ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH

TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN

KHOA HỆ THỐNG THÔNG TIN

BÁO CÁO ĐỒ ÁN MÔN HỌC

HỆ HỖ TRỢ QUYẾT ĐỊNH

ĐỀ TÀI:

ACTOR-CRITIC METHOD FOR DECISION MAKING

GVHD: ThS. Nguyễn Hồ Duy Trí

Nhóm sinh viên thực hiện:

La Hoài Nam	MSSV: 20521629
Lê Trần Anh Quí	MSSV: 21520094
Trần Lê Tứ	MSSV: 21522746
Vương Thanh Linh	MSSV: 21521082

TP. Hồ Chí Minh, tháng 10 năm 2024

LỜI CẢM ƠN

Trước hết, nhóm chúng em xin gửi lời cảm ơn sâu sắc đến tập thể quý thầy cô trường Đại học Công nghệ Thông tin - Đại học Quốc gia TP.HCM và quý thầy cô khoa Hệ thống thông tin đã tạo điều kiện, giúp chúng em học tập và có được những kiến thức cơ bản làm tiền đề giúp chúng em hoàn thành được dự án này. Đặc biệt, nhóm chúng em xin gửi lời cảm ơn chân thành và sâu sắc tới Thầy Nguyễn Hồ Duy Trí. Nhờ sự hướng dẫn tận tình và chu đáo của thầy, nhóm chúng em đã học hỏi được nhiều kinh nghiệm và hoàn thành thuận lợi, đúng tiến độ cho dự án của mình.

Ngoài ra, chúng em cũng gửi lời cảm ơn đến tập thể lớp IS254.P11 khoảng thời gian qua đã đồng hành cùng nhau. Cảm ơn sự đóng góp của tất cả các bạn cho những buổi học luôn sôi nổi, thú vị và dễ tiếp thu. Trong quá trình thực hiện đồ án, nhóm chúng em luôn giữ một tinh thần cầu tiến, học hỏi và cải thiện từ những sai lầm, tham khảo từ nhiều nguồn tài liệu khác nhau và luôn mong tạo ra được sản phẩm chất lượng nhất có thể.

Tuy nhiên, do vốn kiến thức còn hạn chế trong quá trình trau dồi từng ngày, nhóm chúng em không thể tránh được những sai sót, vì vậy chúng em mong rằng quý thầy cô sẽ đưa ra nhận xét một cách chân thành để chúng em học hỏi thêm kinh nghiệm nhằm mục đích phục vụ tốt các dự án khác trong tương lai. Xin chân thành cảm ơn quý thầy cô!

NHẬN XÉT CỦA GIẢNG VIÊN

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

……, ngày…….tháng……năm 2024

Người nhận xét

(Ký tên và ghi rõ họ tên)

DANH MỤC HÌNH ẢNH

MỤC LỤC

Chương 1. GIỚI THIỆU TỔNG QUAN

1.1. Lý do chọn đề tài

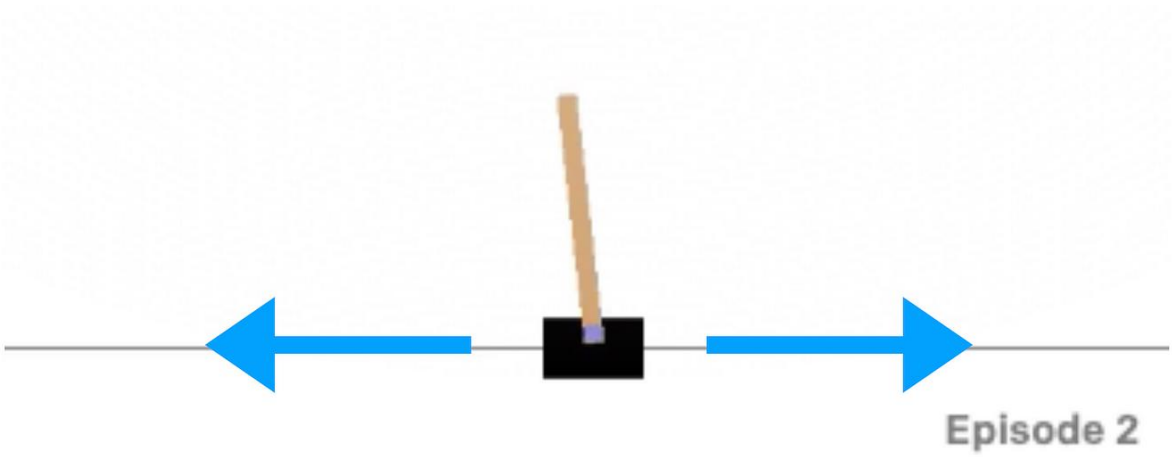
- Trong thời đại công nghệ phát triển mạnh mẽ hiện nay, học tăng cường (Reinforcement Learning - RL) đang đóng vai trò ngày càng quan trọng trong lĩnh vực trí tuệ nhân tạo và khoa học máy tính. Đặc biệt, với khả năng cho phép các agent học cách đưa ra quyết định tối ưu thông qua tương tác với môi trường, RL đã và đang được ứng dụng rộng rãi trong nhiều lĩnh vực từ robotics, điều khiển tự động đến các hệ thống ra quyết định phức tạp.
- Trong số các thuật toán học tăng cường, Actor-Critic nổi bật như một phương pháp tiên tiến kết hợp được ưu điểm của cả hai hướng tiếp cận chính trong RL: học dựa trên giá trị (Value-based learning) và học dựa trên chính sách (Policy-based learning). Với kiến trúc độc đáo gồm hai thành phần Actor và Critic hoạt động đồng thời, thuật toán không chỉ có khả năng học các chính sách liên tục mà còn giảm thiểu được độ biến thiên trong quá trình học, từ đó cải thiện đáng kể hiệu quả và tốc độ hội tụ của quá trình huấn luyện.
- Hơn nữa, tính linh hoạt và khả năng thích ứng cao của Actor-Critic đã được chứng minh qua nhiều ứng dụng thực tế thành công. Từ việc điều khiển robot trong môi trường phức tạp, tối ưu hóa hệ thống năng lượng, cho đến phát triển AI trong game, thuật toán đã thể hiện tiềm năng to lớn trong việc giải quyết các bài toán ra quyết định phức tạp. Đây chính là lý do nhóm lựa chọn nghiên cứu sâu về Actor-Critic, nhằm tìm hiểu rõ hơn về cơ chế hoạt động, ưu nhược điểm cũng như các khả năng ứng dụng của thuật toán này trong thực tiễn.

1.2. Mô tả bài toán

Bài toán CartPole-v1 là một bài toán trong lĩnh vực Reinforcement Learning (RL), trong đó nhiệm vụ của agent (tác nhân) là điều khiển một xe trượt (cart) trên một đường thẳng để giữ một cây gậy (pole) đứng thăng bằng. Mục tiêu là giữ cây gậy

không bị đổ càng lâu càng tốt bằng cách quyết định di chuyển xe trượt sang trái hoặc phải.

- Trạng thái: Được biểu diễn bằng một vector gồm 4 giá trị: vị trí của xe, vận tốc của xe, góc nghiêng của cột, và vận tốc góc.
- Hành động: Có hai lựa chọn: di chuyển xe sang trái hoặc sang phải.
- Phần thưởng: Mỗi bước thời gian mà cột không bị đổ sẽ được cộng thêm một điểm.
- Kết thúc khi:
    - Cây gậy nghiêng quá 15 độ so với trục thẳng đứng.
    - Xe trượt di chuyển ra ngoài biên giới hạn của đường thẳng.
    - Hoặc sau 500 bước (được xem là thành công).



Hình 1. Mô tả bài toán CartPole-V1

1.3. Mô tả dữ liệu sử dụng

Dữ liệu trong CartPole-v1 không phải là một tập dữ liệu cố định mà là các trạng thái và hành động sinh ra trong quá trình tương tác với môi trường. Các thông số chính bao gồm:

- State (trạng thái):
  - x: Vị trí của xe.
  - x_dot: Vận tốc của xe.
  - theta: Góc nghiêng của cột so với trục thẳng đứng.
  - theta_dot: Vận tốc góc của cột.
- Action (hành động):
  - 0: Di chuyển xe sang trái.
  - 1: Di chuyển xe sang phải.
- Reward: Điểm thưởng tại mỗi bước, thường là 1 nếu cột vẫn còn thăng bằng.

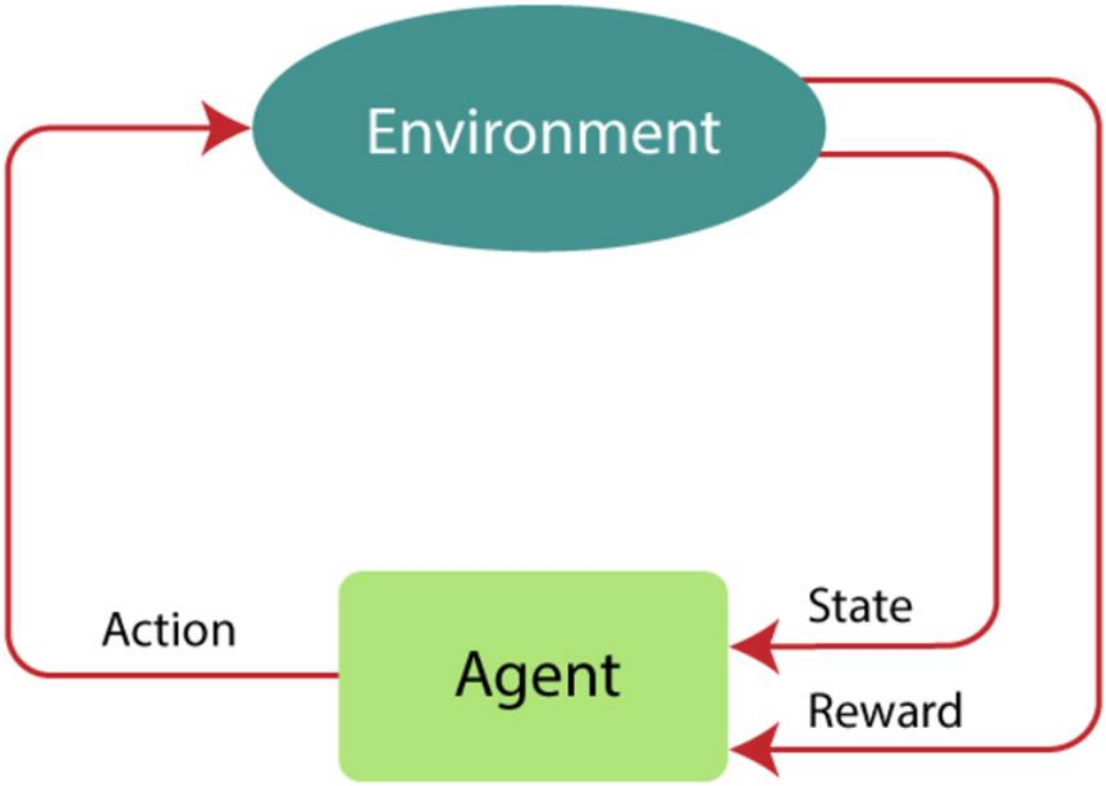
Chương 2. CƠ SỞ LÝ THUYẾT

2.1. Khái niệm học tăng cường

Học tăng cường (**Reinforcement Learning – RL**) là một nhánh của học máy, nghiên cứu cách thức một tác nhân (agent) học cách tương tác với môi trường (enviroment) đang ở một trạng thái (state) thực hiện một hành động (action) và nhận phản hồi dưới dạng phần thưởng (reward) hoặc hình phạt (penalty). Mục tiêu của tác nhân là tối ưu hóa tổng phần thưởng nhận được theo thời gian bằng cách chọn những hành động tốt nhất trong từng tình huống. Gồm 6 thành phần cơ bản:

- Tác nhân (Agent): Thực thể tương tác với môi trường và đưa ra quyết định.
- Môi trường (Environment): Hệ thống bên ngoài hoặc thế giới mà tác nhân hoạt động trong đó. Môi trường cung cấp phản hồi dựa trên các hành động của tác nhân.
- Hành động (Action – A): Tập các hành động của tác nhân.
- Trạng thái (State – S): Tình trạng hiện tại của tác nhân trong môi trường.
- Phần thưởng (Reward – R): Đối với mỗi hành động được chọn bởi tác nhân, môi trường sẽ đưa ra một phần thưởng. Phần thưởng có giá trị dương, âm hoặc bằng không. Tác nhân hướng đến việc tối đa hóa phần thưởng này.
- Chính sách (Policy – π): Chiến lược (ra quyết định) mà tác nhân sử dụng để phản ứng trước môi trường giúp đạt được mục tiêu là tối đa hóa phần thưởng.
- Hàm giá trị (Value Function): Hàm ước tính phần thưởng tích lũy dự kiến từ một trạng thái nhất định, giúp tác nhân dự đoán giá trị dài hạn của các hành động.

- Quy trình hoạt động:



Hình 2. Quy trình hoạt động của các thuật toán Reinforcement Learning

- Tác nhân thực hiện hành động (A) trong trạng thái (S) nhất định của môi trường.
- Môi trường phản hồi bằng phần thưởng (R) và chuyển sang trạng thái mới (S').
- Tác nhân sử dụng phản hồi này để cập nhật chiến lược (π) của mình, dần dần cải thiện khả năng ra quyết định bằng cách tối đa hóa phần thưởng đạt được tương lai.

2.2. Thuật toán Actor-Critic

2.2.1. Giới thiệu về học chính sách (Policy Learning) và học hàm giá trị (Value Function Learning)

2.2.1.1. Học chính sách (Policy Learning)

Học chính sách tập trung vào việc tìm ra một chính sách tối ưu $\pi(a|s)$, tức là một hàm xác định xác suất chọn hành động a khi ở trạng thái s. Mục tiêu là tối ưu hóa chính sách này để tối đa hóa phần thưởng kỳ vọng dài hạn.

Ví dụ: Giả sử bạn đang huấn luyện một robot để tìm đường trong một mê cung. Chính sách của robot là một tập hợp các quy tắc xác định hành động tiếp theo (đi lên, xuống, trái, phải) dựa trên vị trí hiện tại trong mê cung. Robot sẽ học chính sách này bằng cách thử nghiệm các hành động khác nhau và điều chỉnh chính sách dựa trên phần thưởng nhận được (ví dụ, đến gần lối ra sẽ nhận được phần thưởng cao hơn).

2.2.1.2. Học hàm giá trị (Value Function Learning)

Học hàm giá trị tập trung vào việc ước lượng giá trị của các trạng thái hoặc cặp trạng thái-hành động.

Hàm giá trị trạng thái V(s) ước lượng giá trị kỳ vọng của trạng thái s. Điều này có nghĩa là nó tính toán tổng phần thưởng mà bạn mong đợi nhận được khi bắt đầu từ trạng thái s và tiếp tục theo chính sách hiện tại π.

Ví dụ: Nếu robot đang ở trạng thái s (một ô cụ thể trong mê cung), hàm giá trị trạng thái V(s) sẽ ước lượng tổng phần thưởng mà robot mong đợi nhận được từ ô đó cho đến khi thoát khỏi mê cung, dựa trên chính sách hiện tại của nó.

- Nếu robot ở gần lối ra, V(s) sẽ cao vì robot mong đợi nhận được phần thưởng lớn (thoát khỏi mê cung) trong thời gian ngắn.
- Nếu robot ở xa lối ra, V(s) sẽ thấp hơn vì robot cần nhiều bước hơn để đến lối ra và nhận phần thưởng.

Hàm giá trị hành động Q(s,a) ước lượng giá trị kỳ vọng của việc thực hiện hành động a tại trạng thái s, sau đó tiếp tục theo chính sách hiện tại π. Điều này có nghĩa là nó tính toán tổng phần thưởng mà bạn mong đợi nhận được khi thực hiện hành động a tại trạng thái s và sau đó tiếp tục theo chính sách π.

Ví dụ: Nếu robot đang ở trạng thái s (một ô cụ thể trong mê cung) và có thể thực hiện hành động a (ví dụ, di chuyển lên), hàm giá trị hành động Q(s,a) sẽ ước lượng

tổng phần thưởng mà robot mong đợi nhận được từ ô đó khi thực hiện hành động "di chuyển lên" và sau đó tiếp tục theo chính sách hiện tại.

- Nếu hành động "di chuyển lên" từ ô đó dẫn đến một ô gần lối ra, Q(s,a) sẽ cao vì robot mong đợi nhận được phần thưởng lớn (thoát khỏi mê cung) trong thời gian ngắn.
- Nếu hành động "di chuyển lên" từ ô đó dẫn đến một ngõ cụt hoặc xa lối ra, Q(s,a) sẽ thấp hơn vì robot cần nhiều bước hơn để đến lối ra và nhận phần thưởng.

2.2.2. Thuật toán Actor-Critic

Thuật toán Actor-Critic là một phương pháp trong lĩnh vực học tăng cường (Reinforcement Learning) kết hợp giữa học chính sách (Policy Learning) và học hàm giá trị (Value Function Learning). Hai thành phần này được huấn luyện song song, cho phép chúng tương tác và cải thiện lẫn nhau trong quá trình học.

2.2.2.1. Khái niệm về Actor

Actor là thành phần trong thuật toán, thường được biểu diễn bằng một mạng neural, để học và tối ưu hóa chính sách $\pi\theta$. Nói cách khác, Actor là cách mà chính sách $\pi\theta$ được triển khai. Khi được huấn luyện, Actor sẽ cập nhật tham số θ sao cho chính sách $\pi\theta$ có thể chọn hành động tối ưu.

Chính sách $\pi\theta$ là một hàm toán học hoặc mô hình xác suất định nghĩa cách lựa chọn hành động a dựa trên trạng thái s. Biểu thức toán học của chính sách:

$$\pi\theta(a|s) = P(a|s;\theta)$$

Nó là xác suất (hoặc mật độ xác suất) của hành động a khi ở trạng thái s, tham số hóa bởi θ. Tham số θ là các giá trị số mà chúng ta điều chỉnh trong một mô hình hoặc hàm số để tối ưu hóa hiệu suất của mô hình đó. Trong ngữ cảnh của mạng nơ-ron hoặc các mô hình học máy, tham số θ thường bao gồm các trọng số (weights) và

độ lệch (biases) của mạng. Trong thực tế, chính sách πθ(a|s) trong thuật toán Actor-Critic thường được thiết kế dưới dạng một mạng nơ-ron với tham số hóa θ, có thể sử dụng hàm softmax cho bài toán hành động rời rạc hoặc phân phối Gaussian cho hành động liên tục.

Mục tiêu của Actor là tối đa hóa giá trị kỳ vọng U(θ), tức là cải thiện chính sách sao cho phần thưởng kỳ vọng là cao nhất. Hàm giá trị kỳ vọng U(θ) là một hàm mục tiêu đại diện cho phần thưởng mà Actor có thể kỳ vọng nhận được từ việc thực hiện các hành động theo chính sách πθ. Để đơn giản, có thể nghĩ rằng:

$$U(\theta) = E[R] = E\left[\sum_{t=0}^{T} \gamma^t r_t\right]$$

- R: Là tổng phần thưởng nhận được từ môi trường khi thực hiện các hành động theo chính sách πθ.
- γ: Là yếu tố giảm dần (discount factor), điều chỉnh trọng số của phần thưởng trong tương lai.
- r_t: Phần thưởng nhận được tại thời điểm t.
- T: Thời gian kết thúc (hoặc số bước trong một episode)

Tuy nhiên, việc tính trực tiếp giá trị này qua toàn bộ các bước thời gian t là không khả thi trong thực tế khi làm việc với các hệ thống phức tạp hoặc không gian trạng thái lớn.

2.2.2.2. Khái niệm về Critic

Critic (Value Network) là thành phần đảm nhận vai trò đánh giá chất lượng của các hành động do Actor thực hiện. Cụ thể, Critic ước lượng các đại lượng quan trọng sau, được tham số hóa bởi ϕ:

- **Hàm giá trị trạng thái ($V_\phi(s)$):** Đại diện cho giá trị kỳ cọng khi thực hiện hành động a tại trạng thái s và tiếp tục theo chính sách hiện tại.
- **Hàm giá trị hành động ($Q_\phi(s,a)$):** Đại diện cho giá trị kỳ vọng khi thực hiện hành động a tại trạng thái s và tiếp tục theo chính sách hiện tại.
- **Hàm lợi thế ($A_\phi(s,a)$):** Đo lường lợi ích tương đối của hành động a tại trạng thái s so với giá trị trung bình của trạng thái s. Cụ thể:

$$A_\phi(s,a) = Q_\phi(s,a) - V_\phi(s)$$

Critic sử dụng các giá trị ước lượng này để đánh giá hành động đã chọn bởi Actor, xác định hành động nào tốt hơn so với kỳ vọng. Thông tin này được cung cấp dưới dạng phản hồi để Actor điều chỉnh các tham số θ, từ đó cải thiện chính sách $\pi\theta$ nhằm chọn các hành động mang lại phần thưởng cao hơn.

2.2.2.3. Mối quan hệ giữa Actor và Critic

Actor phụ thuộc vào Critic: Critic đánh giá các hành động được Actor thực hiện. Dựa trên đánh giá của Critic, Actor sẽ điều chỉnh chính sách πθ để cải thiện xác suất chọn các hành động tốt hơn trong tương lai.

Critic phụ thuộc vào Actor: Critic cần các hành động từ Actor để đánh giá và học cách ước lượng giá trị trạng thái hoặc hành động.

2.3. Cơ chế hoạt động

2.3.1. Khái quát ban đầu

Tham số chính sách θ: Đây là các tham số của mô hình chính sách (Actor) mà chúng ta muốn tối ưu hóa. Ví dụ, nếu chính sách được biểu diễn bằng một mạng neural, thì θ là các trọng số và bias của mạng này.

Tham số ϕ của Critic: Đây là các tham số của mô hình giá trị (Critic) mà chúng ta muốn tối ưu hóa. Tương tự, nếu Critic được biểu diễn bằng một mạng neural, thì ϕ là các trọng số và bias của mạng này

Gradient là một vector chứa các đạo hàm riêng phần của một hàm số đối với các biến số của nó. Trong học máy, gradient cho biết hướng và độ lớn của sự thay đổi cần thiết để tối ưu hóa hàm mục tiêu. Ví dụ với gradient ∇_θ Uϕ(s(k)) biểu thị đạo hàm bậc nhất của một hàm theo tham số ϕ - mô tả sự thay đổi của hàm Uϕ khi ϕ thay đổi.

Rollout trajectories là quá trình thực hiện các hành động theo chính sách hiện tại từ trạng thái ban đầu đến trạng thái kết thúc, ghi lại các trạng thái, hành động và phần thưởng nhận được. Quá trình này giúp thu thập dữ liệu để ước lượng các giá trị và cập nhật mô hình.

Các bước hoạt động của thuật toán:

- **Bước 1:** Khởi tạo với các tham số ban đầu cho Actor (θ) và Critic (ϕ).
- **Bước 2:** Thực hiện các rollout trajectories từ trạng thái ban đầu, sử dụng chính sách hiện tại πθ để thu thập dữ liệu (các trạng thái s, hành động a, và phần thưởng r).
- **Bước 3:** Critic ước lượng giá trị của trạng thái hiện tại s và trạng thái mới s′. Bằng cách cập nhật tham số ϕ theo hướng gradient descent để giảm thiểu hàm mất mát: $\phi \leftarrow \phi - \beta\nabla\ell(\phi)$.
- **Bước 4:** Actor sử dụng thông tin từ Critic để cập nhật chính sách. Bằng cách sử dụng gradient của hàm mục tiêu, thường là dựa trên hàm lợi ích hoặc hàm giá trị hành động để cập nhật tham số θ theo hướng gradient ascent: $\theta \leftarrow \theta + \alpha\nabla U(\theta)$.
- **Bước 5:** Lặp lại các bước trên cho đến khi hội tụ (Các tham số không thay đổi nhiều giữa các lần cập nhật, cho thấy mô hình đã đạt được hiệu suất tối ưu) hoặc

đạt được số lần lặp tối đa (Đạt đến số lần lặp đã định trước, nhằm tránh việc chạy quá lâu mà không đạt được cải thiện đáng kể).

2.3.2. Các công thức quan trọng

2.3.2.1. Cập nhật Actor

Quá trình cập nhật actor được thực hiện thông qua gradient ascent. Mục tiêu là tối đa hóa hàm giá trị kỳ vọng U(θ) với công thức:

$$\nabla U(\theta) = E_{\tau}\left[\sum_{k=1}^{d} \nabla_{\theta} \log \pi_{\theta}\left(a^{(k)}\middle|s^{(k)}\right)\gamma^{k-1}A_{\theta}\left(s^{(k)}, a^{(k)}\right)\right]$$

Trong đó:

- **$\nabla U(\theta)$**: Gradient của hàm mục tiêu đối với tham số chính sách θ.
- **$E\tau$**: Kỳ vọng trên các quỹ đạo (rollout trajectories) được thu thập từ môi trường. (tính giá trị trung bình)
- **$\sum k=1 d$**: Tổng từ bước 1 đến bước d trong một trajectory.
- **$\nabla_{\theta} \log \pi_{\theta}\left(a^{(k)}\middle|s^{(k)}\right)$** : Gradient của log xác suất chọn hành động a(k) tại trạng thái s(k) theo chính sách πθ..
- **γ**: Hệ số chiết khấu tại bước k−1.
- **$A_{\theta}(s, a)$:** Hàm ước lượng lợi ích (advantage) khi thực hiện hành động a tại trạng thái s. Lợi thế dương có nghĩa là hành động tốt hơn kỳ vọng và cần được Actor ưu tiên, trong khi lợi thế âm cho thấy hành động không tốt và cần bị giảm trọng số.

$A\theta(s, a)$ có thể được tính theo công thức:

$$A_{\theta}(s, a) = E_{r,s'}\left[r + \gamma U_{\pi_{\theta}}(s') - U_{\pi_{\theta}}(s)\right]$$

- **$E_{r,s'}$**: Kỳ vọng trên phần thưởng r và trạng thái tiếp theo s′.
- **r**: Phần thưởng nhận được sau khi thực hiện hành động a tại trạng thái s.

- $\boldsymbol{U_{\pi_\theta}(s')}$**:** Giá trị kỳ vọng của trạng thái s′ theo chính sách πθ
- $\boldsymbol{U_{\pi_\theta}(s)}$**:** Giá trị kỳ vọng của trạng thái s theo chính sách πθ.
- $\boldsymbol{r + \gamma U_{\pi_\theta}(s') - U_{\pi_\theta}(s)}$: được gọi là temporal difference residual, cho phép chúng ta đánh giá hiệu suất của hành động đã thực hiện.

Tuy nhiên, trong thực tế, để giảm phương sai và cập nhật chính sách ổn định hơn, gradient của hàm mục tiêu cho Actor thường được ước lượng thông qua Critic, giúp đánh giá giá trị trạng thái và hành động một cách hiệu quả. Công thức như sau:

$$\nabla U(\theta) \approx \mathbb{E}_\tau \left[\sum_{k=1}^{d} \nabla_\theta log \pi_\theta (a^{(k)} | s^{(k)}) \gamma^{k-1} \left(r^{(k)} + \gamma U_\Phi \left(s^{(k+1)} \right) - U_\Phi \left(s^{(k)} \right) \right) \right]$$

- $\boldsymbol{r^{(k)}}$: Phần thưởng nhận được tại bước k.
- $\boldsymbol{U_\Phi\left(s^{(k+1)}\right)}$: Giá trị ước lượng của trạng thái $\boldsymbol{s^{(k+1)}}$ bởi Critic.
- $\boldsymbol{U_\Phi\left(s^{(k)}\right)}$: Giá trị ước lượng của trạng thái $\boldsymbol{s^{(k)}}$ bởi Critic.

2.3.2.2. Cập nhật Critic

Critic cũng được cập nhật thông qua tối ưu hóa gradient. Mục tiêu là tìm ϕ sao cho giảm thiểu hàm mất mát:

$$\mathrm{l}(\phi) = \frac{1}{2} E_s \left[\left(U_\phi(s) - U_{\pi_\theta}(s) \right)^2 \right]$$

- $\boldsymbol{\mathrm{l}(\phi)}$: Hàm mất mát cho Critic, đo lường sự khác biệt giữa giá trị ước lượng của Critic và giá trị thực tế của chính sách.
- $\boldsymbol{E_s}$: Kỳ vọng trên các trạng thái s.
- $\boldsymbol{U_\phi(s)}$**:** Giá trị ước lượng của trạng thái s bởi Critic với tham số ϕ.

Vì chúng ta không biết chính xác U_{π_θ}, nên có thể ước lượng nó bằng cách sử dụng phần thưởng còn lại dọc theo các quỹ đạo, dẫn đến:

$$\nabla \mathrm{l}(\phi) = E_\tau \left[\sum_{k=1}^{d} \left(U_\phi\left(s^{(k)}\right) - r_{to-go}^{(k)} \right) \nabla_\phi \, U_\phi\left(s^{(k)}\right) \right]$$

- $\boldsymbol{r_{to-go}^{(k)}}$ là phần thưởng từ bước k trở đi trong một quỹ đạo.
- $\boldsymbol{\nabla_\phi U_\phi\left(s^{(k)}\right)}$: Gradient của giá trị ước lượng $U_\phi\left(s^{(k)}\right)$ đối với tham số ϕ. Được tính theo công thức sau:

$$\nabla_\phi U_\phi\left(s(k)\right) = \frac{\partial U_\phi\left(s(k)\right)}{\partial \phi}$$

$\frac{\partial U_\phi(s(k))}{\partial \phi}$:là đạo hàm riêng của $U_\phi\left(s(k)\right)$ theo ϕ

2.4. Ví dụ minh họa thuật toán

Giả sử chúng ta có một robot di chuyển trong một mê cung. Chúng ta sẽ sử dụng thuật toán Actor-Critic để huấn luyện robot này. Các tham số ban đầu của Actor và Critic được khởi tạo ngẫu nhiên

- Khởi tạo ngẫu nhiên:
  - Actor: $\theta = [0.5, -0.3]$
  - Critic: $\phi = [0.2, 0.4]$
- Dữ liệu thu thập được: Giả sử robot thực hiện một hành động và thu thập được dữ liệu sau:
  - Trạng thái hiện tại: s=1
  - Hành động: a=2
  - Phần thưởng nhận được: r=1
  - Trạng thái mới: s′=2
- Cập nhật Critic:
  - Giả định hàm Uϕ(s) là một hàm tuyến tính đơn giản: $U_\phi(s) = \phi_1 s + \phi_2 s$

- Ước lượng giá trị của trạng thái hiện tại và trạng thái mới:
  - $U_\phi(s) = 0.2 \cdot 1 + 0.4 \cdot 1 = 0.6$
  - $U_\phi(s') = 0.2 \cdot 2 + 0.4 \cdot 2 = 1.2$
- Tính toán sai số (TD error): $\delta = r + \gamma U_\phi(s') - U_\phi(s)$

  Giả sử γ=0.9: $\delta = 1 + 0.9 \cdot 1.2 - 0.6 = 1.48$
- Cập nhật tham số của Critic: $\phi \leftarrow \phi - \beta\delta\nabla_\phi U_\phi(s)$

  Giả sử β=0.01 và $\nabla\phi U\phi(s)=[1,1]$:

$$\phi = [0.2,0.4] - 0.01 \cdot 1.48 \cdot [1,1] = [0.2,0.4] - [0.0148,0.0148]$$
$$= [0.1852,0.3852]$$

- Cập nhật Actor:
  - Giả định rằng gradient của log chính sách đã được tính toán trước: $\nabla_\theta \log \pi_\theta(a \mid s) = [0.1, -0.1]$
  - Cập nhật tham số của Actor: $\theta \leftarrow \theta + \alpha\nabla_\theta \log \pi_\theta(a \mid s)\,\delta$

    Giả sử α=0.01:

$$\theta = [0.5, -0.3] + 0.01 \cdot 1.48 \cdot [0.1, -0.1]$$
$$= [0.5, -0.3] + [0.00148, -0.00148] = [0.50148, -0.30148]$$

- Lặp lại đến khi tối ưu.
- Cuối cùng, robot sẽ có thể di chuyển qua mê cung một cách hiệu quả nhất, tối ưu hóa phần thưởng nhận được trong quá trình di chuyển.

2.5. Các phương pháp trong Actor-Critic

2.5.1. Generalized Advantage Estimation (GAE)

2.5.1.1. Tổng quan

Generalized Advantage Estimation (GAE) là một phương pháp trong học tăng cường (reinforcement learning), được phát triển để cải thiện ước lượng lợi ích (advantage) trong các thuật toán actor-critic. GAE không chỉ đơn thuần ước lượng

lợi ích tại một thời điểm mà còn kết hợp thông tin từ nhiều bước thời gian, giúp cân bằng giữa độ thiên lệch (bias) và phương sai (variance) trong ước lượng.

2.5.1.2. So sánh với Actor Critic cơ bản

Phương pháp Generalized Advantage Estimation (GAE) khác với phương pháp actor-critic cơ bản ở một số điểm chính sau đây:

- Ước Lượng Advantage
    - Actor-Critic Cơ Bản: Sử dụng một ước lượng đơn giản cho advantage, thường là dựa vào giá trị temporal difference residual. Điều này có thể dẫn đến bias cao nhưng variance thấp.
    - GAE: Kết hợp nhiều bước ước lượng advantage thông qua một tham số λ, cho phép điều chỉnh giữa bias và variance. Điều này giúp tạo ra các ước lượng advantage chính xác hơn và ổn định hơn.
- Tính đến nhiều bước
    - Actor-Critic Cơ Bản: Thường chỉ xem xét 1 bước trong quá trình cập nhật, dẫn đến việc sử dụng thông tin hạn chế từ các bước trước đó.
    - GAE: Tính toán advantage dựa trên nhiều bước trong một chuỗi rollout, giúp cải thiện khả năng học tập và hiệu suất tổng thể.
- Tham số điều chỉnh
    - Actor-Critic Cơ Bản: Không có tham số điều chỉnh cho bias và variance.
    - GAE: Tham số λ cho phép điều chỉnh mức độ ảnh hưởng của các ước lượng khác nhau, từ đó tối ưu hóa hiệu suất giữa bias và variance.
- Khả năng ổn định
    - Actor-Critic Cơ Bản: Thường có thể gặp phải vấn đề về độ ổn định trong quá trình học do sự tương quan giữa các cập nhật của actor và critic.

- GAE: Cung cấp một cách tiếp cận ổn định hơn nhờ vào việc kết hợp các ước lượng từ nhiều bước, giúp giảm thiểu sự biến động.

2.5.1.3. Các công thức

- Ước lượng hàm lợi ích tổng quát

$$\boldsymbol{A_\theta(s,a) = \mathbb{E}_{r_1,\dots,r_d}[r_1 + \gamma r_2 + \gamma^2 r_3 + \cdots + \gamma^{d-1} r_d - U^{\pi_\theta}(s)]}$$
$$\boldsymbol{= \mathbb{E}_{r_1,\dots,r_d}\left[-U^{\pi_\theta}(s) + \sum_{\ell=1}^{d} \gamma^{\ell-1} r_\ell\right]}$$

Trong đó:

+ $A_\theta(s,a)$: Đo lường giá trị lợi ích của hành động a trong trạng thái s so với việc thực hiện hành động trung bình

+ $\mathbb{E}_{r_1,\dots,r_d}$: Kỳ vọng tính toán giá trị trung bình của một chuỗi các phần thưởng $r_1, \dots, r_d$ mà một tác nhân nhận được sau khi thực hiện hành động a trong trạng thái s

+ γ: Hệ số chiết khấu nằm trong khoảng [0, 1] và dùng để giảm giá trị của các phần thưởng trong tương lai, nhằm phản ánh rằng phần thưởng gần hơn có giá trị hơn phần thưởng xa hơn

+ $U^{\pi_\theta}(s)$: Hàm giá trị cho trạng thái s. Thể hiện giá trị mà tác nhân mong đợi nhận được từ trạng thái đó nếu không thực hiện hành động nào

- **Ý nghĩa:** Đây là một phiên bản tổng quát của hàm lợi ích cơ bản, đo lường sự khác biệt giữa tổng phần thưởng tích lũy theo các rollout $r_1, r_2, \dots, r_d$ (có tính đến hệ số chiết khấu γ) và giá trị kỳ vọng của trạng thái ban đầu $U_{\pi\theta}(s)$.
- **Sử dụng:**
  - Cho phép cân bằng giữa sai số (bias) và phương sai (variance) trong quá trình huấn luyện.

- Giúp đánh giá hiệu quả của hành động a tại trạng thái s, với các phần thưởng được lấy từ các quỹ đạo rollout thực tế.
- Phương pháp này có thể đạt được một ước lượng không sai số (unbiased) thông qua các quỹ đạo rollout.
- Tuy nhiên, ước lượng có phương sai cao, do đó cần nhiều mẫu để đạt được độ chính xác cao.

- Phần dư sai số thời gian (Temporal Difference Residual):

$$\delta_t = r_t + \gamma U(s_{t+1}) - U(s_t)$$

Với st, rt, và st+1 lần lượt là trạng thái, phần thưởng và trạng thái tiếp theo trong một quỹ đạo đã được mẫu.

  - **Ý nghĩa**: Phần dư sai số thời gian đo lường sự khác biệt giữa phần thưởng nhận được cộng với giá trị kỳ vọng của trạng thái tiếp theo và giá trị kỳ vọng của trạng thái hiện tại.
  - **Sử dụng**: Giúp cập nhật hàm giá trị U.
- Ước lượng hàm lợi ích từ k bước rollout:

$$\hat{A}^{(k)}(s,a) = \mathbb{E}_{r_1,\dots,r_k,s'}[r_1 + \gamma r_2 + \cdots + \gamma^{k-1} r_k + \gamma^k U^{\pi_\theta}(s') - U^{\pi_\theta}(s)]$$

$$= \mathbb{E}_{r_1,\dots,r_k,s'}\left[-U^{\pi_\theta}(s) + \gamma^k U^{\pi_\theta}(s') + \sum_{\ell=1}^{k} \gamma^{\ell-1} r_\ell\right]$$

  - **Ý nghĩa**: Công thức này ước lượng hàm lợi ích từ k bước rollout và giá trị kỳ vọng của trạng thái kết quả.
  - **Sử dụng**: Cân bằng giữa độ chệch và phương sai bằng cách sử dụng một số bước rollout cố định.
- Ước lượng hàm lợi ích tổng quát với phần dư sai số thời gian:

$$\hat{A}^{(k)}(s,a) = \mathbb{E}\left[\sum_{\ell=1}^{k} \gamma^{\ell-1} \delta_\ell\right]$$

- **Ý nghĩa**: Công thức này biểu diễn ước lượng hàm lợi ích tổng quát dưới dạng tổng của các phần dư sai số thời gian.
- **Sử dụng:** Giúp giảm phương sai của ước lượng bằng cách sử dụng thông tin từ nhiều bước thời gian.

- Ước lượng hàm lợi ích tổng quát với trọng số mũ λ ∈ [0,1]:

$$\hat{A}_{GAE}(s,a) = (1-\lambda)\left(\hat{A}^{(1)} + \lambda\hat{A}^{(2)} + \lambda^2\hat{A}^{(3)} + \cdots\right)$$

Khi λ=0: GAE trở thành ước lượng dựa trên residual tạm thời với độ thiên lệch cao và phương sai thấp.

Khi λ=1: GAE chuyển thành ước lượng dựa trên toàn bộ quỹ đạo với độ thiên lệch thấp nhưng phương sai cao hơn.

- **Ý nghĩa**: Công thức này sử dụng trọng số mũ λ để kết hợp các ước lượng hàm lợi ích từ nhiều bước rollout khác nhau.
- **Sử dụng**: Cân bằng giữa độ chệch và phương sai bằng cách điều chỉnh giá trị λ.

2.5.1.4. Cách hoạt động của GAE

- **Bước 1:** Thực hiện các hành động theo chính sách hiện tại πθ và thu thập các trạng thái, hành động, và phần thưởng.
- **Bước 2:** Sử dụng công thức **phần dư sai số thời gian** để tính toán phần dư sai số thời gian cho mỗi bước.
- **Bước 3:** Sử dụng công thức $\hat{A}^{(k)}(s,a)$ để ước lượng hàm lợi ích từ k bước rollout.
- **Bước 4:** Sử dụng công thức $\hat{A}_{GAE}(s,a)$ để kết hợp các ước lượng hàm lợi ích từ nhiều bước rollout với trọng số mũ λ.
- **Bước 5:** Sử dụng ước lượng hàm lợi ích tổng quát để cập nhật tham số của Actor và Critic.

2.5.1.5. Ví dụ

Giả sử chúng ta có một robot di chuyển trong một mê cung. Chúng ta sẽ sử dụng GAE để cải thiện hiệu quả của thuật toán Actor-Critic.

- **Bước 1: Thu thập dữ liệu**
  - Robot di chuyển và ghi lại các trạng thái, hành động, phần thưởng, và trạng thái mới.
- **Bước 2: Tính toán phần dư sai số thời gian**
  - Trạng thái hiện tại: s=1
  - Hành động: a=2
  - Phần thưởng nhận được: r=1
  - Trạng thái mới: s′=2
  - Giá trị kỳ vọng của trạng thái hiện tại: U(s)=0.6
  - Giá trị kỳ vọng của trạng thái mới: U(s′)=1.2
  - Phần dư sai số thời gian: δ=1+0.9·1.2−0.6=1.48
- **Bước 3: Ước lượng hàm lợi ích từ các bước rollout**
  - k=1: $\hat{A}^{(1)}(s,a) = \delta = 1.48$
  - k=2: $\hat{A}^{(2)}(s,a) = \delta_1 + \gamma\delta_2 = 1.48 + 0.9 \cdot 1.48 = 2.812$
- **Bước 4: Kết hợp các ước lượng hàm lợi ích**
  - λ=0.5: $\hat{A}_{GAE}(s,a)$ =(1−0.5)(1.48+0.5·2.812)=1.852
- **Bước 5: Cập nhật Actor và Critic**
  - Sử dụng $\hat{A}_{GAE}(s,a)$ =1.852 để cập nhật tham số của Actor và Critic.

2.5.2. Deterministic Policy Gradients (DPG)

2.5.2.1. Tổng quan

Phương pháp Deterministic Policy Gradient (DPG) là một kỹ thuật trong học tăng cường (reinforcement learning) được thiết kế để tối ưu hóa các chính sách xác định (deterministic policies) trong các bài toán có không gian hành động liên tục.

DPG tập trung vào việc tối ưu hóa một chính sách xác định $\pi\theta(s)$ mà không cần sử dụng phân phối xác suất cho hành động, giúp cải thiện tốc độ hội tụ và tính ổn định.

DPG sử dụng một critic có dạng hàm giá trị hành động $Q\phi(s,a)$ với mục tiêu là tìm ϕ để tối đa hóa hàm mục tiêu cho Actor:

$$U(\theta) = Es \sim b\gamma, \theta[Q\phi(s, \pi\theta(s))]$$

Trong đó $b\gamma,\theta$ là phân phối trạng thái với hệ số chiết khấu khi theo chính sách $\pi\theta$.

2.5.2.2. So sánh với thuật toán Actor Critic cơ bản

Phương pháp Deterministic Policy Gradient (DPG) khác với phương pháp actor-critic cơ bản ở một số điểm chính sau đây:

- Loại Chính Sách
  - Actor-Critic Cơ Bản: Thường sử dụng chính sách ngẫu nhiên (stochastic policy), nơi hành động được chọn dựa trên phân phối xác suất.
  - DPG: Sử dụng chính sách xác định (deterministic policy), trong đó hành động được quyết định trực tiếp từ đầu vào mà không có ngẫu nhiên. Điều này giúp cải thiện hiệu suất trong các bài toán có không gian hành động liên tục.
  - Ví dụ: Giả sử bạn đang ở trong một nhà hàng và bạn phải chọn một món ăn từ thực đơn. Bạn có thể chọn món ăn dựa trên hai cách tiếp cận: chính sách xác suất và chính sách xác định.
  - Chính sách xác suất (Stochastic Policy)
    - Trong chính sách xác suất, bạn sẽ chọn món ăn dựa trên một phân phối xác suất. Ví dụ, bạn có thể có các xác suất sau cho các món ăn:
    - Món A: 40%
    - Món B: 30%
    - Món C: 20%
    - Món D: 10%

        - Biểu thức toán học của chính sách này là: $\pi\theta(a|s)$
        - Trong đó, $\pi\theta(a|s)$ là xác suất chọn món ăn a khi ở trạng thái s, tham số hóa bởi θ. Ở đây, θ có thể đại diện cho một mô hình học máy nào đó, chẳng hạn như một mạng neuron, dùng để học các xác suất lựa chọn món ăn dựa trên các yếu tố của trạng thái s (có thể là loại nhà hàng, món ăn yêu thích, thời gian trong ngày, v.v.)
    - Chính sách xác định (Deterministic Policy)
        - Trong chính sách xác định, bạn sẽ luôn chọn một món ăn cụ thể tại mỗi lần đến nhà hàng mà không có yếu tố ngẫu nhiên. Ví dụ, bạn có thể luôn chọn món A: $a=\pi\theta(s)$
        - Trong đó, a là món ăn bạn chọn, và nó được xác định hoàn toàn bởi chính sách $\pi\theta(s)$ tại trạng thái s. Điều này có nghĩa là tại mỗi trạng thái s, bạn luôn chọn món ăn cố định.
- Gradient Tính Toán
    - Actor-Critic Cơ Bản: Cập nhật chính sách dựa trên gradient của giá trị hàm V hoặc Q, thường sử dụng phương pháp gradient ascent dựa trên các giá trị ước lượng từ critic.
    - DPG: Tính toán gradient của chính sách thông qua giá trị Q, sử dụng công thức gradient của Q để tối ưu hóa chính sách. Điều này cho phép DPG tận dụng thông tin từ critic một cách hiệu quả hơn.
- Cách Cập Nhật
    - Actor-Critic Cơ Bản: Cập nhật cả actor và critic đồng thời, thường thực hiện cập nhật theo cách song song.
    - DPG: Tập trung vào việc tối ưu hóa chính sách trước, sau đó cập nhật giá trị hàm Q. Quy trình này có thể giúp cải thiện độ ổn định và hiệu suất.
- Khám Phá
    - Actor-Critic Cơ Bản: Khám phá hành động thường dựa vào tính ngẫu nhiên của chính sách, có thể không hiệu quả trong một số trường hợp.

- DPG: Thường sử dụng noise (chẳng hạn như Gaussian noise) để khám phá, giúp cải thiện khả năng tìm kiếm trong không gian hành động liên tục.

- Ứng Dụng
  - Actor-Critic Cơ Bản: Thích hợp cho các bài toán có không gian hành động rời rạc.
  - DPG: Đặc biệt hiệu quả cho các bài toán điều khiển với không gian hành động liên tục, như trong robot hoặc các hệ thống điều khiển phức tạp.

2.5.2.3. Một số công thức được sử dụng trong DPG

- Định nghĩa hàm mất mát: Để tối ưu hóa chính sách và hàm giá trị hành động, chúng ta định nghĩa hàm mất mát cho giá trị hành động như sau:

$$\ell(\phi) = \frac{1}{2} E_{s,a,r,s'} \left[\left(r + \gamma Q_\phi\left(s', \pi_\theta(s')\right) - Q_\phi(s,a) \right)^2 \right]$$

  - ϕ: Tham số của hàm giá trị hành động $Q_\phi(s,a)$, cần tối ưu.
  - s: Trạng thái hiện tại,
  - a: Hành động hiện tại, được thực hiện bởi Actor $\pi_\theta(s)$
  - r: Phần thưởng từ môi trường khi thực hiện hành động a tại trạng thái s.
  - s': Trạng thái tiếp theo sau hành động a.
  - γ: Yếu tố giảm dần (discount factor), điều chỉnh tầm quan trọng của phần thưởng tương lai.
  - $Q_\phi\left(s', \pi_\theta(s')\right)$: Giá trị dự đoán của hành động ở trạng thái tiếp theo s', hành động theo Actor π_θ.
  - $Q_\phi(s,a)$: Giá trị dự đoán của hành động a ở trạng thái hiện tại s.

Hàm mất mát này cố gắng giảm thiểu độ sai lệch giữa giá trị ước lượng của hàm giá trị hành động và giá trị thực tế từ hệ thống.

- Tính Gradient của hàm mất mát: Để cập nhật các tham số ϕ của hàm giá trị hành động Qϕ, chúng ta cần tính gradient của hàm mất mát.

$$\nabla_\phi \ell(\phi) = E_{s,a,r,s'}\left[r + \gamma Q_\phi\left(s', \pi_\theta(s')\right) - Q_\phi(s,a)\right]\left[\gamma \nabla_\phi Q_\phi\left(s', \pi_\theta(s')\right) - \nabla_\phi Q_\phi(s,a)\right]$$

Gradient này cho phép chúng ta cập nhật tham số ϕ theo hướng tối ưu hóa hàm giá trị hành động.

- Tối Ưu Hàm Mục tiêu

Để tối ưu hóa chính sách, chúng ta cần tìm giá trị của θ sao cho hàm giá trị kỳ vọng U(θ) được tối đa hóa:

$$U(\theta) = E_{s\sim b}\left[Q_\phi\left(s, \pi_\theta(s)\right)\right]$$

Ở đây, b là phân phối trạng thái đầu vào. Để tối ưu hóa θ, chúng ta sử dụng gradient ascent:

$$\nabla U(\theta) = E_s\left[\nabla_\theta Q_\phi\left(s, \pi_\theta(s)\right)\right]$$

Gradient này cho biết cách mà giá trị hành động Qϕ thay đổi khi điều chỉnh tham số của chính sách πθ

- Tính Gradient của Hàm mục tiêu

Gradient của hàm mục tiêu cần được tính toán để cải thiện chính sách:

$$\nabla U(\theta) = E_s\left[\nabla_\theta \pi_\theta(s) \nabla_a Q_\phi(s,a)|_{a=\pi_\theta(s)}\right]$$

Trong đó, ∇θπθ(s) là ma trận Jacobian, thể hiện sự thay đổi của hành động a khi thay đổi tham số θ tại trạng thái s

Trong đó, $\nabla_a Q_\phi(s,a)$ là gradient của hàm giá trị Q_ϕ với hành động a, mô tả cách Q_ϕ, thay đổi khi thay đổi hành động.

Giải thích ∇θπθ(s) là ma trận Jacobian:

$$\nabla_\theta \pi_\theta(s) = \begin{bmatrix} \frac{\partial \pi_i}{\partial \theta_j} & \cdots & \frac{\partial \pi_1}{\partial \theta_j} \\ \vdots & \ddots & \vdots \\ \frac{\partial \pi_m}{\partial \theta_1} & \cdots & \frac{\partial \pi_m}{\partial \theta_n} \end{bmatrix}$$

- $\pi_\theta(s)$: Đầu ra của Actor, là một hành động a.
- Ma trận Jacobian:
  - Hàng i: Biểu diễn gradient của hành động π_i (thành phần thứ i của hành động) với từng tham số θ_j.
  - Cột j: Biểu diễn anhr hưởng của θ_j lên các thành phần π_i của hành động.
  - Đây là ma trận mxn, với m là số chiều của hành động a và n là tham số θ.

2.5.2.4. Cách hoạt động của DPG

DPG hoạt động tương tự với Actor Critic cơ bản.

2.5.2.5. Ví dụ

Một cánh tay robot cần di chuyển đầu tay (end-effector) đến một vị trí mục tiêu (xtarget,ytarget) trên một mặt phẳng.

Không gian trạng thái (s): Góc quay của các khớp s=[θ1,θ2] (hai góc khớp của robot).

Không gian hành động (a): Tốc độ góc tại mỗi khớp a=[θ˙1,θ˙2].

Mục tiêu: Giảm khoảng cách giữa đầu tay và mục tiêu trong không gian 2D.

Các thành phần:

- Actor ($\pi_\theta(s)$):

- Đầu vào: trạng thái $s = [\theta_1, \theta_2]$.
- Đầu ra: hành động $a = [\dot{\theta}_1, \dot{\theta}_2]$.
- Chính sách Actor được mô hình hóa bởi mạng nơ-ron với tham số θ.

- Critic ($Q(s, a, \phi)$):
  - Đầu vào: trạng thái s và hành động a.
  - Đầu ra: giá trị hành động $Q(s, a)$, thể hiện chất lượng của hành động a tại trạng thái s.

- Hàm thưởng:

$$r(s, a) = -\|End - efector\ position - (x_{target}, y_{target})\|$$

  - Đây là khoảng cách Euclidean từ đầu tay robot đến mục tiêu, với giá trị âm để khuyến khích cánh tay tiến gần mục tiêu.

- Thông số khởi tạo:
  - Trạng thái khởi đầu:

$$s_0 = [\theta_1 = 0.5, \theta_2 = 0.2]$$

  - Hành động từ Actor:

$$a = \pi_\theta(s_0) = [0.2, 0.38]$$

  - Trạng thái tiếp theo:

$$s_1 = s_0 + a \cdot \Delta t = [0.52, 0.238]$$

  - Hàm Critic: $Q_\phi(s, a) = \phi_1 s_1 a_1 + \phi_2 s_2 a_2$
  - Tham số Critic khởi tạo: $\phi = [0.5, 0.5]$
  - Hệ số giảm discount: $\gamma = 0.9$

- Hàm thưởng:

$$r = -\sqrt{(x_{eff} - x_{target})^2 + (y_{eff} - y_{target})^2}$$

Giả sử $(x_{eff} - y_{eff}) = (0.8,0.6), (x_{target} - y_{target}) = (1.0,1.0)$

$$r = -\sqrt{(0.8 - 1.0)^2 + (0.6 - 1.0)^2} = -0.447$$

- **Gradient Critic**
  - Gradient của hàm mất mát Critic:

$$\nabla_\phi \ell(\phi) = \left[r + \gamma Q_\phi\left(s_1, \pi_\theta(s_1)\right) - Q_\phi(s_0, a)\right]\left[\gamma \nabla_\phi Q_\phi\left(s_1, \pi_\theta(s_1)\right) - \nabla_\phi Q_\phi(s_0, a)\right]$$

  - Gradient Critic tại s_1:

$$\nabla_\phi Q_\phi\left(s_1, \pi_\theta(s_1)\right) = [s_1^1 \cdot a_1, s_1^2 \cdot a_2] = [0.52 \cdot 0.2, 0.238 \cdot 0.38] = [0.104, 0.09044]$$

  - Gradient Critic tại s_0:

$$\nabla_\phi Q_\phi(s_0, a) = [s_0^1 \cdot a_1, s_0^2 \cdot a_2] = [0.5 \cdot 0.2, 0.2 \cdot 0.38] = [0.1, 0.076]$$

  - Sai số Critic:

$$\delta = r + \gamma Q_\phi\left(s_1, \pi_\theta(s_1)\right) - Q_\phi(s_0, a)$$

Với $Q_\phi(s_0, a) = 0.5 \cdot 0.5 \cdot 0.2 + 0.5 \cdot 0.2 \cdot 0.38 = 0.05 + 0.038 = 0.088$:

$$\delta = -0.447 + 0.9 \,.\, 0.09722 - 0.088 = -0.447 + 0.087498 - 0.088 = -0.4475$$

  - Gradient Critic:

$$\nabla_\phi \ell(\phi) = -0.4475 \,.\, (0.9 \,.\, [0.104, 0.09044] - [0.1, 0.076])$$

$$\nabla_\phi \ell(\phi) = -0.4475 \,.\, ([0.0936, 0.081396] - [0.1, 0.076])$$

$$\nabla_\phi \ell(\phi) = -0.4475 \cdot [-0.0064, 0.005396] = [0.0028656, -0.0024133]$$

- **Cập nhật Critic**
  - Cập nhật tham số Critic:

$$\phi \leftarrow \phi - a\nabla_{\phi}\ell(\phi)$$

Với $a = 0.01$:

$$\phi = [0.5,0.5] - 0.01.[0.0028656, -0.0024133]$$

$$\phi = [0.49997134, 0.50002413]$$

- **Cập nhật Actor**

Actor được cập nhật để tối đa hóa giá trị hành động kỳ vọng. Gradient của Actor được tính như sau:

$$\nabla_{\phi}U \approx E_{s\sim D}\left[\nabla_a Q_{\phi}(s,a)|_{a-\pi_{\theta}(s)} . \nabla_{\theta}\pi_{\theta}(s)\right]$$

  - Tính toán từng thành phần
    - Gradient của Critic theo hành động:

$$\nabla_a Q_{\phi}(s,a) = [\phi_1 s_1, \phi_2 s_2]$$

    - Tại trạng thái $s_0 = [0.5,0.2]$:

$$\nabla_a Q_{\phi}(s_0, \pi_{\theta}(s_0)) = [\phi_1 . 0.5, \phi_2 . 0.2]$$

    - Thay $\phi = [0.49997134, 0.50002413]$:

$$\nabla_a Q_{\phi}(s_0, \pi_{\theta}(s_0)) = [0.49997134 . 0.5, 0.50002413 .0.2]$$
$$= [0.24998567, 0.10000483]$$

    - Jacobian của Actor:

$$\pi_{\theta}(s) = [\dot{\theta}_1 + \dot{\theta}_2 s_1, \dot{\theta}_3 + \dot{\theta}_4 s_2]$$

    - Gradient theo tham số θ là:

$$\nabla_\theta \pi_\theta(s) = \begin{bmatrix} 1 & s_1 & 0 & 0 \\ 0 & 0 & 1 & s_2 \end{bmatrix}$$

- Tại $s_0 = [0.5, 0.2]$:

$$\nabla_\theta \pi_\theta(s) = \begin{bmatrix} 1 & 0.5 & 0 & 0 \\ 0 & 0 & 1 & 0.2 \end{bmatrix}$$

- Gradient của Actor:

$$\nabla_\phi U = \nabla_a Q_\phi(s_0, \pi_\theta(s_0)).\nabla_\theta \pi_\theta(s_0)$$

Thay số:

$$\nabla_a Q_\phi(s_0, \pi_\theta(s_0)) = [0.24998567, 0.10000483]$$

$$\nabla_\theta \pi_\theta(s_0) = \begin{bmatrix} 1 & 0.5 & 0 & 0 \\ 0 & 0 & 1 & 0.2 \end{bmatrix}$$

- Tính tích:

$$\nabla_\theta U = [0.24998567, 0.10000483].\begin{bmatrix} 1 & 0.5 & 0 & 0 \\ 0 & 0 & 1 & 0.2 \end{bmatrix}$$
$$= [0.24998567, 0.12499283, 0.10000483, 0.02000097]$$

- Cập nhật tham số Actor

Actor được cập nhật bằng gradient ascent:

$$\theta \leftarrow \theta - a\nabla_\theta U$$

Với $a = 0.01$và tham số ban đầu $\theta = [0.1, 0.2, 0.3, 0.4]$:

$$\theta = [0.1, 0.2, 0.3, 0.4] + 0.01$$
$$- [0.24998567, 0.12499283, 0.10000483, 0.02000097]$$

$$\theta = [0.10249986, 0.20124993, 0.30100005, 0.40020001]$$

- **Lặp lại các bước**

- **Rollouts**: Actor thực hiện hành động mới dựa trên πθ(s) với tham số đã cập nhật.
- **Cập nhật Critic**: Tính giá trị mục tiêu y, gradient Critic, và cập nhật tham số ϕ.
- **Cập nhật Actor**: Tính gradient Actor dựa trên Critic và cập nhật θ.

2.5.3. Monte Carlo Tree Search (MCTS)

2.5.3.1. Tổng quan

Phương pháp **Actor-Critic** với **Monte Carlo Tree Search (MCTS)** là một kỹ thuật kết hợp giữa học tăng cường và quy hoạch trực tuyến, nhằm tối ưu hóa các chính sách trong các bài toán quyết định Markov (MDP)

Trong ngữ cảnh của phương pháp này, chính sách $\pi\theta(a|s)$ là một hàm xác suất quyết định hành động a dựa trên trạng thái s. Hàm giá trị $U\phi(s)$ thể hiện giá trị kỳ vọng của trạng thái s theo chính sách $\pi\theta$

MCTS được sử dụng để đưa ra các hành động tối ưu bằng cách xây dựng một cây tìm kiếm, trong đó mỗi nút đại diện cho một trạng thái và mỗi nhánh đại diện cho một hành động.

2.5.3.2. Giải thích phương pháp MCTS

Ý tưởng chính: MCTS tìm kiếm hành động tốt nhất bằng cách xây dựng một cây tìm kiếm trong quá trình chơi. Cây tìm kiếm này được mở rộng dựa trên các mô phỏng ngẫu nhiên của trò chơi (Monte Carlo rollouts). Ý tưởng là lặp lại các bước mô phỏng để đánh giá các hành động, dần dần cải thiện độ chính xác của cây tìm kiếm.

Quy trình hoạt động: gồm 4 bước chính được lặp lại nhiều lần để xây dựng cây tìm kiếm

- **Selection (Chọn hành động trong cây)**: Bắt đầu từ nút gốc (root node), đi xuống cây tìm kiếm bằng cách chọn các nút con dựa trên một tiêu chí nhất định. Tiêu chí phổ biến nhất **là UCT (Upper Confidence Bound for Trees)**. Công thức UCT kết hợp giữa giá trị trung bình phần thưởng và độ không chắc chắn để cân bằng giữa khai phá và khai thác.
- **Expansion (Mở rộng cây)**: Khi đạt đến một nút lá (leaf node) chưa được khám phá đầy đủ, MCTS mở rộng cây bằng cách thêm các nút con mới tương ứng với các hành động khả dĩ từ trạng thái đó. Thông thường, một hoặc một vài nút con sẽ được thêm vào mỗi lần mở rộng.
- **Simulation (Mô phỏng ngẫu nhiên)**: Từ trạng thái của nút mới mở rộng, thực hiện một hoặc nhiều mô phỏng Monte Carlo bằng cách chơi ngẫu nhiên đến khi đạt trạng thái kết thúc (terminal state). Kết quả của mô phỏng (phần thưởng đạt được) được ghi nhận để đánh giá chất lượng của nút.
- **Backpropagation (Lan truyền ngược)**: Sau khi hoàn thành mô phỏng, các giá trị phần thưởng thu được được lan truyền ngược từ nút mới mở rộng về gốc cây. Trong quá trình lan truyền, các thông tin sau được cập nhật: Tổng phần thưởng W_i và Số lần nút đã được thăm N_i. Điều này giúp các nút gần gốc có thêm thông tin để cải thiện việc ra quyết định ở các vòng lặp sau.

2.5.3.3. Một số công thức được sử dụng trong MCTS

- Công thức cho hành động tối ưu trong tìm kiếm cây Monte Carlo:

$$a = \underset{a}{arg\max} Q(s,a) + c\pi_\theta(a|s)\frac{\sqrt{N(s)}}{1+N(s,a)}$$

  - $Q(s,a)$: giá trị hành động ước tính qua tìm kiếm cây
  - $N(s,a)$: số lần đã thăm trạng thái s với hành động a
  - $N(s) = \sum_a N(s,a)$: tổng số lần thăm trạng thái s
  - c: hằng số điều chỉnh độ tin cậy của chính sách trong việc khám phá
  - **Ý nghĩa:** Công thức này tìm kiếm hành động tối ưu bằng cách kết hợp giá trị ước tính của hành động Q(s,a) với một thành phần điều chỉnh từ

chính sách ngẫu nhiên π_θ (a | s). Hằng số c điều chỉnh tầm quan trọng của chính sách trong việc khám phá, còn N(s) và N(s,a) cung cấp thông tin về mức độ thăm dò trạng thái và hành động.

- Công thức hàm mất mát chính sách:

$$\mathrm{l}(\theta) = -E_s\left[\sum_a \pi_{\mathrm{MCTS}}(a|s)\log \pi_\theta\,(a|s)\right]$$

$$\nabla \mathrm{l}(\theta) = -E_s\left[\sum_a \pi_{\mathrm{MCTS}}(a|s)\nabla_\theta \pi_\theta(a|s)\right]$$

  - $\pi_{\mathrm{MCTS}}(a|s)$: chính sách thu được từ tìm kiếm cây
  - Gradient cho phép chúng ta cập nhật tham số θ theo hướng tối ưu hóa chính sách, sao cho chính sách hiện tại ngày càng giống với chính sách do MCTS đề xuất.
  - **Ý nghĩa:** Hàm mất mát này đánh giá sự khác biệt giữa chính sách hiện tại π_θ sao cho nó càng giống với chính sách từ MCTS càng tốt, từ đó cải thiện khả năng lựa chọn hành động trong các trạng thái khác nhau.
- Công thức hàm mất mát giá trị:

$$\mathrm{l}(\phi) = \frac{1}{2}E_s\left[\left(U_\phi(s) - U_{\mathrm{MCTS}}(s)\right)^2\right]$$

$$\nabla \mathrm{l}(\phi) = E_s\left[\left(U_\phi(s) - U_{\mathrm{MCTS}}(s)\right)\nabla_\phi U_\phi(s)\right]$$

  - $U_{\mathrm{MCTS}}(s) = \max_a Q\,(s,a)$: Hàm giá trị này đại diện cho giá trị tối đa mà chúng ta có thể đạt được từ trạng thái s bằng cách chọn hành động tốt nhất, dựa trên ước lượng từ MCTS.
  - Gradient cho hàm giá trị $U_\phi(s)$ được xác định bằng cách so sánh giữa giá trị ước lượng và giá trị từ MCTS.
  - Gradient này cho phép chúng ta cập nhật tham số ϕ của hàm giá trị, nhằm giảm thiểu độ sai lệch giữa giá trị ước lượng và giá trị thực tế từ MCTS.

- **Ý nghĩa:** Hàm mất mát này đo lường sai số giữa giá trị ước tính từ hàm giá trị của mô hình $U_{\phi}(s)$ và giá trị ước tính từ kết quả của tìm kiếm cây $U_{\text{MCTS}}(s)$. Mục tiêu là điều chỉnh hàm giá trị để nó khớp với các giá trị ước tính từ MCTS, nhằm cải thiện độ chính xác của các dự đoán giá trị trong các trạng thái.

2.5.3.4. Cách hoạt động của MCTS

Bước 1: Sử dụng Actor làm "Prior" cho MCTS

- Chính sách π(s, a) từ Actor được sử dụng làm prior probability (xác suất ban đầu) để hướng dẫn MCTS.
- Thay vì khởi tạo các xác suất hành động một cách ngẫu nhiên, MCTS sử dụng các xác suất từ Actor để ưu tiên các hành động khả thi hơn trong bước tìm kiếm.
- Điều này giúp MCTS tập trung tìm kiếm vào các nhánh tiềm năng, giảm số lượng mô phỏng cần thiết.

Bước 2: MCTS thực hiện tìm kiếm

- Selection (Chọn nhánh): Tại mỗi nút trong cây tìm kiếm, các hành động được chọn dựa trên công thức cho hành động tối ưu trong tìm kiếm cây Monte Carlo.
- Expansion (Mở rộng nhánh): Khi một trạng thái mới được gặp trong cây, MCTS mở rộng các nhánh con. Các hành động khả thi được gán xác suất ban đầu $\boldsymbol{P(s,a) \propto \pi\theta(a|s)}$. Công thức này cho biết xác suất $\boldsymbol{P(s,a)}$ của hành động a tại trạng thái s trong (MCTS) được xác định tỷ lệ với $\boldsymbol{\pi\theta(a|s)}$, chính sách được cung cấp bởi Actor. Xác suất $\boldsymbol{P(s,a)}$ không cần phải bằng chính xác $\boldsymbol{\pi\theta(a|s)}$ nhưng sẽ **tăng hoặc giảm theo tỉ lệ thuận** với $\boldsymbol{\pi\theta(a|s)}$. Điều này giúp:
  - Tập trung vào các nhánh tiềm năng mà Actor đã đánh giá cao.
  - Giảm số lượng mô phỏng cần thiết để tìm kiếm các hành động tốt.
- Simulation (Mô phỏng): Từ trạng thái hiện tại, MCTS thực hiệnmột hoặc nhiều mô phỏng ngẫu nhiên (Monte Carlo rollouts) đến khi đạt trạng thái kết thúc.

Chính sách Actor có thể được sử dụng làm chính sách mô phỏng để tăng độ chính xác của mô phỏng.

- Backpropagation (Lan truyền ngược): Kết quả từ các mô phỏng (reward) được lan truyền ngược trong cây để cập nhật giá trị Q(s,a) và số lần thăm N(s,a).
- Critic có thể hỗ trợ trong việc định lượng giá trị Q(s,a) tốt hơn, đặc biệt trong các trạng thái chưa được khám phá đầy đủ.

Bước 3: Hành động từ MCTS

- Sau một số lượng vòng lặp (hoặc thời gian giới hạn), MCTS chọn hành động tốt nhất từ nút gốc dựa trên: Số lần thăm N(s,a) (hành động được khám phá nhiều nhất) $\boldsymbol{a} *= \mathbf{a}\boldsymbol{r}\boldsymbol{g}\underset{\boldsymbol{a}}{\mathbf{max}}\boldsymbol{N}(\boldsymbol{s_0}, \boldsymbol{a})$. Hoặc giá trị trung bình Q(s,a) (hành động có phần thưởng dự kiến cao nhất)

$$\boldsymbol{a} *= \mathbf{a}\boldsymbol{r}\boldsymbol{g}\underset{\boldsymbol{a}}{\mathbf{max}}\boldsymbol{Q}(\boldsymbol{s_0}, \boldsymbol{a}).$$

Bước 4: Cập nhật Actor-Critic

- Cập nhật Critic: Giá trị từ MCTS: $\boldsymbol{UMCTS}(\boldsymbol{s}) = \boldsymbol{maxQ}(\boldsymbol{s}, \boldsymbol{a})$ Critic được cập nhật để hàm giá trị $\boldsymbol{U}\boldsymbol{\phi}(\mathbf{s})$ gần với UMCTS(s) thông qua hàm mất mát giá trị.
- Cập nhật Actor: Chính sách từ MCTS: $\boldsymbol{\pi_{MCTS}}(\boldsymbol{a}|\boldsymbol{s}) \propto \boldsymbol{N}(\boldsymbol{s}, \boldsymbol{a})^{\boldsymbol{\eta}}$
  - $\boldsymbol{\eta}$**:** Siêu tham số điều chỉnh mức độ ngẫu nhiên
- Công thức này biểu thị chính sách $\boldsymbol{\pi_{MCTS}}(\boldsymbol{a}|\boldsymbol{s})$ từ MCTS, được xác định tỷ lệ với số lần hành động $\boldsymbol{a}$ được chọn ở trạng thái $\boldsymbol{s}$ trong quá trình tìm kiếm, nâng lên lũy thừa $\boldsymbol{\eta}$**.**
- **Ý nghĩa:**
  - $\boldsymbol{N}(\boldsymbol{s}, \boldsymbol{a})$**:** Số lần hành động $\boldsymbol{a}$ đã được thăm tại trạng thái $\boldsymbol{s}$ trong MCTS
  - $\boldsymbol{\eta} \geq \mathbf{0}$**:** Siêu tham số điều chỉnh mức độ "tham lam" (greediness) của chính sách.
    - Nếu $\boldsymbol{\eta} = \mathbf{0}$**:** Chính sách$\boldsymbol{\pi_{MCTS}}(\boldsymbol{a}|\boldsymbol{s})$ chọn ngẫu nhiên, không ưu tiên hành động nào.

  - Nếu $\boldsymbol{\eta > 0}$**:** Các hành động được chọn nhiều trong quá trình MCTS sẽ có xác suất cao hơn.

- Ứng dụng trong MCTS:Sau khi hoàn tất tìm kiếm, $\boldsymbol{\pi_{MCTS}(a|s)}$ được sử dụng như một chính sách tốt nhất từ dữ liệu MCTS. Công thức này giúp Actor được tối ưu để chính sách $\boldsymbol{\pi\theta(a|s)}$ khớp với $\boldsymbol{\pi_{MCTS}(a|s)}$, thông qua hàm mất mát chính sách.

Chương 3. CHƯƠNG TRÌNH MINH HỌA

3.1. Công cụ sử dụng

- Môi trường mô phỏng: OpenAi Gym - Một môi trường mô phỏng linh hoạt và tiêu chuẩn trong học tăng cường, cung cấp các giao diện và tập môi trường đa dạng để huấn luyện và đánh giá các thuật toán như Actor Critic.
- Ngôn ngữ lập trình: Python
- Thư viện hỗ trợ:
    - Keras là thư viện học sâu cho phép xây dựng và huấn luyện mạng neural network một cách nhanh chóng và dễ dàng.
    - TensorFlow là nền tảng điện toán số phát triển bởi Google, hỗ trợ mạnh mẽ cho các thuật toán học máy và học sâu với hiệu suất cao.
    - NumPy là thư viện tính toán khoa học của Python, cung cấp các công cụ và hàm hiệu quả để thao tác với các mảng và ma trận số học.
    - Thư viện os của Python giúp tương tác và thực hiện các thao tác với hệ điều hành, như quản lý môi trường, đường dẫn tệp và biến hệ thống.

3.2. Thực hiện bài toán CartPole-v1 với Actor Critic cơ bản

```
seed = 42
gamma = 0.99
max_steps_per_episode = 500
env = gym.make("CartPole-v1")
env.seed(seed)
eps = np.finfo(np.float32).eps.item()
```

- seed: Hạt giống ngẫu nhiên giúp tái lập kết quả.
- gamma: Hệ số giảm giá dùng để tính giá trị hồi quy của phần thưởng.
- max_steps_per_episode: Số bước tối đa mỗi tập (episode).
- env: Tạo môi trường CartPole.
- env.seed: Đặt hạt giống cho môi trường.
- eps: Giá trị epsilon rất nhỏ để tránh lỗi chia cho 0.

```
num_inputs = 4
num_actions = 2
num_hidden = 128

inputs = layers.Input(shape=(num_inputs,))
common = layers.Dense(num_hidden, activation="relu")(inputs)
action = layers.Dense(num_actions, activation="softmax")(common)
critic = layers.Dense(1)(common)

model = keras.Model(inputs=inputs, outputs=[action, critic])
```

- Biến num_inputs = 4 thể hiện số chiều của không gian trạng thái trong môi trường CartPole (vị trí xe, vận tốc xe, góc đòn, vận tốc góc đòn).
- Biến num_actions = 2 chỉ ra số hành động có thể thực hiện (di chuyển xe sang trái hoặc phải).
- Biến num_hidden = 128 xác định số lượng nút ẩn trong lớp kết nối dày đặc của mạng neural.
- Xây dựng kiến trúc mạng Actor-Critic với các lớp:
    - Lớp ẩn chung (common) với hàm kích hoạt ReLU.
    - Lớp Actor dự đoán xác suất hành động.
    - Lớp Critic dự đoán giá trị trạng thái.

```
optimizer = keras.optimizers.Adam(learning_rate=0.01)
huber_loss = keras.losses.Huber()

action_probs_history = []
critic_value_history = []
rewards_history = []
running_reward = 0
episode_count = 0
max_episodes = 1000
```

- optimizer: Trình tối ưu hóa Adam với tốc độ học 0.01.
- huber_loss: Hàm mất mát Huber.
- Các biến để lưu lịch sử và theo dõi:
- action_probs_history: Log xác suất các hành động đã chọn.

- critic_value_history: Giá trị Critic theo từng bước.
- rewards_history: Phần thưởng từng bước.
- running_reward: Phần thưởng trung bình động.
- episode_count: Đếm số tập.

- Các danh sách action_probs_history, critic_value_history, và rewards_history được khởi tạo để lưu trữ thông tin về xác suất hành động, giá trị critic, và phần thưởng trong quá trình huấn luyện.
- Biến running_reward được sử dụng để tính toán phần thưởng trung bình động, giúp đánh giá hiệu suất học tập của mô hình.
- Các biến episode_count và max_episodes quản lý số lượng tập (episode) đã thực hiện và giới hạn số tập tối đa trong quá trình huấn luyện.
- Vòng lặp while được sử dụng để thực hiện quá trình huấn luyện cho đến khi đạt số episode tối đa hoặc giải quyết được bài toán.
- Hàm env.reset() khôi phục trạng thái ban đầu của môi trường cho mỗi tập mới.
- Đoạn mã sử dụng tf.GradientTape() để tự động tính toán gradient, hỗ trợ quá trình cập nhật trọng số mô hình.
- Các trạng thái được chuyển đổi thành tensor và mở rộng chiều để phù hợp với đầu vào mô hình.
- Mô hình dự đoán xác suất hành động và giá trị critic cho trạng thái hiện tại.
- Hành động được chọn ngẫu nhiên dựa trên phân phối xác suất được dự đoán.
- Môi trường được cập nhật bằng cách thực hiện hành động và nhận về trạng thái mới, phần thưởng, và trạng thái kết thúc.

```
while episode_count < max_episodes:
    state = env.reset()
    episode_reward = 0

    with tf.GradientTape() as tape:
        for timestep in range(1, max_steps_per_episode + 1):
            state = ops.convert_to_tensor(state)
            state = ops.expand_dims(state, 0)

            action_probs, critic_value = model(state)
            critic_value_history.append(critic_value[0, 0])

            action = np.random.choice(num_actions, p=np.squeeze(action_probs))
            action_probs_history.append(ops.log(action_probs[0, action]))

            state, reward, done, _ = env.step(action)
            rewards_history.append(reward)
            episode_reward += reward

            if done:
                break

        returns = []
        discounted_sum = 0
        for r in rewards_history[::-1]:
            discounted_sum = r + gamma * discounted_sum
            returns.insert(0, discounted_sum)

        returns = np.array(returns)
        returns = (returns - np.mean(returns)) / (np.std(returns) + eps)
        returns = returns.tolist()
```

```
        history = zip(action_probs_history, critic_value_history, returns)
        actor_losses = []
        critic_losses = []
        for log_prob, value, ret in history:
            advantage = ret - value
            actor_losses.append(-log_prob * advantage)
            critic_losses.append(huber_loss(ops.expand_dims(value, 0), ops.expand_dims(ret, 0)))

        loss_value = sum(actor_losses) + sum(critic_losses)

        grads = tape.gradient(loss_value, model.trainable_variables)
        optimizer.apply_gradients(zip(grads, model.trainable_variables))

        action_probs_history.clear()
        critic_value_history.clear()
        rewards_history.clear()

    episode_count += 1
    running_reward = 0.05 * episode_reward + (1 - 0.05) * running_reward

    if episode_count % 10 == 0:
        template = "Episode: {}, Running Reward: {:.2f}"
        print(template.format(episode_count, running_reward))

    if running_reward > 475:
        print("Solved at episode {}!".format(episode_count))
        break

if episode_count == max_episodes:
    print(f"Reached the maximum episode limit of {max_episodes}. Training stopped.")
```

- Vòng lặp chính thực hiện huấn luyện:
  - Khởi tạo trạng thái môi trường và phần thưởng của mỗi tập.
  - Dự đoán hành động thông qua mạng Actor-Critic.
  - Tính giá trị hồi quy từ phần thưởng thu thập được.
  - Tính toán mất mát (actor + critic) và cập nhật mô hình.
- Điều kiện dừng:
  - Nếu running_reward > 475, mô hình được xem là giải quyết bài toán.
  - Nếu đạt đến max_episodes, vòng lặp sẽ kết thúc.

Kết quả:

```
Episode: 600, Running Reward: 141.39
Episode: 610, Running Reward: 155.32
Episode: 620, Running Reward: 182.75
Episode: 630, Running Reward: 225.77
Episode: 640, Running Reward: 335.81
Episode: 650, Running Reward: 381.34
Episode: 660, Running Reward: 268.63
Episode: 670, Running Reward: 217.25
Episode: 680, Running Reward: 192.74
Episode: 690, Running Reward: 225.66
Episode: 700, Running Reward: 223.90
Episode: 710, Running Reward: 303.72
Episode: 720, Running Reward: 354.11
Episode: 730, Running Reward: 396.27
Episode: 740, Running Reward: 437.89
Episode: 750, Running Reward: 414.43
Episode: 760, Running Reward: 446.37
Episode: 770, Running Reward: 467.89
Solved at episode 775!
```

3.3. Thực hiện bài toán CartPole-v1 với Actor Critic với GAE

```python
env = gym.make("CartPole-v1")
num_inputs = 4
num_actions = 2
num_hidden1 = 256
num_hidden2 = 128
gamma = 0.97
lambda_ = 0.95
max_steps_per_episode = 500
max_episodes = 1000
initial_entropy_beta = 0.01
learning_rate = 0.001
```

- env: Khởi tạo môi trường CartPole-v1.
- num_inputs: Số trạng thái đầu vào của môi trường.
- num_actions: Số hành động có thể thực hiện (trái hoặc phải).
- num_hidden1, num_hidden2: Số lượng nơ-ron trong hai lớp ẩn.
- gamma: Hệ số giảm giá để tính phần thưởng hồi quy.
- lambda_: Hệ số GAE (Generalized Advantage Estimation).
- max_steps_per_episode: Số bước tối đa cho mỗi tập.

- max_episodes: Số tập tối đa để huấn luyện.
- initial_entropy_beta: Hệ số điều chỉnh entropy ban đầu.
- learning_rate: Tốc độ học của trình tối ưu hóa.

```
inputs = layers.Input(shape=(num_inputs,))
common = layers.Dense(num_hidden1, activation="relu")(inputs)
common = layers.LayerNormalization()(common)
common = layers.Dense(num_hidden2, activation="relu")(common)
action = layers.Dense(num_actions, activation="softmax")(common)
critic = layers.Dense(1)(common)
model = tf.keras.Model(inputs=inputs, outputs=[action, critic])

optimizer = tf.keras.optimizers.Adam(learning_rate=learning_rate)
huber_loss = tf.keras.losses.Huber()
```

- inputs: Đầu vào của mô hình là trạng thái từ môi trường.
- common: Hai lớp ẩn dùng ReLU, chuẩn hóa đầu ra của lớp đầu tiên.
- action: Đầu ra cho Actor, dự đoán xác suất thực hiện các hành động.
- critic: Đầu ra cho Critic, dự đoán giá trị trạng thái.
- model: Kết hợp Actor và Critic thành một mô hình.
- optimizer: Sử dụng Adam Optimizer để cập nhật trọng số.
- huber_loss: Hàm mất mát Huber cho Critic.

```
def compute_gae_and_returns(rewards, values, gamma=0.97, lambda_=0.95):
    advantages = []
    gae = 0
    returns = []
    for t in reversed(range(len(rewards))):
        delta = rewards[t] + gamma * (values[t + 1] if t + 1 < len(values) else 0) - values[t]
        gae = delta + gamma * lambda_ * gae
        advantages.insert(0, gae)
        returns.insert(0, gae + values[t])
    return np.array(advantages), np.array(returns)
```

- Hàm tính toán Advantage và Returns dựa trên GAE.
- rewards: Danh sách phần thưởng tại mỗi bước.
- values: Giá trị Critic tại mỗi bước.
- delta: Sai lệch giữa phần thưởng thực tế và giá trị Critic.

- gae: Tổng hợp GAE qua các bước.
- Kết quả trả về: Advantage và Returns cho toàn bộ tập.

```
running_reward = 0
entropy_beta = initial_entropy_beta

for episode_count in range(1, max_episodes + 1):
    state = env.reset()
    episode_reward = 0
    action_probs_history = []
    critic_value_history = []
    rewards_history = []

    with tf.GradientTape() as tape:
        for step in range(max_steps_per_episode):
            state = tf.convert_to_tensor(state, dtype=tf.float32)
            state = tf.expand_dims(state, axis=0)

            action_probs, critic_value = model(state)
            action = np.random.choice(num_actions, p=np.squeeze(action_probs))

            action_probs_history.append(tf.math.log(action_probs[0, action]))
            critic_value_history.append(critic_value[0, 0])

            state, reward, done, _ = env.step(action)
            rewards_history.append(reward)
            episode_reward += reward

            if done:
                break

        critic_value_history.append(0 if done else model(tf.expand_dims(state, axis=0))[1][0, 0])

        advantages, returns = compute_gae_and_returns(rewards_history, critic_value_history, gamma, lambda_)
        advantages = (advantages - np.mean(advantages)) / (np.std(advantages) + 1e-8)
```

```
        actor_losses = [-log_prob * adv for log_prob, adv in zip(action_probs_history, advantages)]
        critic_losses = [huber_loss(tf.expand_dims(v, 0), tf.expand_dims(r, 0)) for v, r in zip(critic_value_history[:-1], returns)]
        entropy_loss = -entropy_beta * tf.reduce_mean(tf.reduce_sum(action_probs * tf.math.log(action_probs + 1e-10), axis=1))

        loss_value = tf.reduce_sum(actor_losses) + 0.5 * tf.reduce_sum(critic_losses) + entropy_loss

        grads = tape.gradient(loss_value, model.trainable_variables)
        optimizer.apply_gradients(zip(grads, model.trainable_variables))

    running_reward = 0.1 * episode_reward + 0.9 * running_reward
    entropy_beta = max(0.001, initial_entropy_beta * (1 - running_reward / 500))

    if episode_count % 10 == 0:
        print(f"Episode: {episode_count}, Running Reward: {running_reward:.2f}")

    if running_reward > 475:
        print(f"Đạt kết quả sau {episode_count} tập!")
        break

if episode_count == max_episodes:
    print(f"Đã huấn luyện xong sau {max_episodes} tập.")
```

- Vòng lặp chính thực hiện huấn luyện qua các tập:

- Khởi tạo trạng thái và lưu trữ lịch sử tại mỗi bước.
- Tính toán GAE và Returns để cập nhật mô hình.
- Tính tổng mất mát:
  - Actor loss: Sử dụng Advantage.
  - Critic loss: Dựa trên hàm mất mát Huber.
  - Entropy loss: Điều chỉnh để tăng tính ngẫu nhiên trong hành động.

- Cập nhật trọng số mô hình bằng gradient descent.
- Dừng huấn luyện nếu đạt điều kiện phần thưởng trung bình mục tiêu.

Kết quả:

```
Episode: 600, Running Reward: 280.17
Episode: 610, Running Reward: 257.78
Episode: 620, Running Reward: 220.29
Episode: 630, Running Reward: 205.27
Episode: 640, Running Reward: 224.57
Episode: 650, Running Reward: 205.24
Episode: 660, Running Reward: 194.26
Episode: 670, Running Reward: 162.82
Episode: 680, Running Reward: 151.84
Episode: 690, Running Reward: 160.78
Episode: 700, Running Reward: 268.76
Episode: 710, Running Reward: 352.94
Episode: 720, Running Reward: 258.69
Episode: 730, Running Reward: 180.52
Episode: 740, Running Reward: 149.64
Episode: 750, Running Reward: 209.85
Episode: 760, Running Reward: 382.33
Episode: 770, Running Reward: 326.56
Episode: 780, Running Reward: 257.27
Episode: 790, Running Reward: 231.12
Episode: 800, Running Reward: 233.51
Episode: 810, Running Reward: 366.53
Episode: 820, Running Reward: 453.46
Đạt kết quả sau 826 tập!
```

3.4. Thực hiện bài toán CartPole-v1 với Actor Critic với DPG

```
env = gym.make("CartPole-v1")
num_inputs = 4
num_actions = 1
num_hidden = 256
gamma = 0.99
exploration_noise_std = 0.3
min_exploration_noise_std = 0.05
noise_decay = 0.995
max_steps_per_episode = 500
max_episodes = 1000
```

- env: Tạo môi trường CartPole-v1.
- num_inputs: Số đầu vào của mô hình (trạng thái từ môi trường).
- num_actions: Số đầu ra của mô hình (hành động thực hiện).
- num_hidden: Số lượng nơ-ron trong lớp ẩn.
- gamma: Hệ số chiết khấu dùng để tính giá trị Q.
- exploration_noise_std, min_exploration_noise_std, noise_decay: Tham số điều chỉnh mức độ nhiễu khi chọn hành động.
- max_steps_per_episode: Số bước tối đa trong mỗi tập.
- max_episodes: Số tập tối đa trong quá trình huấn luyện.

```
actor_inputs = layers.Input(shape=(num_inputs,))
actor_hidden = layers.Dense(num_hidden, activation="relu")(actor_inputs)
actor_hidden = layers.Dense(num_hidden, activation="relu")(actor_hidden)
actor_output = layers.Dense(num_actions, activation="tanh")(actor_hidden)
actor_model = tf.keras.Model(inputs=actor_inputs, outputs=actor_output)
```

- Xây dựng mạng Actor với đầu vào là trạng thái từ môi trường.
- Gồm 2 lớp ẩn sử dụng hàm kích hoạt ReLU và một lớp đầu ra sử dụng hàm tanh để dự đoán hành động.

```
critic_inputs = layers.Input(shape=(num_inputs + num_actions,))
critic_hidden = layers.Dense(num_hidden, activation="relu")(critic_inputs)
critic_hidden = layers.Dense(num_hidden, activation="relu")(critic_hidden)
q_value = layers.Dense(1)(critic_hidden)
critic_model = tf.keras.Model(inputs=critic_inputs, outputs=q_value)
```

- Xây dựng mạng Critic với đầu vào là trạng thái và hành động.
- Mạng trả về giá trị Q (Q-value), đại diện cho giá trị của trạng thái-hành động.

```
target_actor_model = tf.keras.models.clone_model(actor_model)
target_critic_model = tf.keras.models.clone_model(critic_model)
```

- Tạo các target network (target_actor_model và target_critic_model) từ mạng chính Actor và Critic.
- Các target network giúp ổn định quá trình huấn luyện bằng cách cập nhật chậm hơn mạng chính.

```
class ReplayBuffer:
    def __init__(self, buffer_capacity=200000, batch_size=128):
        self.buffer_capacity = buffer_capacity
        self.batch_size = batch_size
        self.buffer_counter = 0
        self.state_buffer = np.zeros((buffer_capacity, num_inputs))
        self.action_buffer = np.zeros((buffer_capacity, num_actions))
        self.reward_buffer = np.zeros((buffer_capacity, 1))
        self.next_state_buffer = np.zeros((buffer_capacity, num_inputs))
        self.done_buffer = np.zeros((buffer_capacity, 1))

    def store(self, state, action, reward, next_state, done):
        index = self.buffer_counter % self.buffer_capacity
        self.state_buffer[index] = state
        self.action_buffer[index] = action
        self.reward_buffer[index] = reward
        self.next_state_buffer[index] = next_state
        self.done_buffer[index] = done
        self.buffer_counter += 1

    def sample(self):
        max_buffer = min(self.buffer_counter, self.buffer_capacity)
        batch_indices = np.random.choice(max_buffer, self.batch_size)
        return (
            self.state_buffer[batch_indices],
            self.action_buffer[batch_indices],
            self.reward_buffer[batch_indices],
            self.next_state_buffer[batch_indices],
            self.done_buffer[batch_indices],
        )

buffer = ReplayBuffer()
```

- Replay Buffer là bộ lưu trữ dữ liệu trạng thái, hành động, phần thưởng, trạng thái tiếp theo và cờ trạng thái kết thúc.
- Hai chức năng chính:
    - store: Lưu trữ dữ liệu trải nghiệm vào bộ đệm.
    - sample: Lấy một batch dữ liệu ngẫu nhiên để huấn luyện.

```
running_reward = 0
tau = 0.005

for episode in range(max_episodes):
    state = env.reset()
    episode_reward = 0

    for step in range(max_steps_per_episode):
        state_tensor = tf.convert_to_tensor(state, dtype=tf.float32)
        state_tensor = tf.expand_dims(state_tensor, axis=0)

        action = actor_model(state_tensor).numpy()[0]
        action += np.random.normal(0, exploration_noise_std)
        action = np.clip(action, -1, 1)

        next_state, reward, done, _ = env.step(int(action > 0))
        buffer.store(state, action, reward, next_state, done)
        episode_reward += reward

        if buffer.buffer_counter >= buffer.batch_size:
            states, actions, rewards, next_states, dones = buffer.sample()
            states = tf.convert_to_tensor(states, dtype=tf.float32)
            actions = tf.convert_to_tensor(actions, dtype=tf.float32)
            rewards = tf.convert_to_tensor(rewards, dtype=tf.float32)
            next_states = tf.convert_to_tensor(next_states, dtype=tf.float32)
            dones = tf.convert_to_tensor(dones, dtype=tf.float32)

            with tf.GradientTape() as tape:
                target_actions = target_actor_model(next_states)
                target_q_values = rewards + (1 - dones) * gamma * target_critic_model(
                    tf.concat([next_states, target_actions], axis=1)
                )
                predicted_q_values = critic_model(tf.concat([states, actions], axis=1))
                critic_loss = tf.keras.losses.MSE(target_q_values, predicted_q_values)
            critic_grads = tape.gradient(critic_loss, critic_model.trainable_variables)
            critic_optimizer.apply_gradients(zip(critic_grads, critic_model.trainable_variables))
```

```
        with tf.GradientTape() as tape:
            actions_pred = actor_model(states)
            actor_loss = -tf.reduce_mean(critic_model(tf.concat([states, actions_pred], axis=1)))
        actor_grads = tape.gradient(actor_loss, actor_model.trainable_variables)
        actor_optimizer.apply_gradients(zip(actor_grads, actor_model.trainable_variables))

        for target_param, param in zip(target_critic_model.trainable_variables, critic_model.trainable_variables):
            target_param.assign(tau * param + (1 - tau) * target_param)
        for target_param, param in zip(target_actor_model.trainable_variables, actor_model.trainable_variables):
            target_param.assign(tau * param + (1 - tau) * target_param)

    if done:
        break

    state = next_state

exploration_noise_std = max(min_exploration_noise_std, exploration_noise_std * noise_decay)
running_reward = 0.05 * episode_reward + (1 - 0.05) * running_reward

if episode % 10 == 0:
    print(f"Episode: {episode}, Running Reward: {running_reward:.2f}, Exploration Noise: {exploration_noise_std:.3f}")

if running_reward > 475:
    print(f"Solved at episode {episode}!")
    break
```

- Vòng lặp chính: Thực hiện huấn luyện Actor-Critic qua từng tập.
- Actor: Dự đoán hành động và thêm nhiễu để khám phá.
- Replay Buffer: Lấy dữ liệu để huấn luyện theo batch.
- Critic: Huấn luyện để tính giá trị Q chính xác.
- Actor: Huấn luyện để tăng xác suất thực hiện hành động tối ưu.
- Cập nhật Target Networks: Thực hiện cập nhật trọng số dần dần.
- Điều chỉnh nhiễu: Giảm dần mức độ nhiễu khi chọn hành động.

Kết quả:

```
Episode: 400, Running Reward: 251.64, Exploration Noise: 0.050
Episode: 410, Running Reward: 325.34, Exploration Noise: 0.050
Episode: 420, Running Reward: 318.25, Exploration Noise: 0.050
Episode: 430, Running Reward: 247.25, Exploration Noise: 0.050
Episode: 440, Running Reward: 202.79, Exploration Noise: 0.050
Episode: 450, Running Reward: 187.44, Exploration Noise: 0.050
Episode: 460, Running Reward: 204.92, Exploration Noise: 0.050
Episode: 470, Running Reward: 185.47, Exploration Noise: 0.050
Episode: 480, Running Reward: 173.99, Exploration Noise: 0.050
Episode: 490, Running Reward: 180.70, Exploration Noise: 0.050
Episode: 500, Running Reward: 176.59, Exploration Noise: 0.050
Episode: 510, Running Reward: 166.37, Exploration Noise: 0.050
Episode: 520, Running Reward: 158.14, Exploration Noise: 0.050
Episode: 530, Running Reward: 165.27, Exploration Noise: 0.050
Episode: 540, Running Reward: 246.57, Exploration Noise: 0.050
Episode: 550, Running Reward: 303.78, Exploration Noise: 0.050
Episode: 560, Running Reward: 379.46, Exploration Noise: 0.050
Episode: 570, Running Reward: 418.59, Exploration Noise: 0.050
Episode: 580, Running Reward: 451.26, Exploration Noise: 0.050
Episode: 590, Running Reward: 470.82, Exploration Noise: 0.050
Solved at episode 594!
```

3.5. Thực hiện bài toán CartPole-v1 với Actor Critic với MCTS

```
inputs = layers.Input(shape=(num_inputs,))
common = layers.Dense(num_hidden, activation="relu")(inputs)
common = layers.Dense(num_hidden, activation="relu")(common)
action = layers.Dense(num_actions, activation="softmax")(common)
critic = layers.Dense(1)(common)
model = keras.Model(inputs=inputs, outputs=[action, critic])
```

- Mô hình gồm các tầng:
  - Tầng đầu vào: Nhận trạng thái từ môi trường.
  - Tầng ẩn: 2 lớp ẩn với kích hoạt ReLU để trích xuất đặc trưng.
  - Tầng Actor: Sử dụng softmax để dự đoán xác suất của các hành động.
  - Tầng Critic: Trả về giá trị trạng thái.

```
action_probs_history = []
critic_value_history = []
rewards_history = []
running_reward = 0
episode_count = 0
max_episodes = 1000
entropy_beta = 0.05
num_simulations = 50
running_rewards = []
```

- Biến lưu trữ:
  - action_probs_history: Lịch sử xác suất hành động.
  - critic_value_history: Lịch sử giá trị từ Critic.
  - rewards_history: Lịch sử phần thưởng.
- Các tham số:
  - running_reward: Phần thưởng trung bình.
  - entropy_beta: Hệ số cho entropy regularization.
  - num_simulations: Số lần mô phỏng cho MCTS.

```python
class MCTSNode:
    def __init__(self, state, parent=None, prior_prob=1.0):
        self.state = state
        self.parent = parent
        self.children = {}
        self.visits = 0
        self.value = 0
        self.prior_prob = prior_prob

    def is_fully_expanded(self):
        return len(self.children) == num_actions

    def best_child(self, c_puct=1.0):
        return max(
            self.children.items(),
            key=lambda child: child[1].value / (1 + child[1].visits) +
                              c_puct * child[1].prior_prob * np.sqrt(self.visits) / (1 + child[1].visits)
        )[1]

    def expand(self, action_probs):
        for action, prob in enumerate(action_probs):
            if action not in self.children:
                self.children[action] = MCTSNode(self.state, parent=self, prior_prob=prob)

    def backpropagate(self, reward):
        self.value += reward
        self.visits += 1
        if self.parent:
            self.parent.backpropagate(reward)
```

- MCTSNode: Đại diện cho một nút trong cây MCTS.
- Các chức năng chính:
    - is_fully_expanded: Kiểm tra nút đã mở rộng hết hành động chưa.
    - best_child: Tìm nút con tốt nhất dựa trên công thức UCB.
    - expand: Mở rộng cây dựa trên xác suất hành động.
    - backpropagate: Lan truyền giá trị ngược lên cây.

```python
def run_mcts(root, model, num_simulations=num_simulations, c_puct=1.0):
    for _ in range(num_simulations):
        node = root
        while node.is_fully_expanded() and node.children:
            node = node.best_child(c_puct)
        state_tensor = tf.convert_to_tensor(node.state, dtype=tf.float32)
        state_tensor = tf.expand_dims(state_tensor, 0)
        action_probs, critic_value = model(state_tensor)
        action_probs = action_probs.numpy().squeeze()
        if not node.is_fully_expanded():
            node.expand(action_probs)
        reward = critic_value.numpy()[0, 0]
        node.backpropagate(reward)
    return max(root.children.items(), key=lambda child: child[1].visits)[0]
```

- Thuật toán MCTS thực hiện:
  - Duyệt cây từ gốc đến lá.
  - Dự đoán xác suất hành động và giá trị trạng thái từ mô hình.
  - Mở rộng nút và cập nhật giá trị thông qua backpropagation.
  - Trả về hành động có lượt thăm cao nhất.

```python
def compute_loss(action_probs_history, critic_value_history, returns, entropy_beta):
    actor_losses, critic_losses = [], []
    entropy = 0
    for log_prob, value, ret in zip(action_probs_history, critic_value_history, returns):
        advantage = ret - value
        actor_losses.append(-log_prob * advantage)
        critic_losses.append(huber_loss(tf.expand_dims(value, 0), tf.expand_dims(ret, 0)))
        entropy += -tf.reduce_sum(action_probs_history[-1] * tf.math.log(action_probs_history[-1] + eps))
    total_loss = sum(actor_losses) + sum(critic_losses) - entropy_beta * entropy
    return total_loss
```

- Tính toán mất mát gồm:
  - Actor Loss: Khuyến khích hành động tối ưu dựa trên Advantage.
  - Critic Loss: Mất mát giữa giá trị dự đoán và Returns.
  - Entropy Regularization: Tăng tính ngẫu nhiên trong hành động.

```
while episode_count < max_episodes:
    state = env.reset()
    episode_reward = 0
    with tf.GradientTape() as tape:
        for timestep in range(1, max_steps_per_episode + 1):
            state_tensor = tf.convert_to_tensor(state, dtype=tf.float32)
            state_tensor = tf.expand_dims(state_tensor, 0)
            action_probs, critic_value = model(state_tensor)
            critic_value_history.append(critic_value[0, 0])
            root = MCTSNode(state)
            root.expand(action_probs.numpy().squeeze())
            action = run_mcts(root, model)
            action_probs_history.append(action_probs[0, action])
            state, reward, done, _ = env.step(action)
            rewards_history.append(reward)
            episode_reward += reward
            if done:
                break

        returns = []
        discounted_sum = 0
        for r in rewards_history[::-1]:
            discounted_sum = r + gamma * discounted_sum
            returns.insert(0, discounted_sum)
        returns = np.array(returns)
        returns = (returns - np.mean(returns)) / (np.std(returns) + eps)
        returns = returns.tolist()
        loss_value = compute_loss(action_probs_history, critic_value_history, returns, entropy_beta)
        grads = tape.gradient(loss_value, model.trainable_variables)
        grads, _ = tf.clip_by_global_norm(grads, 1.0)
        optimizer.apply_gradients(zip(grads, model.trainable_variables))
        action_probs_history.clear()
        critic_value_history.clear()
        rewards_history.clear()
```

```
    episode_count += 1
    running_reward = 0.05 * episode_reward + (1 - 0.05) * running_reward
    running_rewards.append(running_reward)

    if episode_count % 10 == 0:
        print(f"Episode: {episode_count}, Running Reward: {running_reward:.2f}")
    if running_reward > 475:
        print(f"Solved at episode {episode_count}!")
        break

if episode_count == max_episodes:
    print(f"Reached the maximum episode limit of {max_episodes}. Training stopped.")
```

- Duyệt môi trường và thực hiện:
    - Dự đoán hành động, mở rộng cây MCTS.

- Tính Advantage, Returns và mất mát.
- Cập nhật trọng số mô hình.

- Dừng nếu đạt được phần thưởng yêu cầu hoặc vượt số tập tối đa.

Kết quả:

```
Episode: 800, Running Reward: 9.43
Episode: 810, Running Reward: 9.55
Episode: 820, Running Reward: 9.42
Episode: 830, Running Reward: 9.53
Episode: 840, Running Reward: 9.49
Episode: 850, Running Reward: 9.30
Episode: 860, Running Reward: 9.44
Episode: 870, Running Reward: 9.41
Episode: 880, Running Reward: 9.27
Episode: 890, Running Reward: 9.34
Episode: 900, Running Reward: 9.32
Episode: 910, Running Reward: 9.27
Episode: 920, Running Reward: 9.36
Episode: 930, Running Reward: 9.29
Episode: 940, Running Reward: 9.23
Episode: 950, Running Reward: 9.15
Episode: 960, Running Reward: 9.25
Episode: 970, Running Reward: 9.28
Episode: 980, Running Reward: 9.18
Episode: 990, Running Reward: 9.36
Episode: 1000, Running Reward: 9.34
Reached the maximum episode limit of 1000. Training stopped.
```

Chương 4. TỔNG KẾT

4.1. Ưu điểm

- **Hiệu quả trong việc học chính sách**: Actor-Critic có khả năng học chính sách trực tiếp thông qua thành phần Actor, giúp tối ưu hóa hành động trong môi trường phức tạp.
- **Giảm phương sai**: Thành phần Critic giúp ước lượng giá trị hành động, giảm phương sai trong quá trình cập nhật chính sách, làm cho việc học ổn định hơn.
- **Khả năng áp dụng rộng rãi**: Thuật toán này có thể áp dụng cho nhiều loại môi trường khác nhau, từ các bài toán đơn giản đến các bài toán phức tạp hơn như trò chơi hoặc robot học

4.2. Hạn chế

- **Phức tạp trong việc triển khai:** Actor-Critic yêu cầu việc triển khai hai mạng nơ-ron riêng biệt (Actor và Critic), điều này có thể làm tăng độ phức tạp và yêu cầu tài nguyên tính toán cao hơn.
- **Khả năng hội tụ chậm:** Trong một số trường hợp, thuật toán có thể hội tụ chậm hoặc không ổn định nếu không được điều chỉnh đúng cách.
- **Cần điều chỉnh nhiều siêu tham số:** Việc điều chỉnh các siêu tham số như tốc độ học, hệ số chiết khấu, và các tham số khác có thể phức tạp và tốn thời gian.

4.3. Hướng phát triển

- **Tối ưu hóa hiệu suất và ổn định:**
    - Giảm phương sai và tăng tốc độ hội tụ: Các nghiên cứu đang tập trung vào việc giảm phương sai trong quá trình cập nhật và tăng tốc độ hội tụ của thuật toán Actor-Critic

- Học không giám sát và bán giám sát: Kết hợp học không giám sát và bán giám sát để cải thiện khả năng học của Actor-Critic trong các môi trường phức tạp và không xác định

- **Kết hợp với các công nghệ khác:**
  - Học sâu (Deep Learning): Kết hợp Actor-Critic với các mô hình học sâu để xử lý các bài toán có không gian trạng thái lớn và phức tạp.
  - Học tăng cường đa tác tử (Multi-Agent Reinforcement Learning): Phát triển các hệ thống Actor-Critic có khả năng tương tác và học hỏi từ nhiều tác tử khác nhau trong cùng một môi trường.
- **Cải thiện khả năng giải thích và minh bạch:**
  - Giải thích quyết định: Nghiên cứu các phương pháp giúp giải thích quyết định của Actor-Critic, làm cho các hệ thống này trở nên minh bạch và dễ hiểu hơn cho người dùng.
  - Đảm bảo đạo đức và trách nhiệm: Đảm bảo các thuật toán Actor-Critic hoạt động theo các tiêu chuẩn đạo đức và có trách nhiệm trong các ứng dụng thực tế.

BẢNG PHÂN CÔNG CÔNG VIỆC

Bảng 5. 1

Công việc	Anh Quí	Thanh Linh	Lê Tứ	Hoài Nam
Lý do chọn đề tài		**X**		
Mô tả bài toán, dữ liệu	**X**			
Cơ sở lý thuyết			**X**	
Thuật toán Actor Critic cơ bản và code	**X**			
Actor Critic với GAE		**X**		
Actor Critic với DPG			**X**	
Actor Critic với MCTS				**X**
Ưu, nhược điểm và hướng phát triển				**X**
	100%	**100%**	**100%**	**100%**

TÀI LIỆU THAM KHẢO

[1] Sutton, R. S., & Barto, A. G. (2018). Reinforcement learning: An introduction. MIT press.

[2] Konda, V. R., & Tsitsiklis, J. N. (2000). Actor-critic algorithms. In Advances in neural information processing systems (pp. 1008-1014).

[3] Kaelbling, L. P., Littman, M. L., & Moore, A. W. (1996). Reinforcement learning: A survey. Journal of artificial intelligence research, 4, 237-285.

[4] Kochenderfer, M. J., Wheeler, T. A., & Wray, K. H. (2022). Algorithms for Decision Making. MIT Press.

[5] Bhatnagar, S., Sutton, R. S., Ghavamzadeh, M., & Lee, M. (2009). Natural Actor-Critic Algorithms. Automatica,45(11), 2471-2482. doi:10.1016/j.automatica.2009.07.008.

[6] Code mẫu Actor Critic cơ bản: actor_critic_cartpole - Colab