**BỘ GIÁO DỤC VÀ ĐÀO TẠO**

**TRƯỜNG ĐẠI HỌC MỞ THÀNH PHỐ HỒ CHÍ MINH**

**TP. HỒ CHÍ MINH, 2025**



**Báo CÁO**

**bÀI TÂP LỚN MÔN: TRÍ TUỆ NHÂN TẠO**

**đỀ TÀI: HUẤN LUYỆN AI CHƠI TRÒ CHƠI RẮN SĂN MỒI**

**gIẢNG VIÊN HƯỚNG DẪN: lÊ vIẾT tUẤN**

**HỌC SINH THỰC HIỆN**

**HỌ VÀ TÊN: LÊ MINH DUY**

**MSSV:2351010033**

**HỌ VÀ TÊN: HUỲNH QUANG VŨ**

**MSSV:2351010247**

**NHẬN XÉT CỦA GIẢNG VIÊN HƯỚNG DẪN**

MỤC LỤC

[Chương 1. Giới thiệu 9](#_Toc208519150)

[1.1. Tổng quan bài toán 9](#_Toc208519151)

[1.1.1. Giới thiệu về trò chơi con Rắn 9](#_Toc208519152)

[1.1.2. Giới thiệu về đề tài 9](#_Toc208519153)

[1.2. Hạn chế của các phương pháp hiện tại 9](#_Toc208519154)

[1.2.1. Phần thưởng khan hiếm. 9](#_Toc208519155)

[1.2.2. Không gian trạng thái lớn dẫn đến chậm hội tụ. 9](#_Toc208519156)

[1.3. Phương pháp đề xuất 10](#_Toc208519157)

[1.4. Đóng góp của đề tài 10](#_Toc208519158)

[Chương 2. Các nghiên cứu liên quan 10](#_Toc208519159)

[1.5. Watkins & Dayan giới thiệu về Q-learning 10](#_Toc208519160)

[1.5.1. Bối cảnh 10](#_Toc208519161)

[1.5.2. Giới thiệu Q-learning 10](#_Toc208519162)

[1.5.3. Ý tưởng cốt lõi 10](#_Toc208519163)

[1.6. Các cải tiến thuật toán Q-learning 11](#_Toc208519164)

[1.6.1. Double Q-learning (Van Hasselt, 2010) 11](#_Toc208519165)

[1.6.2. Dueling Q-network (Wang et al., 2016) 11](#_Toc208519166)

[1.6.3. Prioritized Experience Replay (Schaul et al., 2016) 11](#_Toc208519167)

[1.6.4. Deep Q-Network (DQN) – Mnih et al. (2015, DeepMind) 11](#_Toc208519168)

[1.6.5. Extensions of DQN: 11](#_Toc208519169)

[1.7. Các hướng nghiên cứu mở rộng 12](#_Toc208519170)

[Chương 3. Phương pháp đề xuất 12](#_Toc208519171)

[1.8. Tổng quan phương pháp 12](#_Toc208519172)

[1.9. Môi trường & mã nguồn liên quan 12](#_Toc208519173)

[1.9.1. Mã chính liên quan: 12](#_Toc208519174)

[1.9.2. Một số biến và mặc định xuất hiện trong project: 12](#_Toc208519175)

[1.10. Biểu diễn state và action 13](#_Toc208519176)

[1.11. Kiến trúc mạng (Linear\_QNet) 13](#_Toc208519177)

[1.11.1. Mạng được định nghĩa như sau: 13](#_Toc208519178)

[1.11.2. Chức năng: 14](#_Toc208519179)

[1.12. Thuật toán huấn luyện:Train\_step 14](#_Toc208519180)

[1.12.1. Thuật toán huấn luyện được định nghĩa như sau: 14](#_Toc208519181)

[1.12.2. Chức năng hàm: 15](#_Toc208519182)

[1.12.3. Quy trình bên trong train\_step: 15](#_Toc208519183)

[1.12.4. Giải thích logic: 16](#_Toc208519184)

[1.12.5. Ví dụ tính toán thực tế 16](#_Toc208519185)

[1.13. Replay buffer và chiến lược huấn luyện 16](#_Toc208519186)

[1.14. Cấu trúc lớp Agent 17](#_Toc208519187)

[1.14.1. Hàm khởi tạo \_\_init\_\_ 17](#_Toc208519188)

[1.14.2. Hàm get\_state(game) 18](#_Toc208519189)

[1.14.3. Hàm remember(state, action, reward, next\_state, done) 19](#_Toc208519190)

[1.14.4. Hàm train\_short\_memory(state, action, reward, next\_state, done) 19](#_Toc208519191)

[1.14.5. Hàm train\_long\_memory() 19](#_Toc208519192)

[1.14.6. Hàm get\_action(state) 20](#_Toc208519193)

[1.15. Vòng lặp huấn luyện chính – train() 21](#_Toc208519194)

[1.15.1. Các bước thực hiện: 21](#_Toc208519195)

[1.15.2. Định nghĩa hàm: 21](#_Toc208519196)

[Chương 4. Kết quả và thực nghiệm 22](#_Toc208519197)

[1.16. Kết quả 22](#_Toc208519198)

[1.16.1. Kết quả sau 400 lần chơi: 22](#_Toc208519199)

[1.16.2. Kết quả sau 800 lần chơi: 23](#_Toc208519200)

[1.16.3. Nhận xét 23](#_Toc208519201)

[Chương 5. Kết luận 24](#_Toc208519202)

[1.17. Nhận xét tổng quan: 24](#_Toc208519203)

[1.18. Hạn chế hiện tại & hướng cải tiến 24](#_Toc208519204)

[Tài liệu tham khảo 25](#_Toc208519205)

# DANH MỤC TỪ VIẾT TẮT

|  |  |  |
| --- | --- | --- |
| Viết tắt | Từ đầy đủ | Giải thích |
| AI | Artificial Intelligence | Trí tuệ nhân tạo |
| RL | Reinforcement Learning | Học tăng cường |
| DQN | Deep Q-Network | Mạng Q sâu, dùng Q-learning với mạng nơ-ron |
| Q | Q-function | Hàm giá trị hành động trong Q-learning |
| DDQN | Double Deep Q-Network | Double DQN, cải tiến DQN để tránh overestimation |
| CNN | Convolutional Neural Network | Mạng nơ-ron tích chập (đề xuất cải tiến) |
| RNN | Recurrent Neural Network | Mạng nơ-ron hồi quy (đề xuất cải tiến) |
| PER | Prioritized Experience Replay | Replay buffer ưu tiên các trải nghiệm quan trọng |
| MSE | Mean Squared Error | Hàm mất mát bình phương trung bình |
| LR | Learning Rate | Tốc độ học trong huấn luyện mạng nơ-ron |
| γ (GAMMA) | Discount Factor | Hệ số chiết khấu trong Q-learning |
| ReLU | Rectified Linear Unit | Hàm kích hoạt phi tuyến trong mạng nơ-ron |
| np.array | Numpy array | Cấu trúc dữ liệu mảng trong Python |
| torch | PyTorch | Thư viện học sâu trong Python |
| Atari 2600 | Tên máy chơi game | Nền tảng dùng thử nghiệm DQN |

# DANH MỤC HÌNH VẼ

[Figure 1 Đồ thị sau khi training 400 lần 23](#_Toc208519798)

[Figure 2 Đồ thị sau khi training 800 lần 24](#_Toc208519799)

# Giới thiệu

## Tổng quan bài toán

### Giới thiệu về trò chơi con Rắn

Về trò chơi con rắn thì trò chơi con rắn là một trò chơi arcade phổ biến được chơi trên điện thoại và máy tính. Mục tiêu của trò chơi là ăn càng nhiều táo càng tốt mà không va vào tường hoặc thân rắn. Rắn bắt đầu với chiều dài là một (tức là chỉ có đầu) và mỗi lần rắn ăn một quả táo, nó sẽ tăng chiều dài thêm một ít.

### Giới thiệu về đề tài

Trong những năm gần đây, mô hình học tăng cường (Reinforcement Learning — RL) đã chứng minh khả năng học các hành vi phức tạp từ tương tác thô với môi trường từ chơi game cổ điển tới điều khiển robot. Dự án này tập trung vào một bài toán điển hình và trực quang: huấn luyện một tác tử (agent) để chơi trò Snake tự động. Mục tiêu là thiết kế và triển khai game trò chơi con rắn tự động học được cách chơi qua nhiều lần chơi dựa trên Q-learning kết hợp mạng nơ-ron (Deep Q-Network — DQN) hoàn toàn bằng Python, dùng pygame để dựng môi trường game và torch (PyTorch) để xây dựng/học mạng nơ-ron.

## Hạn chế của các phương pháp hiện tại

### Phần thưởng khan hiếm.

Trong trò chơi con rắn, tại một thời điểm chỉ có một con mồi và xuất hiện ngẫu nhiên trên bản đồ. Agent chỉ được khen thưởng khi ăn được con mồi còn hầu hết thời gian là để “tránh chết” điều này dẫn đến việc training diễn ra rất lâu. Agent dể rơi vào trạng thái sống lâu nhưng không dám ăn mồi.

### Không gian trạng thái lớn dẫn đến chậm hội tụ.

Vị trí đầu rắn, thân rắn, thức ăn, hướng đi → tạo ra state space khổng lồ. DQN phải nén trạng thái qua neural network, nhưng vẫn khó tổng quát tốt khi map lớn hoặc rắn dài, việc học sẽ trở nên kém hiệu quả về sau. Ví dụ:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Số lần chơi | 10 | 20 | 30 | 70 | 100 | 200 | 300 | 400 |
| Điểm trung bình | 2.97 | 2.36 | 2.83 | 6.04 | 13.24 | 24.33 | 25.71 | 26.75 |
| Điểm cao nhất | 4 | 6 | 9 | 20 | 39 | 57 | 69 | 69 |

## Phương pháp đề xuất

Dùng phương pháp Deep Q-Network. Phương pháp này sử dụng mạng neural để xấp xỉ Q-funtion, giải quyết được vấn đề không gian trạng thái lớn. Kết hợp với thưởng thêm, tức là ngoại trừ điểm thưởng khi ăn mồi, có thể thêm vào cộng điểm khi tiến lại gần con mồi và trừ điểm khi rắn tiến xa ra khỏi con mồi. ngoài ra khi số bước đi của Rắn quá lớn mà chưa ăn mồi, ta kết thúc trò chơi và khởi động lại game tiếp theo. Kết hợp những phương pháp này sẽ cho ta tốc độ hội tụ Q-values nhanh hơn

## Đóng góp của đề tài

Lê Minh Duy: hoàn thành 100%

Huỳnh Quang Vũ: hoàn thành 100%

# Chương 2. Các nghiên cứu liên quan

## Watkins & Dayan giới thiệu về Q-learning

### Bối cảnh

Trước khi Watkins & Dayan giới thiệu Q-learning, đa số phương pháp trong **học tăng cường (Reinforcement Learning)** đều dựa trên **Dynamic Programming** (cần biết mô hình môi trường) hoặc **Monte Carlo** (cần tập mẫu dài).  
Vấn đề đặt ra: **làm sao để một agent học chính sách tối ưu mà không biết trước mô hình môi trường?**

### Giới thiệu Q-learning

* **Christopher J.C.H. Watkins** lần đầu đưa ra thuật toán Q-learning trong **luận án tiến sĩ năm 1989**.
* Năm **1992**, Watkins cùng **Peter Dayan** công bố bài báo "Q-learning" trên **Machine Learning Journal**, trình bày chi tiết và chứng minh tính hội tụ.

Thuật toán được giới thiệu như một **phương pháp học tăng cường không cần mô hình (model-free RL)**, giúp agent học được:

1. **Hàm giá trị hành động tối ưu**: Q∗(s,a).
2. **Chính sách tối ưu**: π∗(s)=argmaxa​Q∗(s,a).

### Ý tưởng cốt lõi

* Q-learning cập nhật giá trị Q dựa trên **phương trình Bellman tối ưu:**

Q(st​, at​)←Q(st​, at​)+α[rt+1​+γamax​Q(st+1​,a)−Q(st​,at​)]

Trong đó:

* α: tốc độ học.
* γ: hệ số chiết khấu.
* rt+1: phần thưởng ngay sau hành động.
* Điểm mới: sử dụng **ước lượng cực đại (max over actions)** ở trạng thái tiếp theo, nên **không phụ thuộc chính sách hiện tại** → đây là **off-policy RL**.

## Các cải tiến thuật toán Q-learning

### ****Double Q-learning (Van Hasselt, 2010)****

* + Khắc phục vấn đề **overestimation bias** của Q-learning gốc bằng cách dùng hai hàm Q độc lập.
  + Ứng dụng rộng trong game và robot.

### ****Dueling Q-network (Wang et al., 2016)****

* + Tách giá trị **state value** và **advantage value** → giúp học nhanh hơn và ổn định hơn.

### ****Prioritized Experience Replay (Schaul et al., 2016)****

* + Ưu tiên lưu lại và huấn luyện trên những trải nghiệm “quan trọng” (có sai số dự đoán lớn), tăng hiệu quả học.

### ****Deep Q-Network (DQN) – Mnih et al. (2015, DeepMind)****

* + Kết hợp Q-learning với deep neural network để xử lý **không gian trạng thái lớn** (hình ảnh pixel).
  + Nổi tiếng với thành công trong việc chơi **Atari 2600** vượt trội con người.

### ****Extensions of DQN****:

* + **Double DQN (DDQN, 2016)**: kết hợp Double Q-learning vào DQN.
  + **Dueling DQN (2016):** cải tiến mạng Q-value.
  + **Rainbow DQN (Hessel et al., 2017):** kết hợp nhiều cải tiến (DDQN + PER + Dueling + NoisyNet + Distributional Q-learning).

## Các hướng nghiên cứu mở rộng

* **Multi-agent Q-learning**: nhiều tác tử học song song, áp dụng trong mạng không dây, giao thông thông minh, robot hợp tác.
* **Hierarchical Q-learning**: chia nhiệm vụ phức tạp thành nhiều lớp hành động con, giảm độ khó.
* **Safe Q-learning**: thêm ràng buộc an toàn khi huấn luyện trong môi trường rủi ro (robot, xe tự lái).
* **Continuous action space Q-learning**: kết hợp với Approximation methods (ví dụ Deep Deterministic Policy Gradient – DDPG).

# Chương 3. Phương pháp đề xuất

## Tổng quan phương pháp

Agent chơi trò chơi con rắn sử dụng kỹ thuật **DQN**. Hệ thống gồm một môi trường (class SnakeGameAI), một mạng nơ-ron đơn giản (class Linear\_QNet) để xấp xỉ hàm Q và một bộ huấn luyện (QTrainer) thực hiện cập nhật trọng số theo công thức Q-learning.

## Môi trường & mã nguồn liên quan

### Mã chính liên quan:

* SnakeGameAI (môi trường): cung cấp state, play\_step(action) → trả về (reward, done, score), có \_place\_food, is\_collision, \_move, \_update\_ui.
* Linear\_QNet (mạng): kiến trúc 2 lớp tuyến tính + ReLU.
* QTrainer: xử lý bước huấn luyện train\_step dùng MSELoss và optimizer Adam.

### Một số biến và mặc định xuất hiện trong project:

* STATE\_SIZE = 11 (số đặc trưng trạng thái)
* ACTION\_SIZE = 3 (straight, right, left)
* HIDDEN\_SIZE = 64 (số neuron ẩn)
* BATCH\_SIZE (ví dụ 1000 trong project bạn thấy trước đó)
* MAX\_MEMORY (ví dụ 100\_000)
* LR = 0.001, GAMMA (ví dụ 0.9)

## Biểu diễn state và action

**State (11 phần tử)**: mỗi phần tử là feature mô tả environment (ví dụ: danger straight/right/left, current direction, food location relative to head). (Cụ thể 11 feature có thể là những giá trị 0/1 hoặc số thực biểu diễn vị trí/độ nguy hiểm.)

**Action (3 trạng thái)**:

* [1, 0, 0] — đi thẳng
* [0, 1, 0] — rẽ phải
* [0, 0, 1] — rẽ trái

Sử dụng one-hot để dễ tương tác với hàm torch. argmax và để cập nhật phần tử tương ứng trong vector target.

## Kiến trúc mạng (Linear\_QNet)

### Mạng được định nghĩa như sau:

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 forward(self, x):  
 x = F.relu(self.linear1(x))  
 x = self.linear2(x)  
 return x  
  
 def save(self, file\_name='model.pth'):  
 model\_folder\_path = './model'  
 if not os.path.exists(model\_folder\_path):  
 os.makedirs(model\_folder\_path)  
  
 file\_name = os.path.join(model\_folder\_path, file\_name)  
 torch.save(self.state\_dict(), file\_name)  
  
 #load lai model.pth da train truoc do  
 def load(self, file\_name='model.pth'):  
 model\_folder\_path = './model'  
 file\_name = os.path.join(model\_folder\_path, file\_name)  
 if os.path.exists(file\_name):  
 self.load\_state\_dict(torch.load(file\_name))  
 self.eval()  
 print(f"Model loaded from {file\_name}")  
 else:  
 print("No saved model found, starting fresh.")

### Chức năng:

* **\_\_init\_\_**: Khởi tạo cấu trúc mạng gồm 2 lớp tuyến tính (linear1, linear2).
* **forward**: Xác định cách dữ liệu đi qua mạng (input → hidden → output).
* **save**: Lưu lại các tham số (weights, bias) của model vào file .pth.
* l**oad**: Nạp lại tham số đã lưu từ file để tiếp tục dùng hoặc huấn luyện.
* Input: vector shape [batch\_size, input\_size] .
* Hidden: Linear + ReLU (kích thước 64 là lựa chọn hợp lý cân bằng năng lực biểu diễn và chi phí).
* Output: [batch\_size, 3] — Q-values cho 3 hành động. Output là logits (không qua softmax).

## Thuật toán huấn luyện:Train\_step

Đây là hàm then chốt của QTrainer. Mục tiêu: cập nhật trọng số để pred ≈ target theo MSE.

### Thuật toán huấn luyện được định nghĩa như sau:

class QTrainer:  
 def \_\_init\_\_(self, model, lr, gamma):  
 self.lr = lr  
 self.gamma = gamma  
 self.model = model  
 self.optimizer = optim.Adam(model.parameters(), lr=self.lr)  
 self.criterion = nn.MSELoss()  
  
 def train\_step(self, state, action, reward, next\_state, done):  
 state = torch.tensor(state, dtype=torch.float)  
 next\_state = torch.tensor(next\_state, dtype=torch.float)  
 action = torch.tensor(action, dtype=torch.long)  
 reward = torch.tensor(reward, dtype=torch.float)  
  
 if len(state.shape) == 1:  
 state = torch.unsqueeze(state, 0)  
 next\_state = torch.unsqueeze(next\_state, 0)  
 action = torch.unsqueeze(action, 0)  
 reward = torch.unsqueeze(reward, 0)  
 done = (done, )  
 pred = self.model(state)  
  
 target = pred.clone()  
 for idx in range(len(done)):  
 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()

### Chức năng hàm:

* \_\_init\_\_
* Khởi tạo trainer với **mạng Q-Network, learning rate**, và **gamma**.
* Tạo bộ tối ưu **Adam** để cập nhật trọng số, cùng hàm mất mát **MSELoss** để so sánh Q-value dự đoán và mục tiêu.
* train\_step
* Nhận dữ liệu (state, action, reward, next\_state, done), chuyển thành Tensor và chuẩn hóa dạng batch.
* Tính **Q-value dự đoán** và **Q-value mục tiêu** (theo công thức Bellman).
* Tính loss, lan truyền ngược, và cập nhật trọng số mạng.

### ****Quy trình bên trong**** train\_step****:****

1. Convert inputs sang tensors:
   * state, next\_state → torch.float
   * action → torch.long (dễ dùng cho torch.argmax hoặc gather)
   * reward → torch.float
2. Chuẩn hoá batch dimension: nếu dữ liệu là single-sample (1D state), dùng unsqueeze(0) để chuyển sang shape [1, features]. Đồng thời done = (done,) để làm cho done iterable.
3. pred = self.model(state) → tensor shape [batch\_size, 3].
4. target = pred.clone() → tạo bản sao độc lập của pred để chỉnh sửa.
5. Với mỗi mẫu idx trong batch:
   * Q\_new = reward[idx]
   * Nếu not done[idx]: Q\_new = reward[idx] + gamma \* max(self.model(next\_state[idx])) (điểm max Q của next\_state theo model hiện tại)
   * Gán: target[idx][torch.argmax(action[idx]).item()] = Q\_new
6. Tính loss: loss = MSELoss(target, pred)
7. Backprop: optimizer.zero\_grad(); loss.backward(); optimizer.step()

### ****Giải thích logic:****

Ta chỉ thay đổi phần tử Q tương ứng với action đã thực hiện (off-policy update). Các phần tử Q khác giữ nguyên nhằm giữ thông tin mô hình đã học cho các hành động chưa chọn.

### Ví dụ tính toán thực tế

Giả sử (short memory):

* state → model trả pred = [[-0.5, 1.2, 0.3]] (shape [1,3])
* action = [0, 1, 0] (chọn index=1)
* reward = 10
* next\_state → model(next\_state) trả next\_pred = [[2.0, 0.5, 1.0]], max = 2.0
* gamma = 0.9

Tính Q\_new:

Q\_new = reward + gamma \* max(next\_pred) = 10 + 0.9\*2.0 = 11.8

Tạo target = pred.clone() → target = [[-0.5, 1.2, 0.3]]  
Gán: target[0][1] = 11.8 → target = [[-0.5, 11.8, 0.3]]

Loss (MSE) giữa pred và target = mean((−0.5+0.5)^2, (1.2−11.8)^2, (0.3−0.3)^2) = ((0)^2 + (−10.6)^2 + 0) / 3 ≈ 37.47

Sau loss.backward() và optimizer.step(), trọng số được cập nhật để giảm sai số này. Lưu ý: loss lớn do ví dụ minh hoạ; trong thực tế ổn định dần qua nhiều cập nhật.

## Replay buffer và chiến lược huấn luyện

**Replay buffer** (self.memory = deque(maxlen=MAX\_MEMORY)): lưu tuples (state, action, reward, next\_state, done) sau mỗi bước.

**train\_short\_memory**: huấn luyện ngay lập tức từ trải nghiệm mới (online update 1-sample). Giúp học nhanh từ sự kiện gần nhất.

**train\_long\_memory**: khi game kết thúc, lấy mini\_sample = random.sample(self.memory, BATCH\_SIZE) (nếu có đủ) hoặc toàn bộ memory, unpack batch và gọi trainer.train\_step(states, actions, rewards, next\_states, dones) để huấn luyện theo mini-batch. Việc lấy mẫu ngẫu nhiên phá vỡ correlation giữa các bước liên tiếp, giúp training ổn định hơn.

Kết hợp cả hai giúp cân bằng: cập nhật nhanh (short) + củng cố, tổng quát hóa (long).

**Chiến lược epsilon-greedy:** Xác suất chọn hành động ngẫu nhiên: ϵ=80−n\_games\epsilon = 80 - n\\_gamesϵ=80−n\_games.Khi số game tăng → epsilon giảm → Agent dần dần chuyển từ khám phá sang khai thác kiến thức đã học.

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) #11 phần tử  
 prediction = self.model(state0) #y= W.x+b ( y: vector output, W ma trận trọng số sinh ra ngẫu nhiên lúc đầu, x ma trận input, b là bias, ma trận sai số)  
 move = torch.argmax(prediction).item()  
 final\_move[move] = 1  
  
 return final\_move

## Cấu trúc lớp Agent

### Hàm khởi tạo \_\_init\_\_

* Khởi tạo các tham số chính:
  + n\_games: số lượng ván đã chơi.
  + epsilon: xác suất chọn hành động ngẫu nhiên (exploration).
  + gamma = 0.9: hệ số chiết khấu trong Q-learning, quyết định mức độ ưu tiên của phần thưởng tương lai.
  + memory: bộ nhớ kinh nghiệm dạng deque, lưu tối đa MAX\_MEMORY = 100.000 trải nghiệm, tự động loại bỏ phần cũ khi vượt giới hạn.
* Khởi tạo **mạng nơ-ron**: Linear\_QNet(11, 256, 3)
  + Input 11: số đặc trưng đầu vào (state).
  + Hidden layer: 256 neuron, giúp trích xuất đặc trưng từ state.
  + Output 3: số hành động có thể chọn (thẳng, trái, phải).
* Gọi model.load() để nạp model đã huấn luyện trước (nếu có).
* Tạo trainer QTrainer để thực hiện tối ưu hóa trọng số của mô hình.

Định nghĩa hàm như sau:

def \_\_init\_\_(self):  
 self.n\_games = 0  
 self.epsilon = 0   
 self.gamma = 0.9   
 self.memory = deque(maxlen=MAX\_MEMORY)   
 self.model = Linear\_QNet(11, 256, 3)  
 self.model.load()   
 self.trainer = QTrainer(self.model, lr=LR, gamma=self.gamma)

### Hàm get\_state(game)

* Nhiệm vụ: **chuyển đổi trạng thái của game thành vector số (state vector)**.
* Bao gồm 11 phần tử:  
  1–3: Nguy hiểm (collision) phía trước, bên phải, bên trái.  
  4–7: Hướng di chuyển hiện tại (trái, phải, lên, xuống).  
  8–11: Vị trí tương đối của thức ăn (thức ăn ở bên trái, phải, trên, dưới so với đầu rắn).
* Kết quả trả về: mảng numpy np.array(state, dtype=int) (giá trị 0 hoặc 1).

Định nghĩa hàm như sau:

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  
  
 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)),

(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)),

(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)),  
  
 dir\_l,  
 dir\_r,  
 dir\_u,  
 dir\_d,  
 game.food.x < game.head.x,   
 game.food.x > game.head.x,   
 game.food.y < game.head.y,   
 game.food.y > game.head.y   
 ]  
 return np.array(state, dtype=int)

### Hàm remember(state, action, reward, next\_state, done)

* Lưu một trải nghiệm (experience tuple) vào bộ nhớ:
* (state, action, reward, next\_state, done)
* Đây là dữ liệu dùng cho **experience replay** trong huấn luyện long memory.

Định nghĩa hàm như sau:

def remember(self, state, action, reward, next\_state, done):  
 self.memory.append((state, action, reward, next\_state, done))

### Hàm train\_short\_memory(state, action, reward, next\_state, done)

* Thực hiện huấn luyện ngay lập tức chỉ với **trải nghiệm vừa xảy ra**.
* Mục đích: cho phép AI điều chỉnh nhanh hành vi từ tình huống hiện tại.

Định nghĩa hàm như sau:

def train\_short\_memory(self, state, action, reward, next\_state, done):  
 self.trainer.train\_step(state, action, reward, next\_state, done)

### Hàm train\_long\_memory()

* Khi một ván game kết thúc, Agent sẽ **lấy ngẫu nhiên mini-batch** từ memory để huấn luyện.
* Nếu bộ nhớ lớn hơn BATCH\_SIZE (1000), chọn ngẫu nhiên 1000 trải nghiệm; nếu nhỏ hơn thì lấy toàn bộ.
* Lợi ích:
  + Giúp AI **học lại từ quá khứ** → tăng khả năng ghi nhớ lâu dài.
  + Tránh **correlation** (tương quan chặt chẽ giữa các mẫu liên tiếp) bằng cách trộn dữ liệu.
  + Giúp mô hình **tổng quát hóa tốt hơn** thay vì chỉ học từ các bước gần nhất.

Định nghĩa hàm như sau:

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)

### Hàm get\_action(state)

* Xác định hành động dựa trên **epsilon-greedy policy**:
  + Với xác suất epsilon → chọn hành động ngẫu nhiên (exploration).
  + Ngược lại → dùng model để tính toán Q-value cho từng hành động và chọn hành động có Q lớn nhất (exploitation).
* final\_move là vector one-hot có 3 phần tử:
  + [1,0,0]: đi thẳng.
  + [0,1,0]: rẽ phải.
  + [0,0,1]: rẽ trái.

Định nghĩa hàm như sau:

def get\_action(self, state):  
 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

## Vòng lặp huấn luyện chính – train()

### Các bước thực hiện:

1. Khởi tạo Agent và SnakeGameAI.
2. **Lặp vô hạn (while True):**
   * **Bước 1:** Lấy trạng thái hiện tại (state\_old).
   * **Bước 2:** Quyết định hành động (final\_move).
   * **Bước 3:** Thực hiện hành động → nhận phản hồi từ game:
     + reward: thưởng (+10 khi ăn, -10 khi chết, 0 khi đi bình thường).
     + done: trạng thái kết thúc ván game (True/False).
     + score: điểm số hiện tại.
   * **Bước 4:** Lấy trạng thái mới (state\_new).
   * **Bước 5:** Huấn luyện nhanh với dữ liệu mới (train\_short\_memory).
   * **Bước 6:** Lưu trải nghiệm vào bộ nhớ (remember).
3. Nếu game kết thúc (done=True):
   * Reset game để chơi lại.
   * Tăng số ván (n\_games += 1).
   * Huấn luyện dài hạn bằng train\_long\_memory().
   * Nếu đạt kỷ lục mới thì lưu mô hình.
   * Vẽ biểu đồ tiến trình học (plot\_scores, plot\_mean\_scores).

### Định nghĩa hàm:

def train():  
 plot\_scores = []  
 plot\_mean\_scores = []  
 total\_score = 0  
 record = 0  
 agent = Agent()  
 game = SnakeGameAI()  
 while True:  
 state\_old = agent.get\_state(game)  
 final\_move = agent.get\_action(state\_old)  
 reward, done, score = game.play\_step(final\_move)  
 state\_new = agent.get\_state(game)  
 agent.train\_short\_memory(state\_old, final\_move, reward, state\_new, done)  
 agent.remember(state\_old, final\_move, reward, state\_new, done)  
  
 if done:  
 game.reset()  
 agent.n\_games += 1  
 agent.train\_long\_memory()  
  
 if score > record:  
 record = score  
 agent.model.save()  
 print('Game', agent.n\_games, 'Score', score, 'Record:', record)  
 plot\_scores.append(score)  
 total\_score += score  
 mean\_score = total\_score / agent.n\_games  
 plot\_mean\_scores.append(mean\_score)  
 plot(plot\_scores, plot\_mean\_scores)

# Chương 4. Kết quả và thực nghiệm

## Kết quả

### Kết quả sau 400 lần chơi:

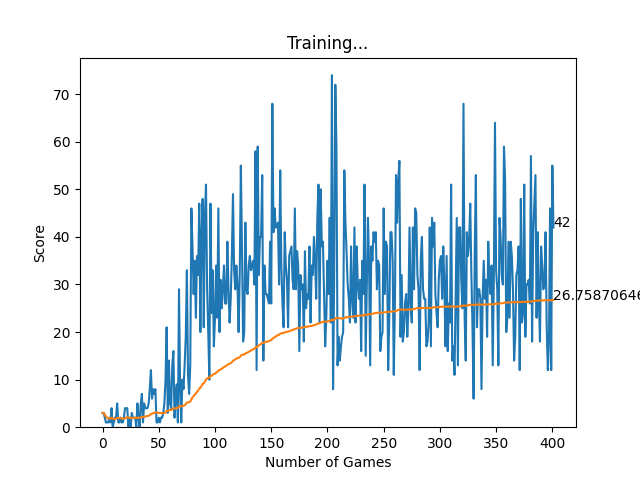


Figure 1 Đồ thị sau khi training 400 lần

### Kết quả sau 800 lần chơi:

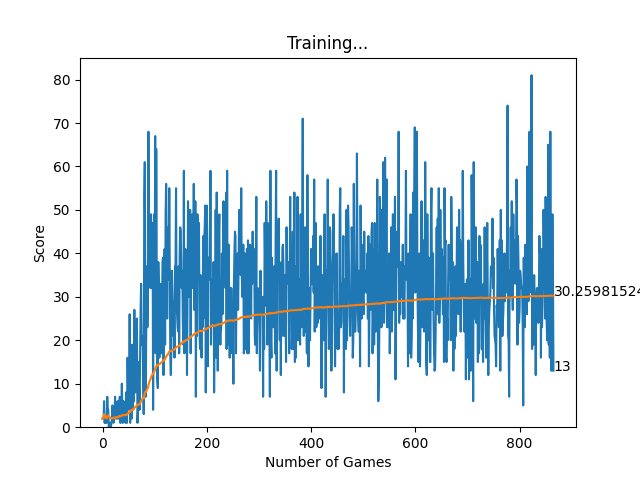


Figure 2 Đồ thị sau khi training 800 lần

### Nhận xét

* **Số điểm theo từng ván chơi (Score – đường xanh dương):**
* Trong khoảng đầu (0–50 game), điểm số còn thấp, gần như bằng 0, điều này phản ánh Agent đang **khám phá môi trường** và chưa học được chiến lược tốt.
* Sau khoảng 50 ván, điểm số bắt đầu tăng lên đáng kể, cho thấy Agent **bắt đầu học cách tránh va chạm và ăn thức ăn** hiệu quả hơn.
* Có nhiều **biến động lớn** trong điểm số, với một số ván đạt tới 60–70 điểm, nhưng cũng có những ván chỉ đạt 10–20 điểm. Điều này xảy ra do tính ngẫu nhiên trong lựa chọn hành động (epsilon-greedy) và môi trường thay đổi.
* **Số lần chết / game kết thúc:**
* Khi số game tăng, Agent có xu hướng **sống lâu hơn** (score trung bình tăng) nhưng vẫn chưa hoàn toàn ổn định.
* **Điểm trung bình (Mean score – đường cam):**
* Đường trung bình tăng dần từ gần 0 lên khoảng **26.7**, phản ánh Agent **đang cải thiện kỹ năng và ổn định hơn theo thời gian**.
* Tuy nhiên, sau khoảng 250–400 game, đường trung bình gần như **bão hòa**, nghĩa là Agent **đã học được chiến lược cơ bản nhưng chưa tối ưu**.

# Chương 5. Kết luận

## ****Nhận xét tổng quan:****

1. Agent đã **học được cách chơi game con rắn ở mức cơ bản**.
2. Kỹ năng ổn định hơn khi chơi, số điểm trung bình tăng, nhưng vẫn còn **biến động lớn** trong từng ván.
3. Đường trung bình chưa đạt cực đại, nghĩa là **Agent chưa hoàn toàn tối ưu** và vẫn có thể cải thiện.

## Hạn chế hiện tại & hướng cải tiến

1. **Không có target network**: hiện tại dùng cùng mạng để tính max(self.model(next\_state)) → dễ gây oscillation. Giải pháp: dùng **target network.**
2. **Prioritized Experience Replay (PER)**: thay random sampling bằng sampling theo lỗi (TD-error) sẽ tập trung học vào experience quan trọng.
3. **Double DQN**: tránh overestimation bias bằng cách tách hành động chọn và hành động đánh giá.
4. **Dueling DQN**: tách giá trị state và advantage cho hành động.
5. **Normalization**: chuẩn hoá input state/feature để ổn định training.
6. **Gradient clipping / learning rate scheduling**: tránh gradient explosion và điều chỉnh tốc độ học.
7. **Tăng cỡ batch / batch balancing**: cân nhắc batch size phù hợp.
8. **Mô hình phức tạp hơn**: nếu state phức tạp (hình ảnh), dùng CNN; nếu chuỗi thời gian, dùng RNN.

# Tài liệu tham khảo

1. Le, V.T., Kim, Y.G., 2023. **Attention-based residual autoencoder for video anomaly detection**. Applied Intelligence 53, 3240–3254.
2. Italo Lelis. LearnSnake: Teaching an AI to play Snake using Reinforcement Learning (Q-Learning) ([LearnSnake: Teaching an AI to play Snake using Reinforcement Learning (Q-Learning) - Italo Lelis - Software Developer](https://italolelis.com/snake))
3. Youtube-( [(8) Python + PyTorch + Pygame Reinforcement Learning – Train an AI to Play Snake - YouTube](https://www.youtube.com/watch?v=L8ypSXwyBds))
4. **Thomas Jiang, Claire Mai – Stanford University-SnakeAI\_report (**[103085287.pdf](https://cs230.stanford.edu/projects_fall_2021/reports/103085287.pdf))**.**