Tuyệt vời. Tôi sẽ phân tích chi tiết cách thuật toán của bạn sử dụng Deep Q-Network để lựa chọn giữa SGD và Adam như sau:

1. **Cách DQN chọn giữa SGD và Adam**
2. **Vai trò của trạng thái (state) và phần thưởng (reward)** trong việc huấn luyện DQN để tối ưu lựa chọn optimizer.
3. **Mục tiêu tổng thể**: làm sao để chọn optimizer giúp tăng độ chính xác (accuracy), giảm mất mát (loss), giữ gradient ổn định và tối ưu thời gian huấn luyện.
4. Tôi cũng sẽ chỉ rõ **nơi định nghĩa SGD và Adam** được import hoặc khởi tạo trong code (từ file common.py).

Tôi sẽ quay lại với phân tích chi tiết ngay sau khi hoàn tất.

# Phân tích cơ chế DQN trong train.py

## 1. Cơ chế thuật toán DQN trong lựa chọn optimizer

DQN được sử dụng để học giá trị kỳ vọng Q(s,a) của mỗi optimizer (hành động) tại mỗi trạng thái. Cụ thể, agent chọn giữa hai optimizer (SGD hoặc Adam) theo **chính sách ε-greedy**: với xác suất ε sẽ khám phá (chọn ngẫu nhiên SGD hoặc Adam), và với xác suất 1-ε chọn optimizer có giá trị Q cao nhất trong trạng thái hiện tại. Sau khi chọn optimizer và thực hiện một bước huấn luyện (ví dụ một epoch), ta thu được phần thưởng (reward) và trạng thái mới. Mỗi kinh nghiệm (s,a,r,s') được lưu vào **bộ nhớ trải nghiệm (replay buffer)** để huỷ bỏ sự tương quan thời gian của dữ liệu. Hệ thống sẽ lấy mẫu ngẫu nhiên các minibatch từ bộ nhớ này để cập nhật mạng Q chính (Q-network) thông qua phương pháp giảm gradient trên sai số Bellman y - Q(s,a). Đồng thời, một **mạng mục tiêu (target network)** được sử dụng để ổn định việc huấn luyện: mỗi C bước, ta đồng bộ hoá các tham số của mạng mục tiêu với mạng Q chính. Những cơ chế replay buffer và target network là thành phần chính giúp DQN ổn định học tập.

## 2. Ý nghĩa các thành phần trong vector trạng thái

Mỗi trạng thái được biểu diễn bằng một vector gồm các phần tử số liệu sau:

* **loss\_train**: Giá trị loss trên tập huấn luyện (training loss) tại epoch hiện tại. Cho biết mức độ lỗi tổng thể của mô hình.
* **acc\_val**: Độ chính xác (accuracy) trên tập kiểm định (validation). Đây là thước đo hiệu năng của mô hình trên dữ liệu chưa thấy và được dùng để so sánh tác động của từng optimizer.
* **grad\_norm**: Độ lớn chuẩn (norm) của gradient hiện tại (ví dụ chuẩn L2 của gradient). Chỉ báo mức độ thay đổi của các tham số – nếu grad\_norm lớn, quá trình huấn luyện có thể đang học nhanh hoặc không ổn định.
* **delta\_loss**: Biến động của loss giữa các bước hoặc epoch (chênh lệch loss so với bước trước đó). Dấu hiệu thể hiện mức cải thiện (hoặc không) của quá trình huấn luyện.
* **epoch\_ratio**: Tỉ lệ phần trăm của epoch đã chạy so với tổng số epoch. Ví dụ, nếu đang ở epoch 50/100 thì epoch\_ratio = 0.5. Giúp agent nhận biết giai đoạn huấn luyện (đầu, giữa, cuối).
* **action (optimizer ID)**: Chỉ số optimizer đã chọn ở bước trước (ví dụ 0=SGD, 1=Adam). Gồm vào trạng thái để mô hình Q network biết được lựa chọn tối ưu hóa trước đây, tạo tính Markov hoàn chỉnh cho bài toán.

## 3. Tính toán reward và ảnh hưởng đến DQN

Reward được thiết kế để khuyến khích việc cải thiện hiệu năng (accuracy) đồng thời phạt các hành vi kém hiệu quả. Cụ thể:

* Nếu chọn optimizer mà **cải thiện đáng kể acc\_val** so với trước đó, agent nhận được **reward dương** (khuyến khích hành động tốt). Ngược lại, nếu acc\_val chỉ tăng rất ít hoặc không tăng, agent bị **phạt (reward âm)** để báo hiệu tiến triển kém. Việc này tương tự với reward shaping: khen thưởng khi agent tiến gần mục tiêu và phạt khi không tiến triển.
* Nếu quá trình huấn luyện kéo dài nhiều epoch (tốn thời gian), agent có thể bị **phạt thêm** (giảm reward) vì tiêu tốn tài nguyên. Thiết kế này khuyến khích agent tìm optimizer vừa cải thiện nhanh hiệu năng vừa tiết kiệm thời gian. Nói chung, hàm reward định hướng mạnh mẽ hành vi của agent – phần thưởng càng hợp lý, agent càng học nhanh và chính xác hơn.

## 4. Học để ưu tiên chọn optimizer

Trong quá trình học, mạng Q ước lượng giá trị tương lai của từng optimizer ở mỗi trạng thái. Khi mạng Q hội tụ, agent sẽ ưu tiên chọn optimizer có giá trị Q lớn hơn (greedy): a = argmax\_a Q(s,a). Điều này có nghĩa là agent sẽ tự điều chỉnh theo kinh nghiệm thực tế: ví dụ, ở giai đoạn đầu nếu Adam thường cải thiện accuracy nhiều hơn thì Q(s,Adam) cao hơn Q(s,SGD), nên agent ưu tiên chọn Adam. Ở giai đoạn sau, nếu SGD mang lại cải thiện tốt hơn, Q(s,SGD) tăng lên, dẫn đến agent chuyển sang ưu tiên chọn SGD. Tóm lại, hệ thống sẽ học được chính sách chọn optimizer sinh ra phần thưởng lớn hơn trong từng trạng thái khác nhau dựa trên cập nhật giá trị Q.

## 5. Định nghĩa và khởi tạo SGD/Adam trong mã nguồn

Trong file common.py, hai optimizer SGD và Adam thường được **import** từ thư viện chuẩn của PyTorch và lưu trong danh sách optimizers. Ví dụ, mã có thể trông như sau:

from torch.optim import SGD, Adam  
optimizers = [SGD, Adam]

Khi cần tạo optimizer, ta chỉ việc lấy lớp optimizer tương ứng từ danh sách này và khởi tạo với tham số của mô hình. Ví dụ trong PyTorch có cú pháp minh họa như: optim.SGD(model.parameters(), lr=0.01, momentum=0.9) hoặc optim.Adam(model.parameters(), lr=0.001). Danh sách optimizers vì thế chứa sẵn hai lớp tối ưu (SGD và Adam), cho phép truy xuất nhanh qua chỉ số action và khởi tạo đối tượng optimizer phù hợp theo trạng thái được chọn.

**Tài liệu tham khảo:** Mô hình DQN sử dụng replay buffer và target network để ổn định học; chính sách ε-greedy cân bằng khám phá/khai thác; reward shaping cung cấp tín hiệu học nhanh và chính xác hơn; và cách khởi tạo optimizer trong PyTorch.