Tetris Battle With Q-Learning

Machine Learning Project Report from Team 38

Cheng – Ning, Huang
Project Manager
hcn1222@gapp.nthu.edu.tw

Hao – Wen, Hsu
Game Producer
vincent88588@gmail.com

Jun – Ping, Chou
Algorithm Designer
terrychou911019@gmail.com

Pei – Jen, Chen
Group Represented
penny11124@gmail.com

Po – Ching, Wen
Gathering Information
annieeric520@gmail.com

You – Cheng, Liu
Program Manager
liu.zack0505@gmail.com

***Abstract*—This project applies Q-Learning to train an agent for Tetris Battle, a multiplayer game. The objective is to surpass other AI agents' performance. The agent's training involves custom state representation and action space design. Results show superior performance in win rates and strategic gameplay compared to another agent from Team 37. The success highlights the potential of Q-Learning in enhancing multiplayer gaming.**

***Keywords*—Q-Learning, Tetris Battle, pygame**

I. Introduction

In recent years, the intersection of artificial intelligence and gaming has yielded remarkable advancements, showcasing the potential of reinforcement learning techniques in mastering complex game scenarios. Tetris Battle, a multiplayer adaptation of the classic Tetris game, provides an engaging and challenging environment for testing the capabilities of intelligent agents.

The motivation behind our topic stems from the increasing interest in developing agents that can adeptly navigate and excel in dynamic, multiplayer gaming environments. Tetris Battle, with its competitive nature, serves as an ideal platform to explore the application of Q-Learning, a widely used reinforcement learning algorithm, in training agents for strategic decision-making.

The central aim of our project is to employ Q-Learning to train an agent for Tetris Battle, with the goal of surpassing the performance of other AI-driven agents. We aim to demonstrate the effectiveness of Q-Learning in advancing the understanding of reinforcement learning techniques in real-world gaming applications.

This report is organized as follows: the next section reviews an introduction to Tetris Battle and Q-Learning. We then detail our methodology and experimental design, presenting and analyzing the results in subsequent sections.

II. Methods

A. Game Environment

In our quest to find a suitable game environment for our project, we discovered the pygame developed by ylsung[1]. This game offers two modes: Single Player Mode and Two Players Mode, aligning with our need to train a model and engage in battles with others.

At the beginning of the game, players control and place seven different shapes of blocks (Tetriminos) to form horizontal complete lines. When a line is filled, it disappears. Clearing two lines at once awards an additional 1 point, three lines yield an extra 2 points, and clearing four lines simultaneously results in an additional 4 points (Tetris). Achieving consecutive line clears generates a Combo, earning additional (combo + 1)/2 points (up to a maximum of 4 points). Moreover, players can employ the T-shaped block for a special rotation known as T-Spin, enabling the connection of multiple T-Spins to achieve a Combo. Successfully executing a T-Spin earns an extra 3 points. If consecutive line clears involve T-Spin or Tetris, a "Back-to-Back Combo" will be triggered, resulting in an additional 2 points. The game concludes when a player's stack of blocks reaches the top of the screen, displaying the final score.

In multiplayer battles, an additional "garbage" gameplay element is introduced. Successfully clearing a line not only earns points but also generates gray blocks on the opponent's screen, creating obstacles and making it more challenging to complete a line.

Players can score points by clearing lines, employing special techniques, or causing disruption in the opponent's area. Swift and strategic block placement, along with clearing multiple lines at once, are key strategies for achieving high scores.

B. Q-Learning

Q-Learning is a reinforcement learning algorithm used to solve problems where an agent operates in an environment, taking actions and receiving feedback. In Q-Learning, the agent aims to learn the optimal action policy to maximize its long-term cumulative reward in the environment. The key concepts include State, Action, Q-Value, Reward and Policy. "State" describes the possible conditions in the environment that affect the agent's behavior. "Action" is the operations that the agent can perform given a specific state. "Q-Value" means measuring the expected return for different actions in a particular state. Higher Q-Values indicate greater expected returns for the corresponding actions. "Reward" refers to Immediate feedback given by the environment after the agent performs an action. "Policy" is the optimal action strategy determined by Q-Values, i.e., choosing the action with the highest Q-Value in each state.

During the Q-Learning process, the agent updates Q-Values through trial-and-error learning, gradually refining its policy. The algorithm uses the Bellman equation to update Q-Values, considering the immediate reward and the potential maximum future reward for the current state.

C. Train Model

In our project, we first define the state representation encompasses four key metrics derived from the game board:

1. Lines Cleared: The number of rows cleared in the game.

2. Holes: The number of holes on the game board. Holes are defined as gaps or openings formed within the rows of the board where there is an empty space below filled spaces.

3. Bumpiness: Bumpiness quantifies the total absolute height differences between vertically adjacent columns on the board.

4. Height: Height represents the cumulative sum of the heights of blocks on the board.

After determining the state, the training process can be broadly divided into the following steps. Firstly, before deciding each action, generate all possible next states under the current state, predicting the potential landing positions of the falling blocks and computing state properties. Secondly, based on the current state, use an epsilon-greedy strategy to select actions, and store the experience in the replay memory. Our available actions include no action, hold, drop, rotate right, rotate left, shift right, shift left, and down. Among them, "hold" means allowing the player to save the current falling block in a reserve area, replacing it with the next falling block when needed. Only one block can be held at a time. Epsilon-greedy action selection involves comparing a random number 'u' with the current exploration rate 'epsilon.' If 'u' is less than or equal to 'epsilon,' a random action is chosen; otherwise, the action with the highest Q-value is selected.

After executing the action, interact with the environment, and calculate the reward and corresponding next state by using the reward function. After many attempts, we finally determined the basic reward function as follows:

$$\boldsymbol{reward} = \mathbf{1} + \mathbf{min}\ (\boldsymbol{lines\ cleard}, \mathbf{4})^{\mathbf{2}} \times \mathbf{10}$$

Furthermore, we also give some penalties for the unwanted situations. If end game, reward will be subtracted by 1000; if max height of blocks is greater than 10, subtract (max height-10) from the reward; if max height of blocks is greater than 15, subtract the square of the excess height beyond 15, multiplied by 5, from the reward. Expressed them mathematically as follows:

If max height > 10,

$$\boldsymbol{reward} -= (\boldsymbol{max\_height} - \mathbf{10})$$

If max height > 15,

$$\boldsymbol{reward} -= (\boldsymbol{max\_height} - \mathbf{15})^{\mathbf{2}} \times \mathbf{5}$$

When finish calculating, store the experience in the replay memory in the form of a quadruple (state, action, reward, next state). When the replay memory is full, a batch of experiences is randomly sampled from the replay memory. The current model is then used to predict the Q-values of the next state, calculate the loss, and perform backpropagation and optimization. Finally, every 1000 epochs, save the trained model. Our model used for the final evaluation is from epoch 10,000.

III. Results

A. *Battle Result*

On December 24, 2023, we engaged in a preliminary match against Team 37, employing the second version of our reward function. Unfortunately, in that round, we experienced defeat across all the matches. Subsequently, recognizing the need for improvement, we revised our approach and implemented the final version of the reward function as described earlier. The updated model showcased significant enhancement, leading to an impressive victory rate of 9 out of 10 rounds during the presentation on December 28. The process of refining the reward function and discussions will be elaborated upon in the forthcoming section.

B. *Single Mode Performance*

Noteworthy achievements extend beyond the realm of two-player mode; our AI model also demonstrated commendable performance in the single-player mode, where no garbage lines were introduced, and a time constraint was imposed. In this setting, our AI consistently sustained throughout the entire game duration and frequently garnered additional points through achievements such as back-to-back clears, Tetris formations, and combos. (see Fig 1. and Fig 2.)

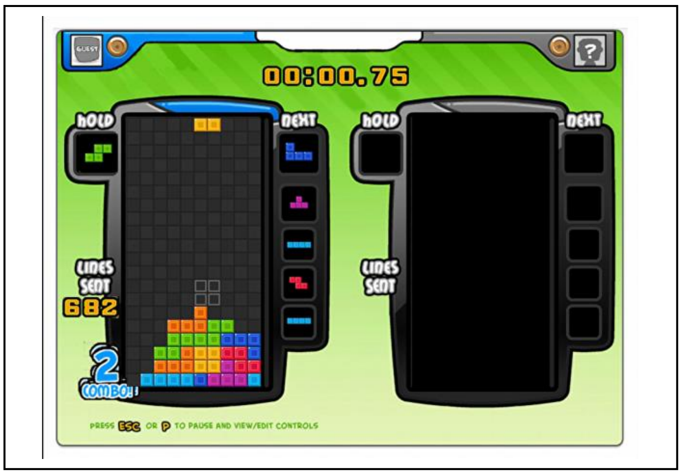

Fig 1. sustained throughout the entire game

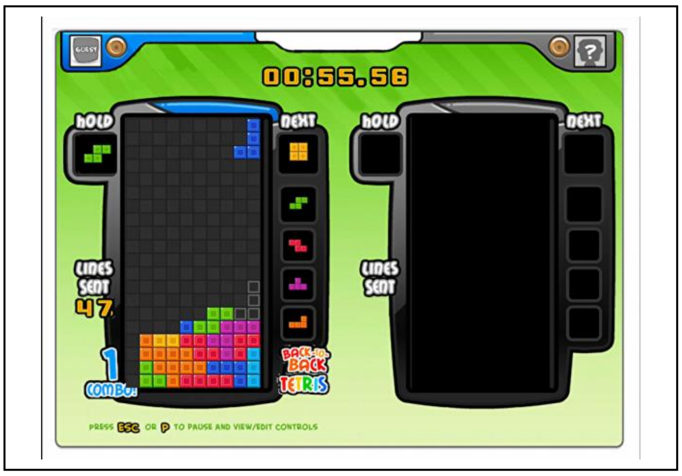

Fig 2. back-to-back, Tetris formations, combos

IV. Discussion And Conclusion

In this section, we will delve into the iterative process of modifying our reward function, outlining the adjustments made to enhance the performance of our AI agent in Tetris Battle. In addition, the contributions of team members will be listed in the latter section.

A. *Modify Reward Function*

Initially, our reward function was simplistic, using only the number of cleared lines as a reward, defined by the equation:

$$\boldsymbol{reward} = \boldsymbol{lines\ cleared}^{\mathbf{2}}$$

However, we observed that the AI struggled to control the stacking height of blocks. To address this, we shifted to training based on the average height of each block. Consequently, we noticed a tendency for the AI to initially build vertically and then transition to horizontal stacking.

To refine the behavior further, we introduced a penalty for creating holes. This adjustment led the AI to penalize actions resulting in lower gaps between heights and holes. If the penalty weight for holes was too significant, the AI would favor building tall towers vertically. Conversely, if the weight for height was too substantial, it would start creating larger holes, effectively providing a slight increase in height.

Subsequently, we incorporated the difference in heights between blocks and the adjacent columns. This modification led the AI back to constructing slanted structures with large holes.

In further experimentation, we considered the count of blocks and empty spaces along the boundaries. However, the AI still tended to favor building tall towers to avoid penalties associated with block and boundary constraints.

During the match against Team 37 on December 24th, we utilized the following reward function:

$$\boldsymbol{reward} \\ = \mathbf{1} + \boldsymbol{lines\ cleared}^{\mathbf{2}} - \boldsymbol{RELU}(\boldsymbol{max\_height} - \mathbf{10})$$

Due to suboptimal results, we continued refining the approach. Ultimately, by intensifying penalties for death and height, and introducing the infrequently used but strategic "hold" action in the model's training, our AI demonstrated significant improvement and prowess in battles.

B. *Author Contribution Statements*

TABLE I. AUTHOR CONTRIBUTION

Team Member	**Contribution**	
	Works	***Percentage***
Cheng – Ning, Huang	Team leader, recording discussion, fixing the bug, training models, final presentation	17%
Hao – Wen, Hsu	Study design, programming main structure, design training methods, fixing the bug	17%
Jun – Ping, Chou	Study design, programming main structure, design training methods	17%
Pei – Jen, Chen	Training models, final presentation, final project report	16%
Po – Ching, Wen	Data collection, training models, final presentation	16%
You – Cheng, Liu	Study design, programming of the interface between game and model, fixing the bug	17%

DATA AND CODE AVAILABILITY

Here is our project code on GitHub:

https://github.com/penny11124/Tetris-Battle-With-Q-Learning

REFERENCES

Websites:

Uvipen. (2020). [PYTORCH] Deep Q-learning for playing Tetris. [2]

shaoeChen. (2022). 李宏毅_DRL Lecture 3: Q-learning (Basic Idea) [3]

[1] https://github.com/ylsung/TetrisBattle?tab=readme-ov-file *(Tetris)*

[2] https://github.com/uvipen/Tetris-deep-Q-learning-pytorch?tab=MIT-1-ov-file

[3] https://hackmd.io/@shaoeChen/Bywb8YLKS/https%3A%2F%2Fhackmd.io%2F%40shaoeChen%2FSyqVopoYr