GitHub Link: https://github.com/ANDREWZHU-06/Platform-Jumping-Game
YouTube Link: https://youtu.be/gkcJrJq_hqU

# Reinforcement Q-Learning Game Design

# Platform Jumping Game

**ZHU Jun - 5523710**

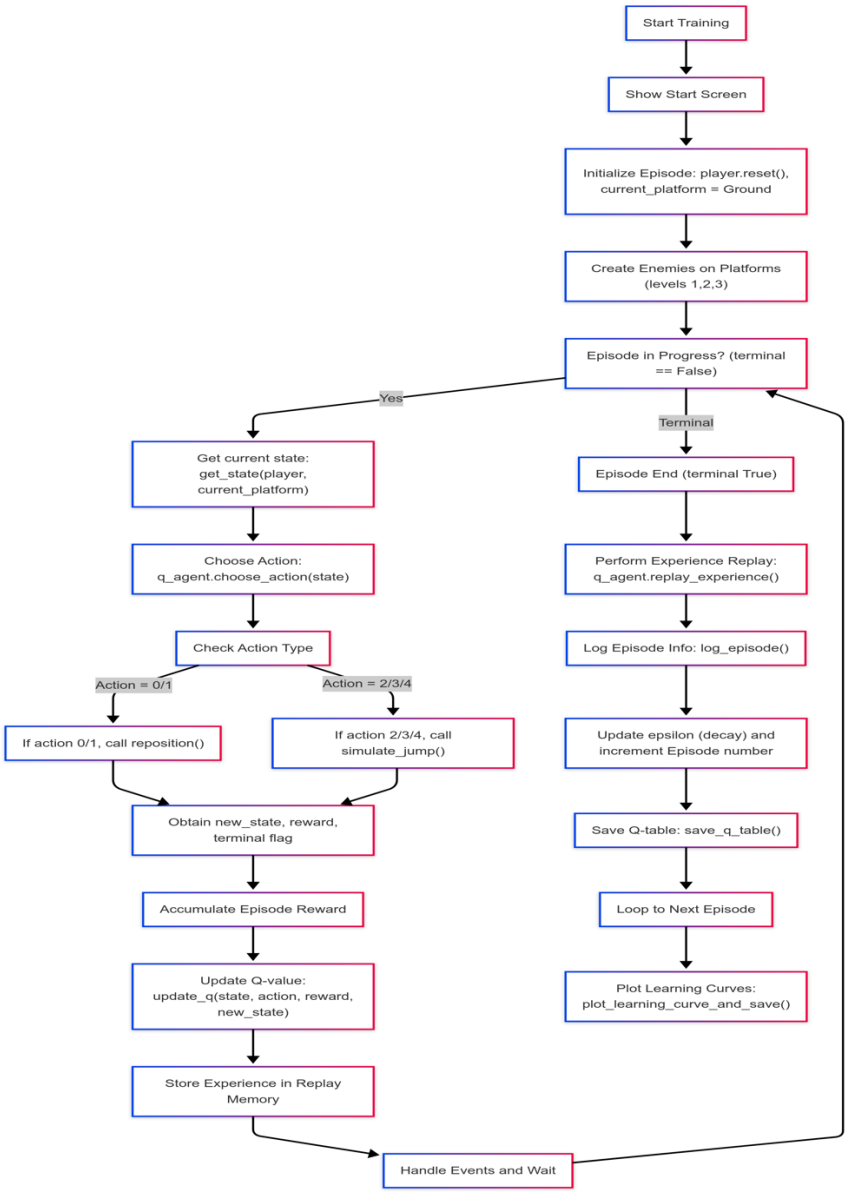
## Introduction

This is a **Q-Learning-based platformer game** built with Pygame, where the agent navigates four levels by jumping between platforms while avoiding moving enemies. The game state is based on the agent's position, and available actions include moving sideways, jumping, and diagonal jumps.

The game has two main parts:

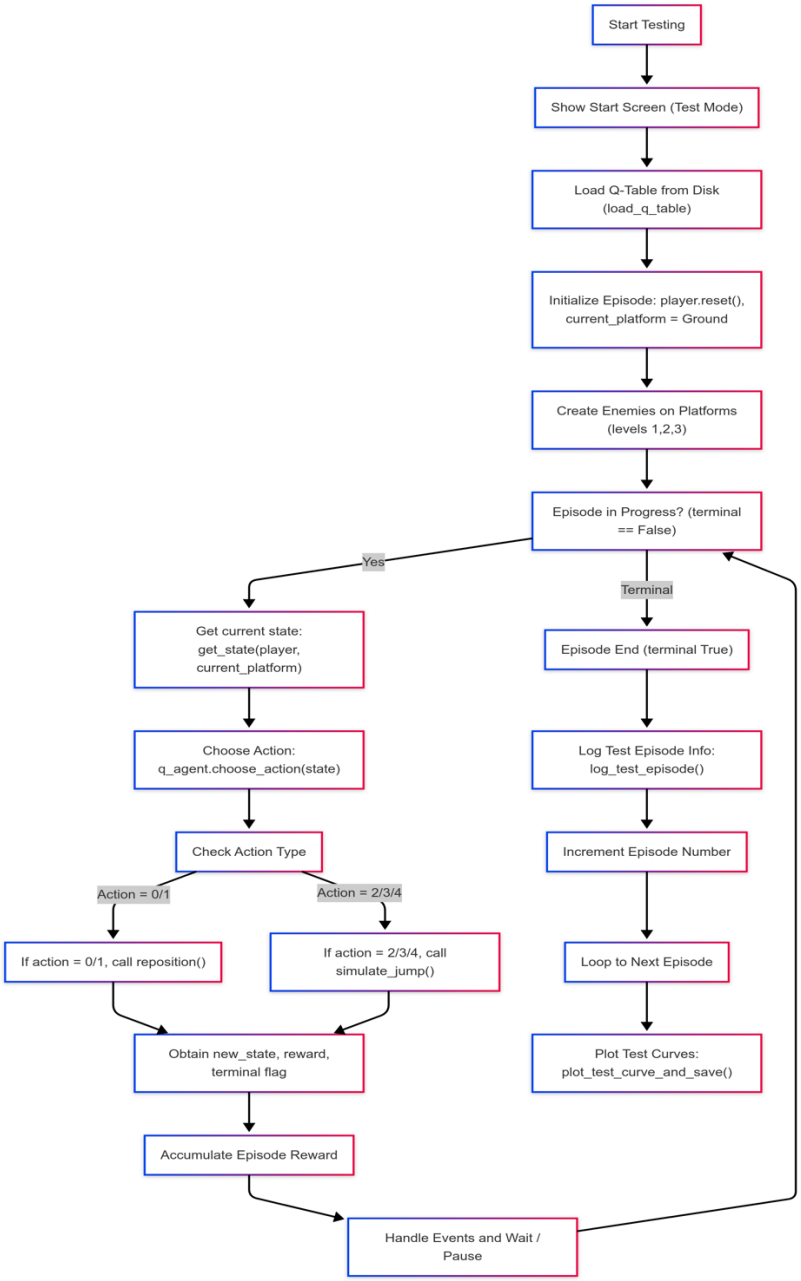
Training (game_train.py): The agent learns by updating its Q-table using an ε-greedy strategy, gradually favoring exploitation over exploration. Experience replay enhances learning stability. Rewards are given for successful jumps, while falls and collisions are penalized.

GitHub Link: https://github.com/ANDREWZHU-06/Platform-Jumping-Game
YouTube Link: https://youtu.be/gkcJrJq_hqU




Testing (game_test.py): The trained agent is evaluated with a low ε-value, testing its ability to perform stable jumps and avoid hazards in real-time gameplay.

GitHub Link: https://github.com/ANDREWZHU-06/Platform-Jumping-Game
YouTube Link: https://youtu.be/gkcJrJq_hqU




## 1. Game Environment Design

(1) Scene construction and platform layout

The game uses the Pygame framework to build a two-dimensional game world consisting of multiple platforms distributed at different heights (levels), with the lowest level usually being the ground and the upper level providing challenges and opportunities. The platforms are designed with discrete ‘level’ attributes to facilitate the discretization of subsequent states.

```python
# ------------------------
# Game initialization
# ------------------------
WINDOW_WIDTH = 400
WINDOW_HEIGHT = 600
FPS = 60

GRAVITY = 0.5
JUMP_VELOCITY = -12         # jump velocity
HORIZONTAL_VELOCITY = 5      # jump horizontal velocity
MOVE_STEP = 10               # move step

platforms = [
    {"level": 0, "rect": pygame.Rect(0, 550, 400, 50)},    # ground
    {"level": 1, "rect": pygame.Rect(20, 450, 150, 10)},     # Level 1-left platform
    {"level": 1, "rect": pygame.Rect(230, 450, 150, 10)},    # Level 1-right platform
    {"level": 2, "rect": pygame.Rect(30, 350, 100, 10)},     # Level 2-left platform
    {"level": 2, "rect": pygame.Rect(270, 350, 100, 10)},    # Level 2-right platform
    {"level": 3, "rect": pygame.Rect(50, 250, 80, 10)},      # Level 3-left platform
    {"level": 3, "rect": pygame.Rect(270, 250, 80, 10)},     # Level 3-right platform
    {"level": 4, "rect": pygame.Rect(150, 150, 100, 10)},    # Level 4-top platform
```

(2) Dynamic elements

The game features dynamic enemies (NPCs) that move side to side on platforms, adding uncertainty and challenge. The agent must avoid collisions to earn rewards and avoid penalties.

The code defines enemy objects, which are randomly generated on different platforms with non-fixed positions. Their movement direction is determined using ‘random.choice’, and they reverse direction upon reaching platform boundaries to ensure they stay on the platform.

```python
class EnemyObj:
    def __init__(self, platform):
        self.platform = platform
        pf_rect = platform['rect']
        self.width = 20
        self.height = 20
        self.x = random.randint(pf_rect.left, pf_rect.right - self.width)
        self.y = pf_rect.top - self.height
        self.speed = 2
        self.direction = random.choice([-1, 1])
    8 usages (8 dynamic)
    def update(self):
        self.x += self.speed * self.direction
        pf_rect = self.platform['rect']
        if self.x <= pf_rect.left or self.x + self.width >= pf_rect.right:
            self.direction *= -1
```

## 2. Agent State and Action Design

(1) State Representation

The game is designed to discretize the state of the agent. The state consists of two main components: the level of the platform on which the agent is located and the relative position of the agent on that platform. For this reason, usually divide the width of the platform into a number of intervals to form a discrete state space that can be easily processed by the Q-learning algorithm.

```
def get_state(agent, current_platform):
    """
    Return a discrete state representation consisting of:
      - The platform level.
      - The bin index (0-9) of the agent's relative x-position within the platform.
    """
    p_rect = current_platform['rect']
    relative_x = agent.x - p_rect.left
    effective_width = p_rect.width - agent.width
    bin_index = int((relative_x / effective_width) * 10) if effective_width else 0
    bin_index = max(0, min(9, bin_index))
    return (current_platform['level'], bin_index)
```

(2) Action Ensemble

The actions of the agent are discretized, and common actions include left-right movement (e.g., moving left or right in fixed steps), jumping in place, and diagonal jumping (combining horizontal and vertical jumps). These five actions provide enough operational options for the agent's decision-making to enable it to explore effective strategies in complex environments.

Base on the 'choose_action' function, the movements of the agent are discretized into 5 fixed movements:

| | |
|---|---|
| 0 | move left |
| 1 | move right |
| 2 | vertical jump |
| 3 | diagonal jump to the left |
| 4 | diagonal jump to the right |

```
def choose_action(self, state):
    """
    Choose an action based on the epsilon-greedy policy.
      - With probability epsilon, choose a random action (exploration).
      - Otherwise, choose the action with the highest Q-value (exploitation).
    """
    if random.random() < self.epsilon:
        return random.randint( a: 0,  b: 4)
    else:
        q_vals = self.get_q_values(state)
        max_q = max(q_vals)
        best_actions = [i for i, q in enumerate(q_vals) if q == max_q]
        return random.choice(best_actions)
```

For the reposition function, it mainly determine the agent to do the action move left and right, when action == 0, the agent executes the action move left. When action == 1, the agent executes the action move right. Meanwhile, to ensure that the agent does not extend beyond the left and right boundary of the platform.

```
def reposition(agent, action, current_platform, enemies):
    """
    Update the agent's position according to a horizontal move action.
      - If action==0: move left.
      - If action==1: move right.
    Also update enemy positions and check for collisions.
    Returns:
      new_state: the new discrete state.
      reward: base reward with collision penalty.
      terminal: game over flag (if agent's health is 0 or below).
    """
    pf_rect = current_platform['rect']
    if action == 0:
        agent.x = max(pf_rect.left, agent.x - MOVE_STEP)
    elif action == 1:
        agent.x = min(pf_rect.right - agent.width, agent.x + MOVE_STEP)
```

For the ‘simulate_jump’ function, it mainly determine the agent to do the action jump:

| | action |
|---|---|
| 2 | Jump vertically (vx = 0, vy = JUMP_VELOCITY) |
| 3 | Jump diagonally left (vx = -HORIZONTAL_VELOCITY) |
| 4 | jump diagonally to the right (vx = HORIZONTAL_VELOCITY) |

```python
def simulate_jump(agent, action, current_platform, enemies):
    """
    Simulate a jump according to the selected action:
      - action 2: vertical jump (no horizontal movement)
      - action 3: jump diagonally to the left
      - action 4: jump diagonally to the right
    The simulation runs frame-by-frame until the agent lands on a platform or a terminal condition occurs.
    Returns:
      landing_platform: the platform the agent landed on (if any)
      new_state: the discrete state after landing
      reward: accumulated reward (including bonus for landing on higher platforms)
      terminal: game over flag.
    """
    if action == 2:
        vx, vy = 0, JUMP_VELOCITY
    elif action == 3:
        vx, vy = -HORIZONTAL_VELOCITY, JUMP_VELOCITY
    elif action == 4:
        vx, vy = HORIZONTAL_VELOCITY, JUMP_VELOCITY
    else:
        vx, vy = 0, JUMP_VELOCITY
```

### 3. Reward Mechanism Design

(1) Positive Rewards

When an agent successfully jumps to a higher platform or completes certain key objectives (e.g. avoiding enemies, landing smoothly), the game gives positive rewards to encourage this behaviour.

(2) Negative Punishment

The game penalizes the agent for mistakes like colliding with enemies, failing a jump, or deviating from objectives. This helps the agent learn through trial and error to optimize strategies and achieve higher cumulative rewards

Three conditions that trigger the Reward mechanism:

(a) Collision with an enemy

At the start of each game, the agent's health is set to 100.Colliding with an enemy reduces health and reward by 20. If health reaches 0 or below, the game ends (terminal = True) and an additional 50 reward is deducted.

```python
    agent_rect = pygame.Rect(int(agent.x), int(agent.y), agent.width, agent.height)
    for enemy in enemies:
        if agent_rect.colliderect(enemy.get_rect()):
            if not collision_occurred:
                collision_occurred = True
                agent.health -= 20
                reward += -20

    # Check for terminal condition (agent's health)
    if agent.health <= 0:
        terminal = True
        reward += -50
        break
```

(b) Agent jumps out of the game activity range

It means that if an agent accidentally jumps out of the game area during the jumping process, the game will be judged as a failure (terminal = True), and the reward value will be reduced by 50.

```
        # If the agent falls below the window (misses landing), mark terminal.
        if agent.y > WINDOW_HEIGHT:
            terminal = True
            reward += -50
            break
```

(c) Jumping up to a higher level or down to a lower platform

The agent earns rewards based on its jumping behavior:

- Jumping to a higher platform increases the reward by 50 × level difference.
- Landing on a lower platform results in a 30-point penalty.
- Jumping on the same level reduces the reward by 1 point per jump.

This reward system encourages the agent to keep jumping to higher platforms to maximize its score.

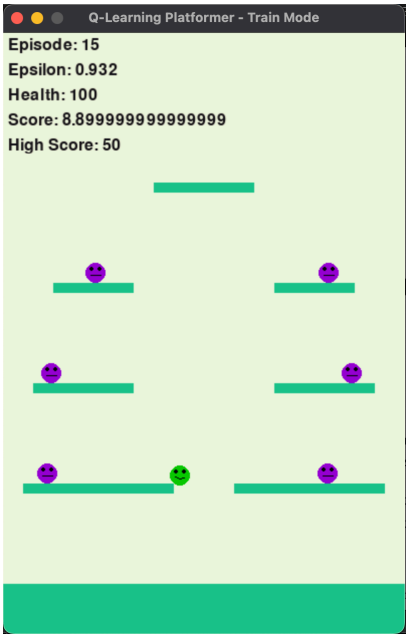
```
        # Reward based on the change in platform level:
        if landing_platform["level"] > current_platform["level"]:
            level_diff = landing_platform["level"] - current_platform["level"]
            reward += 50 * level_diff
        elif landing_platform["level"] == current_platform["level"]:
            reward += -1
        else:
            reward += -30

    return landing_platform, new_state, reward, terminal
```

## 4. User Interaction and Feedback Design

(1) User Interaction

The game adopts an intuitive graphical interface, where platforms, enemies, agent, and real-time scores and health values can be clearly seen on the screen. The player can press any key to start the game and pause it at any time (press P).

GitHub Link: https://github.com/ANDREWZHU-06/Platform-Jumping-Game
YouTube Link: https://youtu.be/gkcJrJq_hqU

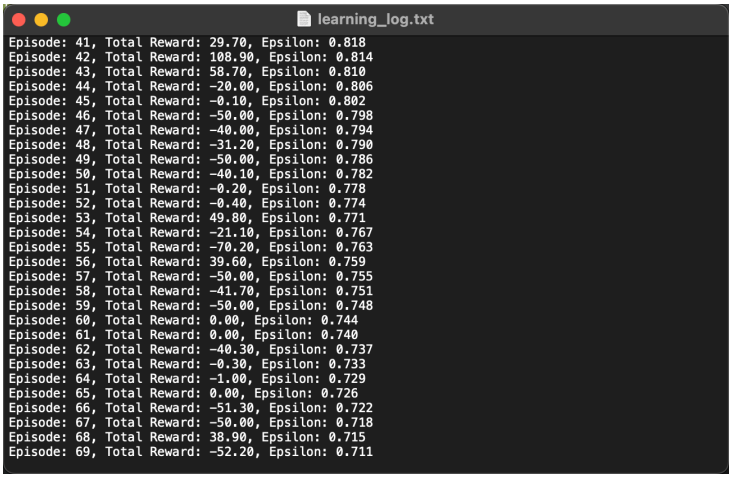



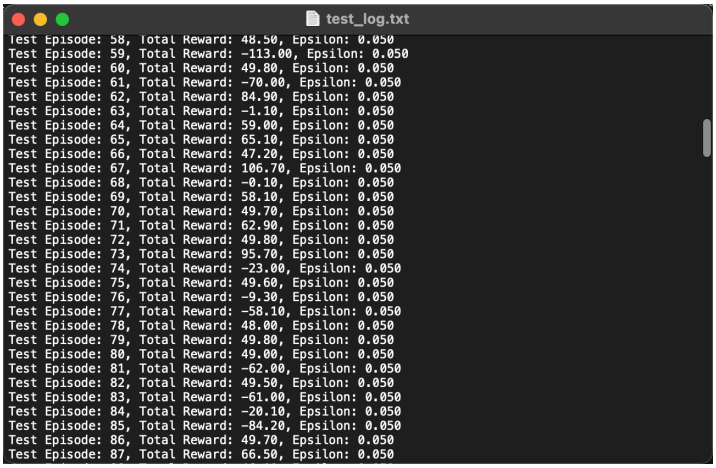

(2) Feedback

Through the log files (learning_log.txt and test_log.txt), the cumulative rewards and ε-value changes of each Episode can be tracked to analyze the algorithm training process and effect, which provides a basis for further tuning of the parameters and reward mechanism.

```
Learning progress : Episode: 58, Total Reward: -41.70, Epsilon: 0.751
Q-table has been saved to: /Users/zhujun/LU/Term-2/CDS524-Machine Learning for Business/Assignment1/train_log(alpha0.2-4)/q_table.pkl
Episode: 59, Total Reward: -50.00, Epsilon: 0.748
Learning progress : Episode: 59, Total Reward: -50.00, Epsilon: 0.748
Q-table has been saved to: /Users/zhujun/LU/Term-2/CDS524-Machine Learning for Business/Assignment1/train_log(alpha0.2-4)/q_table.pkl
Episode: 60, Total Reward: 0.00, Epsilon: 0.744
Learning progress : Episode: 60, Total Reward: 0.00, Epsilon: 0.744
Q-table has been saved to: /Users/zhujun/LU/Term-2/CDS524-Machine Learning for Business/Assignment1/train_log(alpha0.2-4)/q_table.pkl
Episode: 61, Total Reward: 0.00, Epsilon: 0.740
Learning progress : Episode: 61, Total Reward: 0.00, Epsilon: 0.740
Q-table has been saved to: /Users/zhujun/LU/Term-2/CDS524-Machine Learning for Business/Assignment1/train_log(alpha0.2-4)/q_table.pkl
```

GitHub Link: https://github.com/ANDREWZHU-06/Platform-Jumping-Game
YouTube Link: https://youtu.be/gkcJrJq_hqU




In addition, at the end of game training and testing, the corresponding q-learning total reward curves and epsilon decay curves are automatically plotted based on the content of the log files(Curves are shown later in this report ).

## Q-Learning Implementation

In this part, a Q-Learning Agent is defined for reinforcement learning tasks. The QAgent uses Q-learning algorithms for learning, combined with an ε-greedy strategy for decision making, and implements an Experience Replay mechanism.

This code mainly defines the basic parameters of q-learning, as well as creating a dictionary type q-table, and an experience playback pool that contributes to the stability of the training.

```
1 usage
class QAgent:
    def __init__(self, epsilon=1.0, alpha=0.1, gamma=0.9):
        self.epsilon = epsilon  # Exploration rate
        self.alpha = alpha  # Learning rate (set to 0.1/0.2/0.3)
        self.gamma = gamma  # Discount factor
        self.q_table = {}  # Q-table: key is state, value is a list of 5 Q-values corresponding to 5 actions.
        self.experience_replay = []  # Experience replay memory storage
        self.MAX_MEMORY = 5000  # Maximum number of experiences stored
```

Here is used to return the q-value of the action corresponding to the state in the q-table.

```
def get_q_values(self, state):
    """
    Get the Q-values for the current state.
    If the state is not present in the Q-table, initialize it with 5 actions having Q-value 0.0.
    """
    if state not in self.q_table:
        self.q_table[state] = [0.0 for _ in range(5)]
    return self.q_table[state]
```

This code is the core of the decision of the agent to carry out action selection, I set here when the randomly generated number(random.randint(0,4)) is smaller than the epsilon value, the agent will randomly select the corresponding action from 0 to 4 in order to be the subsequent new state in the update function inside the update of the q-table q-value (explore the strategy), if larger than the epsilon value, the agent will only select the maximum q-value from the current state of all the actions of the q-values to select the largest q-value as the new state (using the optimal strategy that already exists).

```
def choose_action(self, state):
    """
    Choose an action based on the epsilon-greedy policy.
      - With probability epsilon, choose a random action (exploration).
      - Otherwise, choose the action with the highest Q-value (exploitation).
    """
    if random.random() < self.epsilon:
        return random.randint( a: 0,  b: 4)
    else:
        q_vals = self.get_q_values(state)
        max_q = max(q_vals)
        best_actions = [i for i, q in enumerate(q_vals) if q == max_q]
        return random.choice(best_actions)
```

The following q-value update function is reproduced based on Q-Learning formula:

$$Q(S,A) = Q(S,A) + \alpha\,(R + \gamma\, Q(S',A') - Q(S,A))$$

or

**Q(S, A) = Q(S, A) + alpha * (reward + gamma * max[Q(next_state)] - Q(S, A))**

```
def update_q(self, state, action, reward, next_state):
    """
    Update the Q-value for the state-action pair using the formula:
        Q(s, a) = Q(s, a) + alpha * (reward + gamma * max[Q(next_state)] - Q(s, a))
    """
    old_q = self.get_q_values(state)[action]
    next_max = max(self.get_q_values(next_state)) if next_state is not None else 0.0
    new_q = old_q + self.alpha * (reward + self.gamma * next_max - old_q)
    self.q_table[state][action] = new_q
```

While storing state, action, reward, and next_state into the experience pool, to ensure that a useful but modest amount of historical experience is retained, ‘store_experience’ function releases the longest stored experience when the experience playback record is too large. Instead, ‘replay_experience’ function is used to randomly sample experiences from the experience pool to be used for q-value training.

```
1 usage
def store_experience(self, state, action, reward, next_state):
    """
    Store a tuple (state, action, reward, next_state) in the experience replay pool.
    Remove the oldest experience if the pool exceeds the maximum memory size.
    """
    if len(self.experience_replay) > self.MAX_MEMORY:
        self.experience_replay.pop(0)
    self.experience_replay.append((state, action, reward, next_state))

1 usage
def replay_experience(self, batch_size=64):
    """
    Randomly sample a batch of experiences from the replay memory and update Q-values for each.
    """
    if len(self.experience_replay) > batch_size:
        batch = random.sample(self.experience_replay, batch_size)
        for state, action, reward, next_state in batch:
            self.update_q(state, action, reward, next_state)
```

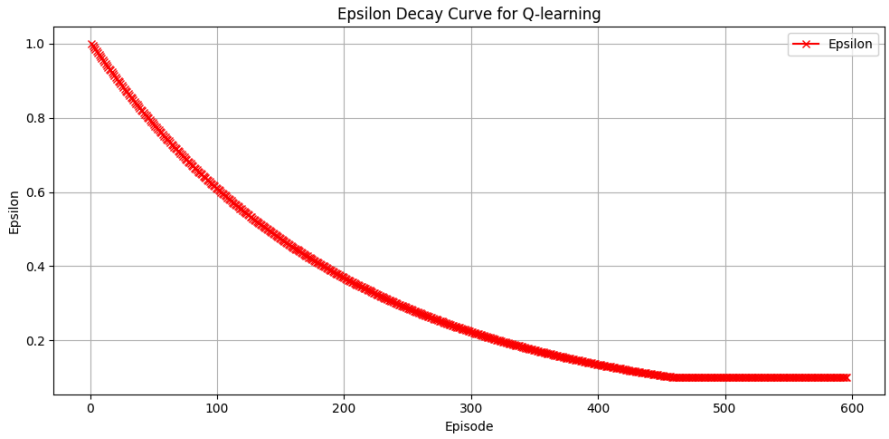
## Results Evaluation

To ensure a more contrasting overall performance between game training and testing, in this process I trained the game separately based on different alpha (0.1, 0.2, 0.3) values and tested the results of the training immediately after.

### 1. alpha=0.3, epsilon=0.1

**Training:**

**Epsilon decay curve:** the epsilon decay curve shows the transition of the ε-greedy strategy. Initially, epsilon = 1, indicating a high exploration phase that helps the agent quickly adapt to the environment. Around episode 460, epsilon converges to 0.1, shifting to a low-exploration phase where the agent primarily follows the learned optimal strategy.

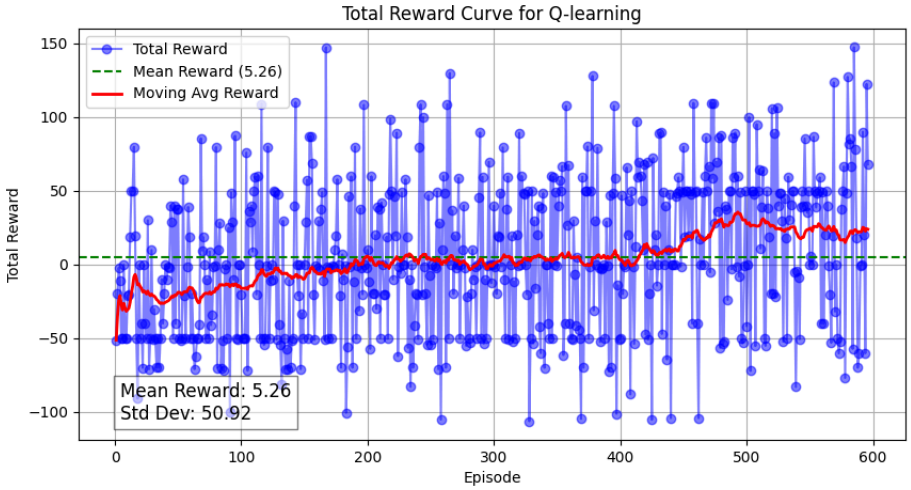



This graph shows the total reward curve, reflecting the agent's performance during training.

- Blue curve(Total Reward): Total reward per round.

- Green dashed line(Mean Reward): Average reward across all rounds.
- Red curve(Moving Average Reward): Moving average reward, showing the overall trend.

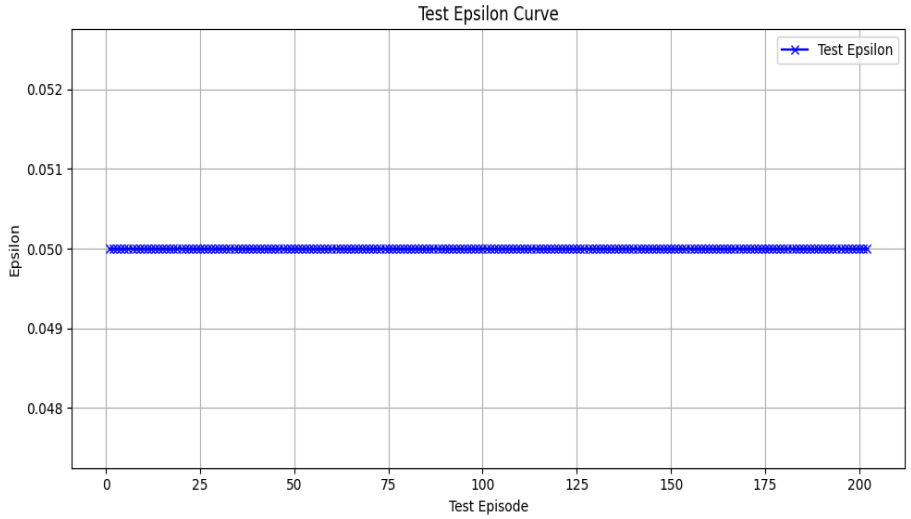
The total reward fluctuates significantly due to the agent's learning and exploration. However, the upward trend in the moving average reward indicates that the agent is continuously improving its strategy.



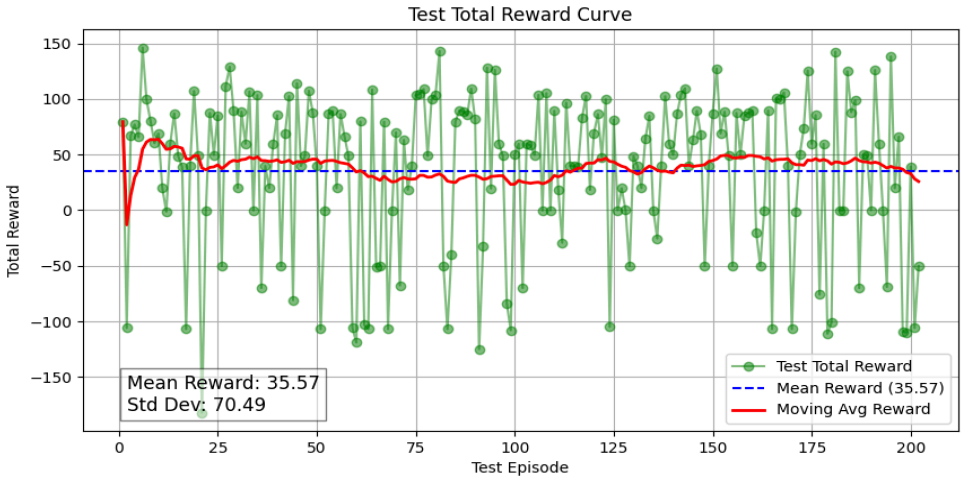

**Testing:**

The first graph shows the Epsilon curve during testing, indicating that the agent mainly relies on learned strategies.

The second graph shows the total reward curve, where the moving average reward fluctuates slightly around the average reward line, suggesting the agent's strategy is relatively stable. With an average reward of 35.57, the agent demonstrates good overall performance in the testing phase.



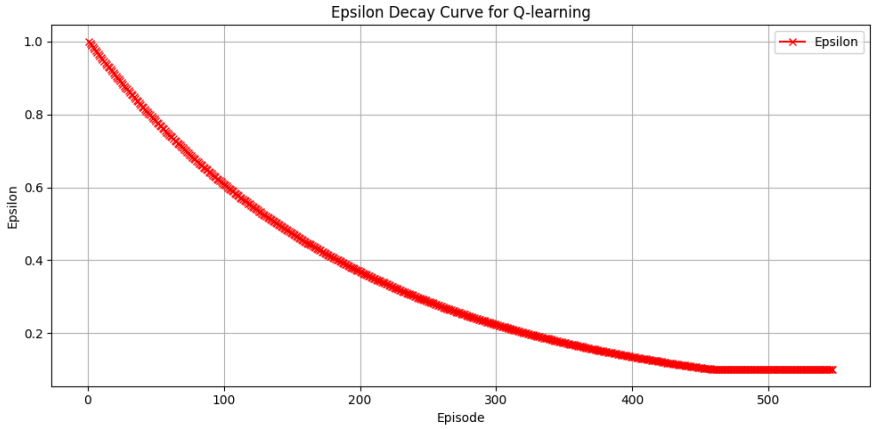
GitHub Link: https://github.com/ANDREWZHU-06/Platform-Jumping-Game
YouTube Link: https://youtu.be/gkcJrJq_hqU



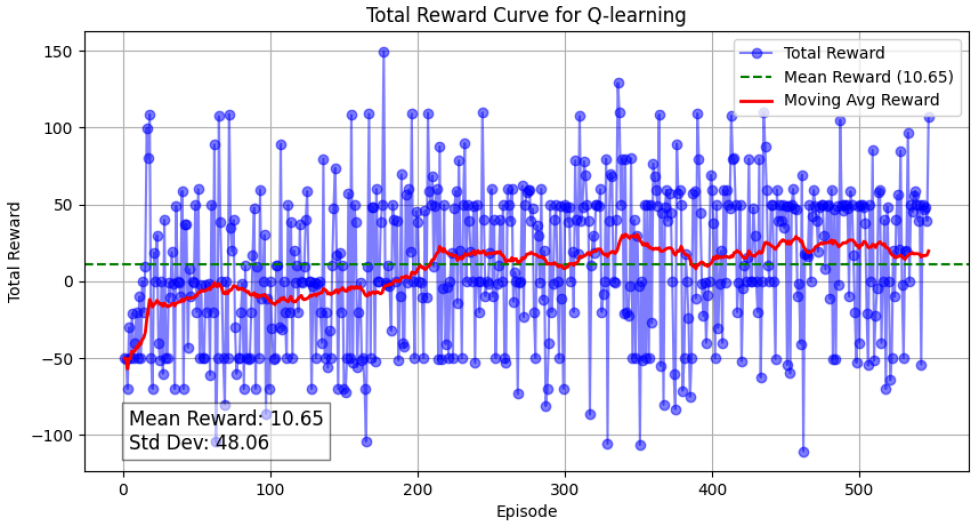

**2. alpha=0.2, epsilon=0.1**

**Training:**

Training is carried out when the value of alpha is equal to 0.2, and when the episodes reach roughly 460, the epsilon curve begins to converge to epsilon = 1 and begins to take the optimal policy in preference to the exploratory policy.




Similarly, as the process of learning and exploration by the intelligences must lead to fluctuations in the overall total reward curve, but if we pay attention to the moving average reward curve, we can see that the overall curve is in an upward state, implying that the intelligences also gradually improve the optimal strategy in the process of learning.




**Testing:**

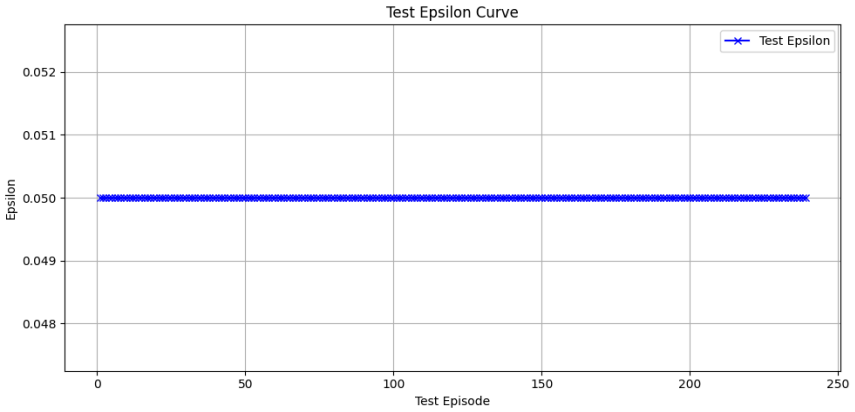
**Initial phase (first 50 episodes):**

Rewards fluctuate widely, and the agent may still be adapting to the test environment.

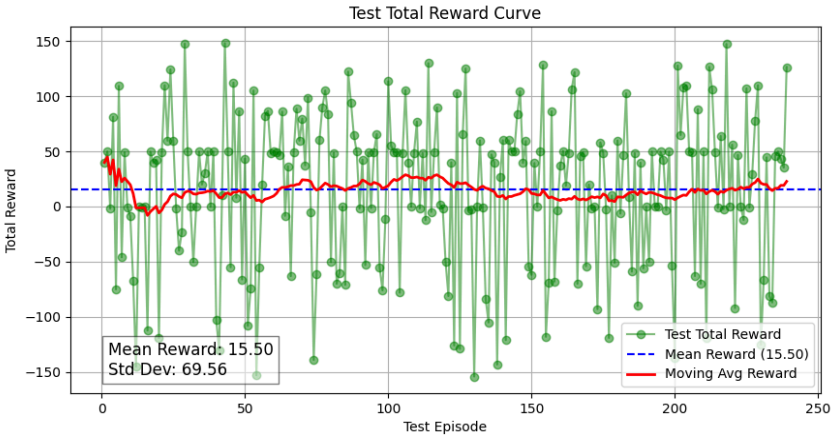
The red moving average reward curve gradually increases, indicating that the performance of the intelligent body has improved.

**Mid & Late Stage (50~250 episodes):**

The reward value still fluctuates, but the moving average reward curve tends to stabilize, indicating that the agent's strategy is more stable.



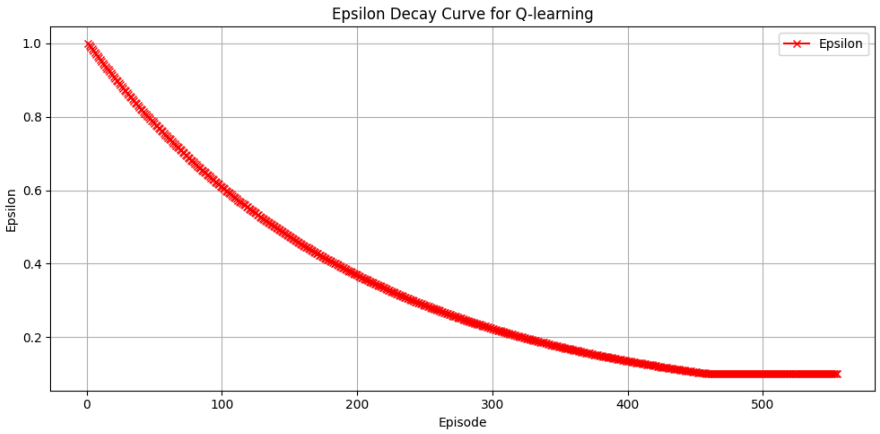
GitHub Link: https://github.com/ANDREWZHU-06/Platform-Jumping-Game
YouTube Link: https://youtu.be/gkcJrJq_hqU

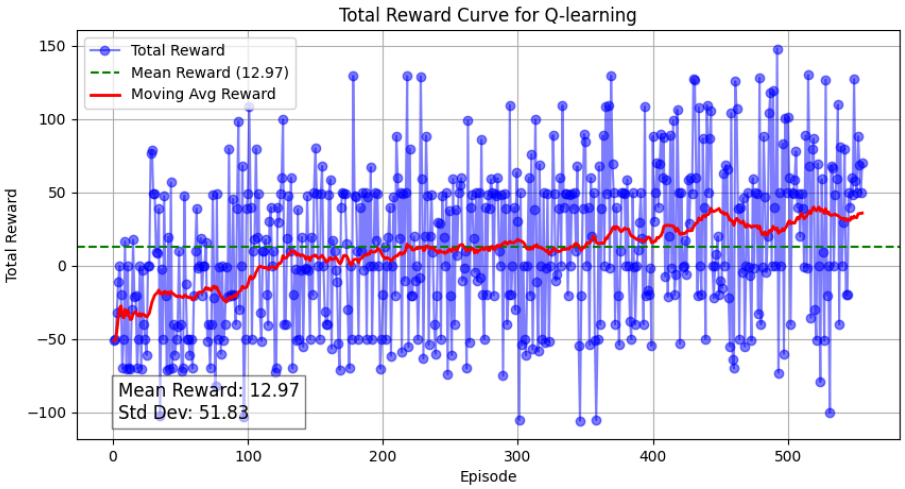



### 3. alpha=0.1, epsilon=0.1

**Training:**

The epsilon decay curve converges to episodes = 460 at alpha = 0.1.



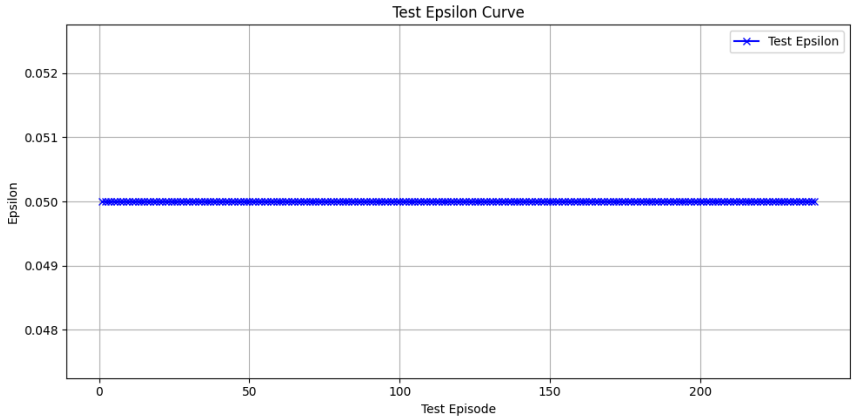
GitHub Link: https://github.com/ANDREWZHU-06/Platform-Jumping-Game
YouTube Link: https://youtu.be/gkcJrJq_hqU

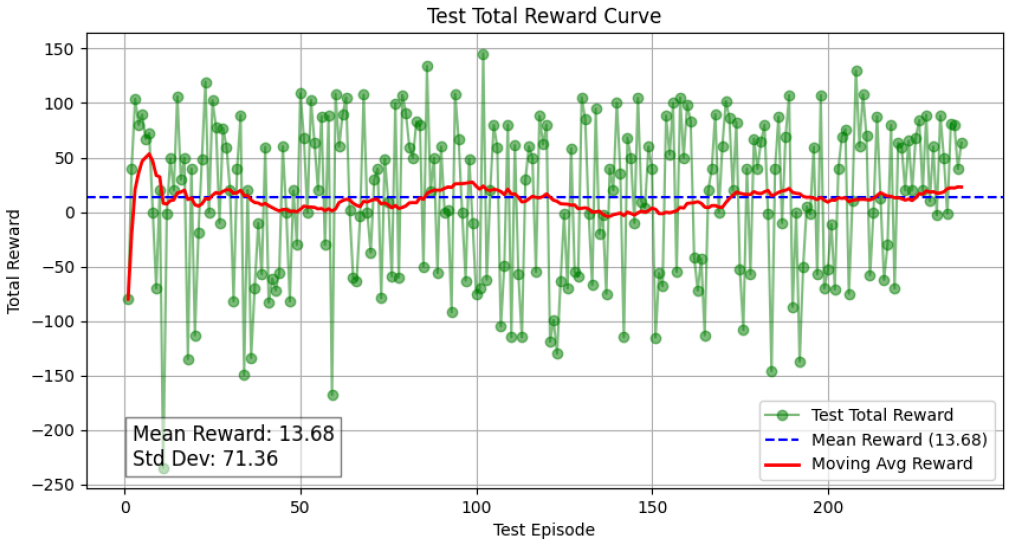
**Testing:**

The initial phase (first 10 episodes) was highly variable in rewards, probably due to the fact that the agent were still adapting to the test environment.

After that, the reward value fluctuates around 0 and the variance is large (Std Dev = 71.36), indicating that the agent's strategy is still unstable.

The red moving average curve flattens out, indicating that the agent's performance in the test environment has stabilized. The overall average reward of 13.68 indicates that the agent has learnt some strategies, but there is still room for improvement.



## Summary of Results

| | Training | | Testing | |
|---|---|---|---|---|
| | Mean Reward | Standard Deviation | Mean Reward | Standard Deviation |
| Alpha=0.1 | 12.97 | 51.83 | 13.68 | 71.36 |
| Alpha=0.2 | 10.65 | 40.06 | 15.50 | 69.56 |
| Alpha=0.3 | 5.26 | 50.92 | 35.57 | 70.49 |

Base on the table above:

- If want a stable training process, choose alpha=0.2 (lowest standard deviation for both training and testing).
- If more concerned about final test performance, choose alpha=0.3, which has the highest final test score, although the training phase is more volatile.
- If want to keep training rewards high, choose alpha=0.1, but test performance may be average.
- If the goal is final test performance, alpha=0.3 is recommended, but the number of training rounds can be increased to compensate for the volatility of the training phase.

You can refer to the project details below:

GitHub Link: https://github.com/ANDREWZHU-06/Platform-Jumping-Game
YouTube Link: https://www.youtube.com/watch?v=gkcJrJq_hqU

## Reference

[1] Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: An introduction* (2nd ed.). MIT Press.

[2] Watkins, C. J. C. H. (1989). *Learning from delayed rewards* (Doctoral dissertation, University of Cambridge).

[3] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., … Hassabis, D. (2013). *Playing Atari with deep reinforcement learning*.
arXiv preprint arXiv:1312.5602.

[4] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., … Hassabis, D. (2015). Human-level control through deep reinforcement learning.
*Nature, 518*(7540), 529–533.

[5] Pygame Community. (n.d.).
*Pygame documentation*. Retrieved from https://www.pygame.org/docs/

[6] van der Walt, S., Colbert, S. C., & Varoquaux, G. (2011). The NumPy array: A structure for efficient numerical computation.
*Computing in Science & Engineering, 13*(2), 22–30.

[7] Hunter, J. D. (2007). Matplotlib: A 2D graphics environment.
*Computing in Science & Engineering, 9*(3), 90–95.

[8] McKinney, W. (2010). Data structures for statistical computing in Python. In *Proceedings of the 9th Python in Science Conference* (pp. 56–61).