Reinforcement Learning

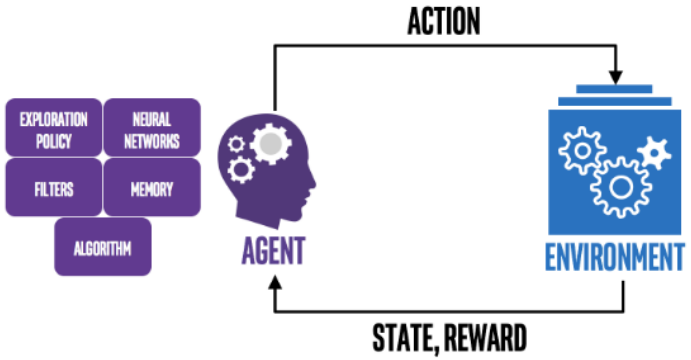
2023. 10

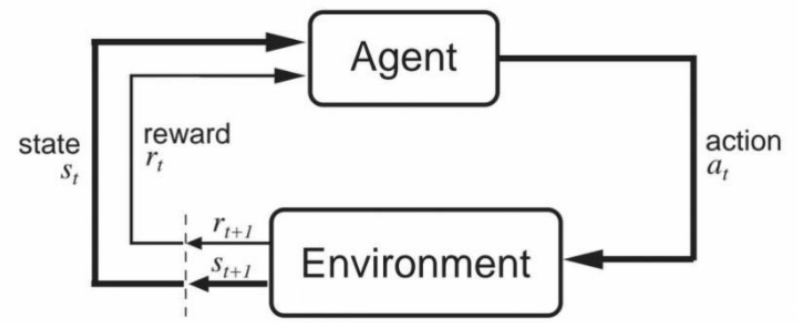
Yongjin Jeong, KwangWoon University

Good reference book site (in Korean): https://dnddnjs.gitbooks.io/rl/

Reinforcement Learning

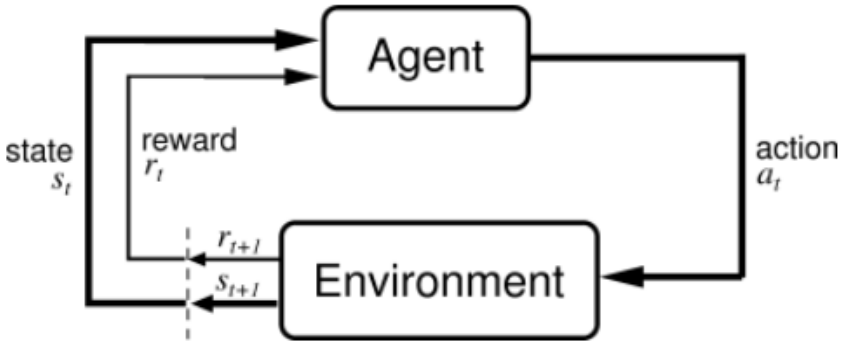
- **Reinforcement Leaning (RL)**
  - Concerned with sequential decision making process
  - Features an agent learning how to act or decide in an environment



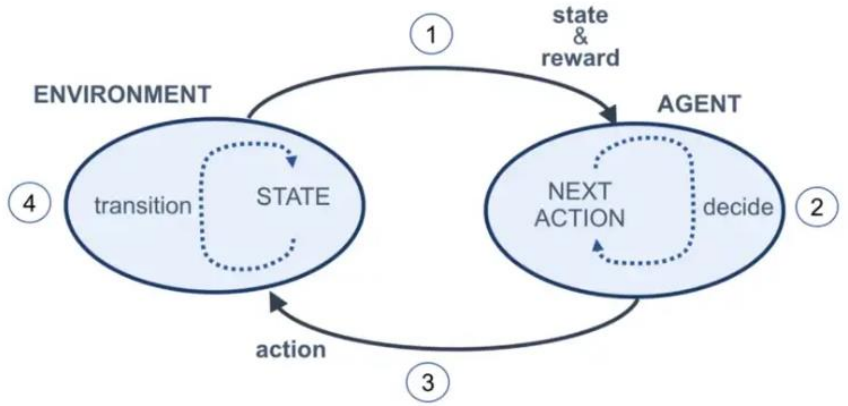


- **Agent** – The program to train (Neural network)
- **Environment** – The world (real or virtual), in which the agent performs actions
- **State** – How the agent sees the environment
- **Action** – A move made by the agent, which causes a status change in the environment
- **Rewards** – The evaluation of an action, which can be positive or negative

Terminologies



Agent-environment interaction [Source]



- **Agent:** It is an assumed entity which performs actions in an environment to gain some reward (robots or human or programs)
- **Environment (e):** A scenario that an agent has to face.
- **Reward (R):** An immediate return given to an agent when he or she performs specific action or task.
- **Action (A):** a set of actions which the agent can perform
- **State (s):** State refers to the current situation returned by the environment.
- **Policy (π):** It is a strategy applied by the agent to decide the next action based on the current state, in other words, a mapping from state *s* to action *a,* where π(*a*/*s*) is the probability of taking action *a* when in the state *s. (either deterministic or stochastic)*

Terminologies

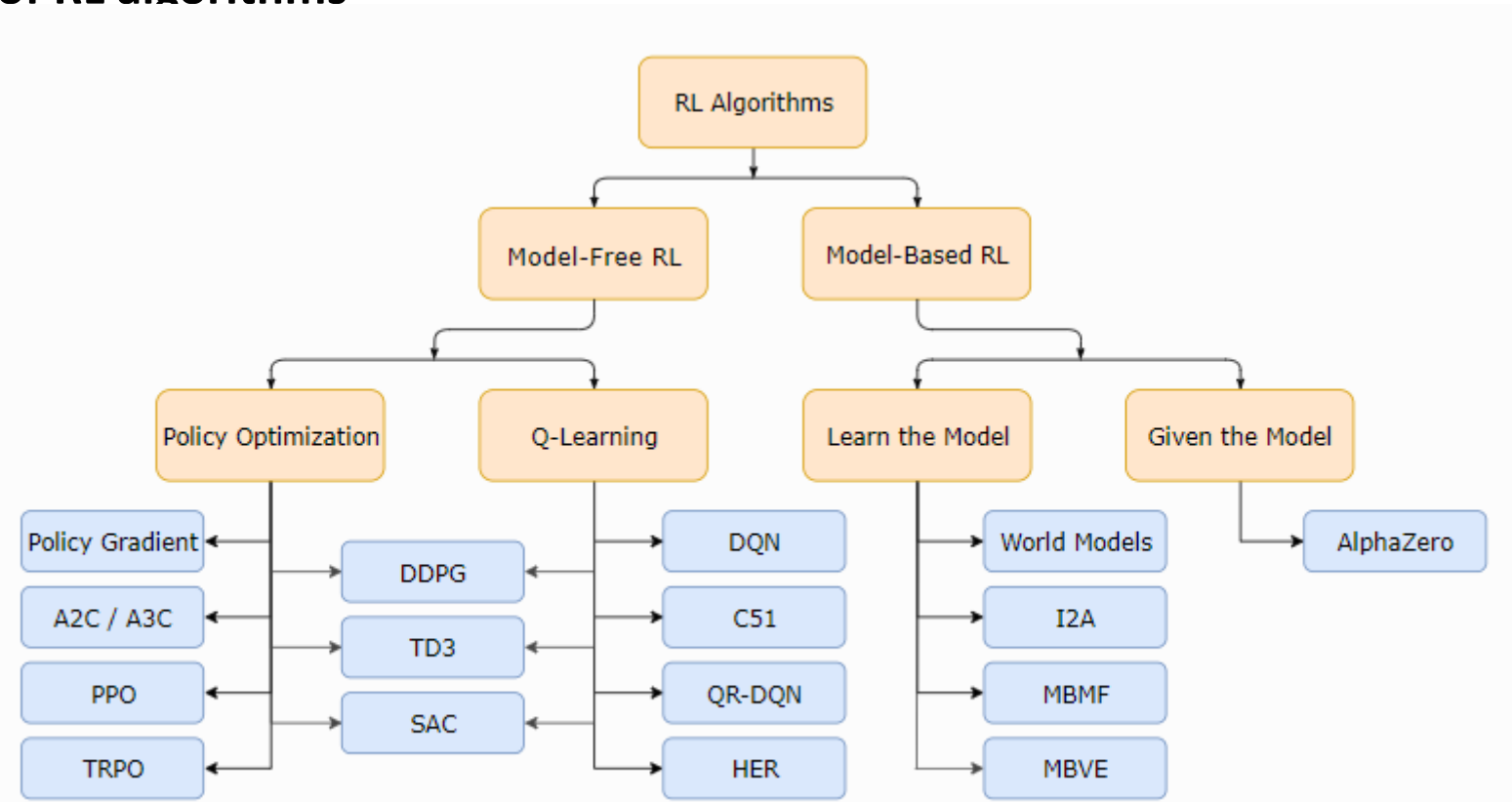
- **Value (V):** It is expected long-term return with discount
- **Value Function:** It specifies the value of a state that is the total amount of reward, which should be expected beginning from that state.
- **Q value or action value (Q):** Q value is quite similar to value. The only difference between the two is that it takes an additional parameter as a current **action**.
- **Model of the environment:** This mimics the behavior of the environment. By a model of the environment, we mean a function that predicts **state transitions and rewards**.
- **Model-Free vs. Model-Based RL**
  - **Model-based:** after learning, the agent can make predictions about what the next state and reward will be before it takes each action. **Dynamic Programming (policy iteration and value iteration)** use the model's predictions or distributions of next state and reward in order to calculate optimal actions.
  - **Model-free: no knowledge of MDP transitions and rewards - a ground-truth model of the environment is usually not available to the agent.** If an agent wants to use a model, it has to learn the model **purely from experience**. **Monte Carlo, Temporal Difference (TD), SARSA, Q-learning, Actor-Critic** rely on real samples from the environment.

RL Methods

- **Dynamic programming (DP) —** good starting point to understand RL algorithms that can solve more complex problems
- **Monte Carlo methods** — A class of methods for learning of value functions, which estimates the value of a state by running many trials starting at that state, then averages the total rewards received on those trials. (episode 마다 update)
- **Temporal Difference (TD) algorithms** — A class of learning methods, based on the idea of comparing temporally successive predictions. Possibly the single most fundamental idea in all of reinforcement learning. (time step 마다 update)
- **What to Learn?**
  - **policies**, either stochastic or deterministic,
  - **action-value functions** (Q-functions),
  - **value functions**, (V)
  - and/or environment models.

Taxonomy

- **OpenAI** — a non-profit AI research company with the mission to build and share safe Artificial General Intelligence (AGI) — launched a program to "spin up" deep RL. (https://spinningup.openai.com/en/latest/index.html)
- **Taxonomy of RL algorithms**



A non-exhaustive, but useful taxonomy of algorithms in modern RL. Citations below.

Real Life Applications

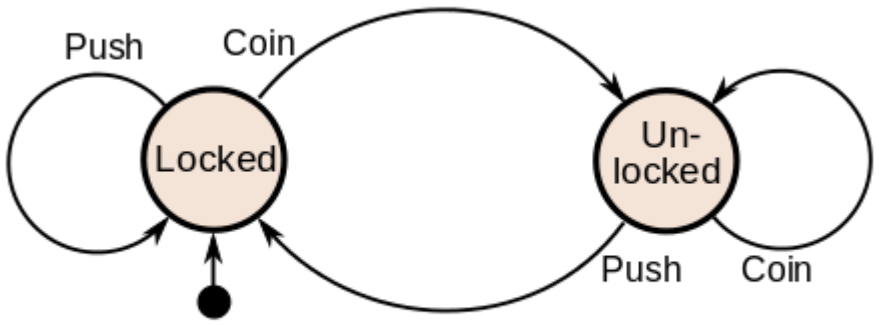
- **Self-driving cars:** trajectory optimization, motion planning, dynamic pathing, controller optimization, and scenario-based learning policies for highways.
- **Industry automation:** energy saving (Google), industrial robots
- **Trading and finance:** better stock exchange than time series model (supervised learning)
- **NLP:** text summarization, question answering, machine translation
- **healthcare:** medical treatment options for every disease stage
- **engineering:** optimize video streaming quality based on the state of video buffer, etc.
- **News recommendation:** track reader's return behavior
- **Gaming:** AlphaGo Zero
- **Marketing and advertisement:** real-time bidding with multi-agent RL
- **Robotics manipulation:** Boston Dynamics robots
- See https://neptune.ai/blog/reinforcement-learning-applications

Reinforcement Learning Algorithms

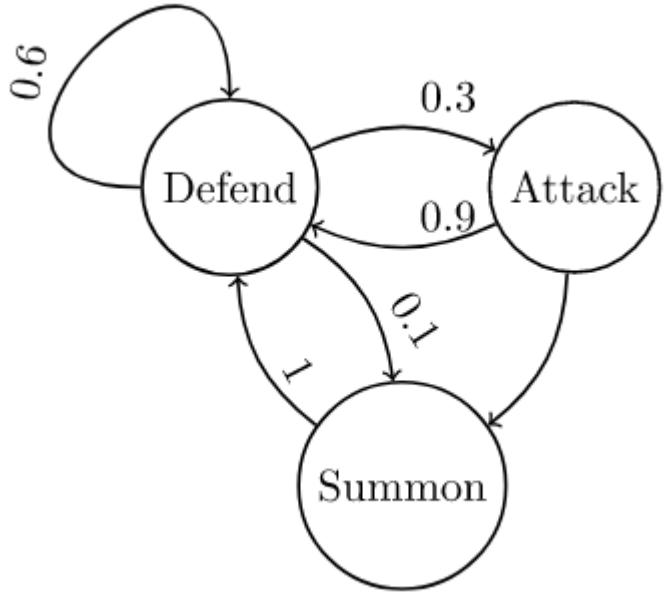
- **Value-based:** aim to find the optimal value function, often denoted as **"Q(s, a)" or "V(s),“**
  - Once the optimal value function is learned, the optimal policy can be derived directly from it.
  - (ex) Q-learning, Deep Q-Networks (DQN), SARSA
  - It is effective in environments with large state and action spaces. They are also known for their stability and ease of convergence.
- **Policy-based:** optimize the policy function **"π(a|s)"** directly
  - It directly parameterize the policy itself, which is a mapping from states to actions. This can be represented as "π(a|s)," where π is the policy function.
    - Deterministic: For any state, the same action is produced by the policy π.
    - Stochastic : Every action has a certain probability.
  - (ex) REINFORCE
  - more suitable for problems with high-dimensional action spaces and complex, continuous action domains.
- **Model-based**
  - You need to create a virtual model for each environment. The agent learns to perform in that specific environment.

Finite State Machine and Markov Chain

- Finite State Machine (FSM) – Deterministic
- Markov Chain – Stochastic (Probabilistic, Random)

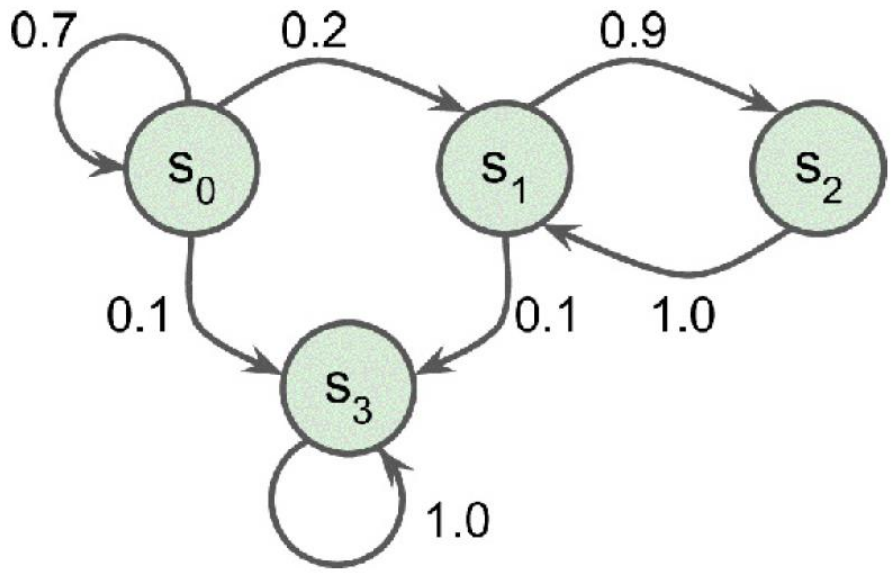


State diagram for a turnstile



Markov Chain (or Markov Process)

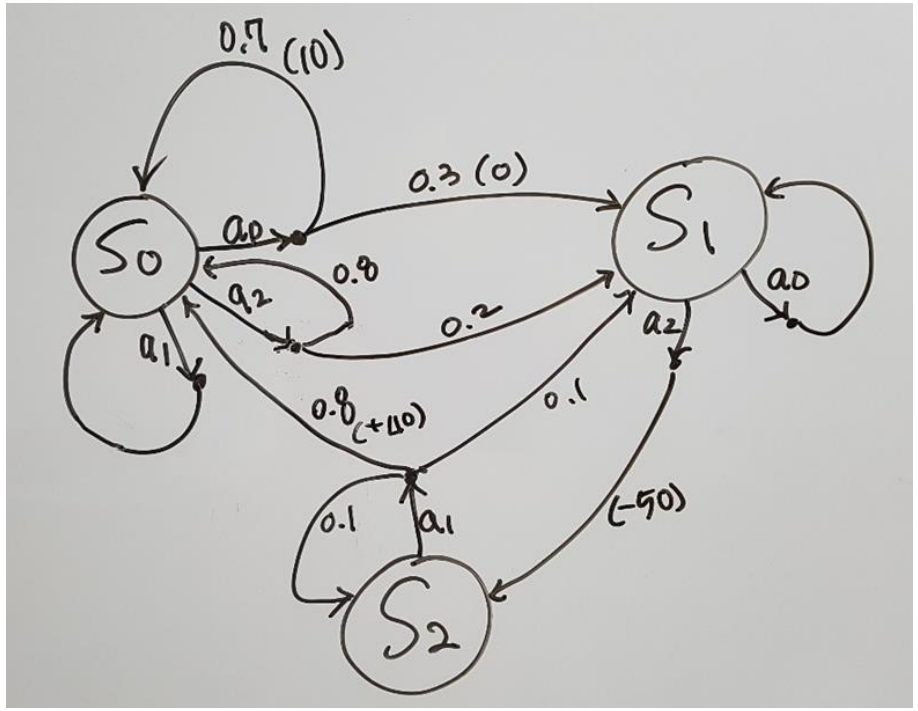
- Stochastic processes with no memory
- Has a **fixed number of states**
- **Probability** from a state s to s' is fixed, and it depends only on the pair (s,s'), not on the past state (the system has no memory)
- **Markov Process**: continuous-time version of a Markov chain




```
transition_probabilities = [
        [0.7, 0.2, 0.0, 0.1],
        [0.0, 0.0, 0.9, 0.1],
        [0.0, 1.0, 0.0, 0.0],
        [0.0, 0.0, 0.0, 1.0]]
```

Markov Decision Processes (MDP)

- They resemble Markov chains but with a twist: at each step, **an agent can choose one of several possible actions**, and
- The state transition probabilities depend on the chosen action. Moreover, some state transitions return some reward (positive or negative), and
- The agent's goal is to find a policy that will maximize rewards over time.



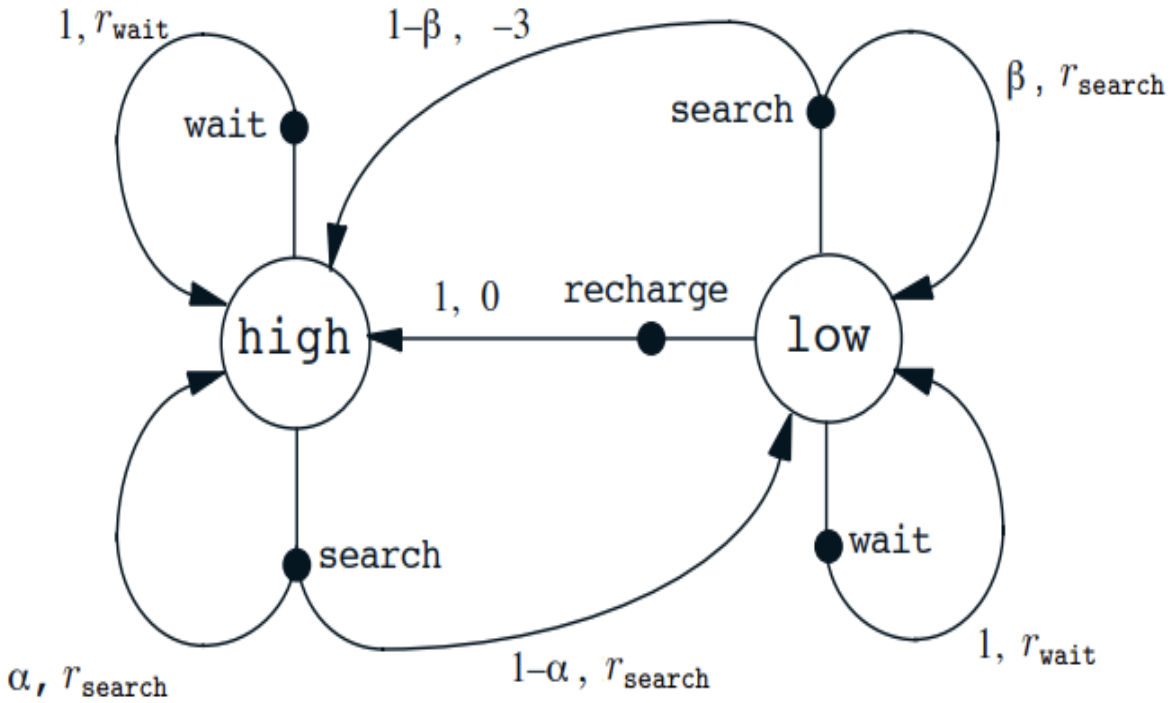

```
transition_probabilities = [
        [[0.7, 0.3, 0.0],
         [1.0, 0.0, 0.0],
         [0.8, 0.2, 0.0]],

        [[0.0, 1.0, 0.0],
         None,
         [0.0, 0.0, 1.0]],

        [None,
         [0.8, 0.1, 0.1],
         None]]
```

```
rewards = [ # shape=[s, a, s']
        [[+10, 0, 0],
         [0, 0, 0],
         [0, 0, 0]],

        [[0, 0, 0],
         [0, 0, 0],
         [0, 0, -50]],

        [[0, 0, 0],
         [+40, 0, 0],
         [0, 0, 0]]]
```

```
possible_actions = [[0, 1, 2], [0, 2], [1]]
```

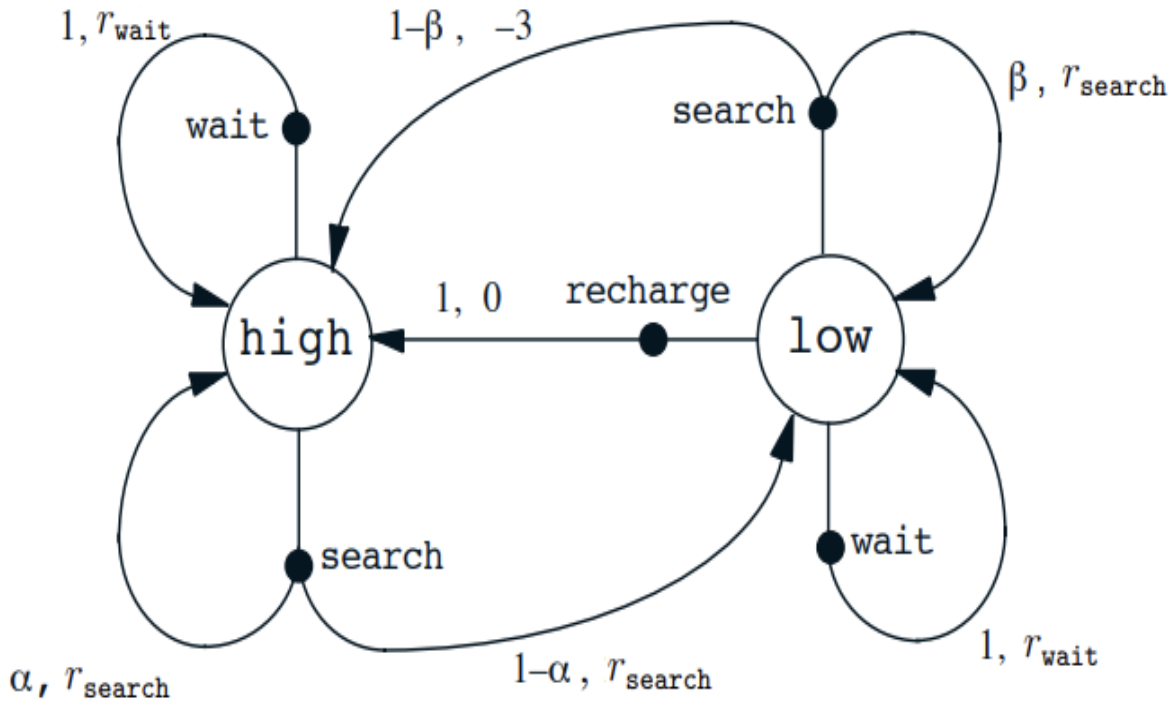
Example: Recycling Robot MDP

- State: Energy level of the battery, S = (High, Low)
- Action:
  - A(high) = {search, wait}
  - A(low) = {search, wait, recharge}
- A transition graph:
  - Two kinds of nodes: state nodes and action nodes.
  - There is a state node for each possible state (a large open circle labeled by the name of the state)
  - an action node for each state-action pair (a small solid circle labeled by the name of the action and connected by a line to the state node)



Transition graph (Sutton and Barto)

Example: Recycling Robot MDP

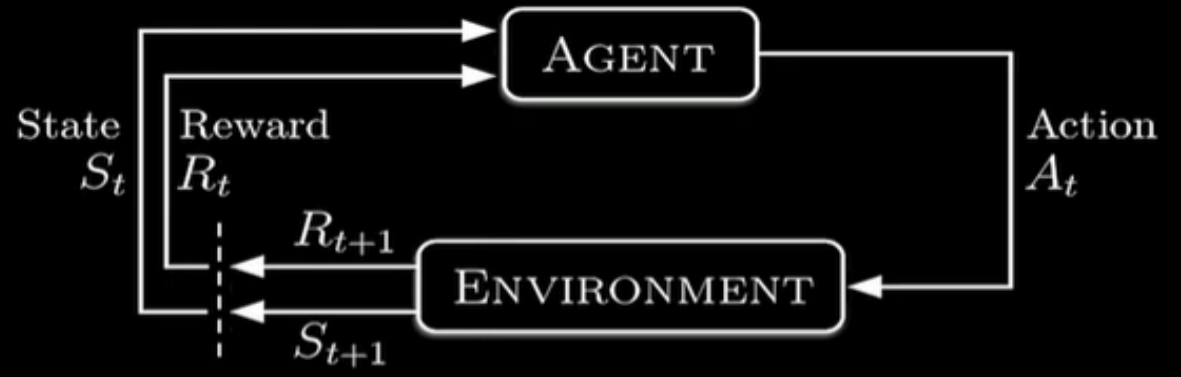


Transition graph (Sutton and Barto)

s	s'	a	$p(s'\|s,a)$	$r(s,a,s')$
high	high	search	α	r_{search}
high	low	search	$1-\alpha$	r_{search}
low	high	search	$1-\beta$	-3
low	low	search	β	r_{search}
high	high	wait	1	r_{wait}
high	low	wait	0	r_{wait}
low	high	wait	0	r_{wait}
low	low	wait	1	r_{wait}
low	high	recharge	1	0
low	low	recharge	0	0.

RL Model

Finite Markov Decision Process



$$t \in \{0, 1, 2, \cdots\} \qquad s \in \mathcal{S} \qquad a \in \mathcal{A}(s) \qquad r \in \mathcal{R} \in \mathbb{R}$$

$$p(s', r|s, a) = \text{Prob}(S_{t+1} = s', R_{t+1} = r \,|\, S_t = s, A_t = a)$$

POLICY: $\pi(a|s) \quad \text{or} \quad a = \pi(s)$

RETURN: $G_t = \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$

GOAL: $\max_{\pi} \mathbb{E}_{\pi}[G_t]$

Returns

- **Agent's Goal:**
  - To maximize the cumulative reward it receives in the long run (to find the optimal policy)
- **Returns**
  - For the sequence of rewards received after time step t : R_{t+1} R_{t+2} R_{t+3}, ····
  - Return G_t : sum of the rewards (*T* is a final time step)

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T$$

  - *t = 1, 2, ····, T (*the sequences are called *episodes.)*
  - Each episode ends in a special state called the terminal state. (episodic task)
  - *t = 1, 2, ····, T=∞* (continuing task)
- **Discounting**
  - determines the present value of future rewards (a reward received at *t=k* time steps in the future is worth only γ^{k-1} times what it would be worth if it were received immediately. (discount rate γ : $0 \le \gamma \le 1$)
  - Discounted return, G_t :

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

Value Functions

- **Value Functions**
  - functions of states (or of state–action pairs) that estimate how good it is for the agent to be in a given state (or how good it is to perform a given action in a given state).
  - The notion of "how good" here is defined in terms of future rewards that can be expected, (or, in terms of expected return, G_t)
  - Of course, the rewards the agent can expect to receive in the future depend on what actions it will take. Accordingly, value functions are defined for particular policies.

(*) note that the value of the terminal state, if any, is always zero.

$$v_\pi(s) = \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s\right]$$

state-value function for policy π

$$q_\pi(s,a) = \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s, A_t = a\right]$$

action-value function for policy π

Policy and Value Functions

- **Policy**: **probability distribution over actions**

$$\text{policy} = \pi(a|s) \quad = P(A_t=a \mid S_t=s)$$

- **State-Value function**, $v_\pi(s)$

"return": t 시점 이후의 모든 discount 된 reward 의 총합

$$G_t = R_{t+1} + \gamma R_{t+2} + r^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi \left[G_t | S_t = s \right] \\ &= \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s \right] \\ &= \mathbb{E}_\pi \left[R_{t+1} + \gamma G_{t+1} | S_t = s \right] \end{aligned}$$

Value(가치) of state s

- **Action-Value function** (Q-Function)

$$\begin{aligned} q_\pi(s,a) &= \mathbb{E}_\pi \left[G_t | S_t = s, A_t = a \right] \\ &= \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s, A_t = a \right] \\ &= \mathbb{E}_\pi \left[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a \right] \end{aligned}$$

Value (가치) of state-action (s,a)

Formal Definitions

Definition

A *Markov Decision Process* is a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$

- $\mathcal{S}$ is a finite set of states
- $\mathcal{A}$ is a finite set of actions
- $\mathcal{P}$ is a state transition probability matrix, $\mathcal{P}^{a}_{ss'} = \mathbb{P}\left[S_{t+1} = s' \mid S_t = s, A_t = a\right]$
- $\mathcal{R}$ is a reward function, $\mathcal{R}^{a}_{s} = \mathbb{E}\left[R_{t+1} \mid S_t = s, A_t = a\right]$
- γ is a discount factor $\gamma \in [0, 1]$.

Definition

A *policy* π is a distribution over actions given states,

$$\pi(a|s) = \mathbb{P}\left[A_t = a \mid S_t = s\right]$$

Definition

The *return* G_t is the total discounted reward from time-step t.

$$G_t = R_{t+1} + \gamma R_{t+2} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

Definition

The *state-value function* $v_\pi(s)$ of an MDP is the expected return starting from state s, and then following policy π

$$v_\pi(s) = \mathbb{E}_\pi\left[G_t \mid S_t = s\right]$$

Definition

The *action-value function* $q_\pi(s, a)$ is the expected return starting from state s, taking action a, and then following policy π

$$q_\pi(s, a) = \mathbb{E}_\pi\left[G_t \mid S_t = s, A_t = a\right]$$

Bellman Equation

- 강화학습에서는 어떤 행동(Action)에 대한 보상(Reward)이 필요하다.
  - 이 때 보상(Reward)은 현재 당장 받는 보상 (immediate reward) 뿐만 아니라 미래의 보상 (future reward)을 함께 고려해야 한다.
- 강화학습에서 미래의 보상을 알기 위해 앞으로 일어날 일을 모두 경험하지 않더라도 현재의 상태에서 예측을 통해서 미래를 가늠해 볼 방법을 이용한다. 벨만 방정식(Bellman Equation)은 이처럼 현재 상태로부터 미래의 성공확률을 계산하기 위한 식이다.
- 가치함수(Value Function)은 다음의 두 개의 부분으로 구성된다.
  - 행동 후 즉각 받는 보상 (immediate reward)
  - 감가율을 고려한 미래상태 (future state)에 대한 가치 (value)

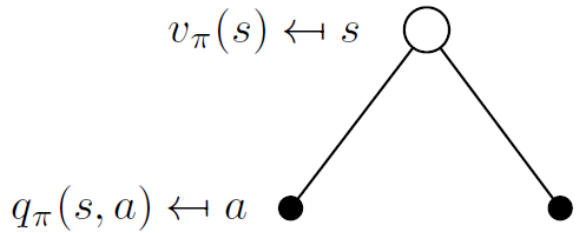
$$
\begin{aligned}
v(s) &= \mathbb{E}\left[G_t \mid S_t = s\right] \\
&= \mathbb{E}\left[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots \mid S_t = s\right] \\
&= \mathbb{E}\left[R_{t+1} + \gamma\left(R_{t+2} + \gamma R_{t+3} + \ldots\right) \mid S_t = s\right] \\
&= \mathbb{E}\left[R_{t+1} + \gamma G_{t+1} \mid S_t = s\right] \\
&= \mathbb{E}\left[R_{t+1} + \gamma v(S_{t+1}) \mid S_t = s\right]
\end{aligned}
$$

immediate reward　　next state value

Bellman (expectation) Equation

- 상태가치 (State value)에 대한 Bellman 방정식
  - 현재의 상태와 뒤이어질 상태들의 모든 가치를 계산

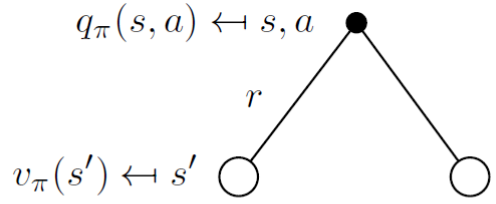
$$v_\pi(s) = \mathbb{E}_\pi \left[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s \right]$$



$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) q_\pi(s,a)$$

- 행동가치 (Action value)에 대한 Bellman 방정식
  - 현재 상태(State)에서 어떤 정책(Policy)에 따라 행동(Action)을 취했을 때 뒤이어질 상태(next State)와 행동(Action)들에 따라 얻게 되는 가치를 계산

$$q_\pi(s,a) = \mathbb{E}_\pi \left[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) \mid S_t = s, A_t = a \right]$$

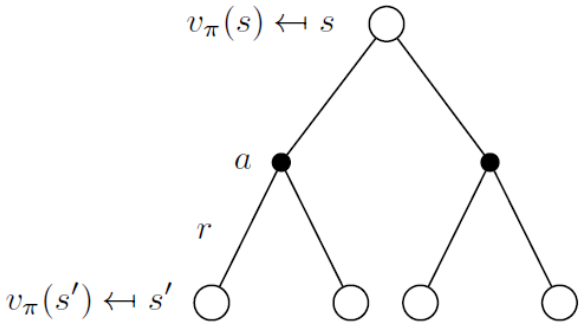


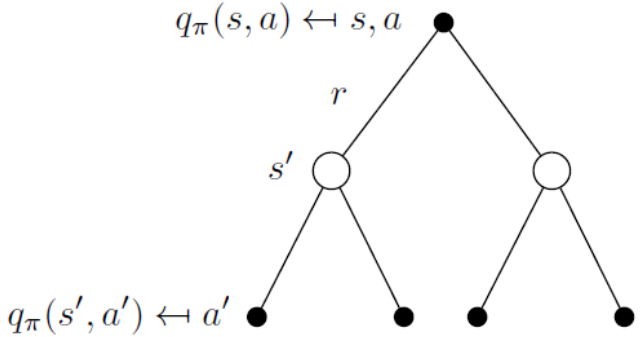
$$q_\pi(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s')$$

(*) Note that $R(s)$ or $R(s,a)$ is often used in the Bellman equations for brevity, but this is a simplified representation of the more detailed $R(s,a,s')$ in the MDP framework.

Bellman (expectation) Equation

- 앞의 두그림을 합치면,





$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s') \right)$$

$$q_\pi(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \sum_{a' \in \mathcal{A}} \pi(a'|s') q_\pi(s',a')$$

Bellman Equation

$$
\begin{aligned}
v_\pi(s) &= \mathbb{E}_\pi\left[G_t \middle| S_t = s\right] \\
&= \mathbb{E}_\pi\left[R_{t+1} + \gamma G_{t+1} \middle| S_t = s\right] \\
&= \mathbb{E}_\pi\left[R_{t+1} + \gamma \mathbb{E}_\pi[G_{t+1} | S_{t+1} = s'] \middle| S_t = s\right] \\
&= \sum_a \pi(a|s) \sum_{s',r} \mathbb{P}(s', r|s, a)\left[r + \gamma \mathbb{E}_\pi\left[G_{t+1} \middle| S_{t+1} = s'\right]\right] \\
&= \sum_a \pi(a|s) \sum_{s',r} \mathbb{P}(s', r|s, a)\left[r + \gamma v_\pi(s')\right] \\
&= \sum_a \pi(a|s) q_\pi(s, a)
\end{aligned}
$$

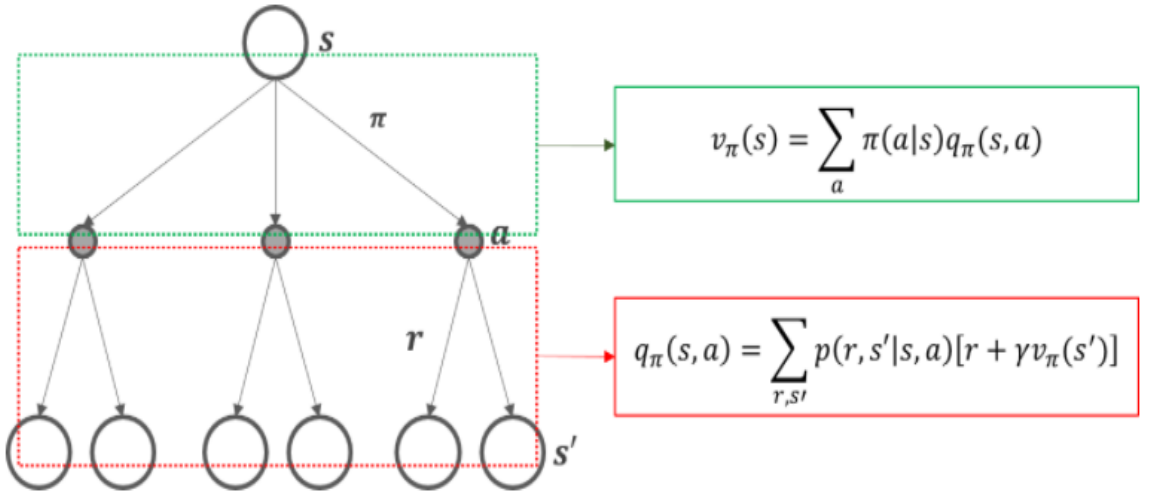


Figure 2: Bellman Equation Back-up Diagram

$$
\begin{aligned}
q_\pi(s, a) =& \mathbb{E}_\pi\left[G_t \middle| S_t = s, A_t = a\right] \\
=& \mathbb{E}_\pi\left[R_{t+1} + \gamma \mathbb{E}_\pi(G_{t+1} | S_{t+1} = s') \middle| S_t = s, A_t = a\right] \\
& \text{or } \mathbb{E}_\pi\left[R_{t+1} + \gamma \mathbb{E}(G_{t+1} | S_{t+1} = s', A_{t+1} = a') \middle| S_t = s, A_t = a\right] \\
=& \mathbb{E}\left[R_{t+1} + \gamma v_\pi(s') \middle| S_t = s, A_t = a\right] \\
& \text{or } \mathbb{E}\left[R_{t+1} + \gamma q_\pi(s', a') \middle| S_t = s, A_t = a\right] \\
=& \sum_{r,s'} \mathbb{P}(s', r|s, a)[r + \gamma v_\pi(s')] \\
& \text{or } \sum_{r,s',a'} \mathbb{P}(r, s', a'|s, a)[r + \gamma q_\pi(s', a')]
\end{aligned}
$$

Bellman Optimality Equation

- 강화학습의 목적은 가치함수의 참값을 찾는 것이 아니라 최대의 보상(Reward)를 얻는 정책(Policy)를 찾는 것이다. 즉, 가치함수의 가장 큰 참값을 찾는 것이다. 이와 같이 최적의 정책을 구하고자 하는 벨만방정식을 Bellman Optimality Equation이라고 한다.
- Optimal policy, π*:

$$\pi_*(a|s) = \begin{cases} 1 & \text{if } a = \underset{a \in \mathcal{A}}{\text{argmax}}\, q_*(s,a) \\ 0 & otherwise \end{cases}$$

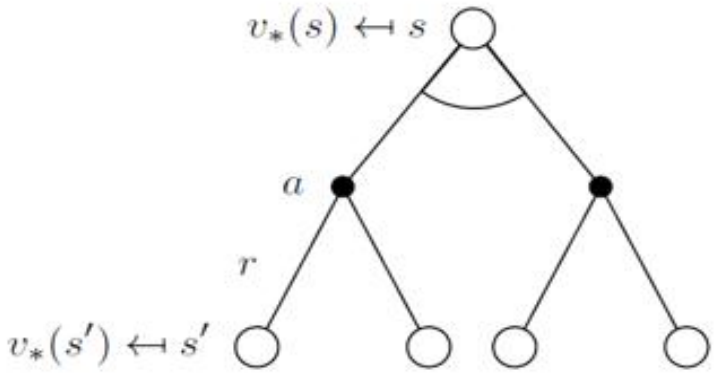
- 최적가치함수(Optimal Value Function)

$$V^*(s) = \max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma . V^*(s')] \quad \text{for all } s$$

  - $T(s, a, s')$ is the transition probability from state s to state s', given that the agent chose action a.
  - $R(s, a, s')$ is the reward that the agent gets when it goes from state s to state s', given that the agent chose action a.
  - γ is the discount rate.

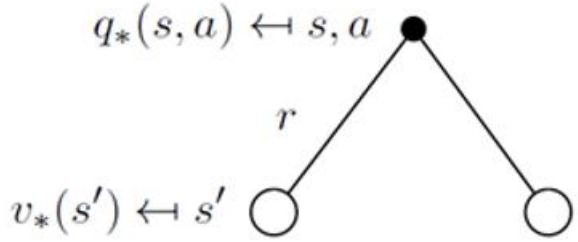
Bellman Optimality Equation

- 최적상태가치함수(Optimal State-Value Function)



$$V_*(s) = \max_\pi V_\pi(s) = \max_a q_\pi(s, a)$$
$$= \max_a \left(R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a V_\pi(s') \right)$$
$$= \max_a R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a V_*(s')$$

- 최적행동가치함수(Optimal Action-Value Function)



$$q_*(s, a) = \max_\pi q_\pi(s, a) = \max_\pi \left(R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a V_\pi(s') \right)$$
$$= R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a V_*(s')$$

Bellman Optimality Equation

1. **Bellman Expectation Function:**
   - The Bellman expectation function is associated with both state-value functions $V(s)$ and action-value functions $Q(s,a)$.
   - It expresses the expected value of a state or state-action pair in terms of the immediate reward and the expected value of the successor states under a given policy.
   - For the state-value function, the Bellman expectation equation is given by:

     $V(s) = \sum_a \pi(a|s) \left(R(s,a) + \gamma \sum_{s'} P(s'|s,a)V(s')\right)$
   - For the action-value function, it is expressed as:

     $Q(s,a) = R(s,a) + \gamma \sum_{s'} P(s'|s,a) \sum_{a'} \pi(a'|s')Q(s',a')$
   - Here, $\pi(a|s)$ represents the policy, indicating the probability of taking action a in state s.

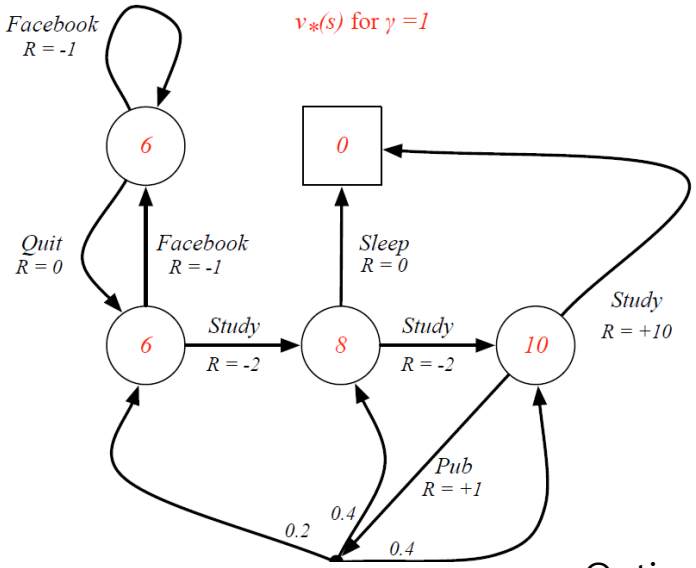
2. **Bellman Optimality Function:**
   - The Bellman optimality function is specifically associated with optimal policies and the optimal value functions.
   - It describes the relationship between the optimal value of a state or state-action pair and the optimal value of its successor states.
   - For the state-value function, the Bellman optimality equation is given by:

     $V^*(s) = \max_a \left(R(s,a) + \gamma \sum_{s'} P(s'|s,a)V^*(s')\right)$
   - For the action-value function, it is expressed as:

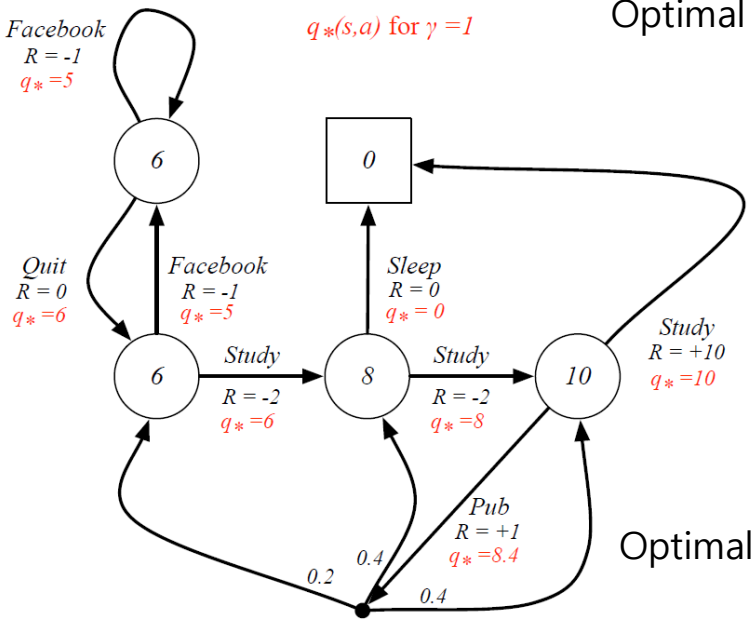
     $Q^*(s,a) = R(s,a) + \gamma \sum_{s'} P(s'|s,a) \max_{a'} Q^*(s',a')$
   - Here, $V^*(s)$ and $Q^*(s,a)$ represent the optimal state-value and action-value functions, respectively.

(*) 즉, 벨만기대함수는 주어진 정책 하에서 기대값으로 표현되고, 벨만최적함수는 가능한 최선의 정책 하에서 최적의 값(최대값)으로 표현됨. Bellman 최적 방정식은 강화 학습에서 최적의 정책을 찾는 데 기본이다.

Bellman Optimality Equation (example)



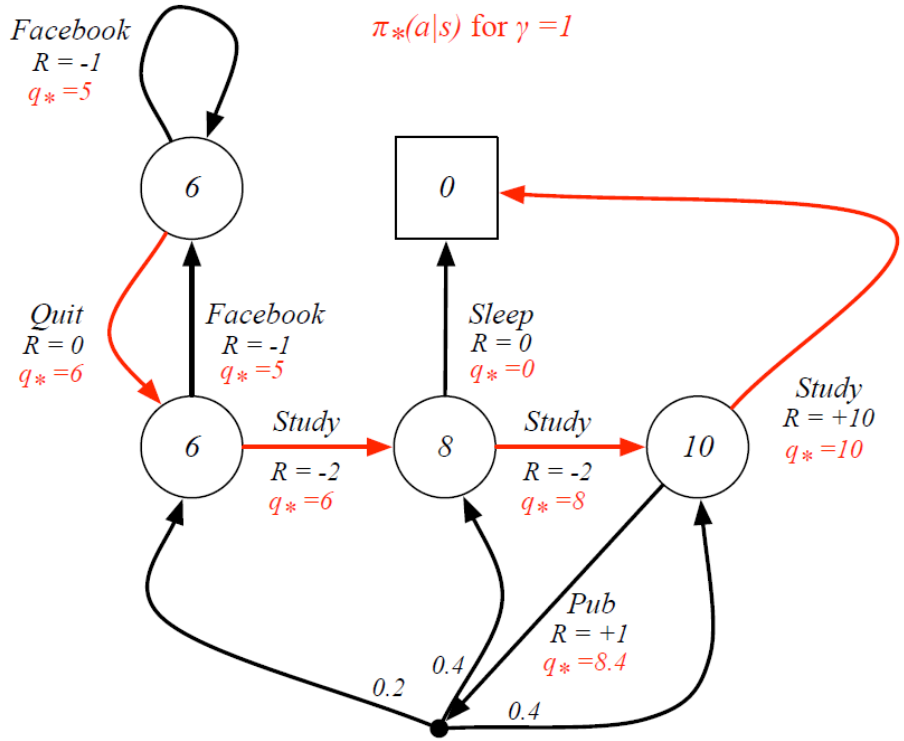
Optimal State-value function



Optimal Action-value function

Optimal policy: take an action that maximizes values

$$\pi_*(a|s) = \begin{cases} 1 & \text{if } a = \underset{a \in \mathcal{A}}{\text{argmax}}\, q_*(s,a) \\ 0 & \text{otherwise} \end{cases}$$



Reinforcement Learning Algorithms

- Dynamic Programming
  - (model-based)
  - Not a Reinforcement Learning, but a **Planning** (in the sense that it assumes complete knowledge about the environment)
  - Policy iteration: alternates between two steps: policy evaluation and policy improvement)
  - Value iteration: iteratively computes and improves the value function for each state until it converges to the optimal value function. The policy is then derived from the optimal value function
- Monte Carlo Learning
- Temporal Difference Learning:
  - SARSA
  - Q-Learning

Reinforcement Learning Algorithms

- Policy iteration
  - obtain a sequence of monotonically improving policies and value functions: (policy evaluation and policy improvement)

$$\pi_0 \xrightarrow{\text{E}} v_{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} v_{\pi_1} \xrightarrow{\text{I}} \pi_2 \xrightarrow{\text{E}} \cdots \xrightarrow{\text{I}} \pi_* \xrightarrow{\text{E}} v_*$$

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Repeat
   $\Delta \leftarrow 0$
   For each $s \in \mathcal{S}$:
   $v \leftarrow V(s)$
   $V(s) \leftarrow \sum_{s',r} p(s', r|s, \pi(s))\left[r + \gamma V(s')\right]$
   $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number)

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
   $a \leftarrow \pi(s)$
   $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r|s, a)\left[r + \gamma V(s')\right]$
   If $a \neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop and return V and π; else go to 2

Reinforcement Learning Algorithms

- Value iteration
  - Special case of policy iteration: when policy evaluation is stopped after just one sweep (one backup of each state) : 벨만 최적 방정식을 업데이트 형식으로 바뀐 것

$$
\begin{aligned}
v_{k+1}(s) &= \max_a \mathbb{E}[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s, A_t = a] \\
&= \max_a \sum_{s',r} p(s', r|s, a)\Big[r + \gamma v_k(s')\Big],
\end{aligned}
$$

$$
\begin{aligned}
v'_\pi(s) &= \max_a \mathbb{E}[R_{t+1} + \gamma v'_\pi(S_{t+1}) | S_t = s, A_t = a] \\
&= \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v'_\pi(s')]
\end{aligned}
$$

Bellman optimality equation

Initialize array V arbitrarily (e.g., $V(s) = 0$ for all $s \in \mathcal{S}^+$)

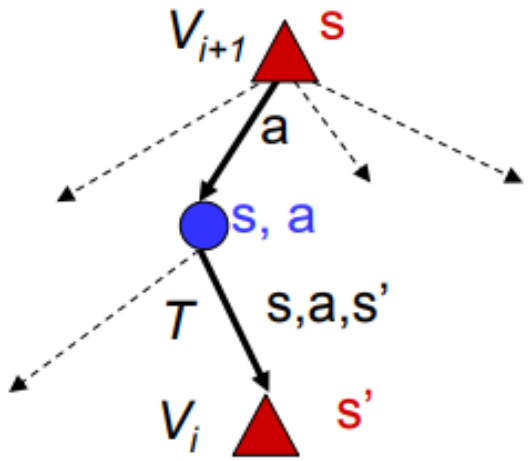
Repeat
  $\Delta \leftarrow 0$
  For each $s \in \mathcal{S}$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a)\big[r + \gamma V(s')\big]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive number)

Output a deterministic policy, π, such that
  $\pi(s) = \arg\max_a \sum_{s',r} p(s', r|s, a)\big[r + \gamma V(s')\big]$

State-Value Iteration algorithm

- An algorithm to precisely estimate the **optimal state value** of every possible state:
  - Initialize all the state value estimates: $V_0(s) = 0$
  - Iteratively update them using **Value Iteration algorithm**

$$V_{i+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V_i(s') \right]$$



- Equivalent to: $V_{i+1}(s) \leftarrow \max_a Q_{i+1}(s,a)$

- Q: why not directly Q-values? -> Q-value iteration

Q-Value Iteration Algorithm

- Knowing the optimal state values can be useful (to evaluate a policy), but it does not tell the agent explicitly what to do.
- A very similar algorithm to estimate the optimal state-action values (Q-values) of the state-action pair (*s*,*a*), noted Q*(*s*,*a*):

$$Q_*(s,a) = \max_{\pi} Q_{\pi}(s,a)$$

- **Precise estimation:**
  - Initialize all the Q-value estimates = 0
  - Iteratively update them using **Q-value Iteration algorithm**

$$Q_{i+1}(s,a) \leftarrow \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma \, V_i(s') \right]$$ i.e.,

$$Q_{i+1}(s,a) \leftarrow \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma \max_{a'} Q_i(s',a') \right]$$

- Once you have the optimal Q-values, the optimal policy π*(s) is trivial:

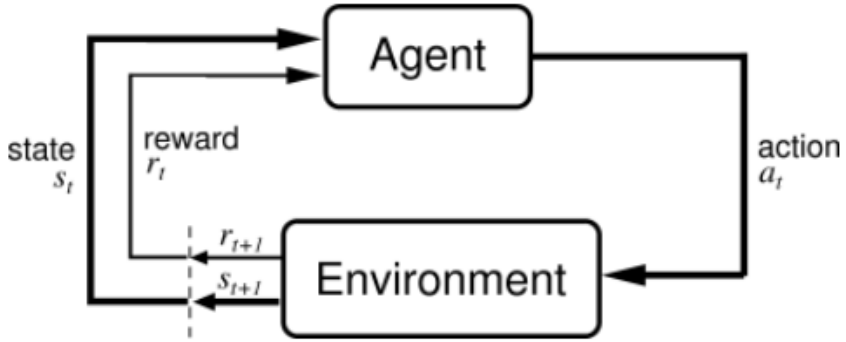
$$\pi^*(s) = \operatorname{argmax}_a Q^*(s,a)$$

Q-iteration -> Q-learning

$$Q_{i+1}(s,a) \leftarrow \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma \max_{a'} Q_i(s',a') \right]$$

In RL, we don't have this!

- But we get a sample at each time step t : (s_t, a_t, r_{t_1}, s_{t+1})



Agent-environment interaction [Source]

- Instead of Expectation under T, what if we compute a running average of Q from all samples received so far?

$$Q(s,a) \leftarrow \frac{1}{t} \sum_{t \text{ samples}} \left(r + \gamma \max_{a'} Q(s',a') \right)$$

Ref: https://courses.cs.washington.edu/courses/cse473/13au/slides/17-mdp-rl.pdf

Q-iteration -> Q-learning

- Running average
  - Running average of t samples of a quantity x:

$$\bar{x}_t = \frac{x_1 + x_2 + \ldots x_{t-1} + x_t}{t}$$

$$= \frac{x_1 + x_2 + \ldots x_{t-1}}{t} \cdot \frac{(t-1)}{(t-1)} + \frac{x_t}{t}$$

$$= \frac{(t-1)}{t} \bar{x}_{t-1} + \frac{1}{t} x_t$$

$$= (1-\alpha)\bar{x}_{t-1} + \alpha\, x_t \quad \text{where } \alpha = 1/t$$

  - Running average of Q:

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha(r + \gamma \max_{a'} Q(s',a'))$$

Ref: https://courses.cs.washington.edu/courses/cse473/13au/slides/17-mdp-rl.pdf

Monte Carlo Methods

- Model-free and it does not require a complete knowledge of the environment's dynamics (i.e., the state transition probabilities and rewards). Instead, they learn directly from episodes of experience.
- only after a **complete episode**, values are updated (Only can be used in **episodic problems**)
- There are a few strategies to be applied for continuing (non-episodic) environments. (e.g. artificial termination)

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
$Q(s,a) \leftarrow$ arbitrary
$Returns(s,a) \leftarrow$ empty list
$\pi(a|s) \leftarrow$ an arbitrary ε-soft policy

Repeat forever:
(a) Generate an episode using π
(b) For each pair s, a appearing in the episode:
$G \leftarrow$ return following the first occurrence of s, a
Append G to $Returns(s,a)$
$Q(s,a) \leftarrow \text{average}(Returns(s,a))$
(c) For each s in the episode:
$a^* \leftarrow \arg\max_a Q(s,a)$
For all $a \in \mathcal{A}(s)$:

$$\pi(a|s) \leftarrow \begin{cases} 1-\varepsilon+\varepsilon/|\mathcal{A}(s)| & \text{if } a = a^* \\ \varepsilon/|\mathcal{A}(s)| & \text{if } a \neq a^* \end{cases}$$

An on-policy first-visit MC control algorithm for ∈-soft policies

ε-greedy (epsilon-greedy)

- Exploration and Exploitation:
  - Exploration: The agent tries new actions to discover more information about the environment.
  - Exploitation: The agent uses known information to maximize rewards, typically by choosing the best-known action.
- How ε-greedy Works: at each time step, the agent selects an action based on:
  - With probability ε, the agent selects an action at random (exploration).
  - With probability 1-ε, the agent selects the best-known action (exploitation).
- Most of the time we choose an action that has the maximal estimated action value, but with probability ε, we instead select an action at random.

$$\text{Prob(action)} = \begin{cases} 1 - \varepsilon + \varepsilon/|\mathcal{A}(s)| & \leftarrow \text{greedy action (minimal probability)} \\ \varepsilon/|\mathcal{A}(s)| & \leftarrow \text{non-greedy action} \end{cases}$$

- In continuous action spaces, different strategies and algorithms are used:

Temporal Difference (TD) Learning

- **Model-free**,
- **Monte Carlo** (learn directly from raw experience) + **Dynamic Programming** (update estimates every time step)
- RL problems with discrete actions can be modeled as MDP, but the agent initially has no idea what the transition probabilities are (does not know $T(s, a, s')$ and $R(s, a, s')$).
- It must experience each state and each transition at least once to know the rewards, and it must experience them multiple times if it is to have a reasonable estimate of the transition probabilities.

Temporal Difference (TD) Learning

- **What does TD(Time Difference) mean?**
  - It means the differences between predicted (expected) true value and current estimation value. (not refer to the time between successive iterations of a learning algorithm)
  - The predicted true value is obtained from the assumption that the Q-values for all other states (s') are optimal. (we use this to find a better estimated (close to optimal) value at state s.)

1. **Temporal Difference (TD)**: In reinforcement learning, the term "temporal difference" refers to the discrepancy or difference between the expected value (or Q-value) of a state or action at a certain time step and the actual observed reward received at that time step.

2. **TD Error**: The TD error is a key concept in TD learning. It's the difference between the estimated value of a state (or action) and the observed reward plus the estimated value of the next state:

$$TD\_error = R + \gamma \cdot Q(S', A') - Q(S, A)$$

Where:

- R is the observed reward.
- γ is the discount factor.
- $Q(S, A)$ is the estimated value of taking action A in state S.
- $Q(S', A')$ is the estimated value of the next state-action pair.

The TD error is used to update the value estimate of the current state-action pair.

Temporal Difference (TD) Learning

- **What does TD(Time Difference) mean?**

3. **TD Learning**: TD learning algorithms, like SARSA and Q-learning, update their value estimates based on these TD errors. They learn from the temporal differences observed during interaction with the environment.

   For example, in Q-learning, the Q-value of a state-action pair is updated as follows:

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

   Where:

   - α is the learning rate.

(*) using the observed reward and the maximum Q-value of the next state over all possible actions.

For SARSA:

$$Q(S, A) \leftarrow Q(S, A) + \alpha \cdot (R + \gamma \cdot Q(S', A') - Q(S, A))$$

(*) using the observed reward and the Q-value of the next state-action pair, which is chosen based on the current policy.

TD Learning Algorithm

- **Temporal Difference Learning (TD)**
  - Similar to Value Iteration algorithm, but, take into account the fact that the agent has only partial knowledge of the MDP. (In general we assume that the agent initially knows only the possible states and actions, and nothing more.)
  - The agent uses an exploration policy (for example, random policy) to explore the MDP, and progresses TD algorithm to update the estimates based on the transitions and rewards that are actually observed.

$$V_{k+1}(s) \leftarrow (1-\alpha)V_k(s) + \alpha(r + \gamma . V_k(s'))$$

- α is the learning rate (e.g., 0.01).

For each state s, it simply keep track of a running average of the rewards (immediate + future) at each state s

  - Many similarities with Stochastic Gradient Descent (SGD):
    - handles one sample at a time
    - converge if you gradually reduce the learning rate (otherwise it will keep bouncing around the optimum Q-value)

Q-Learning algorithm

- **Q-Learning**
  - Adaptation of the Q-value iteration with **no** information about the transition probabilities and the rewards
  - It works by watching an agent play (e.g. randomly) and **gradually improving its estimates** of the Q-values.
  - The optimal policy is choosing the action of the highest Q-value (e.g. greedy policy)

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right]$$

  - For each state-action pair (*s*, *a*), this algorithm **keeps track of a running average of the rewards** (immediate + future)
  - Since the target policy would act optimally, we take the maximum of the Q-Value estimates for the next state.

SARSA algorithm

- **SARSA (State-Action-Reward-State-Action)**
  - Update is done based on:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\Big]$$

  - This update is done after every transition from a nonterminal state S_t. If S_{t+1} is terminal, then $Q(S_{t+1} A_{t+1})$ is defined as zero.
  - This rule uses every element of the quintuple of events, $(S_t\ A_t\ R_{t+1}\ S_{t+1}\ A_{t+1})$, that make up a transition from one state action pair to the next.

SARSA vs. Q-Learning

- **Q-Learning (off-policy)**

Considers the best possible case

$Q_{k+1}(s,a)$ → $$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right]$$

```
Q[s,a] = Q[s,a] + alpha*(reward - Q[s,a]
          + gamma*np.max([Q[s_,a_] for a_ in range(possible_action)]))
```

- **SARSA (on-policy)**

follow the current policy

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\left[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\right]$$

```
a_ = max_action(s_) if rand < (1-epsilon) else env.action_space.sample()
Q[s,a] = Q[s,a] + alpha*(reward - Q[s,a] + gamma*Q[s_,a_])
```

← ε-greedy policy

SARSA vs. Q-Learning

- Policy:
  - ϵ-greedy is applied to balance the exploration and exploration of reinforcement learning.
- Both are updated using the TD learning formula.
- Difference:
  - **Q-Learning (off-policy):** when passing the reward from the next state (s', a') to the current state, it takes the maximum possible reward of the new state (s') and ignores whatever policy we are using. (the agent receives no policy)
  - **SARSA (on-policy):** we still follow the policy (ϵ -greedy), compute the next state (s'), and pass the reward corresponding to that action a'. (the agent is given the policy)
  - Hence, if our policy is greedy, SARSA and QL will be the same. But we are using ϵ-greedy normally, so there is a slight difference.

SARSA vs. Q-Learning

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Choose A from S using policy derived from Q (e.g., ϵ-greedy)
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ϵ-greedy)
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma Q(S',A') - Q(S,A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

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

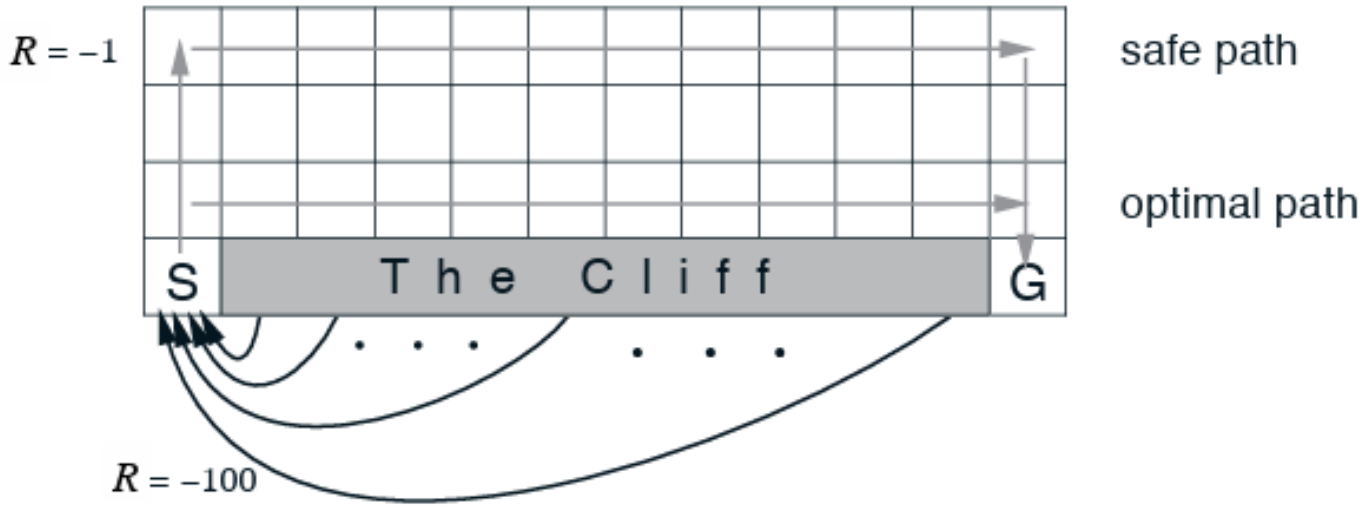
Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Take action A, observe R, 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

SARSA vs. Q-Learning

- **Some notes:**
  - Both approach work in a finite environment (or a discretized continuous environment)
  - QL directly learns the optimal policy while SARSA learns a "near" optimal policy. QL is a more aggressive agent, while SARSA is more conservative.
  - An example is walking near the cliff. QL will take the shortest path because it is optimal (with the risk of falling), while SARSA will take the longer, safer route (to avoid unexpected falling).
  - In practice, if you want to fast in a fast-iterating environment, QL should be your choice. However, if mistakes are costly (unexpected minimal failure — robots), then SARSA is the better option.
  - If your state space is too large, try exploring the deep Q-network.

SARSA vs. Q-Learning

- **The cliff-walking example (Sutten and Barto)**



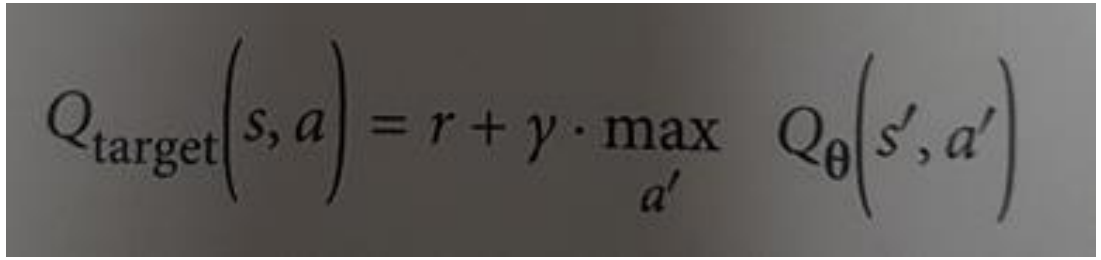
On-policy and Off-policy

- **On-Policy Methods:**
  - the agent learns the value of the policy it is currently following. This means the agent learns about and from the very policy it uses to make decisions.
  - Behavior policy = target policy (decision-making policy = learning policy)
  - SARSA

- **Off-Policy Methods:**
  - It allows the agent to learn about one policy (the target policy) while following another policy (the behavior policy). The agent learns from actions that are outside its current strategy.
  - Behavior policy ≠ target policy (decision-making policy ≠ learning policy)
  - Q-learning, expected SARSA

Approximate Q-Learning (DQN)

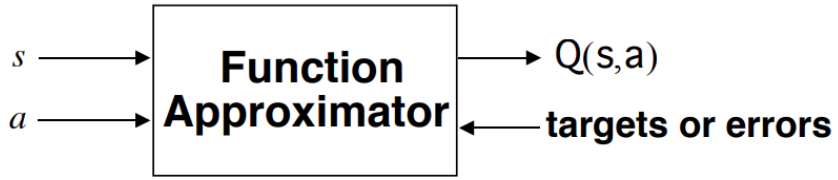
- **Problem in Q-Learning**
  - It does not scale well to large MDPs with many states and actions.
  - The solution is to find a function that approximates the Q-values using some parameters. (Approximate Q-Learning)
  - Uses DQN(deep Q-network) to estimate Q-Values (Deep Q-Learning)
- **How to train DQN?**
  - Thanks to Bellman, we want the approximate Q-value to be as close as to the reward **r** (that we actually observe after playing action a in state s), plus the discounted value of playing optimally from then on.
  - Target Q-value:



when we update the value at state *s*, we're making the assumption that the values at all other states *s'* are accurate or optimal.

Approximate Q-Learning (DQN)

General Idea



Could be:

- table
- Backprop Neural Network
- Radial-Basis-Function Network
- Tile Coding (CMAC)
- Nearest Neighbor, Memory Based
- Decision Tree

gradient-descent methods

Loss = (Target – f(s,a,w))^2
dLoss/dw = -(Target – f)*df/dw
w = w – α*dLoss/dw

$$Q(s,a) = f(s,a,w)$$

weight vector

e.g., gradient-descent Sarsa:

$$w \leftarrow w + \alpha\left[r_{t+1} + \gamma Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)\right] \nabla_w f(s_t,a_t,w)$$

standard backprop gradient

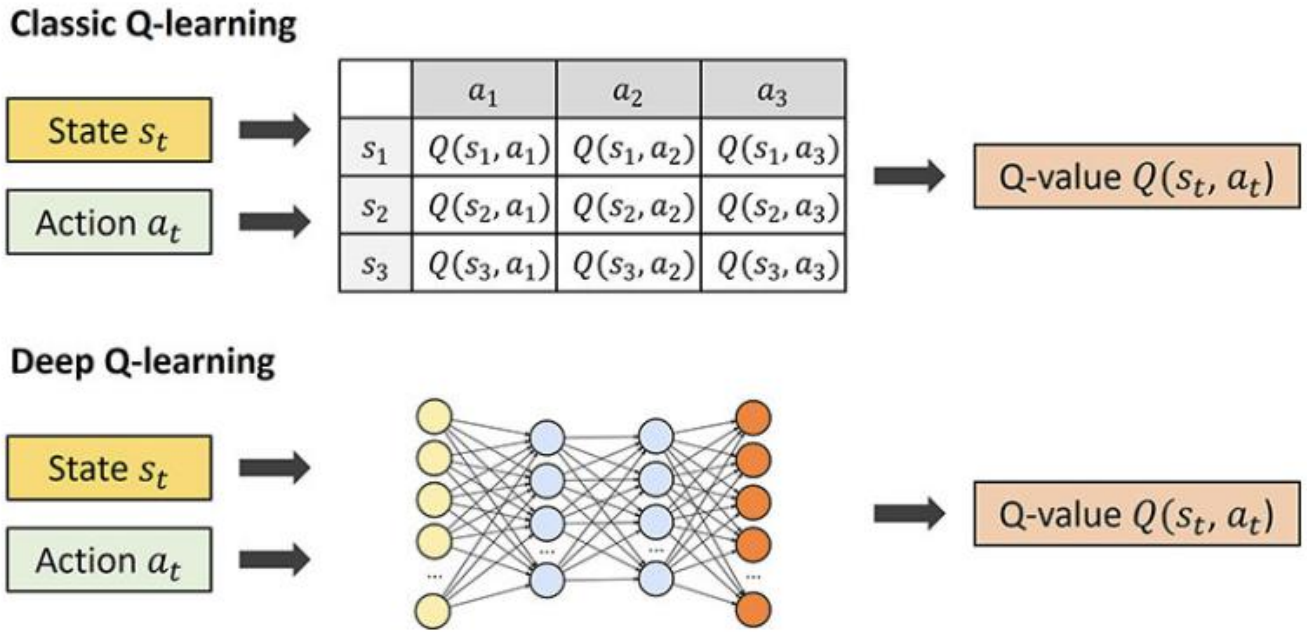
target value

estimated value

NIPS05RLMainTutorial.ppt (umich.edu)

Approximate Q-Learning (DQN)

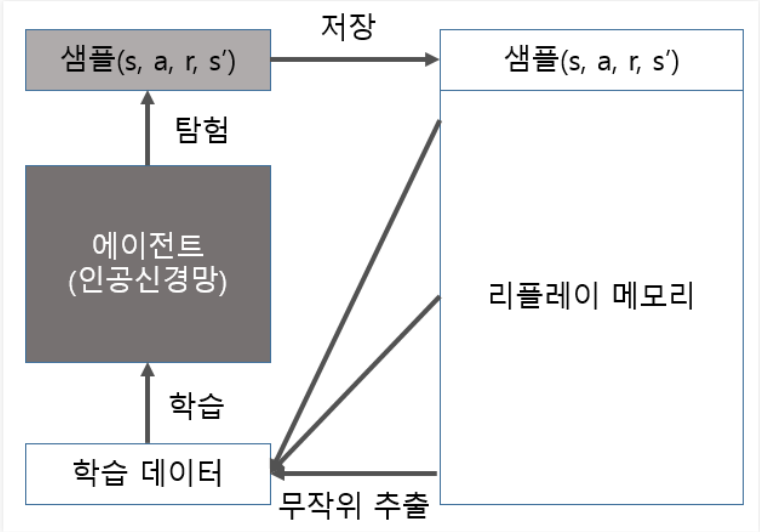
- Neural network: estimate a probability for each action
  - Input: an observation
  - Output: action to be taken



https://ai-com.tistory.com/entry/RL-%EA%B0%95%ED%99%94%ED%95%99%EC%8A%B5-%EC%95%8C%EA%B3%A0%EB%A6%AC%EC%A6%98-1-DQN-Deep-Q-Network

DQN

- **Experience (replay) Memory**
  - Stores experiences including state transitions, rewards and actions (that are necessary to perform Q learning)
  - Makes mini-batches to update neural networks.
  - Merits:
    - Reduces correlation between experiences in updating DNN
    - Increases learning speed with mini-batches
    - Reuses past transitions to avoid catastrophic forgetting
- **Two Q-Networks**
  - Use two separate Q-networks (**Action Network** and **Target Network**)
  - Periodically update (by copying Action Network to Target Network)
- **Gradient Clipping**
  - forces the gradient values (element-wise) to a specific minimum or maximum value (+1 and -1)



Deep Q-learning

Algorithm 1: deep Q-learning with experience replay.

Initialize replay memory D to capacity N

Initialize action-value function Q with random weights θ

Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$

For episode = 1, M **do**

Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$

For $t = 1$,T **do**

With probability ε select a random action a_t

otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$

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 D

Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D

Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$

Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ

Every C steps reset $\hat{Q} = Q$

End For

End For

preprocess example: screen display interface for games

ε-greedy

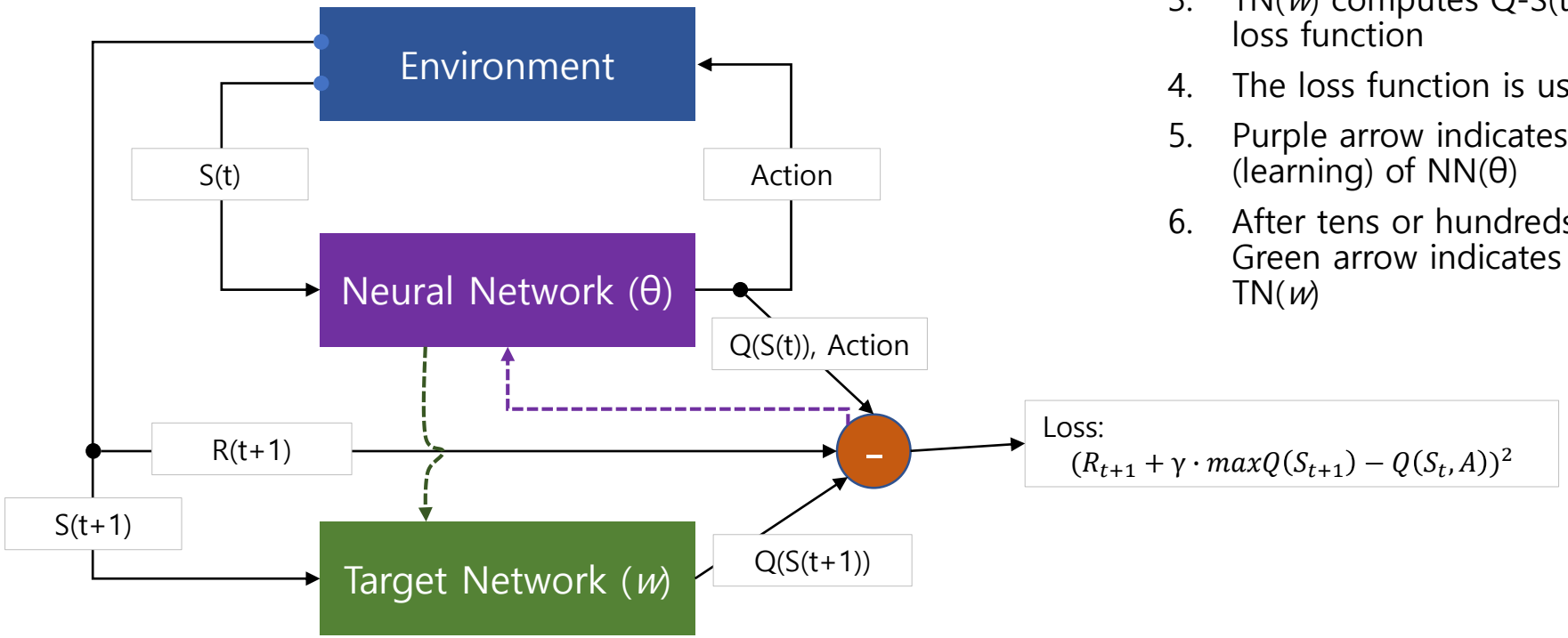
Example code:

- Note that we do not clear out the memory after each episode, this enables us to recall and build batches of experiences from across episodes.
- https://towardsdatascience.com/deep-q-network-dqn-ii-b6bf911b6b2c

Initialize network Q
Initialize target network $\hat{Q}$
Initialize experience replay memory D
Initialize the *Agent* to interact with the Environment
while *not converged* **do**
    /* Sample phase
    $\epsilon \leftarrow$ setting new epsilon with ϵ-decay
    Choose an action a from state s using policy ϵ-greedy(Q)
    Agent takes action a, observe reward r, and next state s'
    Store transition $(s, a, r, s', done)$ in the experience replay memory D
    if *enough experiences in* D **then**
        /* Learn phase
        Sample a random *minibatch* of N transitions from D
        for *every transition* $(s_i, a_i, r_i, s'_i, done_i)$ ***in*** *minibatch* **do**
            if $done_i$ **then**
                $y_i = r_i$
            else
                $y_i = r_i + \gamma \max_{a' \in \mathcal{A}} \hat{Q}(s'_i, a')$
            end
        end
        Calculate the loss $\mathcal{L} = 1/\mathrm{N} \sum_{i=0}^{N-1} (Q(s_i, a_i) - y_i)^2$
        Update Q using the SGD algorithm by minimizing the loss $\mathcal{L}$
        Every C steps, copy weights from Q to $\hat{Q}$
    end
end

Deep Q-learning

- **General flow of Deep Q-Leaning**
  - Deep Q-Network (DQN) Perspective



1. Input S(t) to NN(θ) then outputs an action
2. From the Action, Environment gives S(t+1) and R(t+1)
3. TN(*w*) computes Q-S(t+1), then computes the loss function
4. The loss function is used to train NN(θ)
5. Purple arrow indicates the parameter update (learning) of NN(θ)
6. After tens or hundreds of NN(θ) update, Green arrow indicates parameters copy to TN(*w*)

DDQN

- Q-learning methods are known to be overestimating the Q value (Over-estimating Bias)
  - DQN and other Q learning methods have this common issue and their performances are lowered due to that.
  - Double Q learning is known to be a solution to this problem

DQN Target value $$y_t^{DQN} = r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a'; \theta_t^-)$$

Rewrite it to Double Q form $$y_t^{DQN} = r_{t+1} + \gamma Q(s_{t+1}, \arg\max_{a'} Q(s_{t+1}, a'; \theta_t^-); \theta_t^-)$$

Double Q learning Target $$y_t^{Double} = r_{t+1} + \gamma Q(s_{t+1}, \arg\max_{a'} Q(s_{t+1}, a'; \theta_t); \theta_t')$$