REINFORCEMENT LEARNING

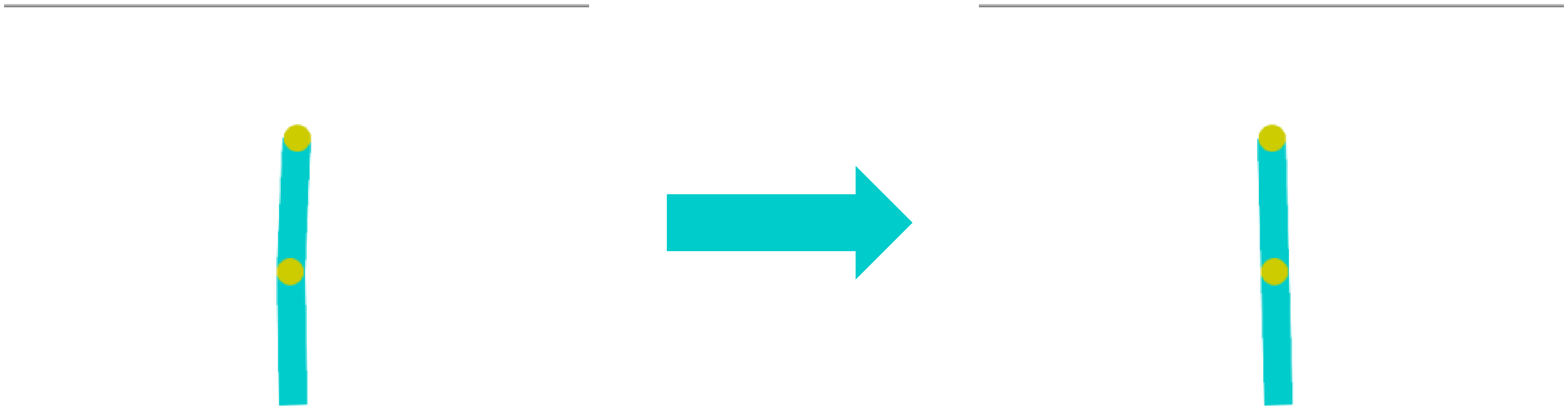
Hyundai Motors Boot-campus

UKJO HWANG, DOHYEOK KWON
INFORMATION AND INTELLIGENCE SYSTEMS LAB.
ELECTRONIC ENGINEERING, HANYANG UNIVERSITY

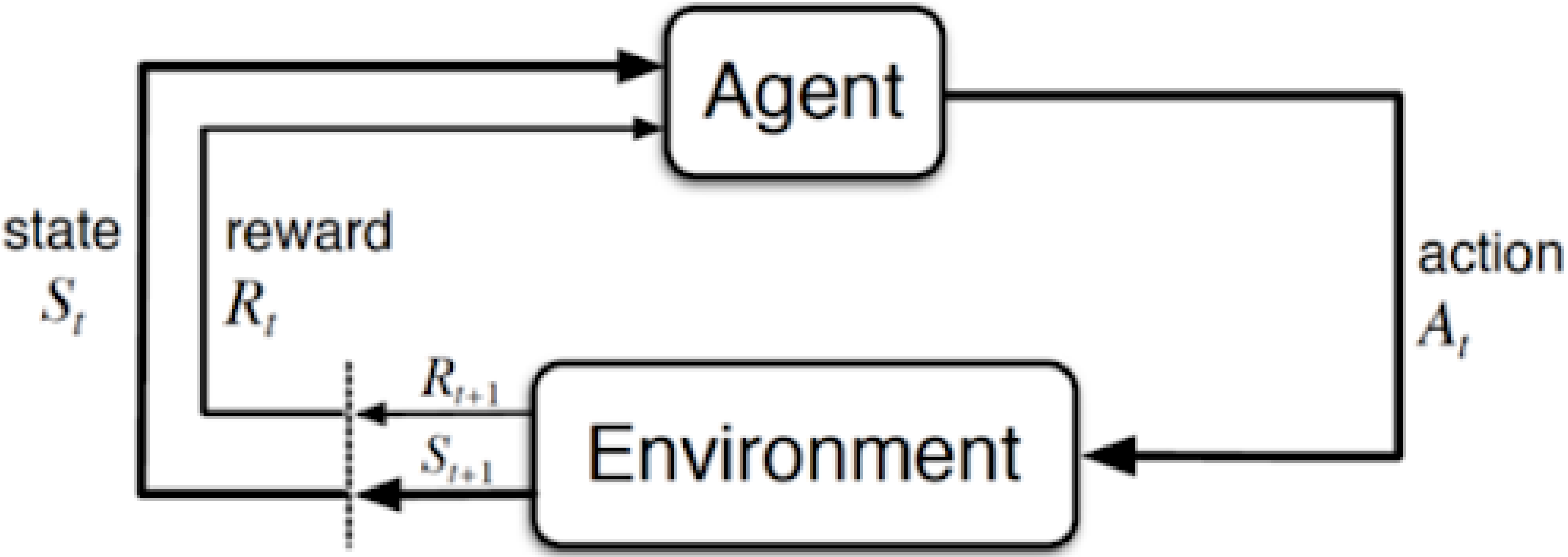
July 7, 2025

GOAL

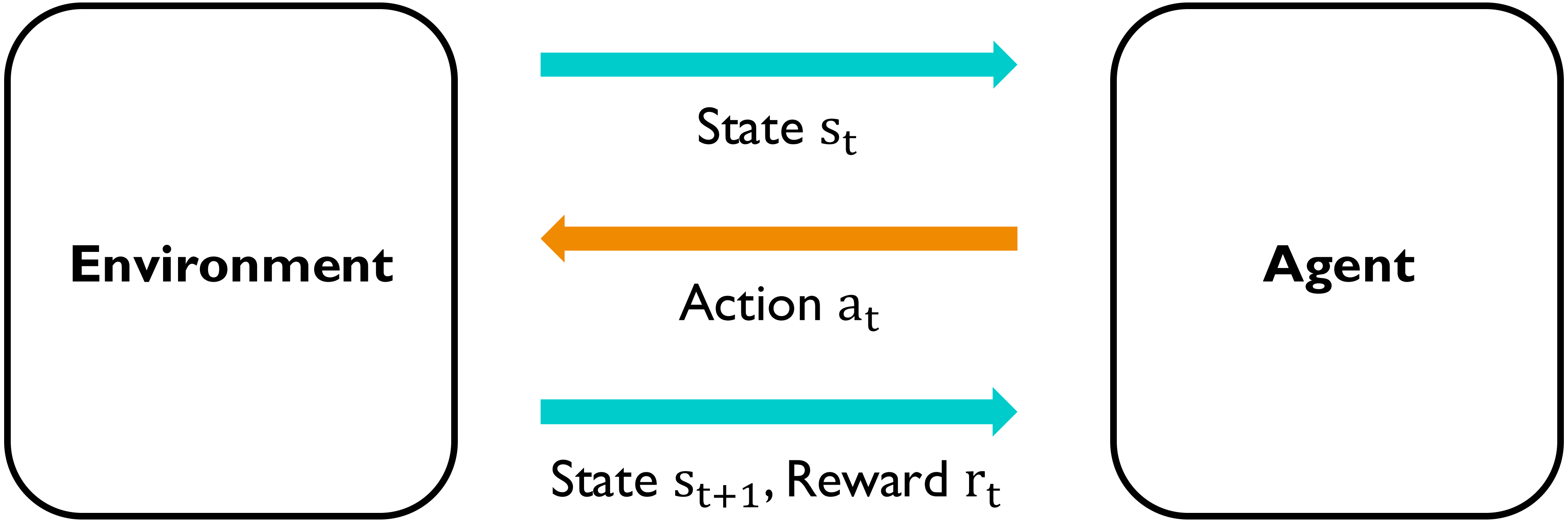
- **How to implement deep RL algorithms**
  - How to utilize environments using the OpenAI Gym API
  - How to train neural networks for reinforcement learning tasks
  - How to train agents using DQN and DDPG algorithms



ENVIRONMENT AGENT INTERACTION



ENVIRONMENT AGENT INTERACTION



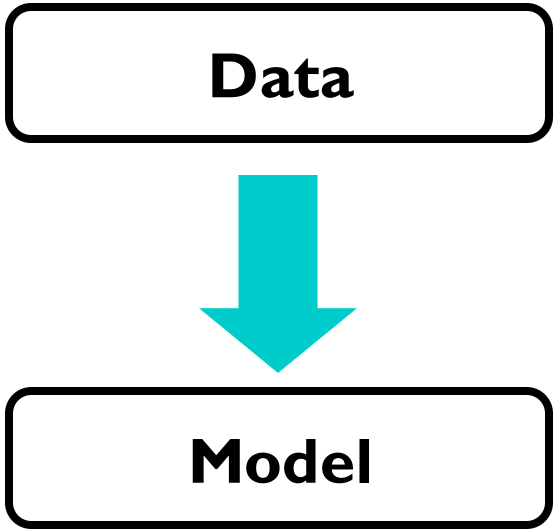
– The environment provides the agent with information about the current **state** and **reward**.

– The agent selects an **action** based on its policy.

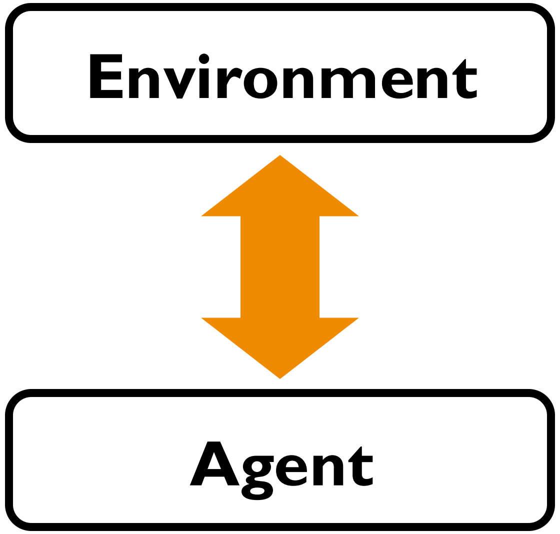
– The environment transitions to a new state according to its **state transition probabilities.**

Supervised Learning vs Reinforcement Learning

- **Supervised Learning**
  - The data is provided in advance.
  - The distribution and quality of data are given beforehand.



- **Reinforcement Learning**
  - The data is **generated through interactions** between the agent and the environment during the learning process
  - The distribution and quality of data continuously change according to the action (policy).



How to implement RL algorithms

Environment

- What is the shape and structure of the data?
- What functions and APIs does the environment provide?

Agent

- How to design and implement the model (e.g., Neural Network)
- How to train the model using the provided data

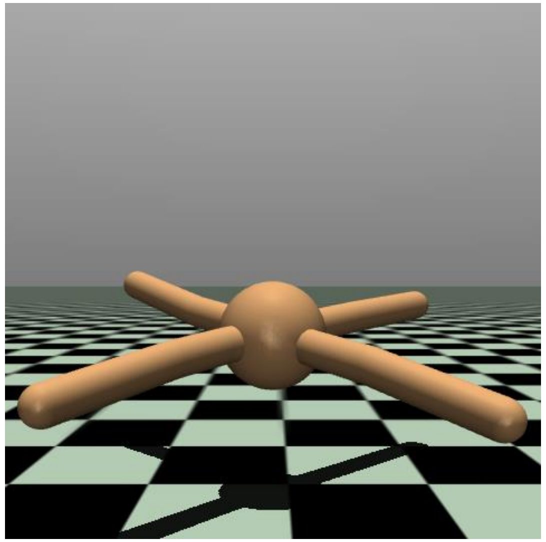
Combination

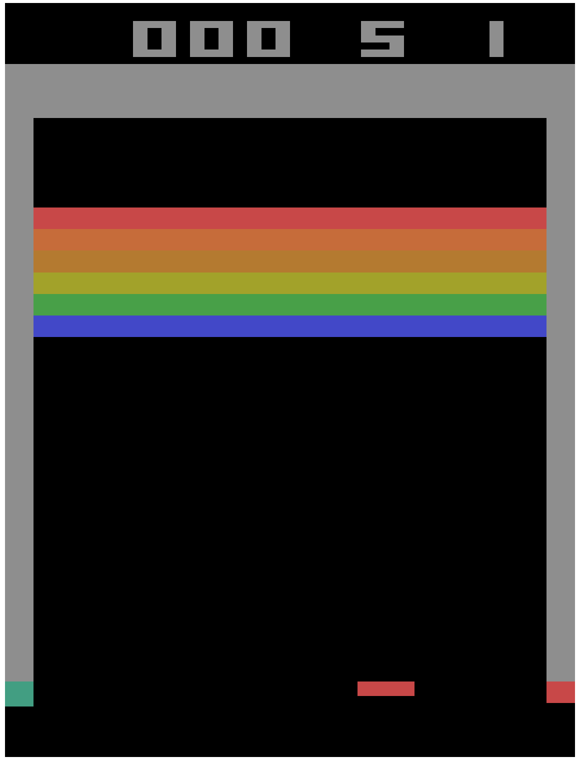
- How the agent interacts with the environment
- How to implement and apply specific RL algorithms

OpenAI Gymnasium

- **OpenAI Gymnasium**
  - Provides a wide range of environments for RL.
  - It's important to understand the basic interface of Gymnasium.
  - To implement a new environment, it's best to follow the Gymnasium API standards.
  - https://gymnasium.farama.org/

Gymnasium





SETUP

- **I. Create Anaconda virtual environment**
  - conda create –n hyundai_rl python=3.10
  - conda activate hyundai_rl
- **II. Install packages**
  - pip install gymnasium matplotlib ipykernel torch

I. Q-LEARNING

BACKGROUND

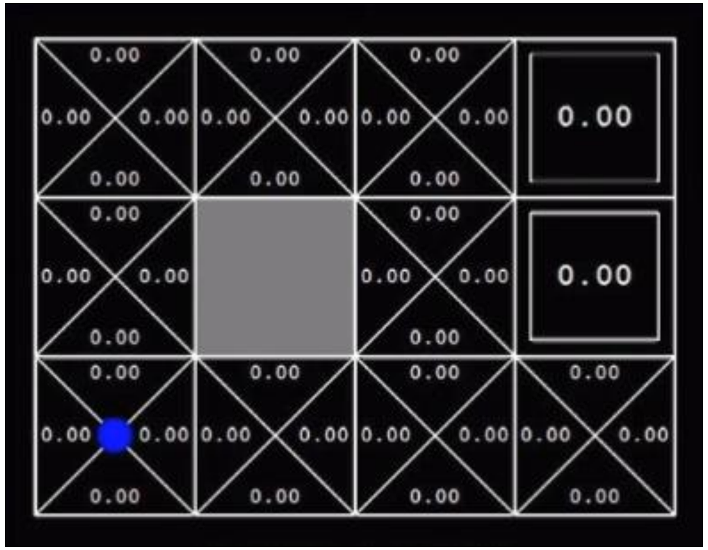
- **Q-Value Function(or State-Action Value Function) $\boldsymbol{Q(S, A)}$**
  - 특정상태 $S(State)$에서 특정 행동 $A(Action)$를 선택했을 때 기대되는 누적 보상.

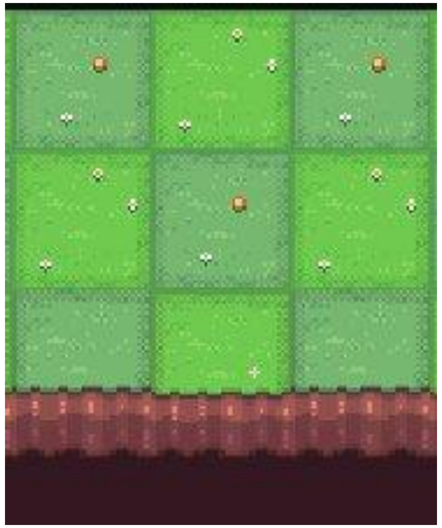
$$Q(S, A) = \mathbb{E}[R_t | s_t = S, a_t = A] = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k} \mid s_t = S, a_t = A\right]$$

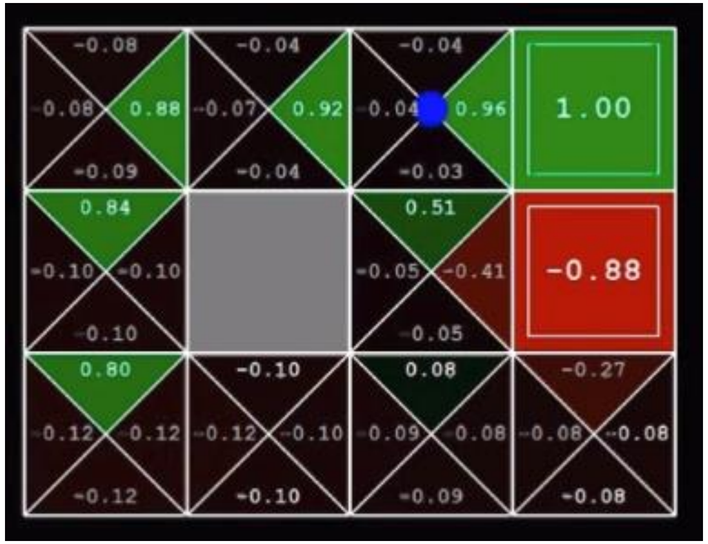
  - 학습을 통해서 모든 state와 action에 대한 Q 값을 찾으면, 특정 state에서 어떤 action을 선택하는 것이 가장 큰 보상을 받을 수 있을지 아는 것과 같음.

Background

- **Q-Table**
  - Q-Value를 저장하는 간단한 방법.
  - 모든 state, action pair에 대해서 Q-value를 저장해야 함.
  - Dimension: $S \times A$







	Left	Right	Up	Down
(1,1)	0	0	0	0
(1,2)	0	0	0	0
(1,3)	0	0	0	0
(2,1)	0	0	0	0
(2,2)	0	0	0	0
(2,3)	0	0	0	0
(3,1)	0	0	0	0
(3,2)	0	0	0	0
(3,3)	0	0	0	0

Q-Learning

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
    Initialize S
    Loop for each step of episode:
        Choose A from S using policy derived from Q (e.g., ε-greedy)
        Take action A, observe R, S'
        $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
        $S \leftarrow S'$
    until S is terminal

Q-LEARNING

- **1. Q-Table 초기화**

$$Q(S, A) = 0 \quad \forall S, A$$

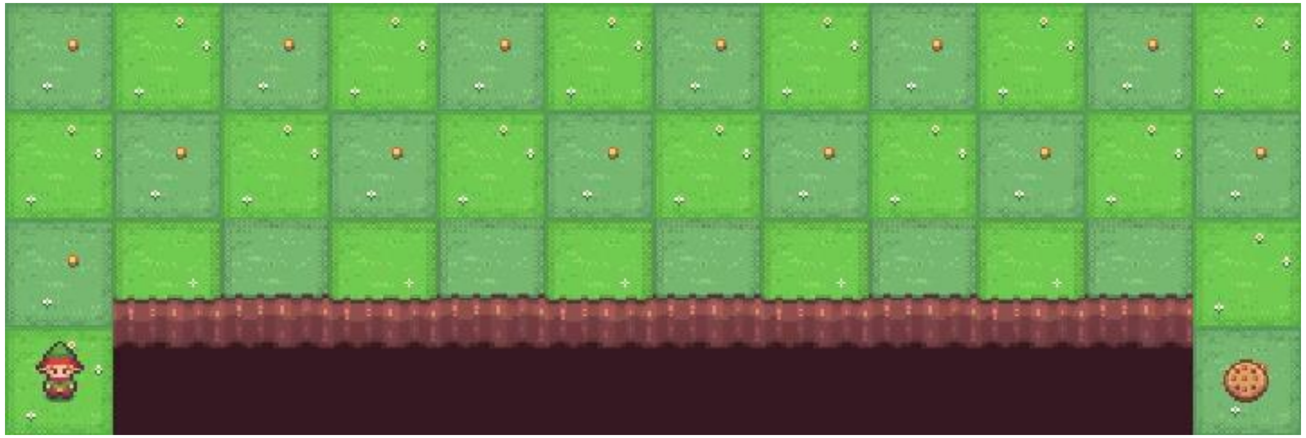
- **2. ϵ-greedy policy**
  - 탐색(Exploration)과 활용(Exploitation)을 적절히 조합하여 다음 action을 선택해야 함.
  - ϵ의 확률로 random action을 선택하고, $1 - \epsilon$의 확률로 현재 state에서 가장 높은 Q-value를 갖는 action 선택

- **3. Q-value update**
  - Bellman equation을 따라 Q-value를 update.

$$Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_{a'} Q(S', a') - Q(S, A)]$$

  - α: Learning rate / γ : Discount factor $\Longrightarrow$ 모두 [0, 1] 사이의 실수 값

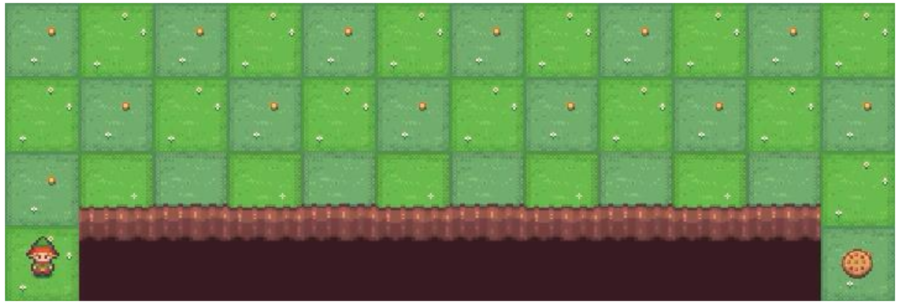
ENV: CLIFF WALKING



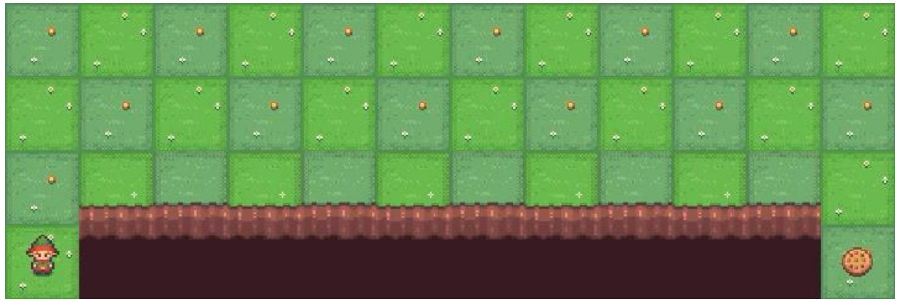
- **Action Space:** Discrete(4) | **Observation Space:** Discrete(48)
  - **[Action]** 0: Move up | 1: Move right | 2: Move down | 3: Move left
- **Starting State:** [36] (3, 0) | **Episode End:** [47] (3, 11)
- **Reward**
  - Each time step incurs -1 reward, unless the player stepped into the cliff, which incurs -100 reward.

ENV: CLIFF WALKING

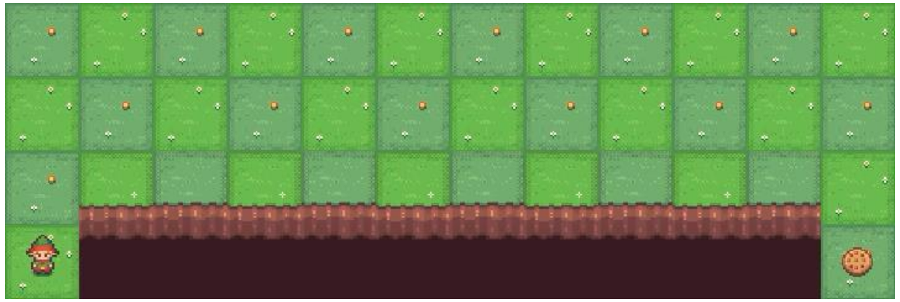
- **[Training]**
  - Episode 0~10 (Average steps: 240.0)



  - Episode 100~110 (Average steps: 38.5)



  - Episode 200~210 (Average steps: 17.1)



Experimental Extensions

- **Exploration probability** 가 더 높은 값이라면?
  - Increase ‘exploration rate’: 0.1→ **0.4**
  - Decrease ‘number of episodes’: 300 → **150**

- 유사한 환경인 `FrozenLake`에서 학습에 **더 많은 step**이 필요한 이유?
  - env = gym.make('FrozenLake-v1', render_mode='rgb_array', is_slippery=False)
  - Increase episodes from 150 to 2000

- **Stochastic transition**의 영향 확인
  - Set ‘is_slippery = True’

EXPERIMENTAL EXTENSIONS

- **Exploration probability** 가 더 높은 값이라면?

- 유사한 환경인 `FrozenLake`에서 학습에 **더 많은 step**이 필요한 이유?
- **Stochastic transition**의 영향 확인

1500

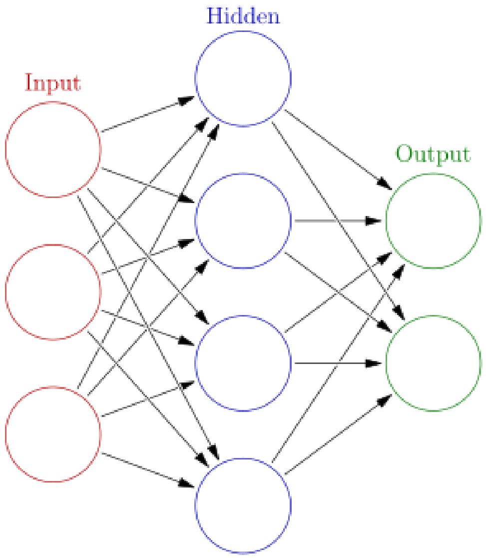
2000

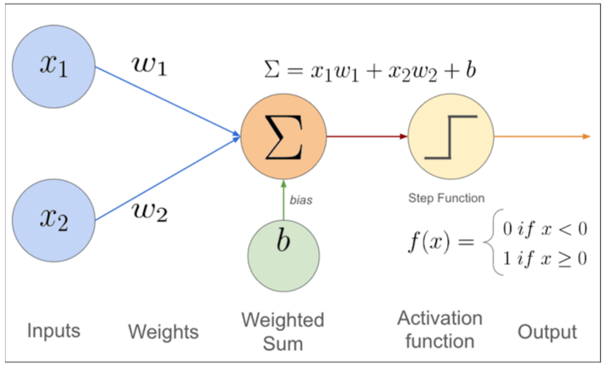
Slippery

II. DQN

DEEP NEURAL NETWORKS

- If the state and action spaces are large, it becomes impractical to implement the Q-function using a table.
- Therefore, deep neural networks are widely used to approximate the Q-function.





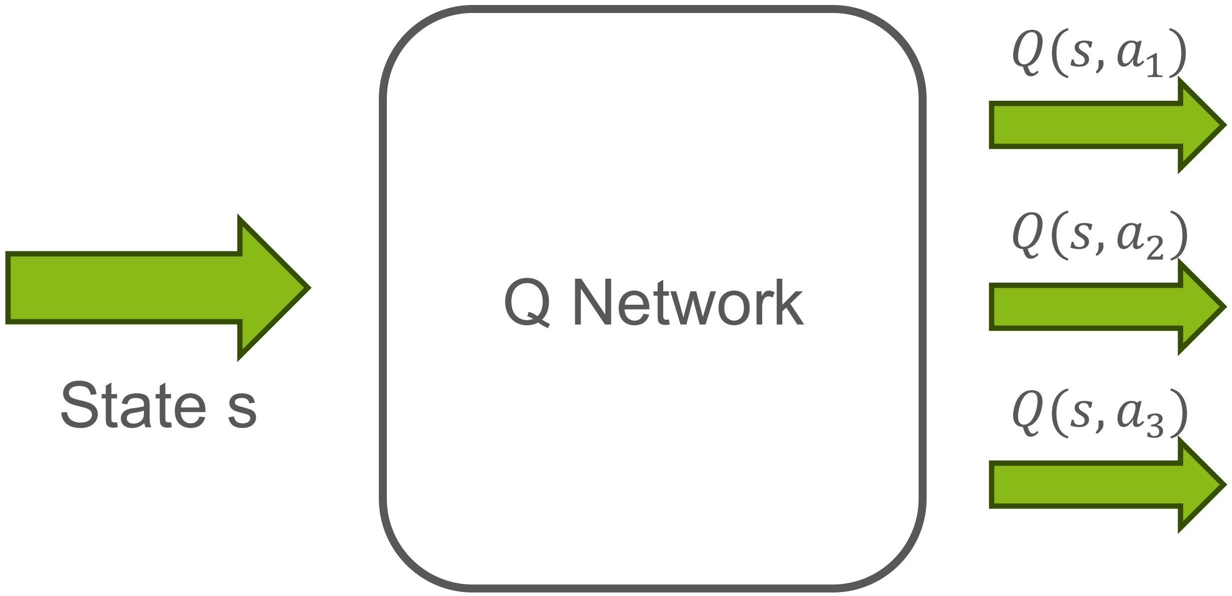
Q-Function

- The output of the DNN is the Q-value corresponding to a given state-action pair.
- Therefore, it is natural to design the DNN to take the state and action as input and output the corresponding Q-value.

Q-FUNCTION

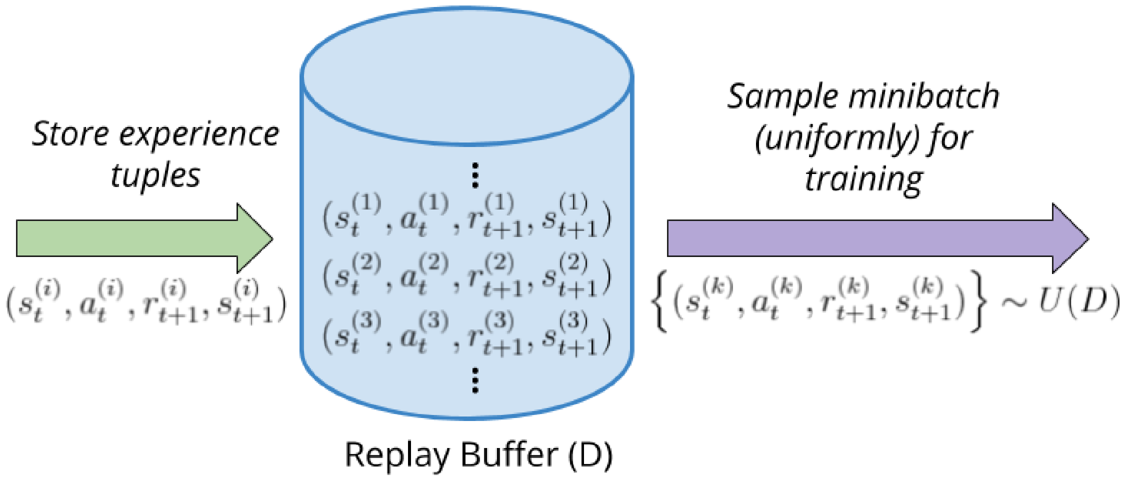
- However, in practice, the DNN is often designed to take only the state as input and output the Q-values for every possible action.
- The feasibility of this implementation relies on the discreteness of the action space.

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot);s'\sim\mathcal{E}}\left[\left(r + \gamma \max_{a'} Q(s',a';\theta_{i-1}) - Q(s,a;\theta_i)\right)\nabla_{\theta_i} Q(s,a;\theta_i)\right]$$



Implementation Details

- Since DQN is an off-policy algorithm, it utilizes a replay buffer to store samples generated while interacting with the environment.
- This design choice is crucial because it helps to reduce strong correlations between samples.



IMPLEMENTATION DETAILS

- DQN employs a target network that gradually updates to follow the parameters of the original network.
- This mechanism contributes to stabilizing the training.

$$\nabla_{\theta_i} L_i\left(\theta_i\right) = \mathbb{E}_{s,a\sim\rho(\cdot);s'\sim\mathcal{E}}\left[\left(r + \gamma \max_{a'} Q(s',a';\theta_{i-1}) - Q(s,a;\theta_i)\right)\nabla_{\theta_i} Q(s,a;\theta_i)\right].$$

DQN Algorithm

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

III. ACTOR-CRITIC ALGORITHM

BACKGROUND

- **Policy-based reinforcement learning**
  - 기존 Value-based algorithm이 어떤 state에서 각 action의 가치를 계산하여 행동했다면, Policy-based algorithm은 각 state에서의 행동 정책(확률)을 학습하는 방식.
    - DQN은 Network의 output으로 특정 (s, a)의 Q-Value를 출력
    - Policy Network는 각 현재 state에서 각 action을 수행할 확률 $\pi(a|s) = P(A_t = a \mid S_t = s)$를 출력.

- **Policy gradient**
  - Objective function: $J(\theta) = V_{\pi_\theta}(s_0)$ (To maximize it, $\theta_{t+1} = \theta_t + \alpha \nabla_\theta J(\theta)$)
    - $V_{\pi_\theta}(s) = \mathbb{E}_{\pi_\theta}\left[R_{t+1} + \gamma V_{\pi_\theta}(S_{t+1}) \,\middle|\, S_t = s\right]$ (Expected return)
  - **Policy gradient theorem:** $\boldsymbol{\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[r(\tau) \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)]}$

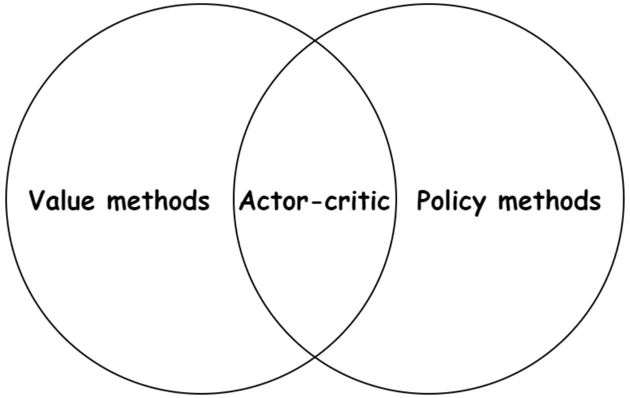
BACKGROUND

- **Policy gradient**
  - **Policy gradient theorem:** $\boldsymbol{\nabla_\theta J(\theta)} = \mathbb{E}_{\pi_\theta}[\boldsymbol{r(\tau)} \sum_{t=0}^{T-1} \boldsymbol{\nabla_\theta \log \pi_\theta(a_t|s_t)}]$

$$
\begin{aligned}
\nabla_\theta J(\theta) &= \mathbb{E}_{\pi_\theta}[\textstyle\sum_{t=0}^{T-1} G_t \nabla_\theta \log \pi_\theta(a_t \mid s_t)] && \text{REINFORCE (Monte Carlo PG)} \\
&= \mathbb{E}_{\pi_\theta}[\textstyle\sum_{t=0}^{T-1} (G_t - V_\phi(s_t)) \nabla_\theta \log \pi_\theta(a_t \mid s_t)] && \text{REINFORCE with baseline} \\
&= \mathbb{E}_{\pi_\theta}[Q_\phi(s, a) \nabla_\theta \log \pi_\theta(a \mid s)] && \text{Q-value Actor-Critic} \\
&= \mathbb{E}_{\pi_\theta}[A_{\phi_1,\phi_2}(s, a) \nabla_\theta \log \pi_\theta(a \mid s)] && \text{Advantage Actor-Critic} \\
&= \boxed{\mathbb{E}_{\pi_\theta}[(r + \gamma V_\phi(s') - V_\phi(s)) \nabla_\theta \log \pi_\theta(a \mid s)] \quad \text{TD Actor-Critic}}
\end{aligned}
$$

Critic (value function) Actor (policy)

ACTOR-CRITIC



- **Actor (Policy Network)**
  - 각 state s에서 다음 액션에 대한 확률(정책- $\pi(s|a)$)을 근사하는 Network
  - Loss: $J(\theta)$
- **Critic (Value Network)**
  - 각 state s에서의 가치(Value function)를 근사하는 Network.
  - Loss: MSE between $V(s)$ and TD target $(r + \gamma V(s'))$

Input : Initial Actor policy parameters $\pi(a|s,\theta)$
: Initial Critic V-function parameters $V(s,\psi)$
Parameters : actor learning rate $\alpha^{\theta} > 0$, critic learning rate $\alpha^{\psi} > 0$
trial_step $= T$
For episode $= 1, M$ **do**
    done $= False$
    Reset environment state.
    For $t = 1, T$ **do** (or **While not** $done$)
        Observe state s and select action $a = \mu_{\theta}(s)$,   # discrete model
        Execute a in the environment
        Observe next state s', reward r, and done signal d to indicate whether s' is terminal
        Reset gradient $d\theta$ and $d\psi$ to 0
        Calculate the TD

$$\boxed{TD \leftarrow r + \gamma(1-d)V(s',\psi) - V(s,\psi) \quad (\text{if } s' \text{ is terminal, then } V(s',\psi) \doteq 0)}$$

where, $Q_{expected} \leftarrow r + \gamma(1-d)V(s',\psi)$

        Accumulate the policy gradient using the critic:

$$d\theta \leftarrow d\theta + \nabla_{\theta} \log \pi_{\theta}(s_t, a_t)(TD)$$

        Accumulate the critic gradient:

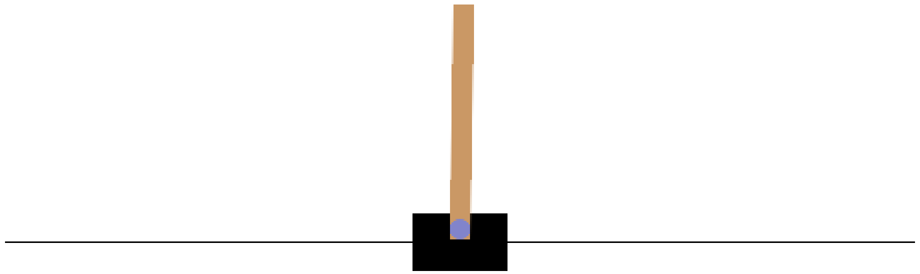
$$d\psi \leftarrow d\psi + \nabla_{\psi}(TD)^2$$

        Update the actor and the critic with the accumulated gradients using gradient descent or similar :

$$\theta \leftarrow \theta + \alpha^{\theta} d\theta \qquad \psi \leftarrow \psi + \alpha^{\psi} d\psi$$

        $s \leftarrow s'$
    End For
End For

Env: Cart Pole

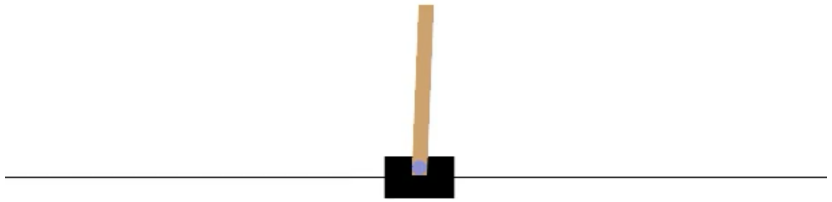


Num	Observation	Min	Max
0	Cart Position	-4.8	4.8
1	Cart Velocity	-Inf	Inf
2	Pole Angle	~ -0.418 rad (-24°)	~ 0.418 rad (24°)
3	Pole Angular Velocity	-Inf	Inf

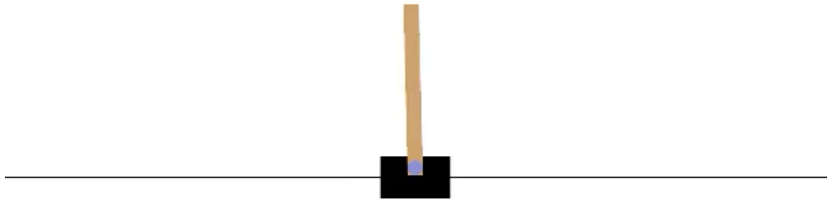
- **Action Space:** Discrete(2) | **Observation Space:** Continuous(4,)
  - **[Action]** 0: Push cart to the left | 1: Push cart to the right
- **Episode End:** Angle $\pm 12°$ / End of display / 500 steps
- **Reward**
  - A +1 reward is given for every step, including the terminal step.

Env: Cart Pole

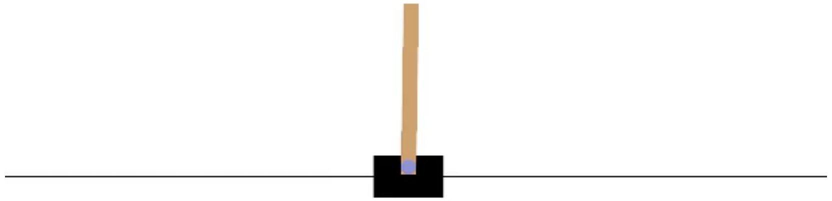
- **[Training]**
  - Episode 100~200 (Average steps: 60.6)



  - Episode 200~210 (Average steps: 144.6)



  - Episode 500~510 (Average steps: 282.0)



IV. DDPG

DQN vs DDPG

- DQN
  - Designed for environments with discrete action spaces
  - A Q-network alone is sufficient for learning the policy

$$\pi(s) = \max_{a \in \mathcal{A}} Q^{\pi}(s, a)$$

$$Q^*(s,a) = \mathbb{E}_{s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q^*(s', a') \middle| s, a \right]$$

- DDPG
  - Suitable for environments with continuous action spaces
  - Requires both a policy network (Actor) and a Q-network (Critic)

$$Q^{\mu}(s_t, a_t) = \mathbb{E}_{r_t, s_{t+1} \sim E} \left[r(s_t, a_t) + \gamma Q^{\mu}(s_{t+1}, \mu(s_{t+1})) \right]$$

Deterministic Policy

- Deterministic policies simplify the computation of gradients during the optimization of the policy network.

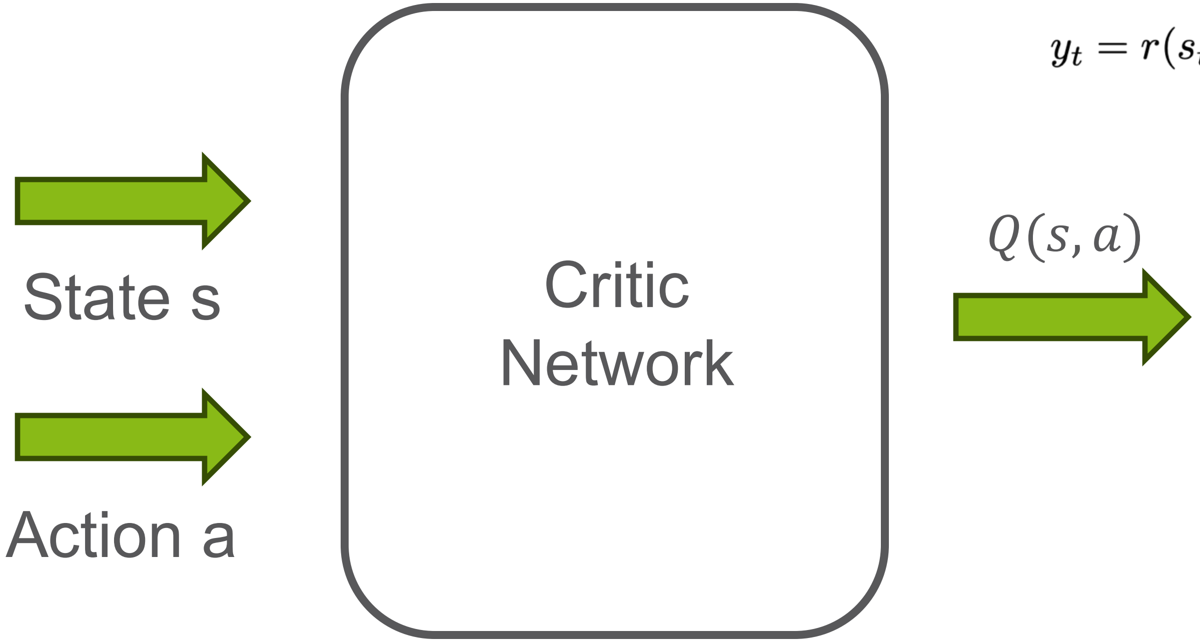
$$
\begin{aligned}
\nabla_{\theta^\mu} J &\approx \mathbb{E}_{s_t \sim \rho^\beta} \left[\nabla_{\theta^\mu} Q(s, a|\theta^Q) |_{s=s_t, a=\mu(s_t|\theta^\mu)} \right] \\
&= \mathbb{E}_{s_t \sim \rho^\beta} \left[\nabla_a Q(s, a|\theta^Q) |_{s=s_t, a=\mu(s_t)} \nabla_{\theta_\mu} \mu(s|\theta^\mu) |_{s=s_t} \right]
\end{aligned}
$$

CRITIC

- The critic network receives the state and action as input and produces the corresponding Q-value as output.

$$L(\theta^Q) = \mathbb{E}_{s_t \sim \rho^\beta, a_t \sim \beta, r_t \sim E}\left[\left(Q(s_t, a_t|\theta^Q) - y_t\right)^2\right]$$

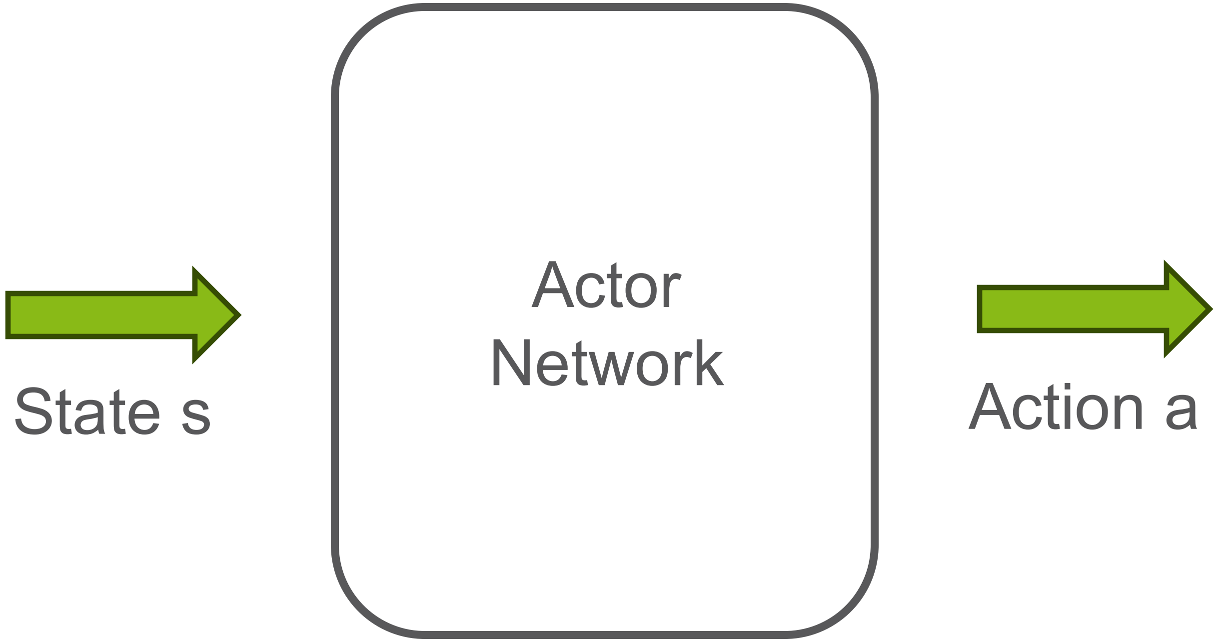
$$y_t = r(s_t, a_t) + \gamma Q(s_{t+1}, \mu(s_{t+1})|\theta^Q).$$



Actor

- The actor receives the state as input and produces the appropriate action for the agent.

$$\begin{aligned} \nabla_{\theta^\mu} J &\approx \mathbb{E}_{s_t \sim \rho^\beta}\left[\nabla_{\theta^\mu} Q(s, a|\theta^Q)|_{s=s_t, a=\mu(s_t|\theta^\mu)}\right] \\ &= \mathbb{E}_{s_t \sim \rho^\beta}\left[\nabla_a Q(s, a|\theta^Q)|_{s=s_t, a=\mu(s_t)} \nabla_{\theta_\mu} \mu(s|\theta^\mu)|_{s=s_t}\right] \end{aligned}$$

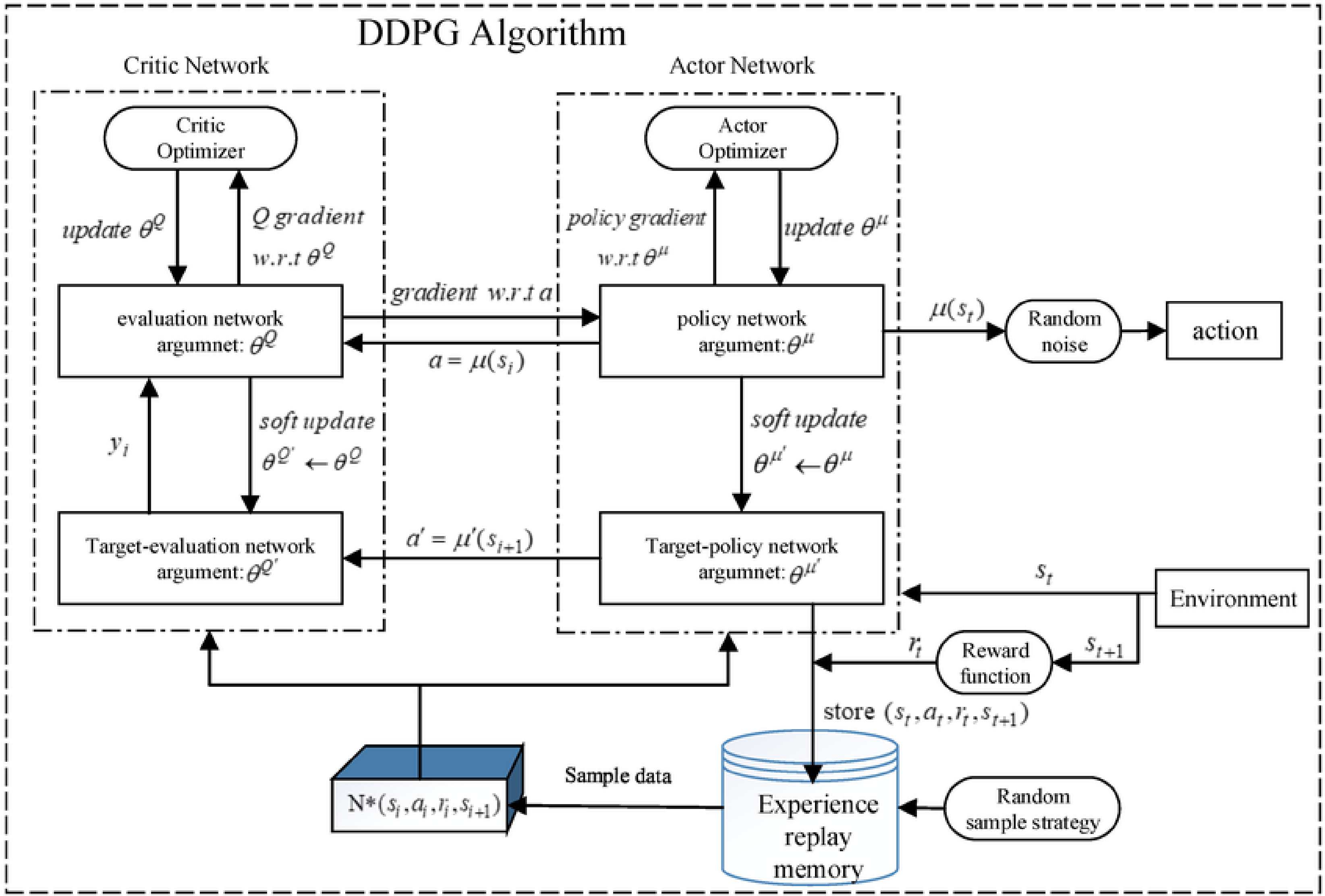


IMPLEMENTATION DETAILS

- Due to the deterministic nature of the policy, the agent may fail to explore enough, which can hinder effective training.
- Therefore, a noise component is incorporated into the agent's policy during training, allowing the agent to explore the environment more effectively.

$$a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$$

DDPG Structure



DDPG ALGORITHM

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$

$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for