep Q-Network, after two weeks of researching about neural network and how former researchers apply deep Q-Network to their study regarding apply deep reinforcement learning to games such as Atari, Go, Chess, etc. [13, 15, 9, 12, 14]. But the complexity of their research is too large for our study. Therefore, we implement our framework for deep reinforcement learning with Pytorch and exclude the convolutions layers. At the end of our study, we can show that we can make a learning agent that can play complex games such as comparative tests or advanced tests by applying neural networks in Q-Learning. While the deep Q-Network agent results are not as expected, it shows superior power compared to normal Q-Learning. Lastly, since our environment is in Java and the Deep Q-Learning agent in Python, we have to use a socket that introduces network latency into our training, which is normally much slower than training normal TD learning methods. Each of our tests requires more than 10 hours to train for deep Q-Learning. Therefore, tinkering with hyperparameters or debugging is extremely difficult.

6. CONCLUSION

In this study, we investigated four reinforcement learning algorithms and two state representations to build a learning agent that can learn how to play the game Bomberman. We performed three tests: learnability test, advance test, and comparative test. The learnability test is used to determine whether it is possible to use a configuration of an agent to play the game Bomberman. The advance test further tests the agents' capability by introducing more states. Lastly, the comparative test is used to determine the most effective configuration by battling the four most suggestive agent configurations on one board. Regarding the overall result, we found out that using State–action–reward–state–action (Sarsa) as the reinforcement learning algorithms and 5-tiles information state representation yields the best results in our tests.

In future work, we would like to further improve our deep Q-Network agent with convolutional layers, which helps to persevere the spatial features that are trimmed by the flattening feature vectors and other latest techniques. Furthermore, we would like to show effective all of the configurations would be in a more complex and more extensive board. On a side note, we also want to investigate the applications of reinforcement learning in real-life applications, such as helping develop intelligent social robots such as waiter, dog, etc. [5] or furthermore improve NPC and bosses by implementing learning agents that can constantly learn and improve to adapt with the players' playstyles in FPS games rather than apply into board game such as Bomberman [6].

7. REFERENCES

[1] R. Bellman. A markovian decision process. *Indiana Univ. Math. J.*, 6:679–684, 1957.

[2] L. Bom, R. Henken, and M. Wiering. Reinforcement learning to train ms. pac-man using higher-order action-relative inputs. In *IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning, ADPRL*, pages 156–163, 2013.

[3] S. Craw. *Manhattan Distance*, pages 639–639. Springer US, Boston, MA, 2010.

[4] Goulart, A. Paes, and E. Clua. *Learning How to Play Bomberman with Deep Reinforcement and Imitation Learning*, volume 11863 LNCS of *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics).* 2019.

[5] J. Ibarz, J. Tan, C. Finn, M. Kalakrishnan, P. Pastor, and S. Levine. How to train your robot with deep reinforcement learning: lessons we have learned. *The International Journal of Robotics Research*, 40(4-5):698–721, Jan 2021.

[6] G. Lample and D. S. Chaplot. Playing fps games with deep reinforcement learning, 2018.

[7] U. V. List. Bomberman series statistics. https://www.uvlist.net/groups/info/bomberman. Accessed: 2021-06-23.

[8] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. 12 2013.

[9] P. G. Patel, N. Carver, and S. Rahimi. Tuning computer gaming agents using q-learning. pages 581–588, 2011.

[10] G. Rummery and M. Niranjan. On-line q-learning using connectionist systems. *Technical Report CUED/F-INFENG/TR 166*, 11 1994.

[11] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall Press, USA, 3rd edition, 2009.

[12] H. M. Schwartz. *Multi-Agent Machine Learning: A Reinforcement Approach*, volume 9781118362082 of *Multi-Agent Machine Learning: A Reinforcement Approach.* 2014.

[13] D. Silver, A. Huang, C. Maddison, A. Guez, L. Sifre, G. Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–489, 01 2016.

[14] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, and D. Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362:1140–1144, 12 2018.

[15] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. Van Den Driessche, T. Graepel, and D. Hassabis. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359, 2017.

[16] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction.* The MIT Press, second edition, 2018.

[17] C. Watkins and P. Dayan. Technical note: Q-learning. *Machine Learning*, 8:279–292, 05 1992.