

Ứng dụng thuật toán DeepQ learning và Reinforcement learning vào game xếp hình

Nguyễn Quang Huy-B20DCCN318

Ngày 17 tháng 5 năm 2024

Tóm tắt nội dung

Dự án này em phát triển 1 AI có thể chơi game xếp hình vô thời hạn

1 Introduction

Ý tưởng thiết kế đó là khi một viên gạch hiện ra nó sẽ xác định vị trí nào được thả xuống là tốt nhất và chọn xoay như nào là tốt nhất .

2 Định nghĩa trò chơi

2.1 Tetris (trò chơi xếp hình) là gì ?

Trò chơi có bảy loại khối hình: I (thẳng đứng), J, L, O (vuông), S, T, Z. Ta thấy mỗi khối gạch được cấu tạo từ 4 hình vuông nhỏ xếp lại với nhau. Ta có thể coi các khối gạch đó như là những hình chữ nhật có kích thước khác nhau. Các hình khác được tạo ra khi xoay các khối cơ bản này các góc tương ứng 90 độ, 180 độ, 270 độ.

Một chuỗi ngẫu nhiên của Tetriminos rơi xuống sân chơi (một trục đứng hình chữ nhật, được gọi là "tốt" hay "ma trận"). Mục tiêu của trò chơi là di chuyển các khối gạch đang rơi từ từ xuống trong kích thước hình chữ nhật 20 hàng x 10 cột (trên màn hình). Chỗ nào có gạch rồi thì không di chuyển được tới vị trí đó. Người chơi xếp những khối hình sao cho khối hình lấp đầy 1 hàng ngang để ghi điểm và hàng ngang ấy sẽ biến mất.

Nếu để cho những khối hình cao quá màn hình, trò chơi sẽ kết thúc.

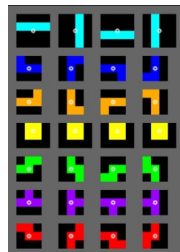
2.2 Nguyên tắc của trò chơi

Bảng Tetris rộng 10 ô và cao 22 ô, với 2 hàng trên cùng bị ẩn.

Tất cả các khối Tetrominoes (các mảnh của Tetris) sẽ bắt đầu ở giữa 2 hàng trên cùng.

Có 7 loại khối Tetrominoes: "I", "O", "J", "L", "S", "Z", "T".

Dưới đây hiển thị một bảng gồm tất cả các phần có thể có theo các quy tắc này.



Hình 1: Các mảnh và góc quay có thể có

2.3 Động thái tốt nhất

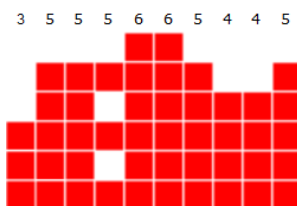
Mục tiêu của chúng ta là xoá càng nhiều dòng càng tốt, và do đó, là thực hiện càng nhiều động thái càng tốt.

Để đạt được mục tiêu này, AI của chúng ta sẽ quyết định động thái tốt nhất cho một mảnh Tetris cho trước bằng cách thử tất cả các động thái có thể (xoay và đặt vị trí). Nó tính điểm cho mỗi động thái có thể (cùng với mảnh nhìn trước), và chọn cái có điểm tốt nhất làm động thái tiếp theo của mình.

Điểm số cho mỗi động thái được tính bằng cách đánh giá lưới mà động thái đó sẽ dẫn đến. Việc đánh giá này dựa trên bốn tiêu chí: tổng chiều cao, số dòng hoàn thành, số lỗ, và bumpiness, mỗi tiêu chí AI sẽ cố gắng tối thiểu hoá hoặc tối đa hoá.

2.4 Aggregate Height (Tổng chiều cao)

Tiêu chí này cho chúng ta biết lưới cao bao nhiêu. Để tính chiều cao tổng, chúng ta lấy tổng chiều cao của từng cột (khoảng cách từ viên gạch cao nhất trong mỗi cột đến đáy của lưới).

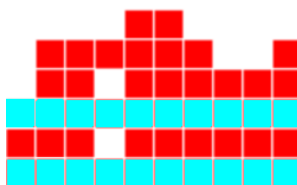


Hình 2: Tổng chiều cao = 48

Chúng ta sẽ muốn giảm giá trị này, bởi vì một chiều cao tổng thấp hơn có nghĩa là chúng ta có thể thả nhiều mảnh hơn vào lưới trước khi chạm đến đỉnh của lưới. Điều này giúp tránh được tình trạng lưới bị đầy sớm, cho phép chúng ta chơi được lâu hơn và có cơ hội xoá nhiều dòng hơn.

2.5 Complete Lines (số dòng hoàn thành)

Đây có lẽ là tiêu chí trực quan nhất trong số bốn tiêu chí. Nó đơn giản chỉ là số dòng hoàn chỉnh trong lưới.



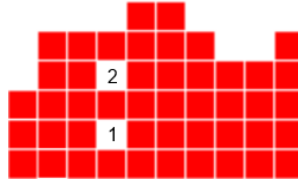
Hình 3: Số dòng hoàn thành = 2

Chúng ta sẽ muốn tối đa hóa số dòng hoàn chỉnh, bởi vì mục tiêu của AI là xoá dòng, và việc xoá dòng sẽ giúp chúng ta có thêm không gian cho nhiều mảnh hơn. Điều này không chỉ giúp tăng điểm số mà còn kéo dài thời gian chơi, cho phép AI tiếp tục tạo ra các kết hợp và chiến lược hiệu quả hơn.

2.6 Holes (Số lỗ)

Một lỗ được định nghĩa là một khoảng trống mà trên nó có ít nhất một viên gạch trong cùng cột.

Việc có lỗ trong lưới sẽ khó xoá hơn, vì chúng ta sẽ phải xoá tất cả các dòng phía trên nó trước khi có thể tiếp cận và lấp đầy lỗ đó. Do đó, chúng ta cần phải cố gắng giảm thiểu số lỗ này. Việc giảm thiểu lỗ không chỉ giúp duy trì khả năng xử lý và tạo hình dễ dàng hơn trên bảng chơi mà còn tăng cơ hội xoá dòng liên tục, qua đó cải thiện hiệu quả tổng thể của trò chơi.



Hình 4: Số lỗ = 2

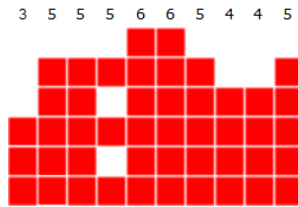
2.7 Bumpiness (Độ gồ ghề)

Hãy xét đến trường hợp có 1 cột thấp hơn hẳn các cột khác trong lưới. Dưới đây là ví dụ :



Hình 5:

Sự hiện diện của những cột này cho thấy những dòng có thể được xoá dễ dàng không được xoá. Nếu một cột bị che phủ, tất cả các hàng mà cột đó trải qua sẽ khó xoá. Để tổng quát hóa ý tưởng về một cột, chúng ta định nghĩa một tiêu chí sẽ gọi là “bumpiness”. dưới đây là ví dụ :



Hình 6: bumpiness=6=|3-5| + |5-5| + ... + |4-5|

Độ gồ ghề của một lưới cho chúng ta biết sự biến thiên về chiều cao của các cột của nó. Nó được tính bằng cách cộng tổng các hiệu số tuyệt đối giữa tất cả các cột liên kề.

2.8 Kết hợp các phương pháp Heuristic lại với nhau

Giờ chúng ta sẽ tính toán điểm của 1 bảng bằng cách lấy kết hợp tuyến tính của 4 phương pháp phỏng đoán trên , chúng ta sẽ được hàm tính điểm sau :

$$a \times (AggregateHeight) + b \times (CompleteLines) + c \times (Holes) + d \times (Bumpiness)$$

Hình 7: Công thức có được

3 Học tăng cường (reinforcement learning)

3.1 Học tăng cường (reinforcement learning) là gì ?

Học tăng cường (Reinforcement Learning - RL) là một lĩnh vực trong học máy (machine learning) liên quan đến cách các tác nhân (agents) hành động trong một môi trường để tối đa hóa một phần thưởng

tích lũy. Trong RL, tác nhân học cách đạt được mục tiêu thông qua thử nghiệm và sai lầm, sử dụng phản hồi từ môi trường dưới dạng phần thưởng hoặc hình phạt.

3.2 Các khái niệm cơ bản

- **Tác nhân (Agent):** Tác nhân là thực thể học cách đưa ra các quyết định. Nó nhận thông tin từ môi trường và thực hiện các hành động dựa trên chính sách học được.
- **Môi trường (Environment):** Môi trường là nơi tác nhân hoạt động. Môi trường cung cấp trạng thái hiện tại cho tác nhân và nhận các hành động từ tác nhân. Sau đó, môi trường trả về trạng thái mới và phần thưởng tương ứng.
- **Trạng thái (State):** Trạng thái là biểu diễn của tình huống hiện tại của môi trường. Trạng thái chứa các thông tin mà tác nhân sử dụng để quyết định hành động tiếp theo.
- **Hành động (Action):** Hành động là những gì tác nhân có thể làm. Tác nhân chọn một hành động dựa trên trạng thái hiện tại để tối đa hóa phần thưởng tích lũy trong dài hạn.
- **Phần thưởng (Reward):** Phần thưởng là phản hồi từ môi trường sau khi tác nhân thực hiện một hành động. Phần thưởng cho biết hành động của tác nhân có tốt hay không.
- **Chính sách (Policy):** Chính sách là chiến lược mà tác nhân sử dụng để chọn hành động dựa trên trạng thái hiện tại. Chính sách có thể là xác suất hoặc quyết định cố định.

3.3 Quá trình học tăng cường

- **Khởi tạo:** Tác nhân khởi tạo chính sách và giá trị hành động (nếu có).
- **Lặp lại cho đến khi kết thúc:**
 - Quan sát trạng thái hiện tại của môi trường.
 - Chọn hành động dựa trên chính sách hiện tại.
 - Thực hiện hành động và quan sát phần thưởng cùng với trạng thái mới từ môi trường.
 - Cập nhật chính sách và giá trị hành động dựa trên phần thưởng và trạng thái mới.
- **Lặp lại quá trình này cho đến khi chính sách hội tụ hoặc đạt được mục tiêu học tập.**

3.4 Các thuật toán phổ biến trong học tăng cường

- **Q-learning:** Q-learning là một thuật toán học không theo dõi mô hình. Nó học giá trị Q (Q-value) cho mỗi cặp trạng thái-hành động. Mục tiêu là tối ưu hóa chính sách chọn hành động dựa trên giá trị Q.
- **Deep Q-Learning (DQN):** DQN là một phiên bản nâng cao của Q-learning, sử dụng mạng nơ-ron sâu (deep neural network) để xấp xỉ giá trị Q. Điều này giúp xử lý các môi trường có không gian trạng thái lớn và phức tạp.
- **SARSA (State-Action-Reward-State-Action):** SARSA là một thuật toán RL khác, tương tự như Q-learning nhưng cập nhật giá trị Q dựa trên hành động thực tế mà tác nhân thực hiện.
- **Policy Gradient:** Thuật toán Policy Gradient tối ưu hóa trực tiếp chính sách bằng cách tối đa hóa hàm mục tiêu dựa trên phần thưởng tích lũy. Một ví dụ phổ biến là phương pháp REINFORCE.
- **Actor-Critic:** Actor-Critic là một phương pháp kết hợp giữa chính sách (Actor) và giá trị (Critic) để tối ưu hóa việc học. Actor cập nhật chính sách, trong khi Critic đánh giá hành động của Actor dựa trên giá trị hành động.

3.5 Ứng dụng của học tăng cường

- **Trò chơi:** Học tăng cường đã được sử dụng để phát triển các tác nhân có khả năng chơi các trò chơi từ đơn giản đến phức tạp như cờ vây (AlphaGo) và trò chơi điện tử (DQN trên Atari).
- **Robot:** Điều khiển và lập kế hoạch cho robot trong các nhiệm vụ phức tạp.
- **Tài chính:** Dự đoán và tối ưu hóa các chiến lược giao dịch.
- **Quản lý tài nguyên:** Tối ưu hóa việc quản lý tài nguyên trong các hệ thống lớn như mạng lưới điện.

4 Deep Q-Learning trong game Tetris

Deep Q-Learning (DQN) là một phiên bản mở rộng của thuật toán Q-learning, sử dụng mạng nơ-ron sâu (deep neural network) để xấp xỉ giá trị Q (Q-value). Điều này giúp xử lý các môi trường có không gian trạng thái lớn và phức tạp. DQN đã được sử dụng rộng rãi trong các trò chơi và các ứng dụng khác.

4.1 Khái niệm cơ bản trong Deep Q-Learning

- **Mạng nơ-ron sâu (Deep Neural Network):** Mạng nơ-ron sâu là mô hình học máy bao gồm nhiều lớp (layer) để xấp xỉ hàm giá trị Q.
- **Giá trị Q (Q-value):** Giá trị Q đại diện cho giá trị kỳ vọng của việc thực hiện một hành động tại một trạng thái cụ thể, theo chính sách tối ưu.
- **Trải nghiệm phát lại (Experience Replay):** Kỹ thuật lưu trữ các trải nghiệm (state, action, reward, next state) trong 1 bộ nhớ đệm và sử dụng chúng để huấn luyện mạng nơ-ron.
- **Mục tiêu giá trị Q (Target Q-value):** Giá trị Q được tính toán bằng cách sử dụng mạng nơ-ron mục tiêu (target network), giúp ổn định quá trình huấn luyện.

4.2 Cấu trúc của DQN

- **Mạng nơ-ron chính (Primary Network):** Mạng nơ-ron chính được huấn luyện để xấp xỉ hàm giá trị Q.
- **Mạng nơ-ron mục tiêu (Target Network):** Mạng nơ-ron mục tiêu được sao chép từ mạng nơ-ron chính sau mỗi số bước cố định và được sử dụng để tính toán mục tiêu giá trị Q.
- **Bộ nhớ đệm trải nghiệm (Experience Replay Buffer):** Bộ nhớ đệm lưu trữ các trải nghiệm để huấn luyện mạng nơ-ron.

4.3 Cấu trúc của DQN

- **Mạng nơ-ron chính (Primary Network):** Mạng nơ-ron chính được huấn luyện để xấp xỉ hàm giá trị Q.
- **Mạng nơ-ron mục tiêu (Target Network):** Mạng nơ-ron mục tiêu được sao chép từ mạng nơ-ron chính sau mỗi số bước cố định và được sử dụng để tính toán mục tiêu giá trị Q.
- **Bộ nhớ đệm trải nghiệm (Experience Replay Buffer):** Bộ nhớ đệm lưu trữ các trải nghiệm để huấn luyện mạng nơ-ron.
- **Mạng nơ-ron mục tiêu (Target Network):** Mạng nơ-ron mục tiêu được sao chép từ mạng nơ-ron chính sau mỗi số bước cố định và được sử dụng để tính toán mục tiêu giá trị Q.
- **Bộ nhớ đệm trải nghiệm (Experience Replay Buffer):** Bộ nhớ đệm lưu trữ các trải nghiệm để huấn luyện mạng nơ-ron.

4.4 Quá trình huấn luyện DQN

– Khởi tạo:

- * Khởi tạo mạng nơ-ron chính và mạng nơ-ron mục tiêu.
- * Khởi tạo bộ nhớ đệm trải nghiệm.

– Lặp lại cho mỗi tập (Episode):

- * Đặt lại môi trường và quan sát trạng thái ban đầu.
- * Lặp lại cho mỗi bước trong tập:
 - Chọn hành động dựa trên chính sách epsilon-greedy.
 - Thực hiện hành động và quan sát phần thưởng và trạng thái tiếp theo.
 - Lưu trải nghiệm vào bộ nhớ đệm.
 - Lấy mẫu ngẫu nhiên các trải nghiệm từ bộ nhớ đệm và huấn luyện mạng nơ-ron.
 - Cập nhật mạng nơ-ron mục tiêu định kỳ.

Chắc chắn rồi! Dưới đây là phần báo cáo về việc sử dụng Deep Q-Learning (DQN) trong game Tetris, bao gồm phần giải thích chi tiết về mã lớp DeepQNetwork.

4. Deep Q-Learning trong game Tetris 4.1 Giới thiệu về Deep Q-Learning Deep Q-Learning (DQN) là một phiên bản mở rộng của thuật toán Q-learning, sử dụng mạng nơ-ron sâu (deep neural network) để xấp xỉ giá trị Q (Q-value). Điều này giúp xử lý các môi trường có không gian trạng thái lớn và phức tạp. DQN đã được sử dụng rộng rãi trong các trò chơi và các ứng dụng khác.

4.2 Khái niệm cơ bản trong Deep Q-Learning

- * **Mạng nơ-ron sâu (Deep Neural Network):** Mạng nơ-ron sâu là mô hình học máy bao gồm nhiều lớp (layer) để xấp xỉ hàm giá trị Q.
less Sao chép mã
- * **Giá trị Q (Q-value):** Giá trị Q đại diện cho giá trị kỳ vọng của việc thực hiện một hành động tại một trạng thái cụ thể, theo chính sách tối ưu.
- * **Trải nghiệm phát lại (Experience Replay):** Kỹ thuật lưu trữ các trải nghiệm (state, action, reward, next_state) trong một bộ nhớ đệm để huấn luyện mạng nơ-ron.
- * **Mục tiêu giá trị Q (Target Q-value):** Giá trị Q được tính toán bằng cách sử dụng mạng nơ-ron mục tiêu (target network), giúp ổn định quá trình huấn luyện.

4.3 Cấu trúc của DQN

- * **Mạng nơ-ron chính (Primary Network):** Mạng nơ-ron chính được huấn luyện để xấp xỉ hàm giá trị Q.
less Sao chép mã
- * **Mạng nơ-ron mục tiêu (Target Network):** Mạng nơ-ron mục tiêu được sao chép từ mạng nơ-ron chính sau mỗi số bước cố định và được sử dụng để tính toán mục tiêu giá trị Q.
- * **Bộ nhớ đệm trải nghiệm (Experience Replay Buffer):** Bộ nhớ đệm lưu trữ các trải nghiệm để huấn luyện mạng nơ-ron.

4.4 Quá trình huấn luyện DQN

* Khởi tạo:

- Khởi tạo mạng nơ-ron chính và mạng nơ-ron mục tiêu.
- Khởi tạo bộ nhớ đệm trải nghiệm.

* Lặp lại cho mỗi tập (Episode):

- Đặt lại môi trường và quan sát trạng thái ban đầu.
- Lặp lại cho mỗi bước trong tập:
 - Chọn hành động dựa trên chính sách epsilon-greedy.
 - Thực hiện hành động và quan sát phần thưởng và trạng thái tiếp theo.
 - Lưu trải nghiệm vào bộ nhớ đệm.
 - Lấy mẫu ngẫu nhiên các trải nghiệm từ bộ nhớ đệm và huấn luyện mạng nơ-ron.
 - Cập nhật mạng nơ-ron mục tiêu định kỳ.

4.5 Áp dụng DQN trong game Tetris

- * **Mô tả môi trường Tetris:** Môi trường Tetris bao gồm một lưới vuông và các khối hình học khác nhau (tetromino). Mục tiêu là xoay và di chuyển các khối để tạo thành các hàng đầy đủ, sau đó xóa chúng và ghi điểm.
- * **Thiết lập mạng nơ-ron:** Mạng nơ-ron bao gồm các lớp đầu vào (input layer), lớp ẩn (hidden layers) và lớp đầu ra (output layer). Đầu vào là trạng thái của lưới, đầu ra là giá trị Q cho các hành động có thể.
- * **Chính sách epsilon-greedy:** Chính sách epsilon-greedy được sử dụng để chọn hành động. Với xác suất epsilon, tác nhân sẽ chọn hành động ngẫu nhiên, và với xác suất $1 - \text{epsilon}$, tác nhân sẽ chọn hành động có giá trị Q cao nhất.
- * **Huấn luyện và đánh giá:** Quá trình huấn luyện bao gồm việc tác nhân chơi nhiều tập Tetris, cập nhật giá trị Q và cải thiện chính sách theo thời gian. Sau khi huấn luyện, tác nhân được đánh giá bằng cách chơi một số tập mà không cập nhật giá trị Q .

5 Xây dựng mã và Deep Q-Learning để ứng dụng vào game Tetris

5.1 Giới thiệu

Trong phần này, chúng ta sẽ đi qua quá trình xây dựng mã nguồn và sử dụng Deep Q-Learning (DQN) để phát triển một tác nhân có thể chơi game Tetris một cách tự động. Chúng ta sẽ xây dựng các thành phần cần thiết, bao gồm môi trường game, mô hình mạng nơ-ron và quá trình huấn luyện.

5.2 Xây dựng môi trường game Tetris

5.2.1 Hàm init

```
def __init__(self, height=20, width=10, block_size=20):
    self.height = height
    self.width = width
    self.block_size = block_size
    self.extra_board = np.ones((self.height * self.block_size, self.width * int(self.block_size // 10),
                                dtype=np.uint8) * np.array([204, 204, 255], dtype=np.uint8)
    self.text_color = (200, 20, 220)
    self.reset()
```

Giải thích:

- * Hàm khởi tạo `__init__` được gọi khi tạo một đối tượng của lớp Tetris.
- * Các biến `height`, `width`, và `block_size` xác định kích thước của lưới Tetris.
- * `extra_board` và `text_color` được sử dụng để hiển thị thêm thông tin trên màn hình.
- * Gọi hàm `reset()` để khởi tạo trạng thái ban đầu của game.

5.2.2 Hàm reset

```
def reset(self):
    self.board = [[0] * self.width for _ in range(self.height)]
    self.score = 0
    self.tetrominoes = 0
    self.cleared_lines = 0
    self.bag = list(range(len(self.pieces)))
    random.shuffle(self.bag)
    self.ind = self.bag.pop()
    self.piece = [row[:] for row in self.pieces[self.ind]]
    self.current_pos = {"x": self.width // 2 - len(self.piece[0]) // 2, "y": 0}
    self.gameover = False
    return self.get_state_properties(self.board)
```

Giải thích:

- * Hàm `reset` đặt lại trạng thái game về trạng thái ban đầu.
- * `self.board` là lưới game, `self.score` là điểm số hiện tại, `self.tetrominoes` là số lượng khối đã rơi, `self.cleared_lines` là số dòng đã xóa.
- * `self.bag` chứa các chỉ số của các khối hình học (tetrominoes) và được trộn ngẫu nhiên.
- * `self.piece` là khối hình học hiện tại.
- * `self.current_pos` xác định vị trí hiện tại của khối hình học trên lưới.
- * Trả về thuộc tính trạng thái của lưới thông qua hàm `get_state_properties`.

5.2.3 Hàm rotate

```
def rotate(self, piece):
    num_rows_orig = num_cols_new = len(piece)
    num_rows_new = len(piece[0])
    rotated_array = []

    for i in range(num_rows_new):
        new_row = [0] * num_cols_new
        for j in range(num_cols_new):
            new_row[j] = piece[(num_rows_orig - 1) - j][i]
        rotated_array.append(new_row)
    return rotated_array
```

Giải thích:

- * Hàm `rotate` xoay một khối hình học (`piece`) theo chiều kim đồng hồ.
- * `num_rows_orig` và `num_cols_new` xác định số hàng ban đầu và số cột mới của khối.
- * `num_rows_new` xác định số hàng mới của khối sau khi xoay.
- * `rotated_array` là mảng mới chứa khối sau khi xoay.
- * Vòng lặp đầu tiên duyệt qua các hàng mới (`num_rows_new`).
- * Vòng lặp thứ hai duyệt qua các cột mới (`num_cols_new`) và gán giá trị từ khối ban đầu vào khối mới theo quy tắc xoay.
- * Trả về mảng `rotated_array` chứa khối sau khi xoay.

5.2.4 Hàm get-state-properties

```
def get_state_properties(self, board):
    lines_cleared, board = self.check_cleared_rows(board)
    holes = self.get_holes(board)
    bumpiness, height = self.get_bumpiness_and_height(board)
    return torch.FloatTensor([lines_cleared, holes, bumpiness, height])
```

Giải thích:

- * Hàm `get_state_properties` tính toán các thuộc tính trạng thái của lưới.
- * `lines_cleared` là số dòng đã xóa.
- * `holes` là số lượng lỗ hổng trên lưới.
- * `bumpiness` là độ gồ ghề của lưới và `height` là tổng chiều cao của các cột.
- * Trả về một tensor Float chứa các thuộc tính này.

5.2.5 hàm get-holes

```
def get_holes(self, board):
    num_holes = 0
    for col in zip(*board):
        row = 0
        while row < self.height and col[row] == 0:
            row += 1
        num_holes += len([x for x in col[row + 1:] if x == 0])
    return num_holes
```

Giải thích:

- * Hàm `get_holes` tính toán số lượng lỗ hổng trên lưới.
- * Số lượng lỗ hổng là số ô trống nằm dưới cùng của các ô không trống trong cùng một cột.
- * Trả về tổng số lượng lỗ hổng trên lưới.

5.2.6 Hàm get-bumpiness-and-height

```
def get_bumpiness_and_height(self, board):
    board = np.array(board)
    mask = board != 0
    invert_heights = np.where(mask.any(axis=0), np.argmax(mask, axis=0), self.height)
    heights = self.height - invert_heights
    total_height = np.sum(heights)
    currs = heights[:-1]
    nexts = heights[1:]
    diffs = np.abs(currs - nexts)
    total_bumpiness = np.sum(diffs)
    return total_bumpiness, total_height
```

Giải thích:

- * Hàm `get_bumpiness_and_height` tính toán độ gồ ghề và tổng chiều cao của các cột trên lưới.
- * `total_height` là tổng chiều cao của tất cả các cột.
- * `total_bumpiness` là tổng độ chênh lệch chiều cao giữa các cột liền kề.
- * Trả về độ gồ ghề và tổng chiều cao của lưới.

5.2.7 get-next-states

```
def get_next_states(self):
    states = {}
    piece_id = self.ind
    curr_piece = [row[:] for row in self.piece]
    if piece_id == 0: # 0 piece
        num_rotations = 1
    elif piece_id == 2 or piece_id == 3 or piece_id == 4:
        num_rotations = 2
    else:
        num_rotations = 4

    for i in range(num_rotations):
        valid_xs = self.width - len(curr_piece[0])
        for x in range(valid_xs + 1):
            piece = [row[:] for row in curr_piece]
            pos = {"x": x, "y": 0}
            while not self.check_collision(piece, pos):
                pos["y"] += 1
            self.truncate(piece, pos)
            board = self.store(piece, pos)
            states[(x, i)] = self.get_state_properties(board)
            curr_piece = self.rotate(curr_piece)
    return states
```

Giải thích:

- * Hàm `get_next_states` tạo ra các trạng thái tiếp theo có thể xảy ra từ trạng thái hiện tại của lưới và khối hiện tại.
- * Tính toán các trạng thái cho mỗi vị trí và mỗi lần xoay hợp lệ của khối hiện tại.
- * Trả về một từ điển các trạng thái tiếp theo với khóa là (x, số lần xoay) và giá trị là các thuộc tính trạng thái của lưới.

5.2.8 get-current-board-state

```
def get_current_board_state(self):
    board = [x[:] for x in self.board]
    for y in range(len(self.piece)):
        for x in range(len(self.piece[y])):
            board[y + self.current_pos["y"]][x + self.current_pos["x"]] = self.piece[y][x]
    return board
```

Giải thích:

- * Hàm `get_current_board_state` trả về trạng thái hiện tại của lưới game bao gồm cả khối hiện tại.
- * Trạng thái lưới bao gồm cả các ô trống và các ô được lấp đầy bởi khối hiện tại.
- * Trả về trạng thái hiện tại của lưới.

5.2.9 new-piece

```
def new_piece(self):
    if not len(self.bag):
        self.bag = list(range(len(self.pieces)))
        random.shuffle(self.bag)
    self.ind = self.bag.pop()
    self.piece = [row[:] for row in self.pieces[self.ind]]
    self.current_pos = {"x": self.width // 2 - len(self.piece[0]) // 2, "y": 0}
    if self.check_collision(self.piece, self.current_pos):
        self.gameover = True
```

Giải thích:

- * Hàm `new_piece` tạo ra một khối hình học mới từ túi chứa các khối và đặt vị trí ban đầu của khối trên lưới.
- * Nếu túi chứa các khối trống, hàm sẽ làm mới và trộn lại các khối.
- * Nếu khối mới tạo ra va chạm với khối khác hoặc rìa của lưới, trạng thái game sẽ chuyển sang game over.

5.2.10 Hàm check-collision

```
def check_collision(self, piece, pos):
    future_y = pos["y"] + 1
    for y in range(len(piece)):
        for x in range(len(piece[y])):
            if future_y + y > self.height - 1 or
               self.board[future_y + y][pos["x"] + x] and piece[y][x]:
                return True
    return False
```

Giải thích:

- * Hàm `check_collision` kiểm tra xem khối hiện tại có va chạm với khối khác hoặc rìa của lưới khi di chuyển xuống dưới hay không.
- * Nếu có va chạm, hàm trả về `True`; nếu không, hàm trả về `False`.

5.2.11 Hàm truncate

```
def truncate(self, piece, pos):
    gameover = False
    last_collision_row = -1
    for y in range(len(piece)):
        for x in range(len(piece[y])):
            if self.board[pos["y"] + y][pos["x"] + x] and piece[y][x]:
                if y > last_collision_row:
                    last_collision_row = y

    if pos["y"] - (len(piece) - last_collision_row) < 0 and last_collision_row > -1:
        while last_collision_row >= 0 and len(piece) > 1:
            gameover = True
            last_collision_row = -1
            del piece[0]
            for y in range(len(piece)):
                for x in range(len(piece[y])):
                    if self.board[pos["y"] + y][pos["x"] + x] and piece[y][x]
                    and y > last_collision_row:
                        last_collision_row = y

    return gameover
```

Giải thích:

- * Hàm `truncate` xử lý va chạm giữa khối và các ô đã được lấp đầy trên lưới, và cắt bỏ phần của khối vượt ra ngoài lưới.
- * Nếu khối va chạm với các ô khác hoặc rìa của lưới, trạng thái game có thể chuyển sang game over.
- * Trả về giá trị `True` nếu game over, ngược lại trả về `False`.

5.2.12 Hàm store

```
def store(self, piece, pos):
    board = [x[:] for x in self.board]
    for y in range(len(piece)):
        for x in range(len(piece[y])):
            if piece[y][x] and not board[pos["y"]][x + pos["x"]]:
                board[pos["y"]][x + pos["x"]] = piece[y][x]
    return board
```

Giải thích:

- * Hàm `store` lưu trữ khối hiện tại vào lưới khi nó đã dừng lại.
- * Các ô của khối sẽ được lấp đầy trên lưới tại vị trí hiện tại của khối.
- * Trả về trạng thái lưới sau khi lưu trữ khối.

5.2.13 Hàm check-cleared-rows

```
def check_cleared_rows(self, board):
    to_delete = []
    for i, row in enumerate(board[::-1]):
        if 0 not in row:
            to_delete.append(len(board) - 1 - i)
    if len(to_delete) > 0:
        board = self.remove_row(board, to_delete)
    return len(to_delete), board
```

Giải thích:

- * Hàm `check_cleared_rows` kiểm tra và xóa các dòng đầy đủ trên lưới.
- * Các dòng đầy đủ sẽ được thêm vào danh sách `to_delete`.
- * Nếu có dòng đầy đủ, hàm sẽ gọi `remove_row` để xóa các dòng này.
- * Trả về số dòng đã xóa và trạng thái lưới sau khi xóa.

5.2.14 Hàm remove-row

```
def remove_row(self, board, indices):
    for i in indices[::-1]:
        del board[i]
        board = [[0 for _ in range(self.width)]] + board
    return board
```

Giải thích:

- * Hàm `remove_row` xóa các dòng đầy đủ từ lưới và thêm các dòng trống ở trên cùng.
- * Các dòng đầy đủ được xác định bởi các chỉ số trong danh sách `indices`.
- * Trả về trạng thái lưới sau khi xóa các dòng đầy đủ.

5.2.15 Hàm step

```
def step(self, action, render=True, video=None):
    x, num_rotations = action
    self.current_pos = {"x": x, "y": 0}
    for _ in range(num_rotations):
        self.piece = self.rotate(self.piece)

    while not self.check_collision(self.piece, self.current_pos):
        self.current_pos["y"] += 1
        if render:
            self.render(video)

    overflow = self.truncate(self.piece, self.current_pos)
    if overflow:
        self.gameover = True

    self.board = self.store(self.piece, self.current_pos)

    lines_cleared, self.board = self.check_cleared_rows(self.board)
    score = 1 + (lines_cleared ** 2) * self.width
    self.score += score
    self.tetrominoes += 1
    self.cleared_lines += lines_cleared
    if not self.gameover:
        self.new_piece()
    if self.gameover:
        self.score -= 2

    return score, self.gameover
```

Giải thích:

- * Hàm `step` thực hiện một hành động và trả về phần thưởng và trạng thái kết thúc của game.
- * Di chuyển và xoay khối theo hành động đã chọn.
- * Kiểm tra va chạm và lưu trữ khối vào lưới khi nó dừng lại.
- * Xóa các dòng đầy đủ và cập nhật điểm số.
- * Tạo khối mới nếu game chưa kết thúc.
- * Trả về điểm số và trạng thái game (kết thúc hoặc không).

5.2.16 Hàm render

```
def render(self, video=None):
    if not self.gameover:
        img = [self.piece_colors[p] for row in self.get_current_board_state() for p in row]
    else:
        img = [self.piece_colors[p] for row in self.board for p in row]
    img = np.array(img).reshape((self.height, self.width, 3)).astype(np.uint8)
    img = img[..., ::-1]
```

```

img = Image.fromarray(img, "RGB")

img = img.resize((self.width * self.block_size, self.height * self.block_size), 0)
img = np.array(img)
img[[i * self.block_size for i in range(self.height)], :, :] = 0
img[:, [i * self.block_size for i in range(self.width)], :] = 0

img = np.concatenate((img, self.extra_board), axis=1)

cv2.putText(img, "Score:", (self.width * self.block_size + int(self.block_size / 2), self.height * self.block_size),
            fontFace=cv2.FONT_HERSHEY_DUPLEX, fontScale=1.0, color=self.text_color)
cv2.putText(img, str(self.score),
            (self.width * self.block_size + int(self.block_size / 2), 2 * self.block_size),
            fontFace=cv2.FONT_HERSHEY_DUPLEX, fontScale=1.0, color=self.text_color)

cv2.putText(img, "Pieces:", (self.width * self.block_size + int(self.block_size / 2), 4 * self.block_size),
            fontFace=cv2.FONT_HERSHEY_DUPLEX, fontScale=1.0, color=self.text_color)
cv2.putText(img, str(self.tetrominoes),
            (self.width * self.block_size + int(self.block_size / 2), 5 * self.block_size),
            fontFace=cv2.FONT_HERSHEY_DUPLEX, fontScale=1.0, color=self.text_color)

cv2.putText(img, "Lines:", (self.width * self.block_size + int(self.block_size / 2), 7 * self.block_size),
            fontFace=cv2.FONT_HERSHEY_DUPLEX, fontScale=1.0, color=self.text_color)
cv2.putText(img, str(self.cleared_lines),
            (self.width * self.block_size + int(self.block_size / 2), 8 * self.block_size),
            fontFace=cv2.FONT_HERSHEY_DUPLEX, fontScale=1.0, color=self.text_color)

if video:
    video.write(img)

cv2.imshow("Deep Q-Learning Tetris", img)
cv2.waitKey(1)

```

Giải thích:

- * Hàm `render` hiển thị trạng thái hiện tại của game Tetris.
- * Tạo hình ảnh từ trạng thái lưới hiện tại, bao gồm các khối hiện tại và các ô đã lấp đầy.
- * Hiển thị điểm số, số khối và số dòng đã xóa trên màn hình.
- * Nếu video được cung cấp, hình ảnh sẽ được ghi lại vào video.

5.3 Hàm `train.py`

```

def train(opt):
    if torch.cuda.is_available():
        torch.cuda.manual_seed(123)
    else:
        torch.manual_seed(123)
    if os.path.isdir(opt.log_path):
        shutil.rmtree(opt.log_path)
    os.makedirs(opt.log_path)
    writer = SummaryWriter(opt.log_path)
    env = Tetris(width=opt.width, height=opt.height, block_size=opt.block_size)
    model = DeepQNetwork()
    optimizer = torch.optim.Adam(model.parameters(), lr=opt.lr)
    criterion = nn.MSELoss()

    state = env.reset()
    if torch.cuda.is_available():
        model.cuda()
        state = state.cuda()

    replay_memory = deque(maxlen=opt.replay_memory_size)
    epoch = 0
    while epoch < opt.num_epochs:
        next_steps = env.get_next_states()
        # Exploration or exploitation
        epsilon = opt.final_epsilon + (max(opt.num_decay_epochs - epoch, 0) * (
            opt.initial_epsilon - opt.final_epsilon) / opt.num_decay_epochs)

```

```

u = random()
random_action = u <= epsilon
next_actions, next_states = zip(*next_steps.items())
next_states = torch.stack(next_states)
if torch.cuda.is_available():
    next_states = next_states.cuda()
model.eval()
with torch.no_grad():
    predictions = model(next_states)[: , 0]
model.train()
if random_action:
    index = randint(0, len(next_steps) - 1)
else:
    index = torch.argmax(predictions).item()

next_state = next_states[index, :]
action = next_actions[index]

reward, done = env.step(action, render=True)

if torch.cuda.is_available():
    next_state = next_state.cuda()
replay_memory.append([state, reward, next_state, done])
if done:
    final_score = env.score
    final_tetrominoes = env.tetrominoes
    final_cleared_lines = env.cleared_lines
    state = env.reset()
    if torch.cuda.is_available():
        state = state.cuda()
else:
    state = next_state
    continue
if len(replay_memory) < opt.replay_memory_size / 10:
    continue
epoch += 1
batch = sample(replay_memory, min(len(replay_memory), opt.batch_size))
state_batch, reward_batch, next_state_batch, done_batch = zip(*batch)
state_batch = torch.stack(tuple(state for state in state_batch))
reward_batch = torch.from_numpy(np.array(reward_batch, dtype=np.float32)[: , None])
next_state_batch = torch.stack(tuple(state for state in next_state_batch))

if torch.cuda.is_available():
    state_batch = state_batch.cuda()
    reward_batch = reward_batch.cuda()
    next_state_batch = next_state_batch.cuda()

q_values = model(state_batch)
model.eval()
with torch.no_grad():
    next_prediction_batch = model(next_state_batch)
model.train()

y_batch = torch.cat(
    tuple(reward if done else reward + opt.gamma * prediction for reward, done, prediction in
        zip(reward_batch, done_batch, next_prediction_batch)))[: , None]

optimizer.zero_grad()
loss = criterion(q_values, y_batch)
loss.backward()
optimizer.step()

print("Epoch: {}/{}", Action: {}, Score: {}, Tetrominoes {}, Cleared lines: {}".format(
    epoch,
    opt.num_epochs,
    action,
    final_score,
    final_tetrominoes,
    final_cleared_lines))
writer.add_scalar('Train/Score', final_score, epoch - 1)
writer.add_scalar('Train/Tetrominoes', final_tetrominoes, epoch - 1)

```

```

writer.add_scalar('Train/Cleared lines', final_cleared_lines, epoch - 1)

if epoch > 0 and epoch % opt.save_interval == 0:
    torch.save(model, "{}tetris_{}".format(opt.saved_path, epoch))

torch.save(model, "{}tetris".format(opt.saved_path))

```

Giải thích:

*** Hàm ‘train(opt)’:**

- Hàm ‘train’ thực hiện huấn luyện tác nhân DQN cho game Tetris với các tham số được truyền vào từ biến ‘opt’.

*** Khởi tạo và thiết lập:**

- Kiểm tra và thiết lập ngẫu nhiên sử dụng CUDA nếu có sẵn, ngược lại sử dụng CPU.
- Xóa và tạo mới thư mục lưu trữ logs cho TensorBoard.
- Khởi tạo môi trường game Tetris với các tham số chiều rộng, chiều cao và kích thước khối.
- Khởi tạo mô hình DeepQNetwork, bộ tối ưu hóa Adam và hàm mất mát MSELoss.

*** Khởi tạo trạng thái ban đầu:**

- Đặt lại trạng thái môi trường và chuyển mô hình, trạng thái sang CUDA nếu có sẵn.

*** Vòng lặp huấn luyện:**

- Vòng lặp huấn luyện chạy cho đến khi đạt số epoch tối đa.
- Lấy các trạng thái tiếp theo từ môi trường.
- Tính toán epsilon cho chiến lược khám phá (exploration) hoặc khai thác (exploitation).
- Chọn hành động ngẫu nhiên hoặc hành động tốt nhất dựa trên mô hình DQN.
- Thực hiện hành động và nhận phần thưởng, trạng thái kết thúc từ môi trường.
- Lưu trữ trải nghiệm vào bộ nhớ đệm.
- Khi trạng thái kết thúc, đặt lại trạng thái môi trường và cập nhật các thông tin cuối cùng (điểm, số khối, số dòng xóa).
- Tiếp tục nếu bộ nhớ đệm chưa đủ lớn.
- Tăng epoch và lấy mẫu ngẫu nhiên các trải nghiệm từ bộ nhớ đệm để huấn luyện mô hình.
- Tính toán giá trị Q và cập nhật mô hình bằng tối ưu hóa Adam.
- In thông tin huấn luyện và ghi logs vào TensorBoard.
- Lưu trữ mô hình sau mỗi khoảng thời gian đã định.

5.4 Hàm test.py

```

import argparse
import torch
import cv2
from src.tetris import Tetris

def get_args():
    parser = argparse.ArgumentParser(
        """Implementation of Deep Q Network to play Tetris""")

    parser.add_argument("--width", type=int, default=10, help="The common width for all images")
    parser.add_argument("--height", type=int, default=20, help="The common height for all images")
    parser.add_argument("--block_size", type=int, default=30, help="Size of a block")
    parser.add_argument("--fps", type=int, default=300, help="frames per second")

```

```

parser.add_argument("--saved_path", type=str, default="trained_models")
parser.add_argument("--output", type=str, default="output.mp4")

args = parser.parse_args()
return args

def test(opt):
    if torch.cuda.is_available():
        torch.cuda.manual_seed(123)
    else:
        torch.manual_seed(123)
    if torch.cuda.is_available():
        model = torch.load("{}tetris".format(opt.saved_path))
    else:
        model = torch.load("{}tetris".format(opt.saved_path),
                             map_location=lambda storage, loc: storage)
    model.eval()
    env = Tetris(width=opt.width, height=opt.height, block_size=opt.block_size)
    env.reset()
    if torch.cuda.is_available():
        model.cuda()
    out = cv2.VideoWriter(opt.output, cv2.VideoWriter_fourcc(*"MJPG"), opt.fps,
                          (int(1.5*opt.width*opt.block_size), opt.height*opt.block_size))
    while True:
        next_steps = env.get_next_states()
        next_actions, next_states = zip(*next_steps.items())
        next_states = torch.stack(next_states)
        if torch.cuda.is_available():
            next_states = next_states.cuda()
        predictions = model(next_states)[: , 0]
        index = torch.argmax(predictions).item()
        action = next_actions[index]
        _, done = env.step(action, render=True, video=out)

        if done:
            out.release()
            break

if __name__ == "__main__":
    opt = get_args()
    test(opt)

```

Giải thích:

* Hàm ‘get-args()’:

- Hàm ‘get-args’ sử dụng argparse để phân tích các đối số đầu vào.
- Các đối số bao gồm ‘width’, ‘height’, ‘block_size’, ‘fps’, ‘saved_path’, ‘output’.

- Trả về các đối số dưới dạng một đối tượng.

* Hàm ‘test(opt)’:

- Hàm ‘test’ thực hiện quá trình kiểm tra mô hình DQN đã huấn luyện trên game Tetris.
- Đặt seed ngẫu nhiên sử dụng CUDA nếu có sẵn, ngược lại sử dụng CPU.
- Tải mô hình đã lưu từ ‘opt.saved_path’. *Chuyển mô hình sang chế độ đánh giá (evaluation mode)*.

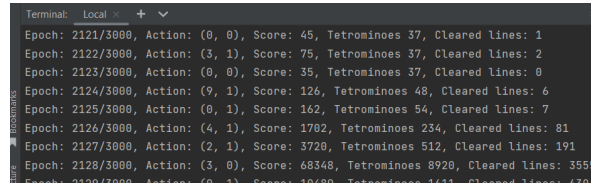
- Khởi tạo môi trường game Tetris với các tham số chiều rộng, chiều cao và kích thước khối.
- Đặt lại trạng thái môi trường.
- Chuyển mô hình sang CUDA nếu có sẵn.
- Khởi tạo VideoWriter để ghi lại video của quá trình chơi game.

* Vòng lặp kiểm tra:

- Lấy các trạng thái tiếp theo từ môi trường.
- Tính toán các hành động và các trạng thái tiếp theo tương ứng.
- Chuyển các trạng thái tiếp theo sang tensor và chuyển sang CUDA nếu có sẵn.

- Sử dụng mô hình để dự đoán giá trị Q cho các trạng thái tiếp theo.
- Chọn hành động có giá trị Q cao nhất.
- Thực hiện hành động và nhận trạng thái kết thúc từ môi trường.
- Ghi lại quá trình chơi game vào video.
- Nếu trạng thái kết thúc, giải phóng VideoWriter và thoát khỏi vòng lặp.

6 DEMO



```

Terminal: Local x + v
Epoch: 2121/3000, Action: (0, 0), Score: 45, Tetrominoes 37, Cleared lines: 1
Epoch: 2122/3000, Action: (3, 1), Score: 75, Tetrominoes 37, Cleared lines: 2
Epoch: 2123/3000, Action: (0, 0), Score: 35, Tetrominoes 37, Cleared lines: 0
Epoch: 2124/3000, Action: (9, 1), Score: 126, Tetrominoes 48, Cleared lines: 6
Epoch: 2125/3000, Action: (0, 1), Score: 162, Tetrominoes 54, Cleared lines: 7
Epoch: 2126/3000, Action: (4, 1), Score: 1702, Tetrominoes 234, Cleared lines: 81
Epoch: 2127/3000, Action: (2, 1), Score: 3720, Tetrominoes 512, Cleared lines: 191
Epoch: 2128/3000, Action: (3, 0), Score: 68348, Tetrominoes 8920, Cleared lines: 3555
Epoch: 2129/3000, Action: (0, 1), Score: 10608, Tetrominoes 5451, Cleared lines: 618
  
```

Hình 8: Các epoch

Giải thích các thông số:

* **Epoch: 2132/3000**

- **Epoch** là một chu kỳ huấn luyện đầy đủ của mô hình.
- Giá trị đầu tiên (2132) cho biết số lượng epoch hiện tại.
- Giá trị thứ hai (3000) là tổng số epoch mà mô hình sẽ được huấn luyện.

* **Action: (6, 2)**

- **Action** là hành động mà tác nhân (agent) thực hiện.
- Hành động được biểu diễn dưới dạng một bộ giá trị (tuple).
- Giá trị đầu tiên (6) là vị trí ngang (x) mà khối Tetris sẽ được đặt.
- Giá trị thứ hai (2) là số lần xoay (rotation) của khối Tetris trước khi đặt.

* **Score: 2720**

- **Score** là điểm số hiện tại của trò chơi.
- Điểm số tăng lên khi người chơi hoặc tác nhân xóa được các hàng trong trò chơi Tetris.

* **Tetrominoes: 512**

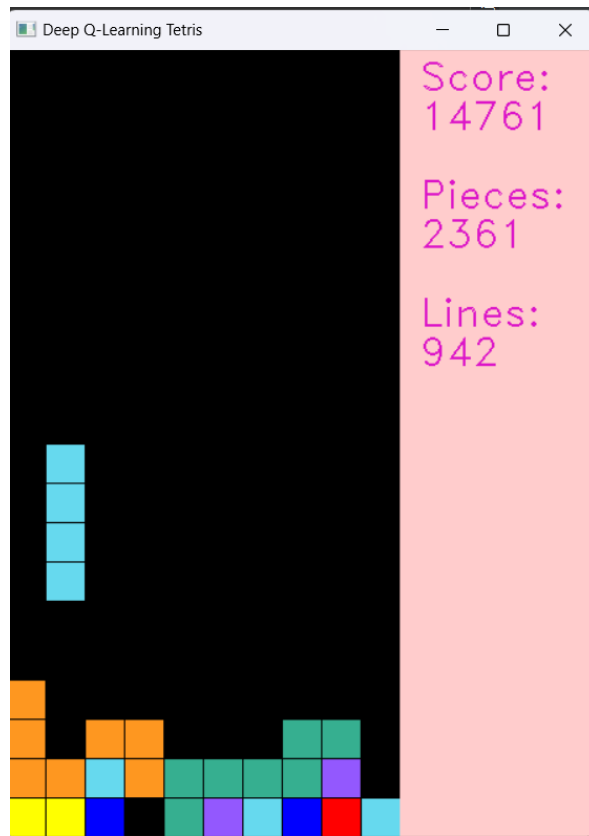
- **Tetrominoes** là số lượng khối Tetris đã rơi xuống trong trò chơi.
- Mỗi khối Tetris rơi xuống sẽ tăng giá trị này lên.

* **Cleared lines: 189**

- **Cleared lines** là số lượng hàng đã được xóa trong trò chơi.
- Hàng được xóa khi tất cả các ô trong hàng đều được lấp đầy bởi các khối Tetris.

7 Kết Luận

Trong dự án này, chúng ta đã khám phá việc sử dụng học tăng cường (Reinforcement Learning) để phát triển một tác nhân AI có khả năng chơi game Tetris. Cụ thể, chúng ta đã áp dụng thuật toán Deep Q-Learning (DQN) để huấn luyện mô hình mạng nơ-ron cho tác nhân.



Hình 9: Enter Caption

7.1 Quá trình xây dựng và huấn luyện

Quá trình xây dựng và huấn luyện bao gồm các bước chính sau:

- * **Xây dựng môi trường game Tetris:**
 - Thiết lập các hàm cần thiết để mô phỏng trò chơi Tetris, bao gồm các hàm để di chuyển, xoay và lưu trữ các khối Tetris, cũng như kiểm tra va chạm và xóa các hàng đầy đủ.
- * **Xây dựng mạng nơ-ron DQN:**
 - Sử dụng PyTorch để xây dựng mô hình Deep Q-Network với ba lớp hoàn toàn kết nối (fully connected layers). Mô hình này chịu trách nhiệm dự đoán giá trị Q cho mỗi hành động trong mỗi trạng thái của game.
- * **Huấn luyện tác nhân DQN:**
 - Sử dụng kinh nghiệm lưu trữ trong bộ nhớ đệm (replay memory) để cập nhật mô hình mạng nơ-ron. Quá trình huấn luyện bao gồm các bước khám phá (exploration) và khai thác (exploitation) để tối ưu hóa hành động của tác nhân.
- * **Kiểm tra và đánh giá:**
 - Sử dụng hàm test để kiểm tra hiệu suất của tác nhân đã được huấn luyện. Tác nhân được yêu cầu chơi game Tetris và kết quả được ghi lại để phân tích.

7.2 Kết quả đạt được

- * **Hiệu suất của tác nhân DQN:** Tác nhân đã học cách chơi game Tetris một cách hiệu quả, với khả năng xóa các hàng và đạt được điểm số cao. Số lượng khối Tetris đã rơi và số lượng hàng đã xóa là các chỉ số chính để đánh giá hiệu suất của tác nhân.

- * **Khám phá và khai thác:** Tác nhân sử dụng chiến lược epsilon-greedy để cân bằng giữa việc khám phá các hành động mới và khai thác các hành động đã biết để tối đa hóa điểm số.

7.3 Thách thức và hạn chế

- * **Yêu cầu tính toán cao:** Quá trình huấn luyện mô hình DQN yêu cầu tài nguyên tính toán lớn, đặc biệt khi số lượng epoch tăng lên.
- * **Khả năng tổng quát hóa:** Mặc dù tác nhân DQN có thể chơi game Tetris tốt trong môi trường huấn luyện, nhưng khả năng tổng quát hóa cho các biến thể khác của game có thể bị hạn chế.

7.4 Hướng phát triển tương lai

- * **Cải thiện kiến trúc mạng nơ-ron:** Thử nghiệm với các kiến trúc mạng nơ-ron khác nhau để cải thiện hiệu suất của tác nhân.
- * **Sử dụng các thuật toán học tăng cường khác:** Khám phá các thuật toán học tăng cường khác như Double DQN, Dueling DQN, và các phương pháp học sâu khác để nâng cao hiệu quả huấn luyện.
- * **Tối ưu hóa bộ nhớ và lấy mẫu:** Cải thiện chiến lược lưu trữ và lấy mẫu kinh nghiệm để tăng tốc độ huấn luyện và nâng cao hiệu quả của tác nhân.

7.5 Tóm tắt

Trong phần này, chúng ta đã tìm hiểu về việc xây dựng mã nguồn và sử dụng Deep Q-Learning (DQN) để phát triển một tác nhân có thể chơi game Tetris. Kết quả đạt được cho thấy tiềm năng của việc áp dụng các kỹ thuật học máy tiên tiến để giải quyết các bài toán trong lĩnh vực game và các ứng dụng khác. Những thách thức và hướng phát triển tương lai mở ra cơ hội để cải thiện và mở rộng nghiên cứu trong lĩnh vực này.