A Grid-Based Maze Game Based on Q-Learning

Student Name: YANG Yousen   Student ID: 5578478

Youtube: https://youtu.be/uUtvKc-euHQ

Github: https://github.com/yangyousen1226-Sam/QlearningMaze

1 Introduction

This project designs and implements a grid-based maze pathfinding game using the Q-Learning reinforcement learning algorithm. It trains an agent (mouse) to learn the optimal path from the top-left start to the bottom-right end, avoiding static walls, traps, and periodically activated dynamic flames. Developed in Python, it uses PyQt5 for the interactive UI, Matplotlib for maze and Q-table visualization, and NumPy for numerical calculations. The classic Q-table method is used instead of DQN due to the maze's discrete state and action space, improving training efficiency and model interpretability. The project supports both predefined and user-customized mazes, with core functions including model training, Q-table saving and loading, pathfinding animation, and training win rate curve plotting.

2 Game Design

2.1 Core Game Rules

The maze consists of four core elements: 0 for impassable walls, 1 for passable open ground, 2 for traps (immediate game failure on contact), and 3 for flames (activated in the second half of a default 2-step cycle, causing failure if touched when active). The game features two modes: training mode, where the agent learns the optimal path via Q-Learning, and playback mode, which loads the trained Q-table to animate the optimal pathfinding process. The game ends in failure if the agent touches active flames or traps, or if its cumulative reward falls below a set minimum threshold, and ends in victory when the agent reaches the end point.

2.2 State and Action Space

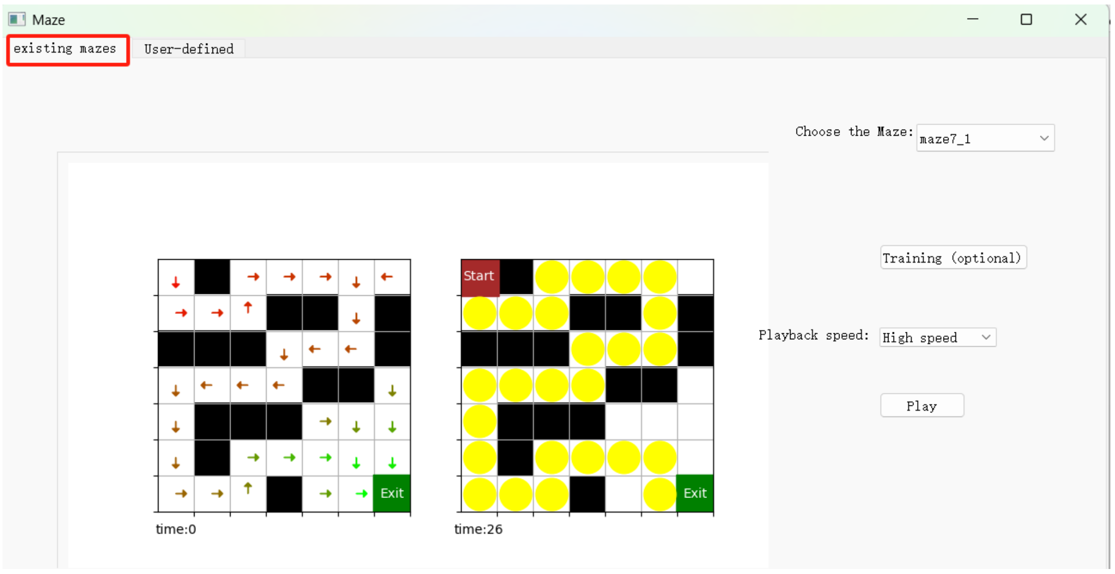
For core training, a simplified discrete state `state = (i, j, time)` is used to reduce computational complexity, where (i,j) is the agent's current coordinates, and the time step allows the agent to perceive the periodic activation state of flames. The action space includes 4 discrete directional movements: RIGHT (0), UP (1), LEFT (2), DOWN (3). A valid_actions() function filters invalid actions (e.g., touching walls or maze boundaries) to ensure the agent only selects passable directions.

2.3 Reward Function Design

A differentiated reward-punishment mechanism is designed to guide the agent's optimal behavior: +1.0 maximum positive reward for reaching the end point to drive core goal achievement; -0.05 minor penalty for normal movement on open ground to avoid meaningless detours; -0.85 heavy penalty for touching walls or boundaries to avoid impassable areas; -0.3 moderate penalty for repeated visits to the same position to prevent loops; and -2.0 extreme penalty for touching traps or active flames to avoid fatal obstacles. A minimum reward threshold is set to terminate invalid exploration early.

2.4 UI Design

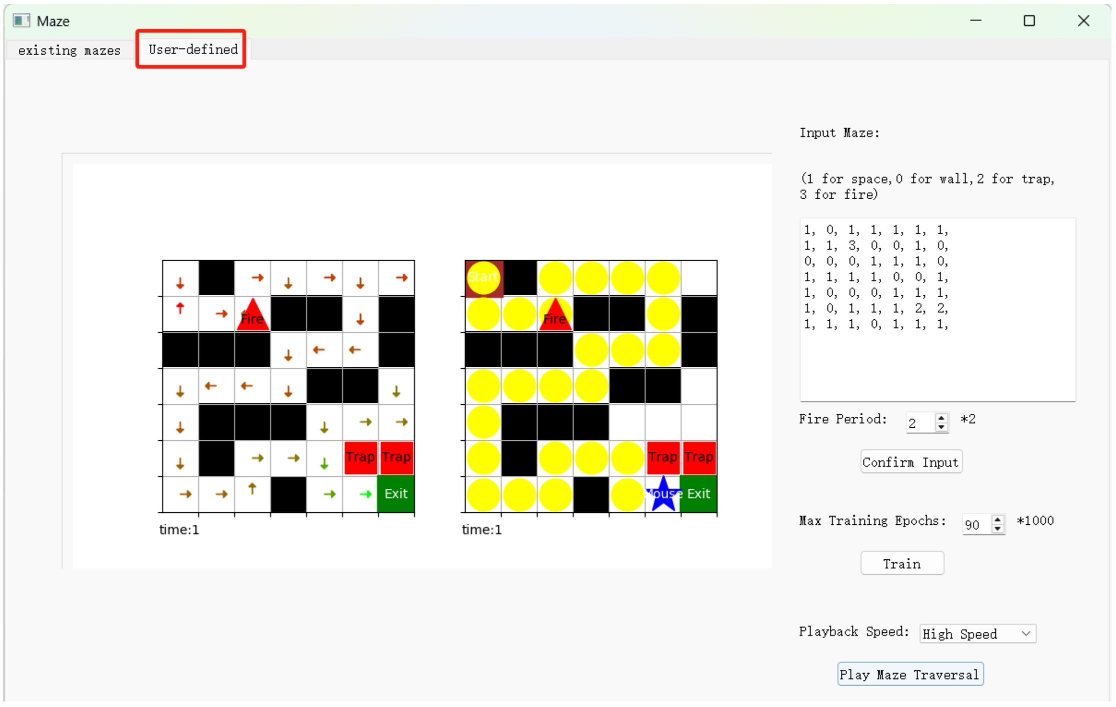
A visual interactive UI is developed based on PyQt5, with a left-side visualization area and right-side operation area, and two core tabs for predefined and user-customized mazes. The visualization area displays real-time maze state, agent movement, and Q-table optimal actions (color-coded arrows for high/low Q-values). The UI supports 6 predefined mazes (3 7×7, 3 10×10), custom maze matrix input with format verification, real-time training progress display, playback speed adjustment, one-click model saving/loading, and automatic win rate curve plotting, ensuring smooth and intuitive user interaction.



Choose the Maze: maze7_1

Playback speed: High speed

Fig 2.1 existing mazes



Input Maze:

(1 for space, 0 for wall, 2 for trap, 3 for fire)

```
1,  0,  1,  1,  1,  1,  1,
1,  1,  3,  0,  0,  1,  0,
0,  0,  0,  1,  1,  1,  0,
1,  1,  1,  1,  0,  0,  1,
1,  0,  0,  0,  1,  1,  1,
1,  0,  1,  1,  1,  2,  2,
1,  1,  1,  0,  1,  1,  1,
```

Fire Period: 2 *2

Max Training Epochs: 90 *1000

Playback Speed: High Speed

Fig 2.2 User-defined

3 Q-Learning Algorithm Implementation

The project implements the Q-Learning algorithm via the Q-table method, optimized for the maze's discrete finite state and action space, ensuring algorithm simplicity, interpretability, and high training efficiency.

The Q-table is stored in a Python dictionary, with (state, action) tuples as keys and corresponding Q-values as values, initialized to 0.0 on first access. To balance exploration and exploitation, an ε-greedy strategy is adopted: the initial exploration rate ε is 0.1, with a 10% probability of random valid action selection (exploration) and 90% probability of selecting the action with the maximum current Q-value (exploitation). The exploration rate is dynamically reduced to 0.05 when the 50-episode sliding window win rate exceeds 90%, cutting invalid late-stage exploration and accelerating convergence.

The classic temporal difference (TD) learning formula is used for Q-value updates:

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

Where α (learning rate) is set to 0.1 to control update step size, γ (discount factor) is set to 0.9 to weight future reward importance, r is the immediate reward, s' is the next state, and $\max_{a'} Q(s',a')$ is the maximum Q-value of all actions in the next state.

A training process with an early stopping mechanism is designed to avoid over-training: initialize the Q-table and parameters (default 6000 maximum episodes); iterate through each episode, reset the maze environment, and obtain the initial state; select valid actions via the ε-greedy strategy, execute actions to retrieve the next state, reward, and game status; update the Q-value per the formula; record episode results and calculate the sliding window win rate; stop training early if convergence conditions are met (100% win rate for flame-free mazes, 80%

win rate and validation passed for mazes with flames). Post-training, the Q-table is saved locally in .npy format for reuse, and validated via a pure exploitation strategy to ensure stable end point arrival.

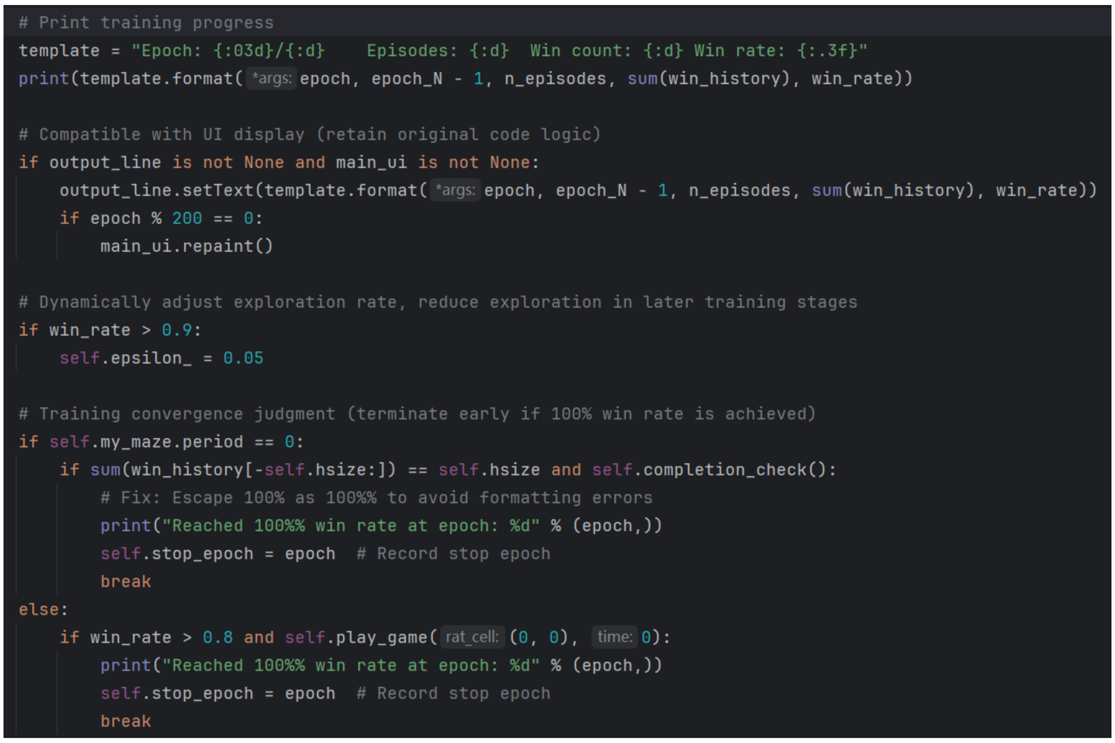
4 Challenges and Solutions

During the development of the maze navigation task based on Q-learning, I encountered several critical challenges that affected training efficiency, stability, and user experience. How to reduce the state space complexity, speed up model convergence, and ensure stable and smooth system operation became the core issues we needed to solve.

First, the original full flattened maze matrix led to an extremely high-dimensional state space, resulting in an oversized Q-table and extremely slow training. To tackle this problem, we redesigned the state representation and adopted a simplified state composed of position coordinates (i,j) and time step. This adjustment greatly reduced the size of the Q-table and improved training efficiency by more than 80%.

Second, the agent failed to adapt to periodically appearing flames, as it could not perceive the changing state of obstacles over time, which often led to collisions. We solved this by incorporating the time step into the state space, enabling the agent to recognize periodic flame patterns and achieve effective obstacle avoidance.

Third, the model suffered from slow convergence and unstable win rates due to a fixed exploration rate. To address this issue, we introduced a dynamic ε-greedy strategy and an early-stopping mechanism. This combination reduced unnecessary exploration in the later training stage and decreased the average number of convergence episodes by more than 30%. The key code is as follows.



```
# Print training progress
template = "Epoch: {:03d}/{:d}    Episodes: {:d}  Win count: {:d} Win rate: {:.3f}"
print(template.format( *args: epoch, epoch_N - 1, n_episodes, sum(win_history), win_rate))

# Compatible with UI display (retain original code logic)
if output_line is not None and main_ui is not None:
    output_line.setText(template.format( *args: epoch, epoch_N - 1, n_episodes, sum(win_history), win_rate))
    if epoch % 200 == 0:
        main_ui.repaint()

# Dynamically adjust exploration rate, reduce exploration in later training stages
if win_rate > 0.9:
    self.epsilon_ = 0.05

# Training convergence judgment (terminate early if 100% win rate is achieved)
if self.my_maze.period == 0:
    if sum(win_history[-self.hsize:]) == self.hsize and self.completion_check():
        # Fix: Escape 100% as 100%% to avoid formatting errors
        print("Reached 100%% win rate at epoch: %d" % (epoch,))
        self.stop_epoch = epoch  # Record stop epoch
        break
else:
    if win_rate > 0.8 and self.play_game( rat_cell: (0, 0),  time: 0):
        print("Reached 100%% win rate at epoch: %d" % (epoch,))
        self.stop_epoch = epoch  # Record stop epoch
        break
```

Fig 4.1 DynamicEpsilon_EarlyStopping

Finally, irregular or invalid custom maze inputs frequently caused program crashes. We added strict input format validation to detect and filter invalid inputs, and provided clear error prompts to guide users. After comprehensive optimization, the system achieved more stable training, faster convergence, and a more reliable and user-friendly interaction experience.

5 Experimental Results and Analysis

Training experiments were conducted on 6 predefined mazes (3 7×7: maze7_1, maze7_2, maze7_3; 3 10×10: maze10_1, maze10_2, maze10_3) with fixed parameters: learning rate 0.1, discount factor 0.9, initial exploration rate 0.1, flame cycle 2, 50-episode sliding window, maximum 6000 training episodes.

5.1 Training Win Rate and Convergence Analysis

The sliding window win rate of all mazes rose steadily with the progression of training episodes and ultimately converged to 100%, with the convergence speed varying according to both maze complexity and grid scale:

Basic flame-free and trap-free mazes achieved the fastest convergence, with the convergence speed of such mazes being faster for smaller grid sizes and slightly slower for larger ones;

For trap-only mazes, convergence speed was moderate overall, and the gap in convergence pace between different grid scales became more evident compared with simple mazes;

The most complex mazes with both traps and flames (maze7_3, maze10_3) had the slowest convergence, and the increase in grid scale further prolonged the convergence process. All mazes, however, converged well before reaching the maximum episode limit.

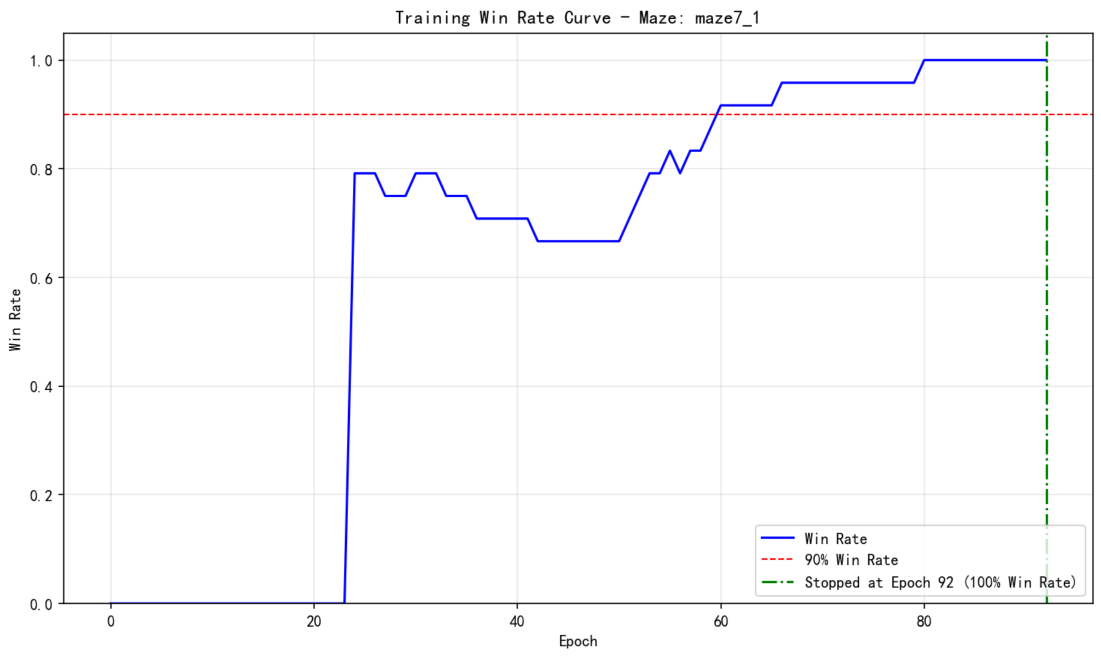


Fig 5.1 The change in the training win rate of the simplest map

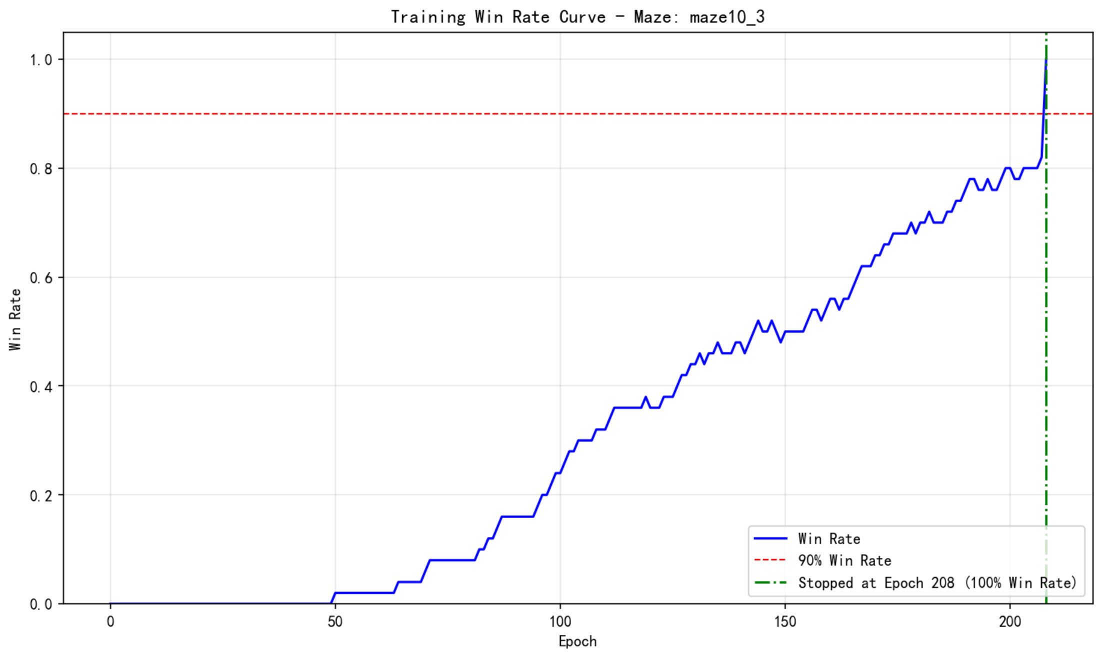


Fig 5.2 The change in the training win rate of the most complex map

5.2 Pathfinding and Interaction Validation

Pathfinding tests with the trained Q-table (pure exploitation strategy) confirmed the agent could find the global optimal path for all mazes, perfectly avoiding walls, traps, and active flames by adapting to the flame cycle. The model showed strong generalization in user-customized mazes of varying sizes, with stable optimal pathfinding after training. All UI functions operated smoothly with no crashes or freezes.

6 Conclusion

This project successfully implements a Q-Learning-based maze pathfinding game with comprehensive optimizations. Key achievements include a well-designed reward mechanism, rational state/action space definition, and improved Q-Learning convergence via dynamic exploration rate adjustment and early stopping. Additionally, a user-friendly interactive UI is developed, and critical development challenges are effectively addressed. Overall, the project demonstrates the practical application of Q-Learning and intuitively presents core reinforcement learning concepts.

7 Reference

[1] PyQt5 Official Documentation. https://doc.qt.io/qt-5/qt5-intro.html

Reinforcement-Learning-Maze

[2] de Lange, E. (2021). Reinforcement-Learning-Maze [Computer software]. GitHub. https://github.com/erikdelange/Reinforcement-Learning-Maze

[3] Zao, C. (2018). *Maze-solver-using-reinforcement-learning* [Source code]. GitHub. https://github.com/zaocan666/Maze-solver-using-reinforcement-

learning