

## Course Materials for GEN-AI

*Northeastern University*

These materials have been prepared and sourced for the course **GEN-AI** at Northeastern University. Every effort has been made to provide proper citations and credit for all referenced works.

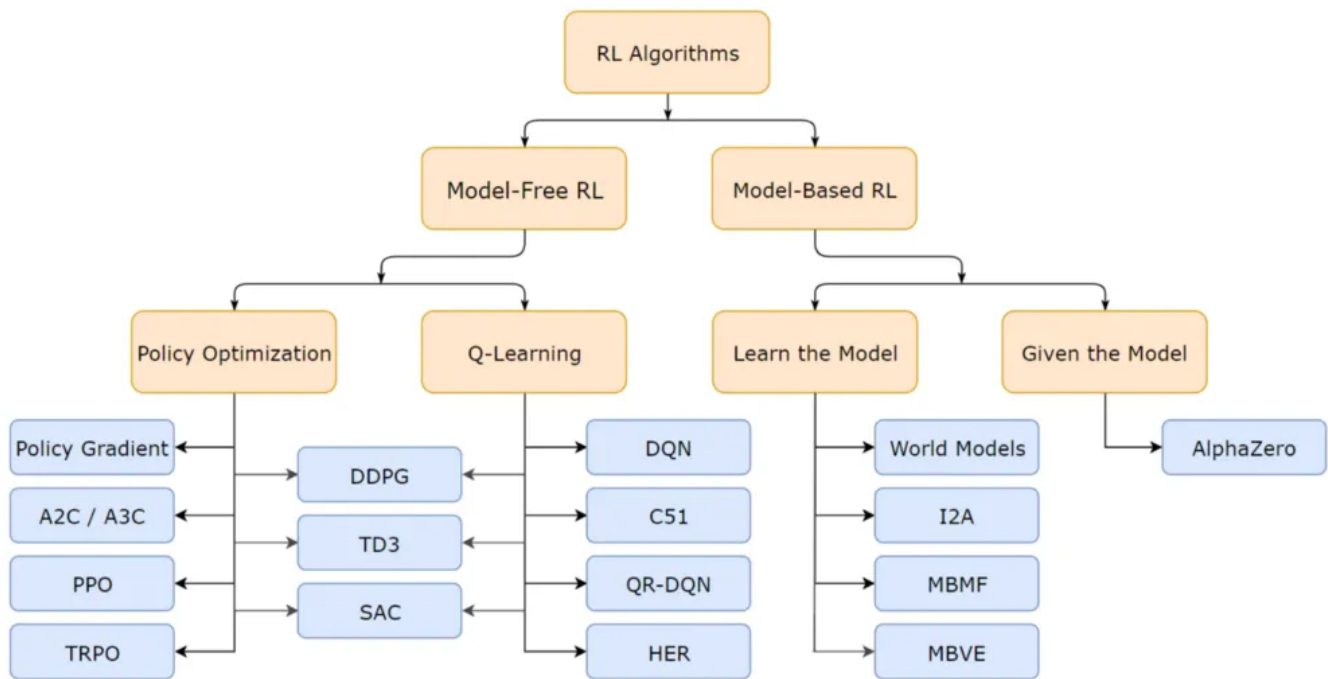
If you believe any material has been inadequately cited or requires correction, please contact me at:

**Instructor: Ramin Mohammadi**  
`r.mohammadi@northeastern.edu`

*Thank you for your understanding and collaboration.*

# Reinforcement Learning

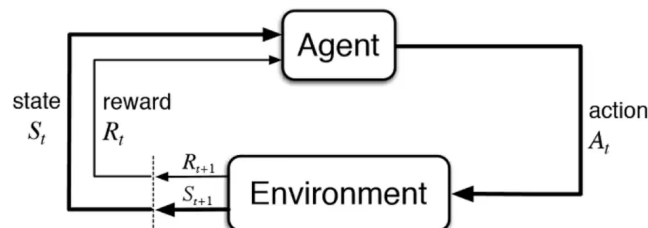
## 1 Introduction



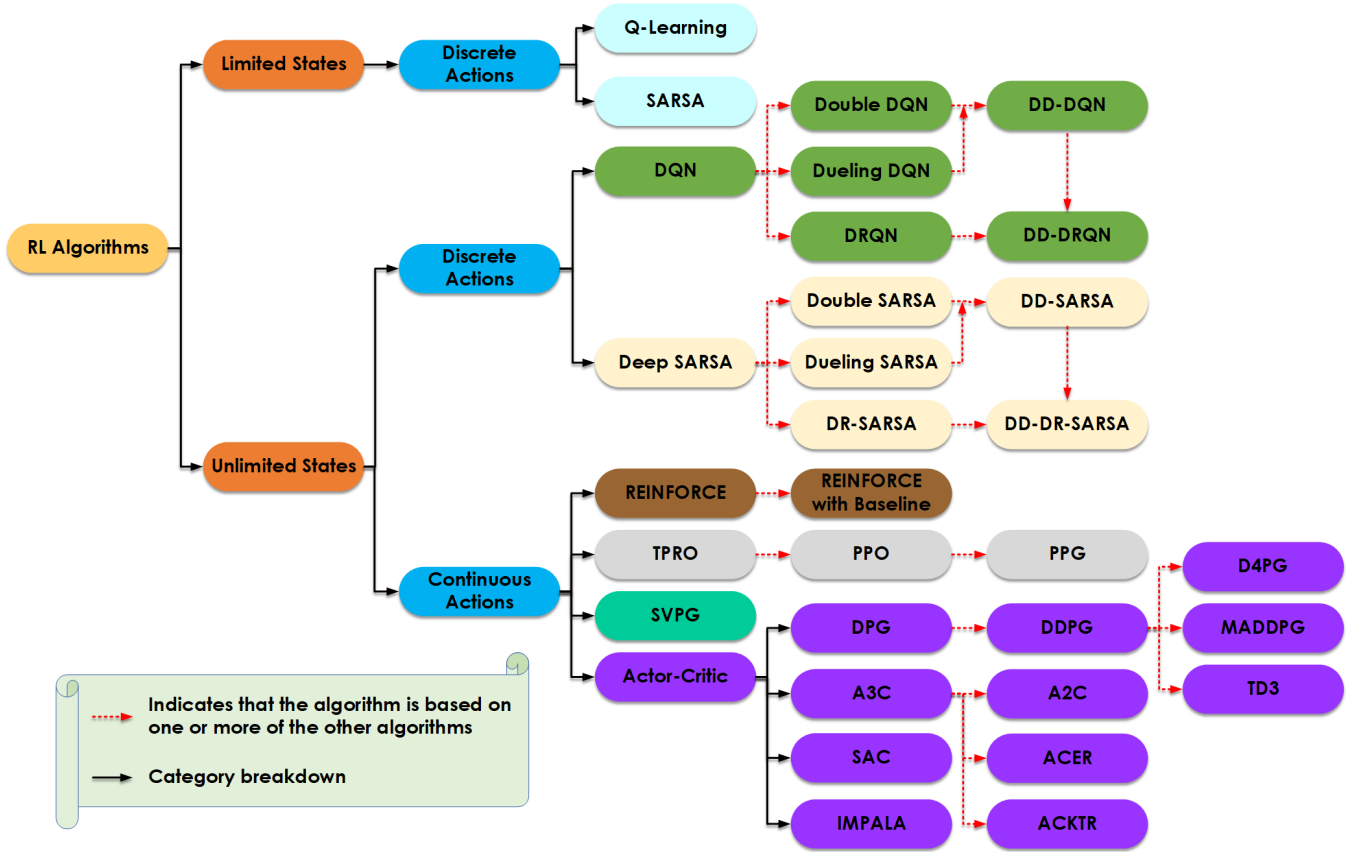
Reinforcement Algorithms Model Types

### 1.1 Reinforcement Learning Definition (Wikipedia)

Reinforcement learning is an area of machine learning inspired by behavioral psychology, concerned with how software agents take actions in an environment to maximize some notion of cumulative reward.



- Positive Reinforcement: e.g., Food.
- Negative Reinforcement: e.g., Hunger.



Reinforcement Algorithms classification based on the environment type

## 2 Markov Decision Process (MDP)

- **States:**  $s \in S$ 
  - The set of all possible situations ( $s$ ) an agent can encounter.
- **Actions:**  $a \in A$ 
  - The set of all possible moves ( $a$ ) the agent can make.
- **Rewards:**  $r \in R$ 
  - Scalar feedback ( $r$ ) received after executing an action in a state (probabilistic / stochastic).
  - Reward model:  $\Pr(r_t|s_t, a_t)$ , the probability of obtaining a reward  $r_t$  given state  $s_t$  and action  $a_t$ .
- **Transition Model:**  $\Pr(s_t|s_{t-1}, a_{t-1})$ 
  - The probability of moving to the next state ( $s_t$ ) from the current state-action pair ( $s_{t-1}, a_{t-1}$ ).
- **Discount Factor:**  $0 \leq \gamma \leq 1$ 
  - $\gamma$  is a coefficient determining the present value of future rewards.
  - Discounted:  $\gamma < 1$
  - Undiscounted:  $\gamma = 1$

### Markov Decision Process

- At every time step  $t = 0, 1, \dots$ , the agent is at state  $S_t$ .

- Takes an action  $A_t$ .
- Transitions to a new state  $S_{t+1}$ , following the probability  $S_{t+1} \sim P(\cdot|S_t, A_t)$ .
- Obtains a reward  $r_t$ .

## 2.1 Policy $\pi$

- Policy  $\pi$  maps states to actions.
- Deterministic:  $A_t = \pi(S_t)$ .
- Stochastic:  $A_t \sim \pi(\cdot|S_t)$ .

**Objective:** Find an optimal policy  $\pi^*$  that maximizes the expected long-term rewards:

$$\pi^* = \arg \max_{\pi} \sum_{t=0}^h \gamma^t \mathbb{E}_{\pi}[r_t].$$

In an MDP, this can be achieved using policy iteration or value iteration.

### Example: Student Deciding on Using LLM for Their Project

**States:**

$$S = [A, B],$$

where:

- $S_1$ : Student got access to an LLM and is deciding how to adopt it for their project.
- $S_2$ : Student has committed to a specific adoption method and is evaluating its performance.

**Actions:**

$$A = [\text{fine-tune}, \text{instruct-tune}],$$

where:

- $a_1$ : Fine-tune — Train on a small, specific dataset.
- $a_2$ : Instruct-tune — Write a detailed set of prompts and instructions to guide the LLM.

**Rewards:**

$$R = \begin{bmatrix} +10 & +8 \\ -5 & -3 \end{bmatrix},$$

where:

- +10: Fine-tune, and the model worked well.
- -5: Fine-tune, but the dataset was too small or biased.
- +8: Instruct-tune, and it was effective.
- -3: Instruct-tune, but it was ineffective.

**Transitions:**

$$P(A \rightarrow B) = \begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix}.$$

**Explanation:**

- Transition probability from  $A$  to  $B$  is 100%.
- Transition probability from  $B$  to  $A$  is 0%.

## 2.2 Deterministic vs Stochastic Policies

- Deterministic: A robot's movement in a maze.
- Stochastic: Marketing strategy for customer interactions. For example, for a sports product browser:
  - 70% recommend similar items.
  - 20% recommend fitness services.
  - 10% recommend supplements.

## 2.3 Challenge

**How do we systematically evaluate whether one policy is better than another?** Or, how do we measure how good a particular action or state is?

To address this, we introduce the concept of the **value function**. A value function tells us how much reward we can expect in the long run starting from the current state and following policy  $\pi$ .

**How do we calculate the value function?** We use **Bellman's Equation**.

## 3 Dynamic Programming

Dynamic Programming (DP) solves the problem by:

- Breaking it into smaller overlapping subproblems.
- Solving these subproblems recursively.

It relies on the **principle of optimality**:

"An optimal solution can be constructed from optimal solutions to its subproblems."

### 3.1 Bellman's Equation

The Bellman equation is a fundamental recursive formula in reinforcement learning. It calculates the optimal value of a current state by balancing immediate reward with the maximum expected future rewards:

$$V_{\pi}(s) = \mathbf{E}_{\pi} [R_{t+1} + \gamma * V_{\pi}(S_{t+1}) | S_t = s]$$

Value of state  $s$       Expected value of immediate reward      + the discounted value of next\_state      If the agent starts at state  $s$

And uses the policy to choose its actions for all time steps

$$V^{\pi}(s) = \sum_a \pi(a|s) \sum_{s'} P(s'|s, a) \left[ \underbrace{R(s, a)}_{\text{immediate reward}} + \gamma V^{\pi}(s') \right].$$

## 4 Bellman's Optimality Equation

**Purpose:** To find the optimal value of a state.

$$V(s_t) = \max_{a_t} \left[ R(s_t, a_t) + \underbrace{\gamma \sum_{s_{t+1}} P(s_{t+1} | s_t, a_t) V(s_{t+1})}_{\text{Expected value of the future state}} \right].$$

**State-Value Function**  $V^\pi(s)$  represents the expected total rewards for state  $s$  under policy  $\pi$ .

### 4.1 Bellman's Equation in Action:

#### 1. Morning Choices:

- Visit the museum (low cost, informative).
- Go to the amusement park (high cost, fun-filled).

#### 2. Considering the Afternoon:

- Museum: More time and money for other activities.
- Amusement Park: Less time and money for other activities.

#### 3. Decision-Making: Balancing immediate rewards with future possibilities.

#### Quantify Outcomes:

- Assign values to outcomes (e.g., enjoyment, cost, time).
- Museum: 40 points (savings on time and money, educational value).
- Amusement Park: 50 points (thrills and fun, higher cost, less time for other activities).

#### Future Rewards:

- Museum: Extra time and money could lead to 30 more points (e.g., visiting a park, enjoying a meal).
- Amusement Park: Fewer resources might limit you to 10 more points (e.g., a quick street food dinner).

#### Total Value:

Museum:  $40 \text{ (immediate)} + 0.9 \times 30 \text{ (future)} = 67 \text{ points.}$

Amusement Park:  $50 \text{ (immediate)} + 0.9 \times 10 \text{ (future)} = 59 \text{ points.}$

Bellman's equation is used in algorithms like **policy evaluation**, **value iteration**, and **Q-learning**.

There are two main approaches to solve for the optimal value function:

- Value Iteration
- Policy Iteration

## 4.2 Value Iteration

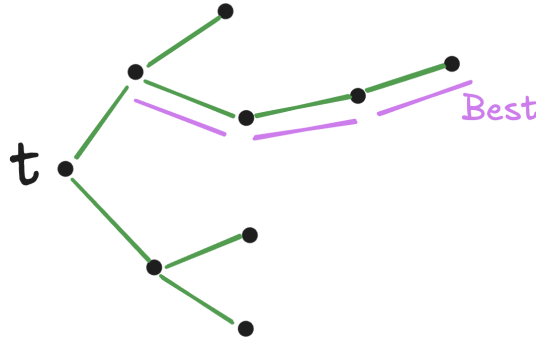
**Definition:** Value iteration is about finding the highest value and policy. Imagine solving a puzzle where you update the value of each state based on its neighbors, gradually refining it until it converges. This is achieved by repeatedly applying Bellman's Optimality Equation to update the value of each state.

$$V^*(s) = 0, \quad \forall s$$

Repeat until convergence:

$$V^*(s) = \max_a \left[ R(s, a) + \underbrace{\gamma \sum_{s'} P(s'|s, a) V^*(s')}_{\text{Expected optimal value of the future state}} \right]$$

$$\pi^*(s) = \arg \max_a \left[ R(s, a) + \gamma \sum_{s'} P(s'|s, a) V^*(s') \right]$$



You are here and will go to every possible path till the end and then pick the best route

Optimal policy will change per iteration.

**State-value function:**  $V^\pi(s)$

- represent the expected total rewards for state  $s$  under policy  $\pi$ .

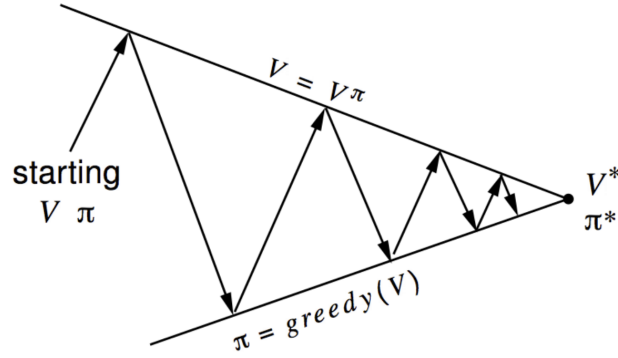
**Optimal state-value function:**  $V^*(s)$

- the highest expected reward attainable from state  $s$  under any policy.

**Key Points:**

- There is an assumption that we know the transition matrix  $P$  (in reality, we often do not).
- Optimal policy will change per iteration.
- At each iteration, you evaluate every possible path until the end and then pick the best route.

### 4.3 Policy Iteration



**Goal:** To find the value function for a given policy. Instead of focusing solely on refining the value estimate, policy iteration alternates between two steps:

- **Evaluating the current policy.**
- **Improving the current policy.**

**Policy Evaluation** Update the value function based on the current policy:

$$V^\pi(s) = R(s, \pi(s)) + \gamma \sum_{s'} P(s'|s, \pi(s)) V^\pi(s') \quad \forall s$$

**Policy Improvement** Adjust the policy based on the updated value function:

$$\pi(s) = \arg \max_a \left[ R(s, a) + \gamma \sum_{s'} P(s'|s, a) V^\pi(s') \right] \quad \forall s$$

#### 4.3.1 Policy Evaluation vs Value Iteration

Both algorithms are to solve MDPs and find the optimal policy:

Policy Iteration	Value Iteration
Alternates between policy evaluation and policy improvement.	Focuses directly on improving the value function to find the optimal policy.
Policy-centric algorithm: Starts with a policy and evaluates its values.	Value-centric algorithm.

#### 4.3.2 Modified Policy Iteration

Modified policy iteration lies between policy and value iteration. Instead of repeating until convergence, it repeats for  $k$  iterations.

- **Partial Policy Evaluation** Repeat for  $k$  times to approximate the value function for a fixed policy  $\pi$ :

$$V^\pi(s) = R(s, \pi(s)) + \gamma \sum_{s'} P(s'|s, \pi(s)) V^\pi(s') \quad \forall s.$$

At each iteration, the value of  $s$  is updated using the immediate reward  $R(s, \pi(s))$  and the discounted values of successor states  $s'$ , which are taken from the previous iteration.



- **Policy Improvement** Update the policy based on the evaluated values:

$$\pi(s) = \arg \max_a \left[ R(s, a) + \gamma \sum_{s'} P(s'|s, a) V^\pi(s') \right] \quad \forall s.$$

## 4.4 Challenge in DP

The main problem in Dynamic Programming (DP) is that we assume we have the transition matrix. In reality, we often do not.

**What if we do not know**  $P(s_{t+1}|s_t, a_t)$ ? Or the environment is too complex to calculate these transition probabilities?

**Imagine:** You are building a robot to play a game. Instead of programming it with all probabilities and rewards, you let it play for itself, record what happens, and learn from these experiences.

## 5 Model-Free Methods

**Goal:** Given a policy  $\pi$ , how can we estimate  $V^\pi(s)$  without a transition model?

- By taking samples: You are in state  $s$ , take action  $a$ , and observe where you end up.

### 5.1 Monte Carlo (MC)

Monte Carlo methods are a model-free approach to finding the value function.

#### 5.1.1 Monte Carlo Estimation for State-Value Function

We want to calculate:

$$V^\pi(s) = \mathbb{E}_\pi \left[ \sum_t \gamma^t r_t \right].$$

However, we do not have the transition probabilities. Instead, we sample and estimate as:

$$V^\pi(s) \approx \frac{1}{n(s)} \sum_{i=1}^{n(s)} \left[ \sum_t \gamma^t r_t^{(i)} \right],$$

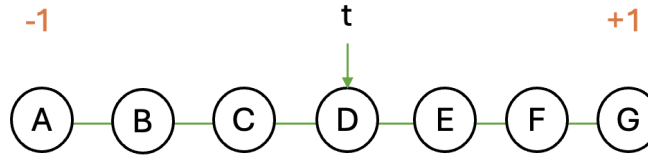
where:

- Calculate the average return for state  $s$  under a series of samples.
- $n(s)$ : Number of samples (trajectories) where  $s$  was visited.
- $r_t^{(k)}$ : Reward at time  $t$  in episode  $k$ .

$$V^\pi(s) = \frac{1}{n(s)} \sum_{k=1}^{n(s)} \left[ \sum_t \gamma^t r_t^{(k)} \right] = \frac{1}{n(s)} \sum_{k=1}^{n(s)} G_k,$$

where  $G_k$  is one sampled trajectory:

$$G_k = \sum_t \gamma^t r_t^{(k)}.$$



example of  $G_k$ :

- DEFEFG +1
- DCBCBA -1

### 5.1.2 Approximate Value Function

$$\begin{aligned}
 V_n^\pi(s) &\approx \frac{1}{n(s)} \sum_{k=1}^{n(s)} G_k \\
 &= \frac{1}{n(s)} (G_{n(s)} + \sum_{k=1}^{n(s)-1} G_k) \\
 &= \frac{1}{n(s)} (G_{n(s)} + (n(s) - 1)V_{n-1}^\pi(s)) \\
 &= \frac{1}{n(s)} G_{n(s)} + V_{n-1}^\pi(s) - \frac{1}{n(s)} V_{n-1}^\pi(s)
 \end{aligned}$$

$$\text{step size} = \frac{1}{n(s)}$$

- Incremental update

$$\underbrace{V(S_t)}_{\text{New value of state } t} \leftarrow \underbrace{V(S_t)}_{\text{Former estimation of value of state } t \text{ (= Expected return starting at that state)}} + \underbrace{\alpha}_{\text{Learning Rate}} \underbrace{[G_t - V(S_t)]}_{\text{Return at timestep } t}$$

### 5.1.3 Key Differences Between MC and DP

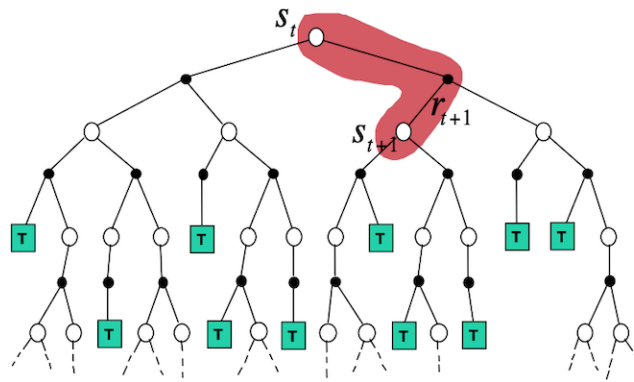
Aspect	Monte Carlo (MC)	Dynamic Programming (DP)
Model Dependency	Model-Free: Learns directly from sampled episodes.	Model-Based: Requires transition probabilities and rewards.
Computation	Relies on averaging observed returns.	Uses Bellman equations to update values recursively.
Updates	Only at the end of an episode (episodic updates).	Updates values incrementally (state by state).
Bootstrapping	Does not bootstrap; uses actual observed returns.	Bootstraps; uses estimated future values.
Usage	Suitable for environments where the model is unknown or complex to compute.	Suitable when the model is known.

### 5.1.4 Challenges of Monte Carlo Methods

- **Sample Efficiency:** Requires waiting until the end of an episode to compute the return and update value estimates.
- **Computational Efficiency:** Calculates the average return over multiple episodes, which can be expensive.
- **Bootstrapping Capability:** None; it won't work for tasks without a clear episode end.
- **High Variance:** Depends on sample returns.

How can we learn the value function without waiting for the episode to end and do it after each step? This is where bootstrapping becomes useful. Instead of waiting for the full return  $G_t$ , approximate it using the current estimate of future state values. This is where Temporal Difference (TD) comes into play.

## 6 Temporal Difference (TD) Evaluation



Temporal Difference (TD) evaluation is similar to Monte Carlo (MC), but it addresses challenges as follows:

- It updates the value estimate incrementally after each time step, using the observed reward and estimated value of the next state.
- Updates the value function using a single return.
- Uses the current value function estimate to predict future rewards (bootstrapping).

### 6.1 Value Function

$$V_n^\pi(s) \leftarrow V_{n-1}^\pi(s) + \alpha_n \left( \underbrace{r + \gamma V_{n-1}^\pi(s')}_{\text{lookahead}} - V_{n-1}^\pi(s) \right)$$

Instead of waiting for a full episode to finish in order to calculate  $G_k$ , TD uses one-step look ahead estimate based on Bellman's equation.

#### 6.1.1 Incremental Update

$$V(S_t) \leftarrow V(S_t) + \alpha [R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$$

New value of state t   Former estimation of value of state t   Learning Rate   Reward   Discounted value of next state   TD Target

$$\text{TD Target: } \underbrace{\approx r + \gamma V^\pi(s')}_{\text{one step bootstrap estimate}}$$

### 6.1.2 Theorem:

If  $\alpha_n$  is appropriately decreased with the number of visits to a state  $s$ , then  $V_n^\pi(s)$  converges to the correct value.

### 6.1.3 Sufficient Conditions for $\alpha_n$ :

1.  $\sum_n \alpha_n \rightarrow \infty$ .
2.  $\sum_n (\alpha_n)^2 < \infty$ .

Often,  $\alpha_n(s) = \frac{1}{n(s)}$ , where  $n(s)$  is the number of times state  $s$  is visited.

### 6.1.4 TD Evaluation Algorithm:

1. Set all  $v(s) = 0 \forall s$
2. Repeat - For number of episodes:
  - Execute  $\pi(s)$ .
  - Observe  $(s', r)$ .
  - Update counts:  $n(s) \leftarrow n(s) + 1$ .
  - Update learning rate:  $\alpha \leftarrow \frac{1}{n(s)}$ .
  - Update value:

$$V_n^\pi(s) \leftarrow V_{n-1}^\pi(s) + \alpha_n \left( \underbrace{r + \gamma V_{n-1}^\pi(s')}_{\text{TD target}} - \underbrace{V_{n-1}^\pi(s)}_{\text{Temporal difference / TD error}} \right)$$

- Update state:  $s \leftarrow s'$ .
3. Until  $V^\pi$  converges.

$V^\pi(s)$  represents the expected return starting from state  $s$  and following policy  $\pi$ .

$$V^\pi(s) = \mathbb{E}_\pi \left[ \sum_{t=0}^{\infty} \gamma^t R_t \mid S_0 = s \right]$$

## Example

- $A \rightarrow B$  with  $r = +10$
- $B \rightarrow A$  with  $r = +1$
- $\gamma = 0.9$
- Initial values:  $V(A) = 0, V(B) = 0$

## TD Evaluation

- **Episode 1:**
  - $A \rightarrow B$ :

- $r = 10$
- $n(A) = 1$
- $\alpha = \frac{1}{1} = 1$
- TD target =  $10 + 0.9(V(B)) = 10$
- TD update  $\Rightarrow V(A) = 0 + 1(10-0) = 10$
- $B \rightarrow A$
- $r = 1$
- $n(B) = 1$
- TD target =  $1 + 0.9V(A) = 1 + 0.9(10) = 10$
- TD update  $\Rightarrow V(B) = 0 + 1(10-0) = 10$

• **Episode 2**

- $A \rightarrow B$
- $r = +10$
- $n(A) = 2$
- $\alpha = \frac{1}{2} = 0.5$
- TD target =  $10 + 0.9(10) = 19$
- TD update  $\Rightarrow V(A) = 10 + 0.5(19-10) = 14.5$
- $B \rightarrow A$
- $r = +1$
- $n(B) = 2$
- TD target =  $1 + 0.9V(A) = 1 + 0.9(14.5) = 14.05$
- TD update  $\Rightarrow V(B) = 10 + 0.5(14.05-10) = 12.025$

• **Episode 3**

- $A \rightarrow B$
- $r = +10$
- $n(A) = 3$
- $\alpha = \frac{1}{3} \approx 0.333$
- TD target =  $10 + 0.9(12.025) = 20.8225$
- TD update  $\Rightarrow V(A) = 14.5 + 0.333(20.8225-14.5) = 16.6017$
- $B \rightarrow A$ :
- $r = +1$
- $n(B) = 3$
- TD target =  $1 + 0.9V(A) = 1 + 0.9(16.6017) = 16.0415$
- TD update  $\Rightarrow V(B) = 12.025 + 0.333(16.0415-12.025) = 13.345$

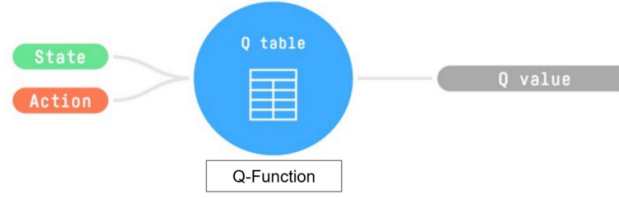
•  $V^\pi(A) \approx 16.60$

•  $V^\pi(B) \approx 13.30$

In many problems, it is not enough to just evaluate the state values. To make decisions, we need to evaluate specific actions in each state. This is where  $Q(s, a)$  comes into play.

## 6.2 State-Action Value $Q^\pi(s, a)$

$Q^\pi(s, a)$  represents the expected return after taking action  $a$  in state  $s$  and following policy  $\pi$ . What is the value if I take action  $a$  and then behave optimally after?



### 6.2.1 TD Update for $Q(s, a)$

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left[ \underbrace{r + \gamma Q(s', a')}_{\text{TD target}} - Q(s, a) \right]$$

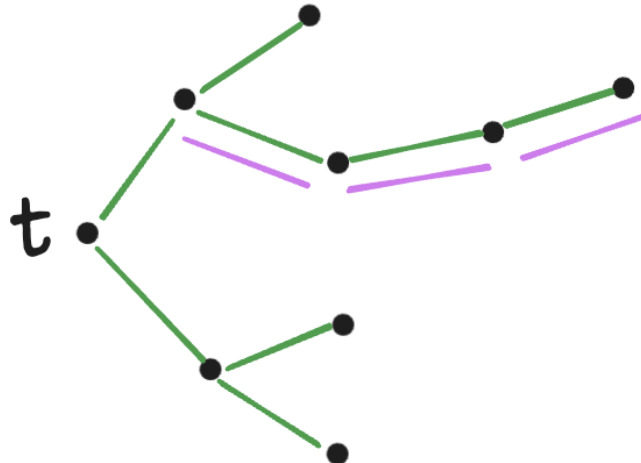
Additionally, we can write  $Q(s, a)$  as:

$$\begin{aligned} Q^\pi(s, a) &= \underbrace{\mathbb{E}[r(s, a)]}_{\text{expected immediate reward}} + \gamma \underbrace{\sum_{s'} P(s'|s, a) V^\pi(s')}_{\text{expected value of next state}} \\ &= \mathbb{E} \left[ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid S_t = s, a_t = a, \pi \right] \end{aligned}$$

where  $r_{t+k+1}$  is reward after  $k$  transition starting from  $s$  and taking action  $a$ .

$$\begin{aligned} V^\pi(s) &= E_\pi[Q^\pi(s, a)] \\ Q^\pi(s, a) &= E_\pi[r + \gamma V^\pi(s')] \\ V^\pi(s') &= \sum_{a'} \pi(a'|s') Q^\pi(s', a') \end{aligned}$$

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



We are here, our agent knows the optimal value from each possible next states. So in  $Q$ , we only focus on time  $t$  and what is the next action to take if I know the optimal values after.

### 6.2.2 Key Concepts

- In Q-learning, the first action is unknown and needs to be found, but the assumption is that after taking the first action, the remaining ones are known/optimal.
- Upto this point, in order to find  $\pi^*$ , we needed to find  $V^*$  which means we need to find the expected value of  $s$  by repeating the trajectory from  $s$  to the end series of times.
- Instead,  $Q$  tells us the expected maximum reward at state  $s$  if you take one action and then behave optimally
- The assumption is that all future states are already optimized and we only focus on current state
- We don't know the optimal but assume that if any current action is optimal the rest will be optimal

**Goal:** The goal of  $Q(s, a)$  is to evaluate the effectiveness of each action within each state. By comparing the overall action values of all actions in state  $s$ , the agent can determine which action is the best at state  $s$ .

### 6.2.3 Example

**State A:**

- $a_1 : A \rightarrow B, r = +10$
- $a_2 : A \rightarrow C, r = 0$
- $a_3 : A \rightarrow A, r = +2$

**State B:**

- $a_1 : B \rightarrow A, r = +1$
- $a_2 : B \rightarrow C, r = +15$
- $a_3 : B \rightarrow B, r = +3$

**Policy:**

	$a_1$	$a_2$	$a_3$
$A$	0.5	0.3	0.2
$B$	0.4	0.4	0.2

**Initial Q-values:**

$$Q(A, a_1) = 0, Q(A, a_2) = 0, Q(A, a_3) = 0$$
$$Q(B, a_1) = 0, Q(B, a_2) = 0, Q(B, a_3) = 0$$

**Initial  $n(s)$  values:**

$$n(A, a_1) = 0, n(A, a_2) = 0, n(A, a_3) = 0$$
$$n(B, a_1) = 0, n(B, a_2) = 0, n(B, a_3) = 0$$

**Episodes**

**Step 1:**  $\pi(A) \rightarrow a_1$

- $r = +10$
- $n(A, 1) \pm 1$

- $\alpha = \frac{1}{1} = 1$
- TD target =  $10 + 0.9 \max(Q(B, \cdot)) = 10$
- TD update  $\Rightarrow Q(A, 1) = 0 + 1(10 - 0) = 10$
- $\pi(B) \rightarrow a_1$
- $r = +1$
- $n(B, 1) \pm 1$
- TD target =  $1 + 0.9 \max(Q(A, \cdot)) = 2 + 0.9(10) = 10$
- TD update  $\Rightarrow Q(B, a_1) = 0 + 1(10 - 0) = 10$

**Step 2:**  $\pi(A) \rightarrow a_3$

- $r = +2$
- $n(A, 3) \pm 1$
- $\alpha = \frac{1}{1} = 1$
- TD target =  $2 + 0.9 \max(Q(A, \cdot)) = 2 + 0.9(10) = 11$
- TD update  $\Rightarrow Q(A, 3) = 0 + 1(11 - 0) = 11$

$\pi(\mathbf{A}) \rightarrow \mathbf{a}_1$

- $r = +10$
- $n(A, 1) \pm 1$
- $\alpha = \frac{1}{2} = 0.5$
- TD target =  $10 + 0.9 \max(Q(B, \cdot)) = 10 + 0.9(10) = 19$
- TD update  $\Rightarrow Q(A, 1) = 10 + 0.5(19 - 10) = 14.5$

$\pi(\mathbf{B}) \rightarrow \mathbf{a}_3$

- $r = +3$
- $n(B, 3) \pm 1$
- $\alpha = 1$
- TD target =  $3 + 0.9 \max(Q(A, \cdot)) = 3 + 0.9(10) = 12$
- TD update  $\Rightarrow Q(B, 3) = 0 + 1(12 - 0) = 12$

### 6.3 Bellman's Equation

- Bellman's equations describe the relationship between the value of a state (or state-action pair) and the expected utility of future rewards in a Markov Decision Process (MDP).
- These equations are foundational for solving problems in Reinforcement Learning (RL), particularly in algorithms like Temporal Difference (TD) learning and model-based approaches.



### 6.3.1 Bellman Optimality Equation for State Value $V^*(s)$ :

$$\begin{aligned} V^*(s) &= \mathbb{E}[r + \gamma V^*(s') \mid s, a] \\ &= \sum_r \Pr(r \mid s, a) \cdot r + \gamma \sum_{s'} \Pr(s' \mid s, a) V^*(s'). \end{aligned}$$

**Key Insight:**  $V^*(s)$  represents the maximum expected return starting from state  $s$  and acting optimally.

### 6.3.2 Bellman Optimality Equation for Action-Value $Q^*(s, a)$ :

$$Q^*(s, a) = \mathbb{E}[r \mid s, a] + \gamma \sum_{s'} \Pr(s' \mid s, a) \max_{a'} Q^*(s', a').$$

**Key Insight:**  $Q^*(s, a)$  represents the maximum expected return for taking action  $a$  in state  $s$  and following the optimal policy thereafter.

### 6.3.3 Relationship Between Optimal State Values and $\pi$ (Policy Values)

1. The optimal state value  $V^*(s)$  is obtained by choosing the best action-value  $Q^*(s, a)$ :

$$V^*(s) = \max_a Q^*(s, a).$$

2. The optimal policy  $\pi^*(s)$  is the action that maximizes the action-value  $Q^*(s, a)$ :

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

## 6.4 Monte Carlo Methods

- Monte Carlo (MC) methods are particularly useful as they do not require a model of the environment (i.e., transition probabilities and rewards are not required beforehand).
- These methods rely on the completion of episodes and the accumulation of actual rewards to learn the action-value function  $Q(s, a)$ .

### 6.4.1 Steps

#### Step 1: Policy Execution

- Follow a given policy (which could be fixed or derived from  $Q(s, a)$ ).
- Execute the policy and generate episodes.

#### Step 2: Episode Recording

- Record all states visited, actions taken, and rewards collected up to the terminal state for each episode.

#### Step 3: Calculate Returns

- For each state-action pair  $(s, a)$  appearing in each episode, calculate the return  $G_t$  from that state-action pair onward until the end of the episode:

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots + \gamma^{T-t-1} r_T$$

#### Step 4: Update $Q(s, a)$

- Use either the **first-visit** or **every-visit** MC method to update  $Q(s, a)$ :
  - **First visit:** Update  $Q(s, a)$  for the first occurrence of each state-action pair.
  - **Every visit:** Update  $Q(s, a)$  for every time a state-action pair is visited.
- Update Rule:

$$Q(s, a) \leftarrow Q(s, a) + \alpha [G_t - Q(s, a)]$$

Where:

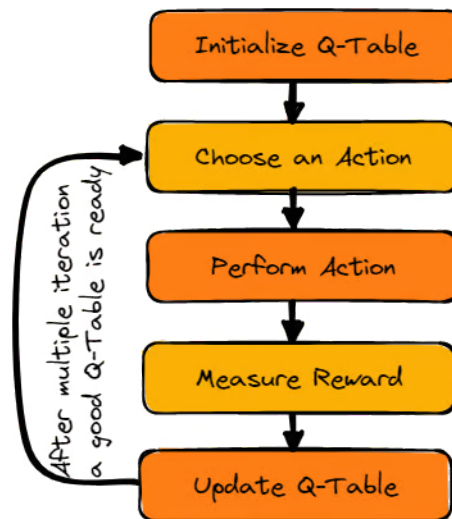
- $G_t$  is the return calculated for that state-action pair.
- $\alpha$  is the learning rate.

#### Step 5: Policy Improvement

- If performing a policy improvement step, update the policy  $\pi(s)$  as follows:

$$\pi(s) \leftarrow \operatorname{argmax}_a Q(s, a)$$

## 6.5 Q-Learning



- Q-learning is a model-free, off-policy Temporal Difference (TD) control algorithm.
- It is based on Bellman's optimality equation for action values.
- The method updates the Q-value function  $Q(s, a)$  incrementally using bootstrapping and observed transitions.

#### 6.5.1 Bellman's Optimality Equation for Q-Learning:

$$Q(s, a) = \mathbb{E} [r + \gamma \max_{a'} Q(s', a')]$$

This represents the discounted value of the best possible action in the next state  $s'$ .

### 6.5.2 Q-Learning Update Rule:

Q-learning uses bootstrapping to update  $Q(s, a)$  incrementally. It's off-policy because it updates  $Q(s, a)$  based on optimal next action and not a policy that agent might follow:

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

New  
Q-value  
estimation

Former  
Q-value  
estimation

Learning  
Rate

Immediate  
Reward

Discounted Estimate  
optimal Q-value  
of next state

Former  
Q-value  
estimation

TD Target

---

TD Error

The term  $r + \gamma \max_{a'} Q(s', a')$  is known as the **bootstrap target**, and the difference is the **TD error**.

1	2	3	4	5					
6	7	8	9	10					
11	12	13	14	15					
16	17	18	19	20					
21	22	23	24	25					

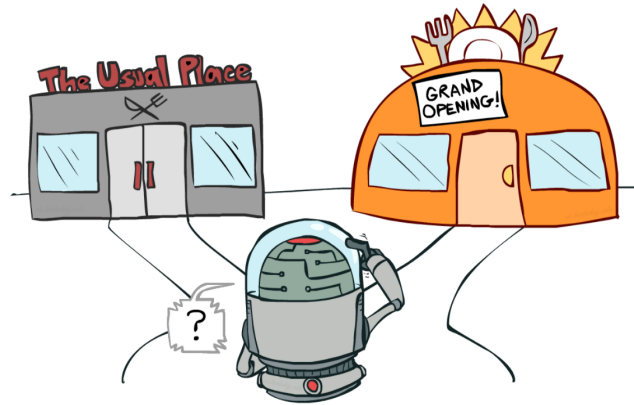
  

	↑	↓	←	→
1	-	+1	-	+1
2	-	+1	-1	+1
3	-	+1	-1	+1
4	-	+1	-1	-1
5	-	+1	+1	-
...				
23	+1	-	-1	+1
24	+1	-	-1	-1
25	+1	-	+1	-

### 6.5.3 Steps to Update $Q(s, a)$ :

1. Initialize  $Q(s, a)$  values for all state-action pairs.
2. Repeat until convergence (for a certain number of episodes):
  - (a) Choose an action  $a$  from state  $s$ .
  - (b) Take action  $a$ , observe reward  $r$  and the next state  $s'$ .
  - (c) Update  $Q(s, a)$  using:
 
$$Q(s, a) \leftarrow Q(s, a) + \alpha [r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$$
  - (d) Move to the next state  $s'$ .
3. Continue until the next episode.

#### 6.5.4 Exploration vs Exploitation:



Q-learning uses strategies like  $\epsilon$ -greedy to balance exploration and exploitation:

- **Exploration:** Taking random actions to discover new paths and outcomes.
- **Exploitation:** Choosing the action that promises the maximum reward based on current knowledge.

## 7 SARSA (State-Action-Reward-State-Action)

- SARSA is a model-free, on-policy Temporal Difference (TD) learning algorithm used to learn  $Q(s, a)$ .
- The name SARSA comes from the sequence of events it uses to update  $Q(s, a)$ : **State, Action, Reward, Next State, and Next Action**.
- **On-Policy Nature:** SARSA updates  $Q(s, a)$  based on the current policy the agent follows. This means the update depends on the actual actions the agent takes.

### 7.1 SARSA Update Rule

$$Q(s, a) \leftarrow Q(s, a) + \alpha [r + \gamma Q(s', a') - Q(s, a)]$$

### 7.2 SARSA Algorithm

#### 1. Initialization:

- Initialize  $Q(s, a)$  randomly for all  $s, a$ .
- Set learning rate  $\alpha$ , discount factor  $\gamma$ , and exploration rate  $\epsilon$ .

#### 2. For each episode:

- (a) Initialize  $s$ .
- (b) Choose an action  $a$  based on the  $\epsilon$ -greedy policy (balancing exploration and exploitation).



3. Repeat until the terminal state is reached:

- Take action  $a$ , observe reward  $r$ , and the next state  $s'$ .
- Choose the next action  $a'$  using the  $\epsilon$ -greedy policy.
- Update  $Q(s, a)$  using:

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)]$$

- Set  $s = s'$  and  $a = a'$ .
- Move to the next state-action pair.

4. End of episode.

### 7.3 Comparison to Q-Learning

Aspect	SARSA	Q-Learning
Policy	On-Policy: Updates based on actions taken.	Off-Policy: Updates using optimal actions.
Update Rule	$r + \gamma Q(s', a')$	$r + \gamma \max_{a'} Q(s', a')$
Exploration	Reflects current policy (e.g., $\epsilon$ -greedy).	Assumes optimal policy, even during exploration.
Risk Behavior	More cautious, as it accounts for exploratory actions.	More aggressive, assuming optimal future behavior.

## Example

The reward function:

- **0**: Going to a state with no cheese in it.
- **+1**: Going to a state with a small cheese in it.
- **+10**: Going to the state with the big pile of cheese.
- **-10**: Going to the state with the poison and thus die.













Details:

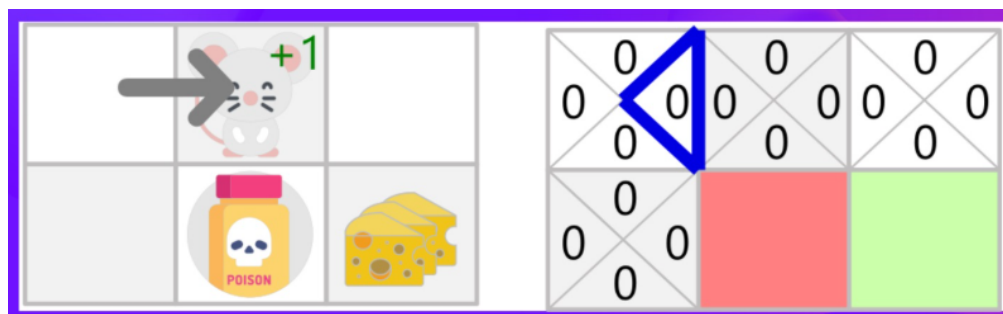
- You always start at the same starting point.
- The goal: eat the big pile of cheese (at the bottom right-hand corner) and avoid the poison.
- The episode ends if we:
  - Eat the poison,
  - Eat the big pile of cheese,
  - Spend more than 5 steps.
- **Learning rate:** 0.1
- **Gamma:** 0.99

$t = 1$

Step 1: Initialize Q-Table











				
	0	0	0	0
	0	0	0	0
	0	0	0	0
	0	0	0	0
	0	0	0	0
	0	0	0	0

Step 2: Choose an action using the Epsilon Greedy Strategy



Step 3:

Take the action and observe  $R_{t+1}$  and  $S_{t+1}$

				
	0	0.1	0	0
	0	0	0	0
	0	0	0	0
	0	0	0	0
	0	0	0	0
	0	0	0	0

#### 7.4 Step 4: Update Q function

$$\underbrace{Q(S_t, A_t)}_{\text{New Q-value estimation}} \leftarrow \underbrace{Q(S_t, A_t)}_{\text{Former Q-value estimation}} + \underbrace{\alpha}_{\text{Learning Rate}} [\underbrace{R_{t+1}}_{\text{Immediate Reward}} + \underbrace{\gamma \max_a Q(S_{t+1}, a)}_{\text{Discounted Estimate optimal Q-value of next state}} - \underbrace{Q(S_t, A_t)}_{\text{Former Q-value estimation}}]$$

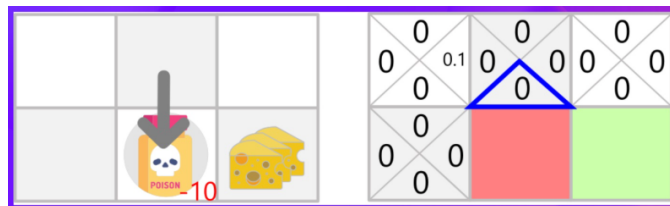
TD Target
TD Error

$$Q(\text{Initial state, Right}) = 0 + 0.1 \times [1 + 0.99 \times 0 - 0]$$

$$Q(\text{Initial state, Right}) = 0.1$$

$t = 2$

Step 1: Choose an action using the Epsilon Greedy Strategy



Step 3:

Take the action and observe  $R_{t+1}$  and  $S_{t+1}$



## 7.5 Step 4: Update Q function

$$\underbrace{Q(S_t, A_t)}_{\text{New Q-value estimation}} \leftarrow \underbrace{Q(S_t, A_t)}_{\text{Former Q-value estimation}} + \underbrace{\alpha}_{\text{Learning Rate}} [\underbrace{R_{t+1}}_{\text{Immediate Reward}} + \underbrace{\gamma \max_a Q(S_{t+1}, a)}_{\text{Discounted Estimate optimal Q-value of next state}} - \underbrace{Q(S_t, A_t)}_{\text{Former Q-value estimation}}]$$

TD Target
TD Error

$$Q(\text{Initial state, Right}) = 0 + 0.1 \times [-10 + 0.99 \times 0 - 0]$$

$$Q(\text{Initial state, Right}) = -1$$

	←	→	↑	↓
☠	0	0.1	0	0
🧀	0	0	0	-1
□	0	0	0	0
■	0	0	0	0
🧀	0	0	0	0
🧀	0	0	0	0

## 7.6 Challenges of Q-Learning

When dealing with large or continuous state spaces, Q-learning faces the following challenges:

- Table-Based Representation
  - In Q-learning, we store  $Q(s, a)$  values in a table.
  - This is problematic when dealing with continuous or large action spaces as memory usage grows exponentially.
- Generalization
  - If an agent encounters a state it hasn't seen before, it cannot estimate  $Q(s, a)$ .



### 7.6.1 Solution

The idea is to replace the Q-table with a Neural Network (NN) that can approximate  $Q(s, a)$ .

- Before Deep Learning (DL): Linear estimators were used to approximate unknown values (e.g.,  $Q, \pi, V$ ).
- With Neural Networks: NNs can estimate these values effectively, forming the foundation of Deep Reinforcement Learning (Deep RL).

## 7.7 Deep Q-Learning

### 7.7.1 Q-Function Approximation

- Assume we have a neural network (NN) parameterized by weights  $w$  and inputs  $x$ .
- Let  $s$  be a  $d$ -dimensional state vector:

$$s = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix}$$

- The Q-function is approximated as:

$$Q(s, a) \approx g(w, s) \quad (\text{using a neural network}).$$

### 7.7.2 Neural Network Architecture

The architecture used in Deep Q-Learning involves convolutional layers for feature extraction, followed by fully connected layers for decision making. The final output represents the Q-values for each possible action.

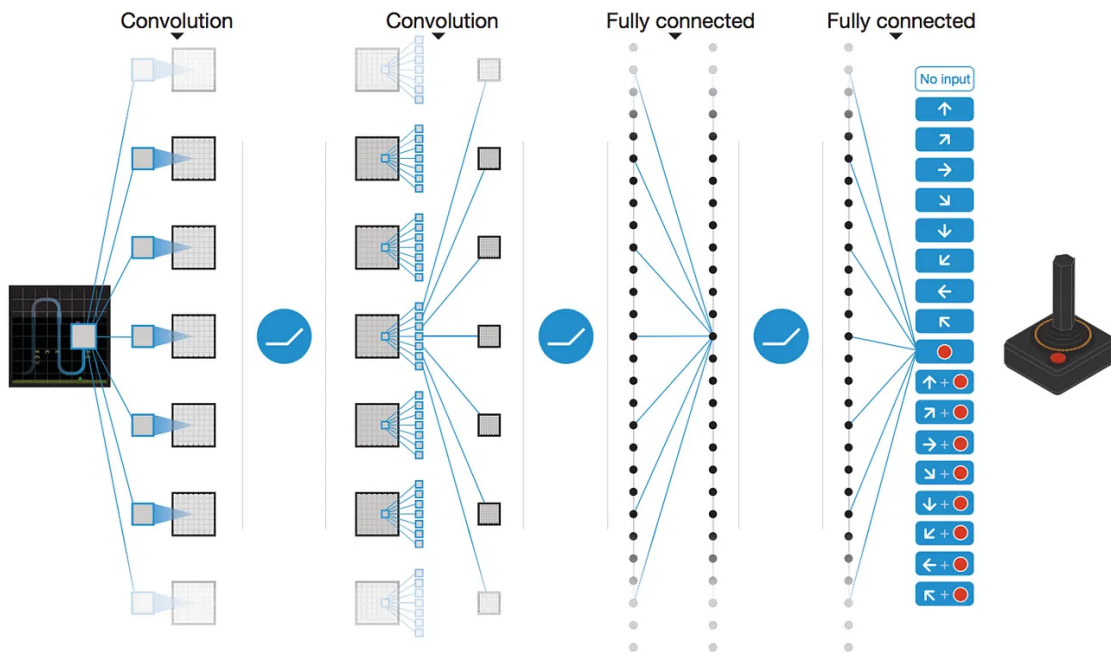


Figure 1: Deep Q-Learning Architecture: Convolutional layers extract features from the input state, fully connected layers compute Q-values for each action.

### 7.7.3 Q-Learning Recap

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

### 7.7.4 Components in Deep Q-Learning

#### Q-value Estimate

- $Q_w(s, a)$ : The Q-value estimated by the neural network (referred to as the Q-network).

#### Target Network

- $Q'_w(s', a')$ : A separate network used to calculate the target value to improve stability.

#### Loss Function

- The mean-squared error between the predicted and target Q-values:

$$\text{Error}(w) = J(w) = \frac{1}{2} [Q_w(s, a) - (r + \gamma \max_{a'} Q'_w(s', a'))]^2$$

#### Gradient Descent

- Compute the gradient of the loss with respect to  $w$ :

$$\frac{\partial J}{\partial w} = [Q_w(s, a) - (r + \gamma \max_{a'} Q'_w(s', a'))] \cdot \frac{\partial Q_w(s, a)}{\partial w}$$

#### Weight Update

- Update the weights using gradient descent:

$$w \leftarrow w - \eta \frac{\partial J}{\partial w}$$

#### Transition to Next State

- Set  $s \leftarrow s'$  and repeat.

### 7.7.5 Challenges

- **Instability Issues:** Gradients might diverge during training, leading to unstable behavior in deep Q-learning.

### 7.7.6 Solutions to Stabilize Training

- Dual Network Approach
  - Use two separate networks:
    - \* One for  $Q(s, a)$  (Q-network).
    - \* One for the target value  $r + \gamma \max_{a'} Q(s', a')$  (Target network).
  - **Key Idea:**
    - \* Keep the weights  $w'$  of the target network fixed for some iterations.
    - \* Update  $w'$  using the weights  $w$  of the Q-network after a fixed number of steps.

- \* Repeat this process iteratively.
- Experience Replay
  - Store previous experiences and sample from them during training.
  - **Purpose:** Break the correlation among consecutive updates by randomly sampling from the replay buffer.

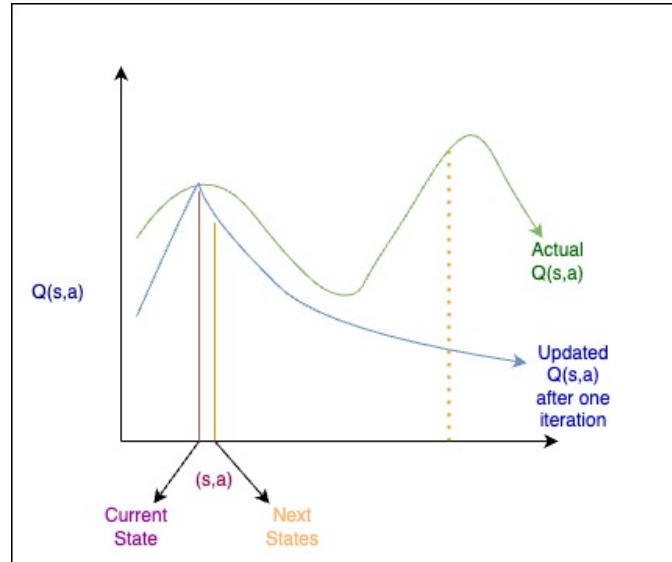


Figure 2: The graph illustrates the relationship between predicted and actual Q-values in reinforcement learning.

- $Q(s, a)$ : The predicted Q-values.
- Actual  $Q(s, a)$ : Target values computed from past experiences.
- **Current State**  $(s, a)$ : The state-action pair being updated.
- **Next States**: Used to compute the target values.

#### 7.7.7 Key Points

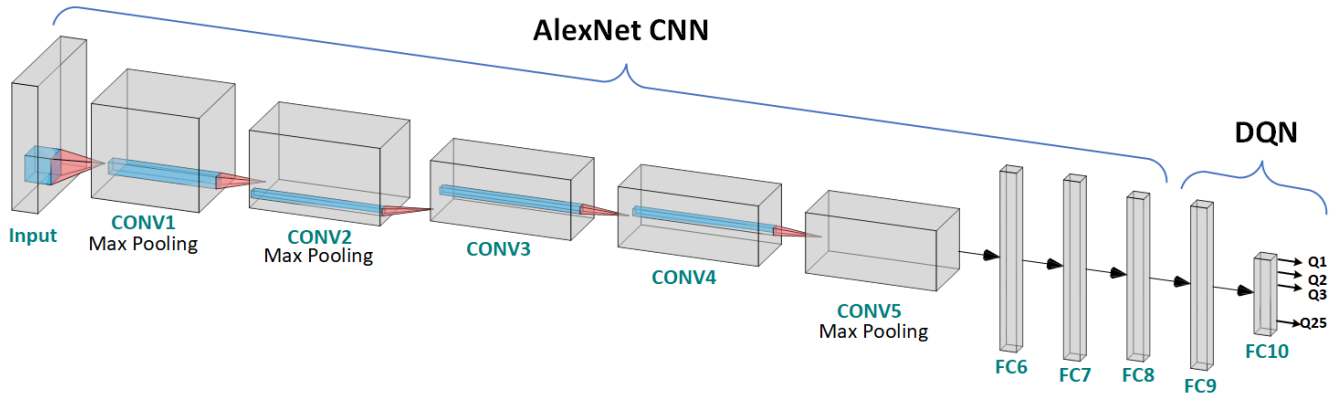
- When we update the weights  $w$ , it provides a new estimate of  $Q(s, a)$ .

#### 7.7.8 Challenges

- **Correlated States:** The next states are technically correlated and not independent and identically distributed (i.i.d.).
- **Divergence for Future States:** The updates for  $Q(s, a)$  work well for immediate states but might diverge for future states.

#### 7.7.9 Solution

- Randomly replay a point from past experiences to ensure that  $Q(s, a)$  does not diverge.
- Alternatively, use multiple points and add them to the input batch during training.



DQN using AlexNet CNN

### 7.7.10 DQN Algorithm

#### Steps

##### 1. Initialization:

- Initialize replay buffer  $D$ .
- Initialize Q-network with random weights  $w$ .
- Initialize target network with  $w' \leftarrow w$ .

##### 2. For each episode:

- Initialize state  $s$ .

##### 3. Loop until the terminal state is reached:

- Choose action  $a$  using an  $\epsilon$ -greedy policy based on  $Q(s, a; w)$ .
- Take action  $a$ , observe reward  $r$ , and next state  $s'$ .
- Store transition  $(s, a, s', r)$  in  $D$ .
- Sample random minibatches of  $(s, a, s', r)$  from  $D$ .
- Compute TD target for each transition:

$$y = r + \gamma \max_{a'} Q(s', a'; w').$$

- Calculate the loss:

$$\text{Loss} = [y - Q(s, a; w)]^2 \approx \mathbb{E}_{(s, a, s', r) \sim D} [y - Q(s, a; w)]^2.$$

- Compute gradient of  $w$  and update  $w$ :

$$w \leftarrow w - \eta \nabla_w [y - Q(s, a; w)]^2.$$

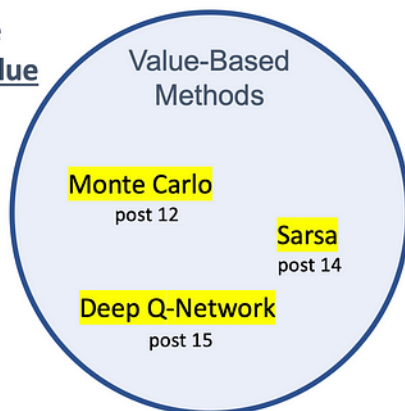
- Set  $s \leftarrow s'$ .

##### 4. Periodically:

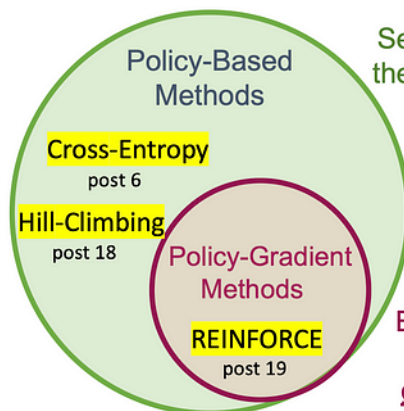
- Update  $w' \leftarrow w$ .

There are further advancements methods developed over Deep Q-Networks (DQN), which are designed to find the policy  $\pi$  directly instead of solving for it through value-based methods like Q-learning or DQN.

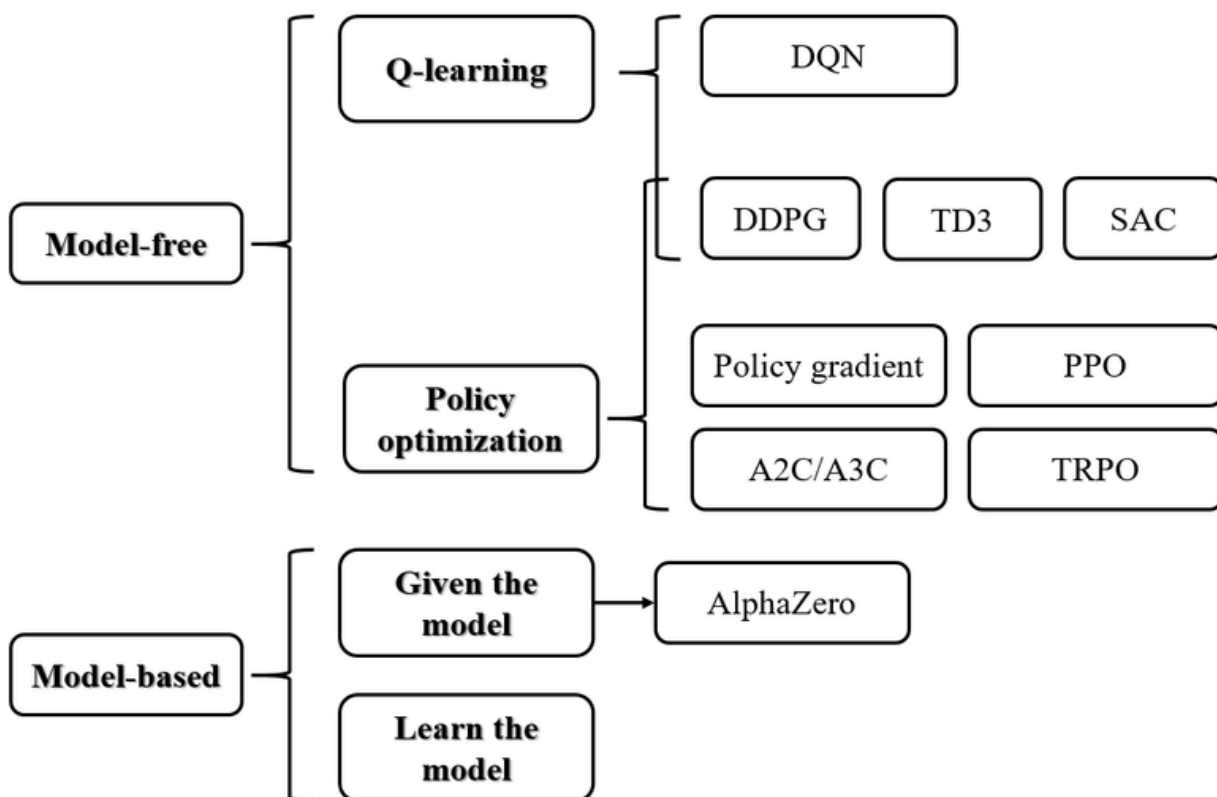
Estimate of the optimal action-value function



Search directly for the optimal policy



Estimate the best weights by gradient ascent



## References

- [1] Ali Ghodsi. *DataScienceCoursesUW*.
- [2] HuggingFace. “q-learning-recap.”.