

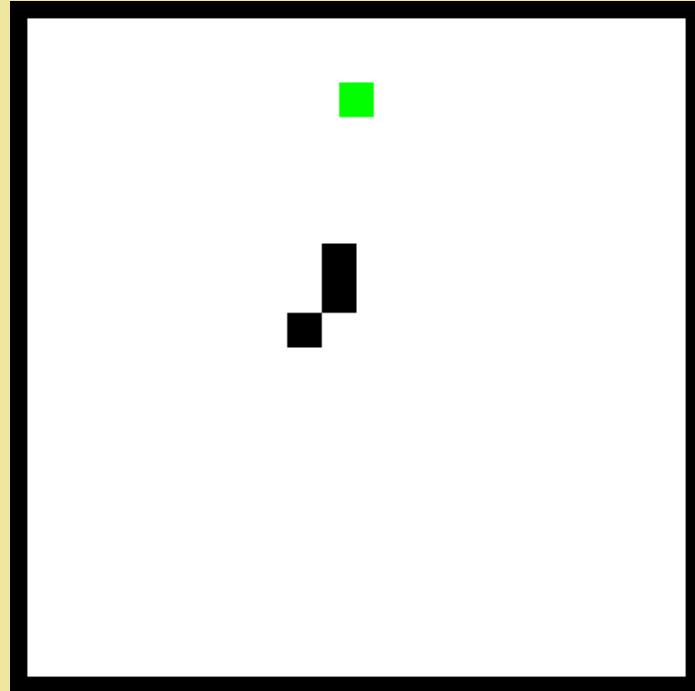


SNAKE GAME AND DRL

**DÙNG DRL ĐỂ CHƠI
RẮN SĂN MỒI**

Le Nhung

CÁC THÀNH PHẦN TRONG GAME



Tác nhân (agent): tác nhân là thực thể đưa ra quyết định dựa trên các quan sát của mình về môi trường. Trong trò chơi Snake, tác nhân điều khiển chuyển động của con rắn.

Cơ chế học tập: Tác nhân học thông qua hệ thống phần thưởng và hình phạt:

- Phần thưởng: Tác nhân nhận được phản hồi tích cực (phần thưởng) khi ăn thức ăn thành công, điều này khuyến khích tác nhân lặp lại hành động đó.
- Hình phạt: Tác nhân nhận được phản hồi tiêu cực (hình phạt) khi va chạm với tường hoặc chính nó, ngăn cản các hành động đó.

CODE

- n_games: Theo dõi số lượng trò chơi do tác nhân chơi.
- epsilon: Kiểm soát sự đánh đổi giữa khám phá và khai thác.
 - giá trị cao hơn biểu thị nhiều khám phá hơn (di chuyển ngẫu nhiên)
 - giá trị thấp hơn biểu thị nhiều khai thác hơn (sử dụng các giá trị đã học)
- gamma: Tỷ lệ chiết khấu cho các phần thưởng trong tương lai.
- memory: Một deque (hàng đợi hai đầu) để lưu trữ các trải nghiệm trong quá khứ (trạng thái, hành động, phần thưởng, trạng thái tiếp theo, cờ đã hoàn thành) lên đến kích thước tối đa (MAX_MEMORY). Các trải nghiệm cũ hơn sẽ bị loại bỏ khi đạt đến giới hạn này.
- model: Một thể hiện của Linear_QNet, là mô hình mạng nơ-ron được sử dụng để dự đoán các giá trị Q cho các hành động khác nhau dựa trên trạng thái hiện tại.
- trainer: Một thể hiện của QTrainer, xử lý việc đào tạo mô hình.

```
class Agent:
    def __init__(self):
        self.n_games = 0
        self.epsilon = 0 # randomness
        self.gamma = 0.9 # discount rate
        self.memory = deque(maxlen=MAX_MEMORY) # popleft()
        self.model = Linear_QNet(11, 256, 3)
        self.trainer = QTrainer(self.model, lr=LR, gamma=self.gamma)
```

```
MAX_MEMORY = 100_000
BATCH_SIZE = 1000
LR = 0.001
```

- Đào tạo hàng loạt bằng cách sử dụng một tập hợp các trải nghiệm được lấy mẫu ngẫu nhiên từ self.memory. Phương pháp này có mục đích hợp nhất quá trình học của tác nhân từ nhiều trải nghiệm trong quá khứ để cải thiện hiệu suất chung.
- Nếu có đủ trải nghiệm được lưu trữ trong bộ nhớ (lớn hơn BATCH_SIZE), phương pháp này sẽ chọn ngẫu nhiên một lô có kích thước BATCH_SIZE.
- Nếu không có đủ trải nghiệm, phương pháp này sẽ sử dụng tất cả các trải nghiệm khả dụng.
- Giải nén các trải nghiệm thành các biến riêng biệt (trạng thái, hành động, phần thưởng, next_states, dones) và chuyển chúng đến phương thức train_step của self.trainer, phương pháp này sẽ cập nhật mạng nơ-ron.

```
def train_long_memory(self):
    if len(self.memory) > BATCH_SIZE:
        mini_sample = random.sample(self.memory, BATCH_SIZE) # list of tuples
    else:
        mini_sample = self.memory

    states, actions, rewards, next_states, dones = zip(*mini_sample)
    self.trainer.train_step(states, actions, rewards, next_states, dones)
```

CODE MEMORY

- Thực hiện đào tạo ngay lập tức dựa trên kinh nghiệm gần đây nhất. Nó được thiết kế để giúp tác nhân học hỏi từ hành động mới nhất của mình và điều chỉnh chính sách của mình nhanh nhất có thể
- Cơ chế:
- Truyền một kinh nghiệm duy nhất cho train_step
- Điều này cho phép tác nhân kết hợp thông tin mới vào mô hình của mình mà không cần chờ tích lũy nhiều kinh nghiệm

```
def train_short_memory(self, state, action, reward, next_state, done):  
    self.trainer.train_step(state, action, reward, next_state, done)
```

CODE MODEL

- Định nghĩa một mô hình mạng nơ-ron đơn giản được sử dụng cho Q-learning trong thiết lập học tăng cường
- `self.linear1`: Đây là lớp đầu tiên được kết nối đầy đủ, tiếp nhận các đặc điểm `input_size` và đầu ra là `hidden_size`
- `self.linear2`: Lớp thứ hai lấy đầu ra từ `linear1` và ánh xạ nó tới `output_size`, tương ứng với không gian hành động (số lượng hành động có thể mà tác nhân có thể thực hiện)
- `F.relu`: Hàm ReLU (Đơn vị tuyến tính chỉnh lưu) được áp dụng sau lớp tuyến tính đầu tiên để đưa vào tính phi tuyến tính, giúp mạng học các mẫu phức tạp hơn trong dữ liệu
- Lớp thứ hai, `self.linear2`, đưa ra các giá trị Q cuối cùng cho mỗi hành động có thể mà không có bất kỳ hàm kích hoạt nào. Điều này thường thấy trong Q-learning, trong đó các giá trị đầu ra thô được sử dụng để chọn các hành động dựa trên các giá trị Q của chúng.
- Lưu các tham số của mô hình đã được đào tạo vào tệp (model.pth)
- `torch.save(self.state_dict(), file_name)` lưu các tham số đã học của mô hình (trọng số và độ lệch), cho phép tải và sử dụng mô hình sau này mà không cần đào tạo lại

```

7  class Linear_QNet(nn.Module):
8      def __init__(self, input_size, hidden_size, output_size):
9          super().__init__()
10         self.linear1 = nn.Linear(input_size, hidden_size)
11         self.linear2 = nn.Linear(hidden_size, output_size)
12
13     def forward(self, x):
14         x = F.relu(self.linear1(x))
15         x = self.linear2(x)
16         return x
17
18     def save(self, file_name='model.pth'):
19         model_folder_path = './model'
20         if not os.path.exists(model_folder_path):
21             os.makedirs(model_folder_path)
22
23         file_name = os.path.join(model_folder_path, file_name)
24         torch.save(self.state_dict(), file_name)
25

```



CODE MODEL

- Đào tạo mạng Q bằng cách cập nhật trọng số của nó dựa trên thuật toán Q-learning.
- lr (Tốc độ học): Xác định kích thước của các bước được thực hiện trong quá trình tối ưu hóa.
- gamma (Hệ số chiết khấu): Kiểm soát tầm quan trọng của phần thưởng trong tương lai trong phép tính giá trị Q.
- Bộ tối ưu hóa Adam được sử dụng để cập nhật trọng số mô hình dựa trên độ dốc.
- Hàm mất mát: nn.MSELoss() tính toán lỗi bình phương trung bình giữa các giá trị Q dự đoán và mục tiêu.

```
26 class QTrainer:  
27     def __init__(self, model, lr, gamma):  
28         self.lr = lr  
29         self.gamma = gamma  
30         self.model = model  
31         self.optimizer = optim.Adam(model.parameters(), lr=self.lr)  
32         self.criterion = nn.MSELoss()  
33  
34     def train_step(self, state, action, reward, next_state, done):  
35         state = torch.tensor(state, dtype=torch.float)  
36         next_state = torch.tensor(next_state, dtype=torch.float)  
37         action = torch.tensor(action, dtype=torch.long)  
38         reward = torch.tensor(reward, dtype=torch.float)  
39         # (n, x)  
40  
41         if len(state.shape) == 1:  
42             # (1, x)  
43             state = torch.unsqueeze(state, 0)  
44             next_state = torch.unsqueeze(next_state, 0)  
45             action = torch.unsqueeze(action, 0)  
46             reward = torch.unsqueeze(reward, 0)  
47             done = (done, )
```

```
49         # 1: predicted Q values with current state  
50         pred = self.model(state)  
51  
52         target = pred.clone()  
53         for idx in range(len(done)):  
54             Q_new = reward[idx]  
55             if not done[idx]:  
56                 Q_new = reward[idx] + self.gamma * torch.max(self.model(next_state[idx]))  
57  
58             target[idx][torch.argmax(action[idx]).item()] = Q_new  
59  
60         # 2: Q_new = r + y * max(next_predicted Q value) -> only do this if not done  
61         # pred.clone()  
62         # preds[argmax(action)] = Q_new  
63         self.optimizer.zero_grad()  
64         loss = self.criterion(target, pred)  
65         loss.backward()  
66  
67         self.optimizer.step()
```



CODE MODEL

- Phương pháp train_step
 - Chuyển đổi trạng thái, next_state, hành động và phần thưởng thành tensor, mà PyTorch yêu cầu để tính toán.
 - Nếu một trải nghiệm duy nhất (thay vì một lô) được truyền qua, nó sẽ thêm một chiều bổ sung để cho phép xử lý lô nhất quán.
- Điều này tính toán các giá trị Q cho tất cả các hành động có thể có trong trạng thái hiện tại bằng cách sử dụng mô hình.

```
33
34     def train_step(self, state, action, reward, next_state, done):
35         state = torch.tensor(state, dtype=torch.float)
36         next_state = torch.tensor(next_state, dtype=torch.float)
37         action = torch.tensor(action, dtype=torch.long)
38         reward = torch.tensor(reward, dtype=torch.float)
39         # (n, x)
40
41         if len(state.shape) == 1:
42             # (1, x)
43             state = torch.unsqueeze(state, 0)
44             next_state = torch.unsqueeze(next_state, 0)
45             action = torch.unsqueeze(action, 0)
46             reward = torch.unsqueeze(reward, 0)
47             done = (done, )
```



CODE MODEL

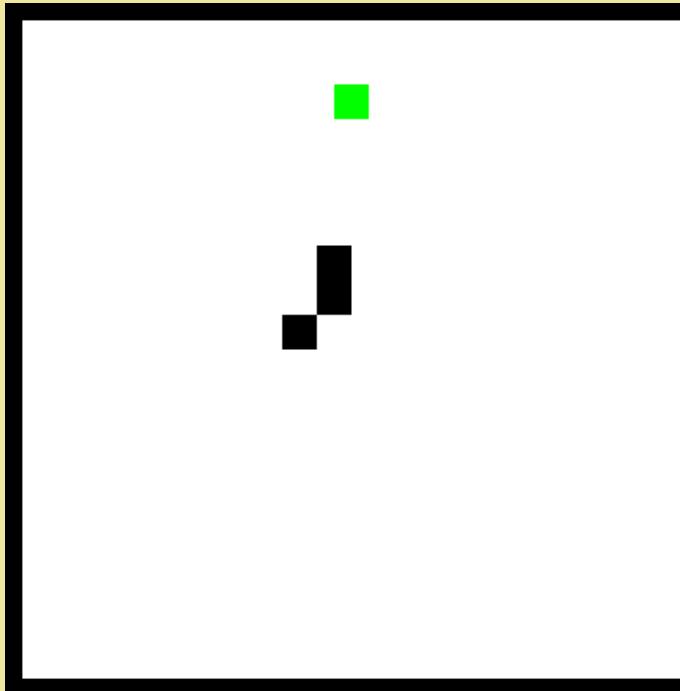
- Cập nhật giá trị Q:
 - Nếu episode đã hoàn thành (done[idx] là True), giá trị Q chỉ đơn giản là phần thưởng ngay lập tức ($Q_{\text{new}} = \text{reward}[\text{idx}]$).
 - Nếu chưa hoàn thành, nó sẽ sử dụng phương trình Bellman:

$$Q_{\text{new}} = \text{reward} + \gamma \times \max(\text{Q-value of next state})$$

- `torch.max(self.model(next_state[idx]))`: cung cấp giá trị Q cao nhất cho trạng thái tiếp theo
- Tensor mục tiêu được sửa đổi để chỉ giá trị Q của hành động đã thực hiện (`torch.argmax(action[idx]).item()`) được cập nhật thành Q_{new}
- Tính toán giá trị mất mát MSE giữa mục tiêu và pred, trong đó mục tiêu giữ các giá trị Q đã cập nhật
 - Các gradient được tính toán bằng `loss.backward()` và trọng số mô hình được cập nhật bằng `self.optimizer.step()`

```
49 | # 1: predicted Q values with current state
50 | pred = self.model(state)
51 |
52 | target = pred.clone()
53 | for idx in range(len(done)):
54 |     Q_new = reward[idx]
55 |     if not done[idx]:
56 |         Q_new = reward[idx] + self.gamma * torch.max(self.model(next_state[idx]))
57 |
58 |     target[idx][torch.argmax(action[idx]).item()] = Q_new
59 |
60 | # 2: Q_new = r + y * max(next_predicted Q value) -> only do this if not done
61 | # pred.clone()
62 | # preds[argmax(action)] = Q_new
63 | self.optimizer.zero_grad()
64 | loss = self.criterion(target, pred)
65 | loss.backward()
66 |
67 | self.optimizer.step()
```

CÁC THÀNH PHẦN TRONG GAME



2

Môi trường (environment):

- Môi trường là mọi thứ mà tác nhân tương tác. Nó bao gồm lưới 2D xung quanh, con rắn, thức ăn và ranh giới.
- Động lực của môi trường xác định cách trạng thái thay đổi để phản ứng với hành động của tác nhân.
 - Ví dụ, di chuyển con rắn theo một hướng nhất định sẽ cập nhật vị trí của nó và có thể dẫn đến việc ăn thức ăn hoặc va vào tường hoặc chính nó.
- Chuyển đổi trạng thái: Môi trường cung cấp phản hồi cho tác nhân dưới dạng các trạng thái mới và phần thưởng sau mỗi hành động.

2

CODE ENVIRONMENT

- Lớp SnakeGameAI bao gồm toàn bộ logic của trò chơi: khởi tạo trạng thái, đặt thức ăn, di chuyển, tính toán phần thưởng, phát hiện va chạm và cập nhật hiển thị.
- Hiển thị và Đồng hồ: Khởi tạo cửa sổ trò chơi và đặt FPS.
- Reset(): Đặt lại trạng thái trò chơi mỗi khi trò chơi mới bắt đầu. Điều này bao gồm khởi tạo vị trí của con rắn, đặt hướng sang PHẢI, đặt lại điểm, đặt thức ăn và đặt bộ đếm khung hình.

```
class SnakeGameAI:  
    def __init__(self, w=640, h=480):  
        self.w = w  
        self.h = h  
        # init display  
        self.display = pygame.display.set_mode((self.w, self.h))  
        pygame.display.set_caption('Snake')  
        self.clock = pygame.time.Clock()  
        self.reset()  
  
    def reset(self):  
        # init game state  
        self.direction = Direction.RIGHT  
  
        self.head = Point(self.w/2, self.h/2)  
        self.snake = [self.head,  
                     Point(self.head.x-BLOCK_SIZE, self.head.y),  
                     Point(self.head.x-(2*BLOCK_SIZE), self.head.y)]  
  
        self.score = 0  
        self.food = None  
        self._place_food()  
        self.frame_iteration = 0
```

2

CODE ENVIRONMENT

```
def _place_food(self):
    x = random.randint(0, (self.w-BLOCK_SIZE )//BLOCK_SIZE )*BLOCK_SIZE
    y = random.randint(0, (self.h-BLOCK_SIZE )//BLOCK_SIZE )*BLOCK_SIZE
    self.food = Point(x, y)
    if self.food in self.snake:
        self._place_food()

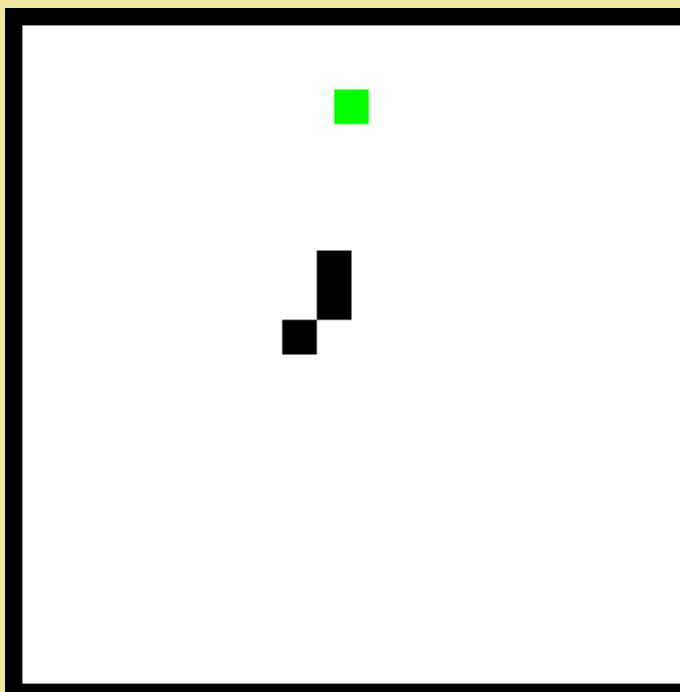
def play_step(self, action):
    self.frame_iteration += 1
    # 1. collect user input
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            quit()

    # Store old state for reward calculation
    old_head = self.head

    # 2. move
    self._move(action) # update the head
    self.snake.insert(0, self.head)
```

- đảm bảo rằng thức ăn được đặt ngẫu nhiên trong ranh giới trò chơi và không chồng lên con rắn
- đảm bảo trò chơi tiến triển từng bước, cập nhật vị trí của con rắn, xử lý va chạm, ăn thức ăn và tính phần thưởng dựa trên hành động của con rắn

CÁC THÀNH PHẦN TRONG GAME



3

Trạng thái (state):

- Biểu thị tình hình hiện tại của môi trường theo nhận thức của tác nhân. Trạng thái này bao gồm tất cả thông tin có liên quan cần thiết để tác nhân đưa ra quyết định.
- Thành phần:
 - Nguy hiểm phía trước
 - Nguy hiểm bên phải
 - Nguy hiểm bên trái
 - Đồ ăn bên trái
 - đồ ăn bên phải
 - Đồ ăn bên trên
 - Đi thẳng
 - Rẽ phải
 - Rẽ trái
 - Đi xuống
- Biểu diễn trạng thái: Trạng thái có thể được biểu diễn dưới dạng vectơ hoặc lưới, tùy thuộc vào cách tác nhân nhận thức môi trường.

CÁC THÀNH PHẦN CỤ THỂ

Trạng thái tác nhân

- sự phản ánh của trò chơi rắn mà tác nhân có thể hiểu
- chứa dữ liệu mà tác nhân phải xử lý để quyết định hành động nào cần thực hiện
- ảnh hưởng đến hiệu suất hoạt động của tác nhân và tốc độ xử lý thông tin
- Nếu cung cấp nhiều dữ liệu có liên quan hơn, tác nhân sẽ hoạt động tốt hơn nhiều nhưng xử lý thông tin chậm hơn
- Mỗi giá trị, có thể là 0 hoặc 1, biểu thị một điều khác nhau

No	State	Value
1	Danger straight	0 or 1
2	Danger right	0 or 1
3	Danger left	0 or 1
4	Direction left	0 or 1
5	Direction right	0 or 1
6	Direction up	0 or 1
7	Direction down	0 or 1
8	Food-left	0 or 1
9	Food-right	0 or 1
10	Food-up	0 or 1
11	Food-down	0 or 1

CODE STATE

```
def get_state(self, game):
    head = game.snake[0]
    point_l = Point(head.x - 20, head.y)
    point_r = Point(head.x + 20, head.y)
    point_u = Point(head.x, head.y - 20)
    point_d = Point(head.x, head.y + 20)

    dir_l = game.direction == Direction.LEFT
    dir_r = game.direction == Direction.RIGHT
    dir_u = game.direction == Direction.UP
    dir_d = game.direction == Direction.DOWN

    # Current direction
    dir_l,
    dir_r,
    dir_u,
    dir_d,

    # Relative food position
    game.food.x < game.head.x, # Food to the left
    game.food.x > game.head.x, # Food to the right
    game.food.y < game.head.y, # Food above
    game.food.y > game.head.y # Food below
]

return np.array(state, dtype=int)
```

```
# Define state representation
state = [
    # Danger straight
    (dir_r and game.is_collision(point_r)) or
    (dir_l and game.is_collision(point_l)) or
    (dir_u and game.is_collision(point_u)) or
    (dir_d and game.is_collision(point_d)),

    # Danger right
    (dir_u and game.is_collision(point_r)) or
    (dir_d and game.is_collision(point_l)) or
    (dir_l and game.is_collision(point_u)) or
    (dir_r and game.is_collision(point_d)),

    # Danger left
    (dir_d and game.is_collision(point_r)) or
    (dir_u and game.is_collision(point_l)) or
    (dir_r and game.is_collision(point_u)) or
    (dir_l and game.is_collision(point_d)),
```



CÁC THÀNH PHẦN TRONG GAME

4

Hành động (action):

- Hành động là tập hợp tất cả các động thái có thể mà tác nhân có thể thực hiện trong môi trường.
- Không gian hành động: Trong trò chơi Snake, không gian hành động thường bao gồm:
 - Di thẳng
 - Di chuyển sang trái
 - Di chuyển sang phải
- Lựa chọn hành động: Tác nhân lựa chọn hành động dựa trên chính sách của mình, chính sách này hướng dẫn quá trình ra quyết định của tác nhân.

- Sử dụng chiến lược epsilon-greedy để lựa chọn hành động
 - với xác suất epsilon, nó sẽ chọn một nước đi ngẫu nhiên (để khuyến khích khám phá)
 - nếu không, nó sẽ chọn hành động có giá trị Q cao nhất do mô hình dự đoán.
- epsilon giảm dần, thúc đẩy khám phá trong các trò chơi đầu và chuyển sang khai thác khi tác nhân có thêm kinh nghiệm.

CODE

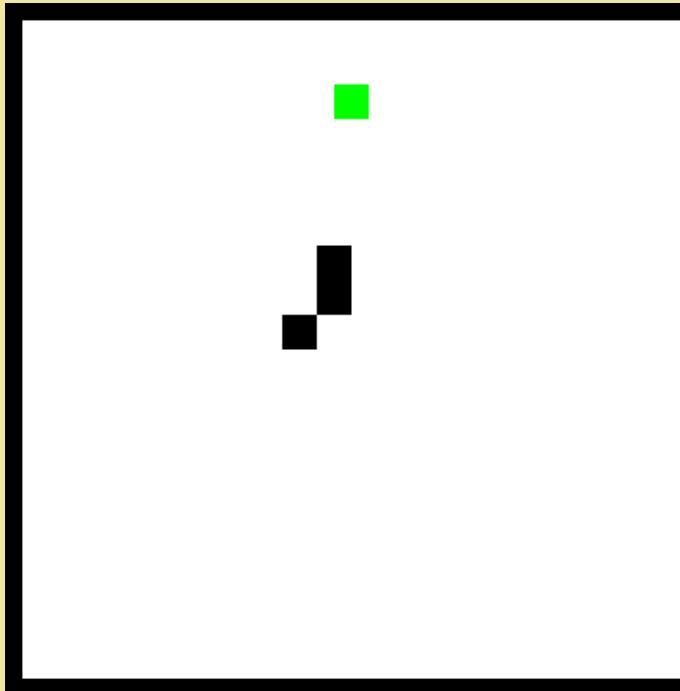
ACTION



- Chịu trách nhiệm quyết định bước đi tiếp theo của tác nhân, cân bằng giữa khám phá với khai thác
- Sử dụng chiến lược epsilon-greedy, trong đó tác nhân đôi khi chọn hành động ngẫu nhiên và đôi khi chọn hành động dựa trên dự đoán của mô hình. Sự cân bằng này giúp tác nhân học hỏi và cải thiện hiệu suất của mình theo thời gian bằng cách ban đầu khám phá môi trường trò chơi và dần dần dựa vào các mẫu đã học
- self.epsilon: giá trị epsilon, kiểm soát tính ngẫu nhiên của hành động, giảm dần khi số lượng trò chơi tăng lên. Epsilon cao hơn có nghĩa là khám phá nhiều hơn, trong khi epsilon thấp hơn có lợi cho việc khai thác
- Nếu giá trị ngẫu nhiên nhỏ hơn epsilon, tác nhân sẽ chọn hành động ngẫu nhiên: [0, 1, 2]=[thắng, phải, trái]
 - Nếu tác nhân quyết định khai thác (sử dụng mô hình đã được đào tạo) nó sẽ chuyển đổi trạng thái hiện tại thành một tensor và đưa vào mạng nơ-ron (self.model)
 - Mô hình đưa ra các giá trị Q dự đoán cho mỗi hành động có thể xảy ra và hành động có giá trị Q cao nhất (torch.argmax(prediction).item()) sẽ được chọn
 - Hành động đã chọn được biểu diễn dưới dạng dlist ([1, 0, 0] = thắng, [0, 1, 0] = phải và [0, 0, 1] = trái), mà hàm play_step sẽ diễn giải để di chuyển con rắn theo đó

```
def get_action(self, state):  
    # random moves: tradeoff exploration / exploitation  
    self.epsilon = 80 - self.n_games  
    final_move = [0,0,0]  
    if random.randint(0, 200) < self.epsilon:  
        move = random.randint(0, 2)  
        final_move[move] = 1  
    else:  
        state0 = torch.tensor(state, dtype=torch.float)  
        prediction = self.model(state0)  
        move = torch.argmax(prediction).item()  
        final_move[move] = 1  
  
    return final_move
```

CÁC THÀNH PHẦN TRONG GAME



5

Chính sách (policy):

- Là chiến lược mà tác nhân sử dụng để xác định hành động nào sẽ thực hiện trong trạng thái nhất định. Nó có thể là
 - xác định (luôn chọn cùng một hành động cho một trạng thái)
 - ngẫu nhiên (chọn hành động dựa trên phân phối xác suất)
- Biểu diễn chính sách: Trong DRL, chính sách thường được biểu diễn bằng mạng nơ-ron lấy trạng thái hiện tại làm đầu vào và đưa ra phân phối xác suất cho các hành động có thể.
- Học chính sách: Tác nhân cập nhật chính sách của mình dựa trên phần thưởng nhận được và các trạng thái đã truy cập, sử dụng Q learning

- mô hình mạng nơ-ron được sử dụng để dự đoán các giá trị Q cho các hành động khác nhau dựa trên trạng thái hiện tại
- Mạng nơ-ron này xấp xỉ hàm giá trị hành động (hàm Q) của tác nhân
- Mạng nơ-ron có hai lớp tuyến tính:
 - linear1 chuyển đổi đầu vào (trạng thái hiện tại) thành một biểu diễn ẩn.
 - linear2 đưa ra các giá trị Q tương ứng với mỗi hành động

CODE

Lựa chọn hành động (Thực thi chính sách):

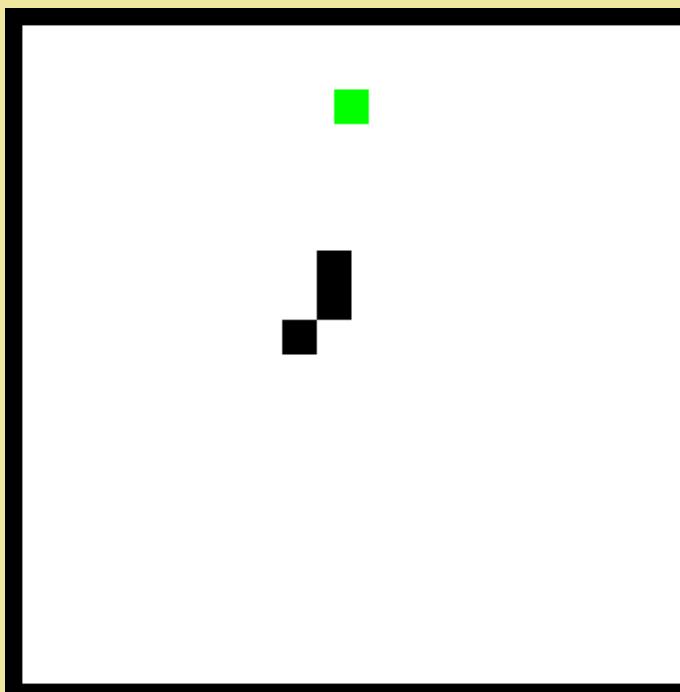
- Chính sách được thực thi trong phương thức get_action
- Tác nhân quyết định giữa việc khám phá và khai thác
- Sự đánh đổi này được kiểm soát bởi epsilon, giảm dần theo thời gian khi tác nhân có thêm kinh nghiệm.

```
class Linear_QNet(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super().__init__()
        self.linear1 = nn.Linear(input_size, hidden_size)
        self.linear2 = nn.Linear(hidden_size, output_size)
```

```
def get_action(self, state):
    # random moves: tradeoff exploration / exploitation
    self.epsilon = 80 - self.n_games
    final_move = [0,0,0]
    if random.randint(0, 200) < self.epsilon:
        move = random.randint(0, 2)
        final_move[move] = 1
    else:
        state0 = torch.tensor(state, dtype=torch.float)
        prediction = self.model(state0)
        move = torch.argmax(prediction).item()
        final_move[move] = 1

    return final_move
```

CÁC THÀNH PHẦN TRONG GAME



6

Phần thưởng (reward):

- Phần thưởng là tín hiệu phản hồi vô hướng mà tác nhân nhận được sau khi thực hiện hành động ở trạng thái cụ thể. Nó chỉ ra lợi ích tức thời của hành động đó
- Cấu trúc phần thưởng:
 - Ăn trái cây: 10
 - Trạng thái mới gần trái cây hơn so với trạng thái trước đó: 1
 - Trạng thái mới xa trái cây hơn so với trạng thái trước đó: -1
 - Chết: -10
 - Bất kỳ phần thưởng nào khác: 0
- Mục tiêu: Mục tiêu của tác nhân là tối đa hóa phần thưởng tích lũy theo thời gian, dẫn đến các chiến lược tốt hơn để đạt được thành công lâu dài.

6

CODE

- Trạng thái mới gần trái cây hơn so với trạng thái trước đó: 1
- Trạng thái mới xa trái cây hơn so với trạng thái trước đó: -1

```
# 4. place new food or just move
if self.head == self.food:
    self.score += 1
    reward = 10
    self._place_food()
else:
    self.snake.pop()
```

Ăn trái cây: 10

```
# 5. Reward based on distance to food
new_reward = self.reward_distance_to_fruit(old_head, self.head)
reward += new_reward # Add distance-based reward

def reward_distance_to_fruit(self, old_head, new_head):
    # Calculate Manhattan distance to food for the old and new states
    old_distance = abs(old_head.x - self.food.x) + abs(old_head.y - self.food.y)
    new_distance = abs(new_head.x - self.food.x) + abs(new_head.y - self.food.y)
```

```
# 3. check if game over
reward = 0
game_over = False
if self.is_collision() or self.frame_iteration > 100 * len(self.snake):
    game_over = True
    reward = -10
return reward, game_over, self.score
```

Chết: -10

TƯƠNG TÁC GIỮA CHÍNH SÁCH, PHẦN THƯỞNG VÀ TRẠNG THÁI

Biểu diễn trạng thái: Tác nhân nhận được trạng thái hiện tại của môi trường.

Ứng dụng chính sách: Mô hình dự đoán các giá trị Q cho mỗi hành động dựa trên trạng thái hiện tại. Hành động có giá trị Q cao nhất được chọn làm bước tiếp theo.

Phần thưởng: Sau khi thực hiện một hành động, tác nhân nhận được phần thưởng từ môi trường. Phần thưởng này được sử dụng trong quá trình đào tạo để cập nhật chính sách.

Cập nhật chính sách: Loss được tính toán dựa trên các giá trị Q dự đoán và các giá trị Q mục tiêu và trình tối ưu hóa cập nhật các tham số của mô hình để cải thiện chính sách.

```
state0 = torch.tensor(state, dtype=torch.float)
prediction = self.model(state0)
```

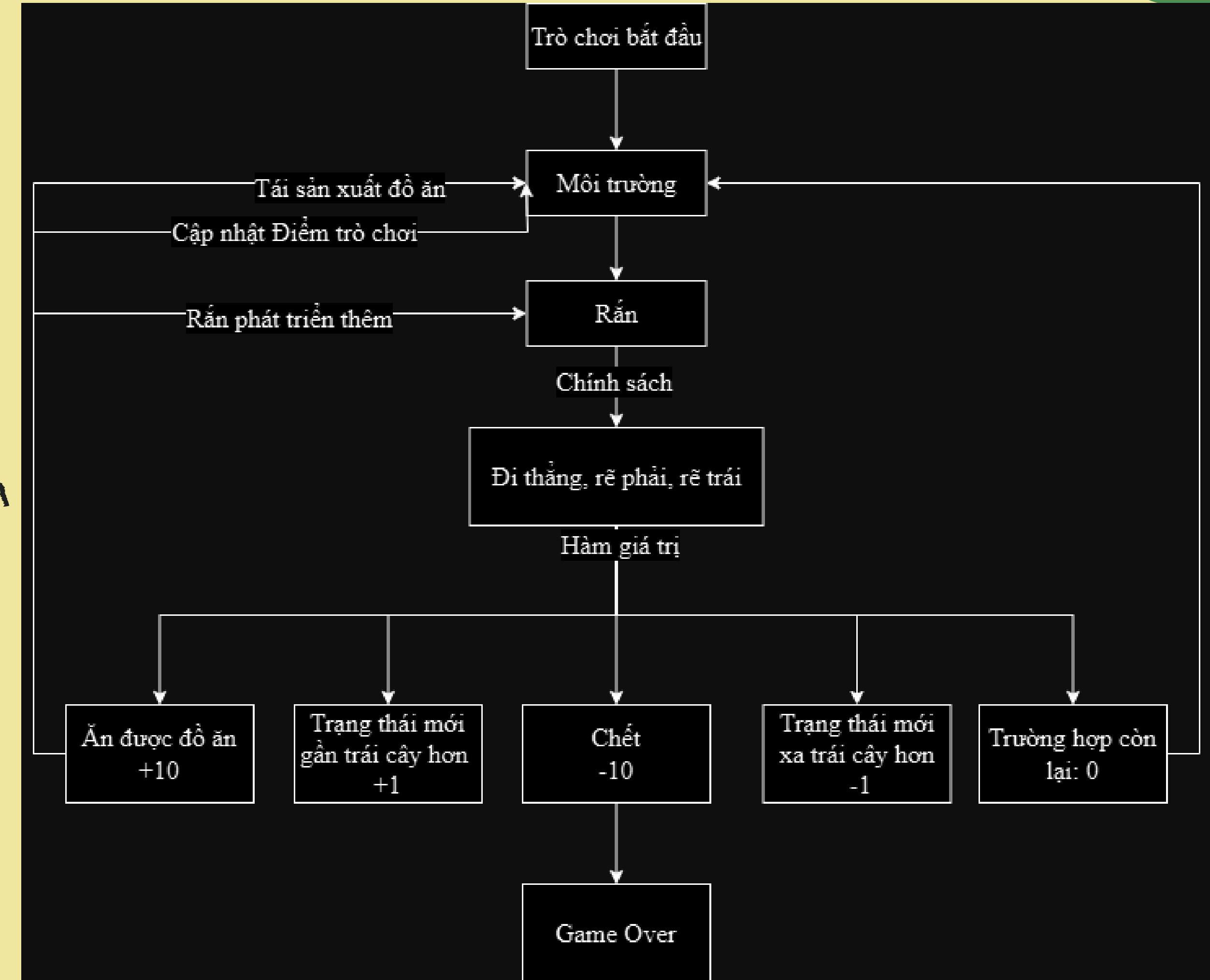
```
move = torch.argmax(prediction).item()
```

```
Q_new = reward[idx]
if not done[idx]:
    Q_new = reward[idx] + self.gamma * torch.max(self.model(next_state[idx]))
target[idx][torch.argmax(action[idx]).item()] = Q_new
```

```
self.optimizer.zero_grad()
loss = self.criterion(target, pred)
loss.backward()

self.optimizer.step()
```

FLOW CHART



CÁC BƯỚC CỤ THỂ

Bước 1: Khởi tạo trò chơi

- Khởi tạo khung trò chơi 800 x 600 làm môi trường
- Khởi tạo rắn và thức ăn

Bước 2: Bắt đầu trò chơi

- Quan sát trạng thái: Ghi lại trạng thái hiện tại của trò chơi (vị trí rắn, vị trí thức ăn, v.v.).



CÁC BƯỚC CỤ THỂ

Bước 3: Dùng policy để dự đoán hướng đi tiếp theo

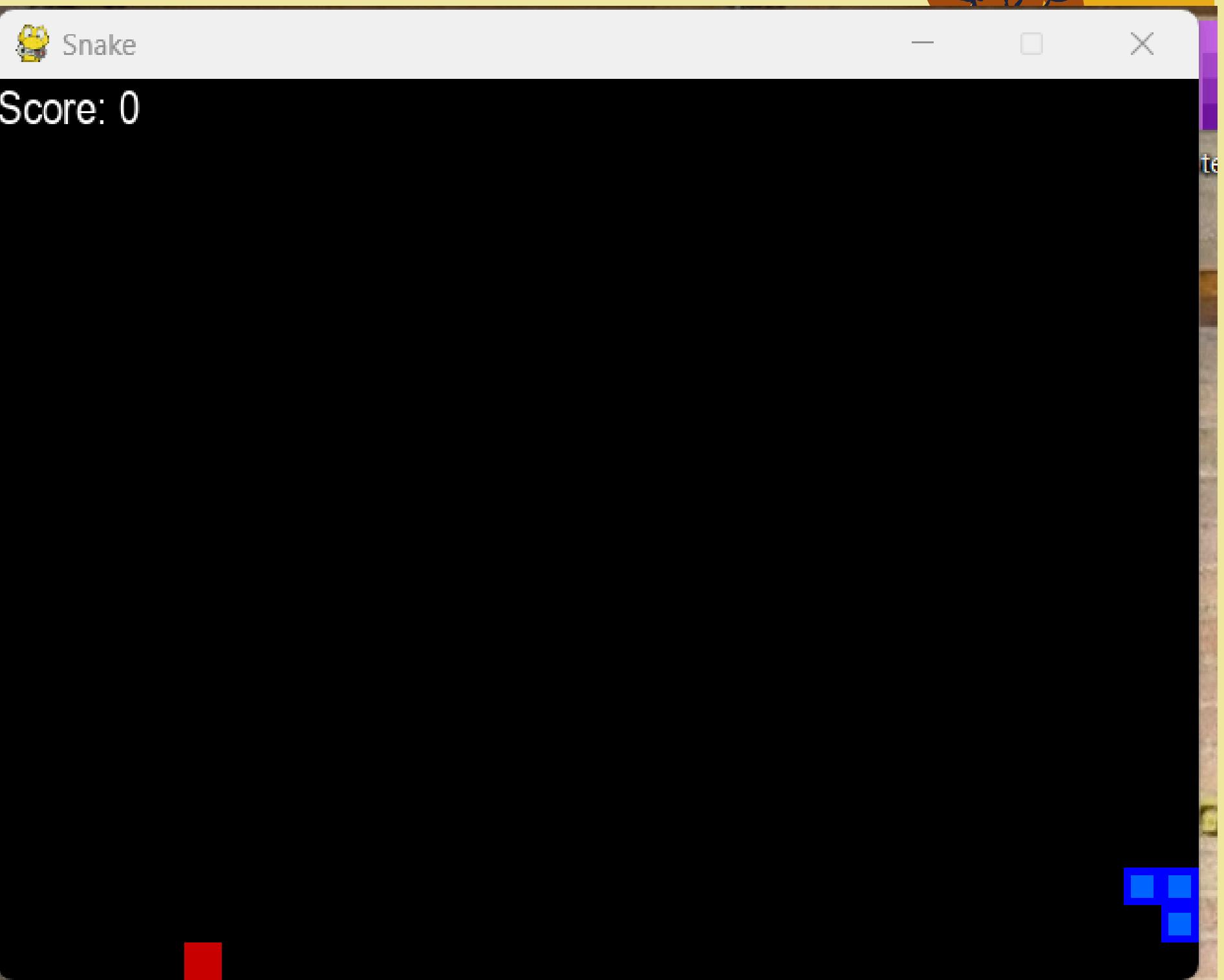
- Sử dụng mạng nơ-ron để đánh giá trạng thái hiện tại và chọn hành động (di chuyển lên, xuống, trái, phải)
- Di chuyển rắn dựa trên hành động đã chọn
- Cập nhật trạng thái trò chơi dựa trên hành động
- Kiểm tra va chạm (rắn với tường hoặc chính nó)



CÁC BƯỚC CỤ THỂ

Bước 4: Tính toán phần thưởng

- Phần thưởng dựa trên hành động mà tác nhân đã làm trước đó
- ăn trái cây: +10
- trạng thái mới gần trái cây hơn trạng thái cũ: +1
- trạng thái mới xa trái cây hơn trạng thái cũ: -1
- chết: -10
- các trường hợp khác: 0
- Lưu trải nghiệm (trạng thái, hành động, phần thưởng, trạng thái tiếp theo) để học trong tương lai



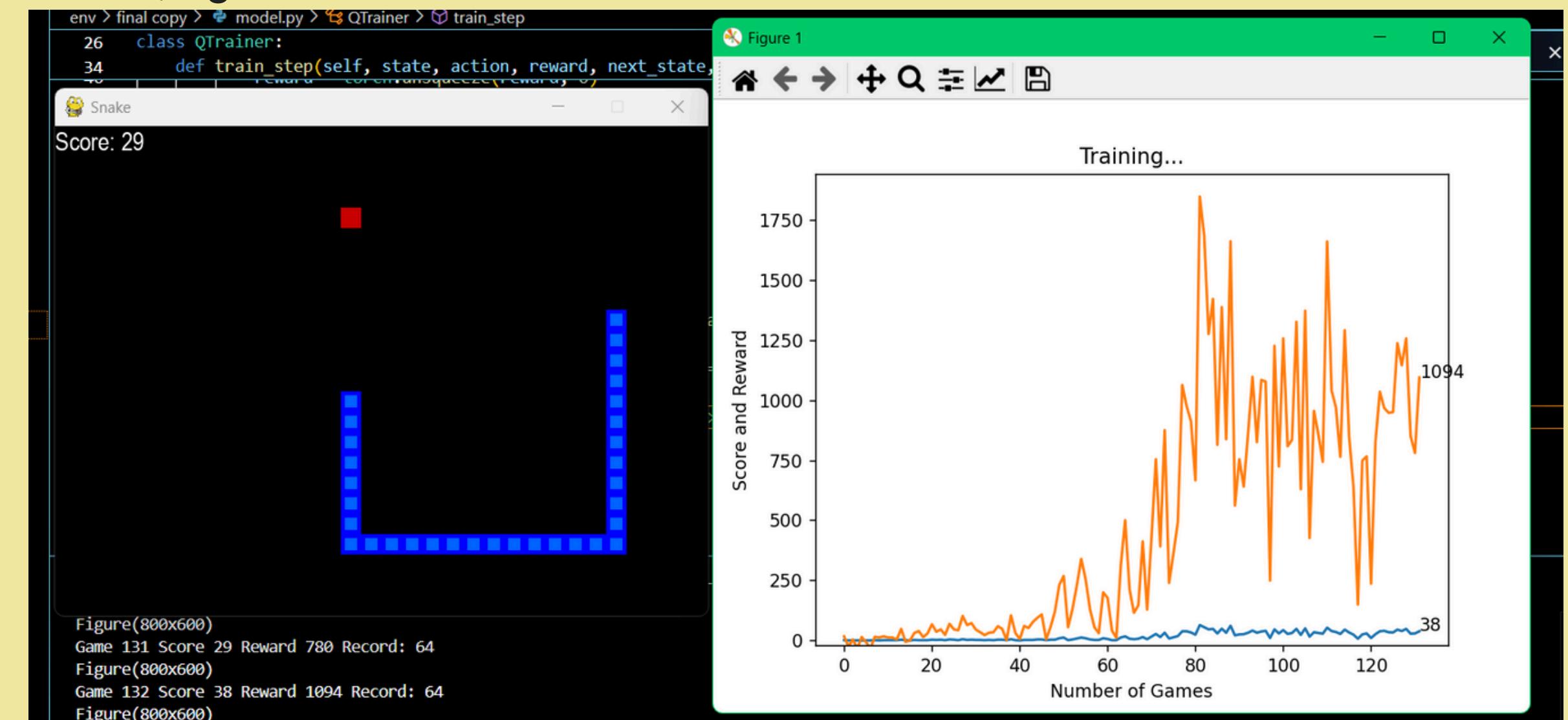
CÁC BƯỚC CỤ THỂ

Bước 5: Đào tạo mạng nơ-ron

- Sử dụng các trải nghiệm đã lưu trữ để đào tạo mạng nơ-ron
- Điều chỉnh giá trị Q dựa trên phần thưởng và hành động đã thực hiện
- Kiểm tra xem trò chơi đã kết thúc chưa (phát hiện va chạm)
- Nếu chưa, hãy quay lại Quan sát trạng thái.

Bước 6: Điều chỉnh tham số

- Điều chỉnh các tham số và cải thiện mô hình nếu cần





A vibrant, cartoon-style illustration of a castle's outer wall and courtyard. The wall is made of grey stone blocks with several tall, thin towers. In the foreground, there's a large white speech bubble containing the text. To the left of the bubble, a tiger stands on a small hill. To the right, a lion sits near a blue pond where a grey dragon is coiled. A green tree with a red trunk is on the left, and a brown gate is at the bottom center.

**THANK YOU
FOR LISTENING**