QlearningDungeonBattle

Student Name: Xue Tingkun

Student ID: 5563215

This project applies Q-learning to the design of a 2D top-down action game where a human-controlled player fights against an AI-controlled enemy in a single room. The visual and gameplay style are inspired by “The Binding of Isaac”: the player moves in a grid-based dungeon room, uses keyboard controls to dodge and shoot, and tries to survive under the pressure of an intelligent enemy. The enemy’s movement policy is learned by a Deep Q-Network (DQN) interacting with a simplified environment that shares the same map layout as the actual game. To make the game engaging and suitable for a demonstration video, we add a health system, hit effects, and a start menu with in-game UI. Although the pure DQN agent struggles to surpass a hand-crafted chasing rule on complex maps, combining DQN with a rule-based policy on a simplified cross-shaped map yields reasonable behaviour and provides a clear example of how Q-learning can be used in game AI.

1. Introduction

Reinforcement learning (RL) studies how an agent can learn an optimal policy by interacting with an environment and receiving rewards. Q-learning is a classical off-policy RL algorithm that estimates the action-value function and can, in principle, learn optimal behaviour without a model of the environment. However, when the state space is continuous or high-dimensional, tabular Q-learning is infeasible and neural networks are used to approximate the Q-function, leading to Deep Q-learning (DQN).

The goal of this project is to design and implement a small game where a human player controls a character and an AI enemy attempts to chase and defeat the player using Q-learning. The game must have a clear set of rules, a well-defined state and action space, and a reward function suitable for Q-learning. In addition, the project requires a user interface that makes the game easy to play and to demonstrate in a video, as well as a written report that explains the design choices and evaluates the performance of the learned agent.

2. Game Design

2.1 Map and Characters

The game uses a grid-based map of size 28×18 cells, with each cell rendered as a 32×32 pixel tile in Pygame. The room contains a simple cross-shaped obstacle structure:

- One horizontal wall approximately across the middle of the room.
- Two short vertical walls close to the left and right edges.
- Wide gaps are left in the walls so that both the player and the enemy can pass through and navigate around the obstacles.

There are two main entities:

- Player: A green square controlled by the human via keyboard. The player can move in four directions and aim independently.
- Enemy: A blue square controlled either by a hand-crafted rule or by a DQN policy. The enemy's goal is to chase and collide with the player.

Both entities share the same health system:

- Each has 3 hit points (HP).
- When the enemy touches the player, the player loses 1 HP and briefly flashes a

lighter colour to indicate damage.

- When the player’s bullet hits the enemy, the enemy loses 1 HP and flashes yellow.

- When HP reaches 0, that entity dies and the round ends.

2.2 Controls and UI

The control scheme is designed to mimic classic twin-stick shooters:

- Movement: W, A, S, D.

- Aiming: Arrow keys (↑, ↓, ←, →) rotate a small arrow drawn on the player to indicate the current shooting direction.

- Shooting: SPACE; bullets travel in a straight line in the aimed direction and disappear when they hit walls or leave the room.

The game includes a simple user interface:

- Start menu: On launch, the game shows a menu with the title “Q-learning Dungeon Battle” and textual instructions in English:
    - “Move: WASD, Aim: Arrow keys, Shoot: SPACE”
    - “Both player and enemy have 3 HP”
    - “Press ENTER to start, R to restart, ESC to quit”
- In-game HUD:
    - Top-left corner: “Player HP: x/3”.
    - Top-right corner: “Enemy HP: x/3”.
    - When the round ends, a large message “Player Wins – Press R to restart” or “Enemy Wins – Press R to restart” is shown in the centre.

2.3 Win and Lose Conditions

A round of the game ends in one of two ways:

- Player win: The enemy’s HP decreases to zero due to being hit by bullets three times.
- Enemy win: The player’s HP decreases to zero after three collisions with the enemy.

The player can press R at any time to reset positions, HP, and bullets, and start a new round, or press ESC to exit.

3. Reinforcement Learning Formulation

3.1 State Representation

We formulate the RL problem from the enemy’s point of view. At each time step the state is encoded as an 11-dimensional continuous feature vector:

1. Relative position of the player:
    - dx = (player_x – enemy_x) / GRID_WIDTH
    - dy = (player_y – enemy_y) / GRID_HEIGHT
2. Normalised Manhattan distance:
    - dist = (|dx_cells| + |dy_cells|) / (GRID_WIDTH + GRID_HEIGHT)
3. Local obstacle information around the enemy:
    - up_block: 1 if the cell above is a wall or outside the map, else 0.
    - down_block: 1 if the cell below is blocked.
    - left_block: 1 if the cell to the left is blocked.

- right_block: 1 if the cell to the right is blocked.

4. Normalised absolute positions:

   - px_norm = player_x / GRID_WIDTH

   - py_norm = player_y / GRID_HEIGHT

   - ex_norm = enemy_x / GRID_WIDTH

   - ey_norm = enemy_y / GRID_HEIGHT

This representation balances simplicity and informativeness: it gives the DQN awareness of the relative direction of the player, approximate distance, and whether immediate moves are blocked by walls, without encoding the full map as an image.

3.2 Action Space

The enemy's action space consists of four discrete actions:

- 0: move one cell up

- 1: move one cell down

- 2: move one cell left

- 3: move one cell right

If an action would cause the enemy to step into a wall or outside the map, the position remains unchanged.

3.3 Reward Function

The reward function is designed to encourage the enemy to approach and eventually catch the player while discouraging idle wandering:

- Distance change:
    - If the Manhattan distance to the player decreases: +0.1.
    - Otherwise: −0.1.
- Close-range pressure:
    - If the new distance is less than or equal to 3 cells: additional +0.05.
- Step penalty:
    - Each time step incurs −0.01 to encourage faster completion.
- Catching the player:
    - If the enemy and player occupy the same cell: +10.0 and the episode terminates.
- Timeout penalty:
    - If the maximum step limit (400 steps) is reached without catching the player: −5.0 and the episode terminates.

The player in the training environment is not controlled by a human but by a scripted policy that mostly moves away from the enemy, with some randomness added. This provides a moving target that the enemy needs to track.

3.4 DQN Architecture and Training

We use a standard DQN setup implemented in PyTorch:

- Network architecture:
    - Input layer: 11 units (state features).
    - Two hidden fully connected layers with 128 units each and ReLU activation.
    - Output layer: 4 units, giving Q-values for the four actions.

- Training details:
    - Optimiser: Adam with learning rate 1×10^{-3}.
    - Discount factor: $\gamma = 0.99$.
    - Replay memory size: 50 000 transitions.
    - Mini-batch size: 64.
    - ε-greedy exploration: ε starts at 1.0 and decays towards 0.05 with factor 0.997.
    - Target network: parameters are copied from the policy network every 10 episodes to stabilise learning.

The environment used for training mirrors the game map and obstacle layout but omits bullets and HP for simplicity: episodes terminate when the enemy catches the scripted player or when the step limit is reached.

4. Implementation and Hybrid Control

The game logic, UI, and physics are implemented with Pygame. The RL environment is implemented as a separate class that exposes reset and step functions and shares the same wall layout as the visual game. The DQN and training loop are implemented in a separate Python script.

In practice, we observed that a pure DQN agent trained on more complex maze-like layouts tended to get stuck near obstacles and performed worse than a simple hand-crafted chasing rule that always moves along the axis with the larger distance to the player. Even on the simplified cross-shaped map, the DQN agent alone sometimes exhibits sub-optimal behaviour. To obtain better in-game behaviour while still demonstrating Q-learning, the final playable version uses a hybrid controller in which:

- Both the DQN policy and the rule-based policy are evaluated at each decision step.

- With a certain probability (e.g. 50%), the enemy takes the rule-based action.
- Otherwise, it takes the DQN action.

This hybrid approach preserves the stability and predictability of the rule-based AI, ensuring that the enemy continues to chase the player, while allowing the learned policy to influence local movement decisions.

5. Results and Discussion

Qualitatively, the hybrid enemy on the simplified cross-shaped map behaves noticeably better than the pure DQN agent on the earlier maze-like maps. The enemy is able to:

- Move roughly towards the player instead of staying in distant corners.
- Navigate through the wide gaps in the walls and maintain pressure on the player.
- Recover from some local oscillations near obstacles due to the contribution of the rule-based policy.

However, the project also highlights several limitations:

- The state representation is local and does not encode the full map layout, which makes it difficult for a pure DQN agent to plan multi-step routes around obstacles.
- The scripted player introduces stochasticity; sometimes the enemy receives positive distance rewards due to the player accidentally moving closer, which adds noise to the learning signal.
- Even after increasing the number of training episodes, tuning rewards, and simplifying the map, the pure DQN agent does not reliably outperform the simple rule-based chasing strategy.

From a game design perspective, the health system, hit effects, and English UI significantly improve the playability and make the behaviour of the AI easier to observe and explain in a demonstration video. The player can clearly see when damage occurs and can judge whether the enemy is successfully closing in over time.

6. Conclusion and Future Work

This project implemented a small but complete pipeline for applying Q-learning to game AI: from designing a grid-based dungeon room and player controls, through building an RL environment and DQN training script, to integrating the learned policy into an interactive Pygame application with a start menu and in-game UI. Although the pure DQN enemy is not yet as strong as the hand-crafted chasing rule, the hybrid control scheme demonstrates a practical way to combine reinforcement learning with classic game AI techniques.

Future work could explore several directions:

- richer state features, such as local occupancy grids or short action histories, to give the agent more information about the environment;
- more advanced DQN variants, such as Double DQN, dueling architectures, or prioritised replay, to improve stability and performance;
- curriculum training, starting from an open map without walls and gradually introducing obstacles;
- extending the health system so that the enemy also learns to avoid bullets, not just chase the player.

Overall, the project meets the assignment requirements by designing a non-trivial game, implementing Q-learning in a meaningful way, and critically evaluating the behaviour of the learned agent in comparison with a rule-based baseline.

GitHub repository:

XueTingkun/Qlearning-dungeon-battle: Game overview - Player controls a green square using WASD to move and arrow keys to aim. - SPACE shoots bullets; both player and enemy have 3 HP. - The room contains simple cross-shaped walls, so both sides need to navigate around obstacles. - When HP reaches 0, the round ends and can be restarted with R.

Project overview:

- The player controls a green square, moving with WASD and aiming with the arrow keys.

- SPACE shoots bullets; both the player and the enemy have 3 HP.

- The room contains a simple cross-shaped wall structure, so both sides must navigate around obstacles.

- When HP reaches zero, the round ends and can be restarted with R.

YouTube demo link: https://youtu.be/piZ_CtX_1O4?si=xbr15XH58WyLvZh0