Project Background

This project is inspired by the classic RTS game "Red Alert 3" and constructs a simplified turn-based strategy confrontation scenario: players control 10 infantry units to confront a powerful Apocalypse tank that has the ability to crush and attack in an area. Traditional game AI relies on pre-programmed rules, while this project innovatively uses the Q-learning reinforcement learning algorithm to enable infantry units to form optimal combat strategies through autonomous learning.

Video Link: https://drive.google.com/file/d/1eMiAzq4YBNE3Hugr_GHDW3v9E791aPqy/view?usp=drive_link

Github Link:https://github.com/fangxweiaidedbtx/Q-learning-Demo.git

Challenges:

The core objective is to train the agent to maximize damage to the tank while minimizing casualties through algorithmic training.

Dynamic Battlefield Environment: The tank has a 3×3 crushing area and long-range attacks.

Multi-Unit Coordination: The movement and attack rhythm of 10 infantry units need to be coordinated.

Exploration and Exploitation Balance: Finding safe attack windows in the unknown movement patterns of the tank.

Optimization Goals:

Maximize: Σ(Damage to the tank per turn)

Minimize: Σ(Number of infantry crushed)

Game Rules

1.Basic Rules

Victory Condition	Tank HP ≤ 0 or all infantry HP ≤ 0
Turn Process	Infantry action → Tank movement/crushing → Tank attack → Environment settlement
Battlefield Specifications	60×60 grid

2.Unit And Tank Parameters

```python
class Infantry:
    def __init__(self, x, y):
        self.hp = 8          # Health points
        self.attack_power =1 # Attack power
        self.range = 18      # Attack range (Manhattan distance)
        self.movement =4     # Maximum movement steps per turn
        self.moved = False   # Action status flag
```

```python
class Tank:
    def __init__(self, x, y):
        self.hp = 75          # Health points
        self.attack_power=2   # Single-target attack power
        self.movement=7       # Movement per turn
        self.size = (3,3)     # Occupies a 3x3 grid
        self.route = []        # Movement path record
```

3. Special Rules

(1) Infantry units in the 3x3 area covered by the tank's movement path are immediately killed .

(2) Infantry units deal damage when their distance to the tank is ≤ range.

(3) Infantry units can only move or attack once per turn.

Game Implementation

Tank Movement Decision and Path Planning

1.Movement Decision Logic (from TankAI.decide_movement)

- **Target Selection:**

Iterate through all surviving infantry units and calculate the distance

distance = |target.x - tank.x| + |target.y - tank.y|.

Select the infantry unit with the shortest distance as the movement target.

```python
min_distance = 1000
target_infantry = None
for infantry in infantry_units:
    distance = abs(infantry.x - self.tank.x) + abs(infantry.y - self.tank.y)
    if distance < min_distance and infantry.hp > 0:
        min_distance = distance
        target_infantry = infantry
```

- **Path Planning**

If the target is within the tank's movement range (Manhattan distance ≤ 7), move directly towards the target (dx, dy).

If it exceeds the movement range, prioritize horizontal movement to the maximum number of steps, and then use the remaining steps for vertical movement.

```
def find_shortest_path(self, start_x, start_y, end_x, end_y,tank):
    dx = end_x - start_x
    dy = end_y - start_y
    if abs(dx) - tank.movement > 0:
        return (tank.movement*dx//abs(dx), 0)
    else:
        return (dx,(tank.movement-abs(dx))*dy//abs(dy))
```

- Movement Execution (Tank.move)

Record the movement path to the route list for subsequent crushing judgment.

Update the tank's coordinates (x, y).

```
class Tank:
    .......
    def move(self, path):
      new_x = self.x + path[0]
      new_y = self.y + path[1]
      for x in range(min(self.x, new_x), max(self.x, new_x) + 1):
          self.route.append((x, self.y))
      self.x = new_x
      for y in range(min(self.y, new_y), max(self.y, new_y) + 1):
          self.route.append((self.x, y))
      self.y = new_y

dx = target_infantry.x - self.tank.x
dy = target_infantry.y - self.tank.y
if abs(dx) + abs(dy) > self.tank.movement:
    path = self.find_shortest_path(self.tank.x, self.tank.y, target_infantry.x,
target_infantry.y, self.tank)
            self.tank.move(path)
        else:
            self.tank.move((dx, dy))
```

2. Tank Attack Decision (TankAI.decide_attack)

Iterate through all infantry units and select the first surviving unit that enters the attack range .

```
def decide_attack(self, infantry_units: list):
    for infantry in infantry_units:
        if infantry.hp <= 0:
            continue
        if abs(infantry.x - self.tank.x) + \
            abs(infantry.y - self.tank.y) <= self.tank.range:

            infantry.hp -= self.tank.attack_power
            break
```

3. Map Scene Rendering (panal.py)

Grid Drawing:

Map the logical grid (60×60) to a 1400×1200 pixel area on the screen.

Cell size: CELL_WIDTH = (1400 - 200) // 60 = 20px.

```
WINDOW_WIDTH = 1400
WINDOW_HEIGHT = 1200
BOARD_WIDTH = 60
BOARD_HEIGHT = 60
CELL_WIDTH = (WINDOW_WIDTH - 200)  // BOARD_WIDTH
CELL_HEIGHT = WINDOW_HEIGHT // BOARD_HEIGHT

PANEL_WIDTH = 200
PANEL_X = WINDOW_WIDTH - PANEL_WIDTH


for x in range(BOARD_WIDTH):
    for y in range(BOARD_HEIGHT):
        rect = pygame.Rect(x * CELL_WIDTH, y * CELL_HEIGHT, CELL_WIDTH, CELL_HEIGHT)
        pygame.draw.rect(screen, "#2F4F4F", rect, 1)
```

Unit Rendering:

Infantry: 1×1 picture.

Tank: 3×3 picture, occupying 9 grid cells.

```
infantry_image = pygame.image.load("unit.png")
infantry_image = pygame.transform.scale(infantry_image, (CELL_WIDTH, CELL_HEIGHT))

tank_image = pygame.image.load("tank.png")
tank_image = pygame.transform.scale(tank_image, (CELL_WIDTH * 3, CELL_HEIGHT * 3))

for infantry in player_units:
    rect = pygame.Rect(infantry.x * CELL_WIDTH, infantry.y * CELL_HEIGHT,
CELL_WIDTH, CELL_HEIGHT)
    if infantry.hp <= 0:
        continue
    screen.blit(infantry_image, rect.topleft)
    if infantry.selected:
        pygame.draw.rect(screen, "#00FFFF", rect, 1)
    elif not infantry.moved:
        pygame.draw.rect(screen, "#4444AA", rect, 1)
    else:
        pygame.draw.rect(screen, RED, rect, 1)

tank_rect = pygame.Rect(
    tank.x * CELL_WIDTH - CELL_WIDTH,
    tank.y * CELL_HEIGHT - CELL_HEIGHT,
    CELL_WIDTH * tank.size[0],
    CELL_HEIGHT * tank.size[1]
)
screen.blit(tank_image, tank_rect.topleft)
pygame.draw.rect(screen, BLACK, tank_rect, 1)
```

Health Bar Panel:

A 200px wide gray panel on the right side, dynamically updating the status of units.

Use pygame.font to render text.

```
def draw_hp_panel(tank,player_units,screen):
    pygame.draw.rect(screen, (128, 128, 128), (PANEL_X, 0, PANEL_WIDTH,
WINDOW_HEIGHT))
    font = pygame.font.Font(None, 36)
    tank_hp_text = font.render(f"Tank HP: {tank.hp}", True, WHITE)
    screen.blit(tank_hp_text, (PANEL_X + 10, 10))
    for i, infantry in enumerate(player_units):
        infantry_hp_text = font.render(f" {i + 1} HP: {infantry.hp}", True, WHITE)
        screen.blit(infantry_hp_text, (PANEL_X + 10, 50 + i * 30))
```

Implementation of Reinforcement Learning

1. Environment Definition

step(): Receives a list of actions for the 10 infantry units and executes the following process in order:

- Infantry Movement: Each infantry unit executes its action (movement vector (dx, dy)).
- Tank Action: Calls TankAI to decide the movement path and attack target.
- Reward Calculation: Computes the immediate reward based on the survival status of the infantry and the effectiveness of the attack.
  - Positive Rewards:Effective Attack: +6 reward is given when infantry enters the attack range and attacks the tank.
  - Negative Punishments:
    - Crushed: -35 punishment is given when infantry is covered by the tank's movement path.
    - Ineffective Movement: -1 reward is deducted for each step when not entering the attack range.
- State Update: Generates a new state and returns it to the agent.

```
def step(self, actions):
    rewards = [0 for i in range(10)]
    for index,unit in enumerate(self.player_units):
        unit.move(actions[index][0],actions[index][1])

    under_crush_units = []
    self.tank_ai.decide_movement(self.player_units)
    for x, y in self.tank.route:
        for i in self.tank.crush(x, y, self.player_units):
            #被碾压了但是未被记录
            if i not in under_crush_units:
                under_crush_units.append(i)
    for index in under_crush_units:
        rewards[index] -= 35
    self.tank.route = []
    self.tank_ai.decide_attack(self.player_units)
    for index,infantry in enumerate(self.player_units):
        if infantry.hp <= 0:
            continue
        if abs(infantry.x - self.tank.x) + abs(infantry.y - self.tank.y) <=
infantry.range:
            self.tank.hp -= infantry.attack_power
            rewards[index] += 6
        else:
            rewards[index] -= 1
```

reset(): Resets the battlefield state, including:

- Randomly generating the initial positions of the 10 infantry units (in the right half of the logical grid).
- Resetting the tank to a fixed position (BOARD_WIDTH - 5, BOARD_HEIGHT // 2).

```
def reset(self):
    self.game_end = False
    self.player_units = [Infantry(random.randint(BOARD_WIDTH // 2, BOARD_WIDTH // 4
* 3), random.randint(int(BOARD_HEIGHT*0.3), int(BOARD_HEIGHT*0.7))) for _ in
                    range(10)]
    self.tank = Tank(BOARD_WIDTH - 5, BOARD_HEIGHT // 2)
    self.tank_ai = TankAI(self.tank, None)
```

2. State Definition and Encoding

State Representation:

Individual Infantry State: A triplet (dx, dy, hp).

dx: The horizontal distance between the infantry and the tank (tank.x - infantry.x).

dy: The vertical distance (tank.y - infantry.y).

hp: The current health points of the infantry, discretized into [0, 2, 4, 6, 8].

```
def transform_state(unit, tank):
    dx = tank.x - unit.x
    dy = tank.y - unit.y
    return (dx,dy,unit.hp)
```

State Truncation:

If |dx| > 20, it is set to sign(dx)*20,this means that in the state space, the maximum and minimum values of dx and dy are 20 and -20.

The same treatment applies to dy, ensuring the state space is finite.

```
def get_q_value(self, state, action):
  if abs(state[0]) > 20 and abs(state[1]) > 20:
      state = (int(state[0] / abs(state[0]) * 20), int(state[1] / abs(state[1]) *
20), state[2])
  elif abs(state[0]) > 20:
      state = (int(state[0] / abs(state[0]) * 20), state[1], state[2])
  elif abs(state[1]) > 20:
      state = (state[0], int(state[1] / abs(state[1]) * 20), state[2])
  return self.q_table.get(state).get(action)
```

3. Action Space Definition

Legal Action Set

Generation Rule: All movement combinations satisfying the Manhattan distance |dx| + |dy| ≤ 4.

Total Number of Actions: 41 in total (generated by pre-calculation).

```
ACTION = []
for i in range(-INFANTRY_MOVEMENT,INFANTRY_MOVEMENT+1):
    for j in range(-(INFANTRY_MOVEMENT - abs(i)),INFANTRY_MOVEMENT - abs(i)+1):
        ACTION.append((i,j))
```

4. Q-Table Structure and Initialization

Q-Table Design

Key: The discretized state triplet (dx, dy, hp_level).

Value: A dictionary recording the Q-values for each action in that state (initialized to 0).

```
def create_q_table(self):
    import pickle
    if pathlib.Path('q_table.pkl').exists():
        with open('q_table.pkl', 'rb') as f:
            q_table = pickle.load(f)
        return q_table
    state = []
    for i in range(-20, 20 + 1):
        for j in range(-20, 20 + 1):
            state.append((i, j))
    result = {}
    for i in state:
        for j in [0, 2, 4, 6, 8]:
            result[(i[0], i[1], j)] = {}
            for k in ACTION:
                result[(i[0], i[1], j)][k] = 0
    return result
```

5. Hyperparameter Explanation

Parameter	Symbol	Value	Description
Learning Rate	α	0.1	Controls the speed of Q-value updates; higher values rely more on new experiences
Discount Factor	γ	0.99	Measures the importance of future rewards; a value close to 1 emphasizes long-term gains
Initial Exploration Rate	ε	1.0	Starts with complete random exploration, which decays with training
Minimum Exploration Rate	ε_min	0.01	Ensures a minimum exploration probability is always retained
Exploration Decay Rate		0.995	Updates after each training round using ε = max(ε*0.995, ε_min)

6.Implementation of the Q-learning Algorithm

Q-learning is a reinforcement learning algorithm that guides an agent to select optimal actions in different states by learning state-action values (Q-values). The core formula of Q-learning is as follows:

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

Formula Explanation:

Q(s,a): The expected return for selecting action a in state s.

r:The immediate reward, which is the feedback the agent receives from the environment after performing an action in the current state.

maxa' Q(s',a'): The maximum Q-value among all possible actions in the next state s', representing the optimal expected return in s'.

Formula Purpose: The term r+γmaxa' Q(s',a')−Q(s,a) calculates the temporal difference error (TD Error), which represents the difference between the actual return obtained and the expected return.

Objective: To gradually approximate the actual expected return with the Q-values, thereby guiding the agent to select optimal actions and maximize cumulative rewards.

```
    def update_q_table(self, state, action, reward, new_state):
        current_q = self.get_q_value(state, action)
        future_q = max([self.get_q_value(new_state, a) for a in ACTION])
        new_q = current_q + self.alpha * (reward + self.gamma * future_q -
current_q)
        self.q_table[state][action] = new_q
```

7.Multi-Round Training Process

Training Framework Construction

The training process adopts a classic "episode-step" double-layer loop structure. Each training round (episode) simulates a complete battle until a win or loss is determined. The system iteratively optimizes the agent's strategy through repeated iterations, enabling it to gradually master the optimal action patterns.

Single Episode Training Process

(1) Environment Initialization: At the beginning of each training round, the battlefield environment is reset.

(2) In-Episode Loop

a. State Perception: Each infantry unit independently perceives the environment and obtains its relative position to the tank.

b. Action Decision: The system reads the Q-values for all actions in the current state from the Q-table and selects the movement direction corresponding to the maximum value.If multiple

actions have the same highest Q-value, one is randomly selected to avoid path fixation.

c. Action Execution and Feedback: Collective Infantry Action and Tank AI Response:

d. Reward Calculation

e. Q-Table Update: The decision strategy for each infantry unit is updated using the Q-learning formula.

f. Termination Condition Check: After each step, the win or loss conditions are checked:

Player Victory: Tank HP ≤ 0.

Player Defeat: All infantry HP ≤ 0.

(3) Exploration Rate Decay: After each training round, the exploration rate is reduced .

High Exploration in Early Stages: For the first 100 rounds, ε ≈ 1.0, encouraging extensive exploration of different strategies.

Focus on Exploitation in Later Stages: After 1000 rounds, ε ≈ 0.01, primarily relying on the learned optimal strategies.

```
SHOW_TIME = 10
env = Environment()
agent = QLearningAgent(env)
num_episodes = 150
show_state(env.player_units, env.tank)

for episode in range(num_episodes):
    env.reset()
    if episode > num_episodes - SHOW_TIME:
        show_state(env.player_units, env.tank)
    total_reward = 0

    while True:
        try:
            current_states = [transform_state(unit, env.tank) for unit in
env.player_units]
            actions = []
            for idx, state in enumerate(current_states):
                if state is not None:
                    action = agent.choose_action(state)
                    actions.append(action)
            next_states, rewards, done = env.step(actions, episode > num_episodes -
SHOW_TIME)
            for idx, (state, action, reward, next_state) in enumerate(
                    zip(current_states, actions, rewards, next_states)):
                if abs(state[0]) > 20 and abs(state[1]) > 20:
                    state = (int(state[0] / abs(state[0]) * 20), int(state[1] /
abs(state[1]) * 20), state[2])
                elif abs(state[0]) > 20:
                    state = (int(state[0] / abs(state[0]) * 20), state[1], state[2])
                elif abs(state[1]) > 20:
                    state = (state[0], int(state[1] / abs(state[1]) * 20), state[2])
                agent.update_q_table(state, action, reward, next_state)
            total_reward += sum(rewards)
            if done:
                print(f"Episode {episode + 1}: Total Reward = {total_reward}! ")
                break
            if episode > num_episodes - SHOW_TIME:
                show_state(env.player_units, env.tank)
        except :
            pass
    agent.decay_epsilon()
```

Conclusion

This project constructs a turn-based strategy confrontation environment based on Q-learning, simulating a tactical game between 10 anti-aircraft infantry units and an Apocalypse tank. By defining the state space (relative position and health), action space (41 movement combinations), and reward-punishment mechanism (attack reward +6 / crushing penalty -35), the agent is trained to balance firepower output and survival evasion in multi-unit coordination. The ε-greedy strategy drives exploration and exploitation, enabling the infantry to eventually learn human-like tactics such as dispersed encirclement and safe concentrated fire.