

# Reinforcement Learning

## Lecture 5: Value-Based Learning I - Q-Learning

Taehoon Kim

Sogang University MIMIC Lab  
<https://mimic-lab.com>

Fall Semester 2025

# Learning Objectives

## By the end of this lecture, you will:

- 1 Understand the Bellman optimality operator and its properties
- 2 Implement tabular Q-learning from scratch
- 3 Design effective exploration and learning rate schedules
- 4 Analyze Q-learning behavior in stochastic environments
- 5 Compare Q-learning with SARSA
- 6 Implement Double Q-learning to reduce bias

## Prerequisites:

- Markov Decision Processes (Lecture 4)
- Dynamic programming concepts
- PyTorch basics (Lecture 2)

# Why Q-Learning?

## Model-Based Limitations:

- Requires complete MDP model
- $P(s'|s, a)$  often unknown
- Complex dynamics hard to model
- Computational complexity

## Q-Learning Advantages:

- Model-free learning
- Learns from experience
- Off-policy algorithm
- Converges to optimal policy

## Key Insight

Learn action-values  $Q(s, a)$  directly from samples without knowing transition dynamics

# Action-Value Function

## Definition (Action-Value Function)

The action-value function  $Q^\pi(s, a)$  of policy  $\pi$  is:

$$Q^\pi(s, a) = \mathbb{E}_\pi \left[ \sum_{t=0}^{\infty} \gamma^t r_{t+1} \mid s_0 = s, a_0 = a \right]$$

### Interpretation:

- Expected return starting from state  $s$
- Taking action  $a$  first
- Following policy  $\pi$  thereafter

### Optimal action-value function:

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

## Connection to last lecture

Q-learning extends the Bellman equation for the state value  $v(s)$  to the action-value  $Q(s, a)$  so that we learn not just “how good a state is” but also “how good each action is in that state.”

# Bellman Optimality Equation

## Theorem (Bellman Optimality for $Q^*$ )

$$Q^*(s, a) = R(s, a) + \gamma \sum_{s'} P(s'|s, a) \max_{a'} Q^*(s', a')$$

### Key Components:

- $R(s, a)$ : Immediate reward
- $P(s'|s, a)$ : Transition probability
- $\max_{a'} Q^*(s', a')$ : Best future value
- $\gamma$ : Discount factor

*Intuition:* the term  $\max_{a'} Q^*(s', a')$  means that when we evaluate taking action  $a$  at  $s$ , we assume that from the next state  $s'$  onward we will act optimally. Thus, current value equals immediate reward plus the best possible future value. **Optimal Policy:**

$$\pi^*(s) = \arg \max_a Q^*(s, a)$$

# Bellman Optimality Operator

## Definition (Operator $T^*$ )

For any  $Q : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ :

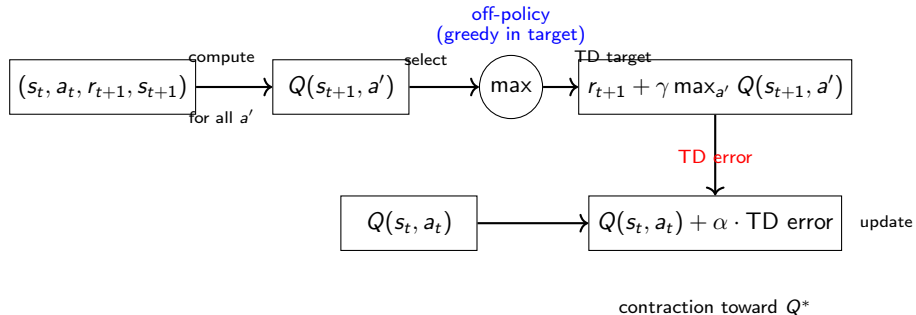
$$(T^*Q)(s, a) = R(s, a) + \gamma \sum_{s'} P(s'|s, a) \max_{a'} Q(s', a')$$

## Properties:

- ➊ **Contraction:**  $\|T^*Q_1 - T^*Q_2\|_\infty \leq \gamma \|Q_1 - Q_2\|_\infty$
- ➋ **Fixed Point:**  $Q^* = T^*Q^*$
- ➌ **Uniqueness:**  $Q^*$  is the unique fixed point
- ➍ **Convergence:**  $\lim_{n \rightarrow \infty} (T^*)^n Q = Q^*$

*Why contraction implies convergence:* applying  $T^*$  repeatedly pulls any two Q-functions closer by at least a factor  $\gamma < 1$ , so iterations must collapse to the unique fixed point  $Q^*$ .

# Q-Learning Target Construction



# From Value Iteration to Q-Learning

## Value Iteration (Model-Based):

```
for each iteration do
  for all  $(s, a)$  pairs do
     $Q(s, a) \leftarrow$ 
       $R(s, a) + \gamma \sum_{s'} P(s'|s, a) \max_{a'} Q(s', a')$ 
  end for
end for
```

**Requires:** Full MDP model

## Q-Learning (Model-Free):

```
for each sample  $(s, a, r, s')$  do
   $Q(s, a) \leftarrow$ 
     $Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$ 
end for
```

**Requires:** Only samples

## Key Difference

Q-learning uses samples to approximate the Bellman backup



# Q-Learning Update Rule

## Temporal Difference (TD) Update

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \cdot \text{TD error}$$

where

$$\text{TD error} = \underbrace{r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t)}_{\text{TD target}}$$

*From Bellman to samples:* this update is a sample-based approximation to the Bellman optimality backup—replace the expectation over  $s'$  by a single observed transition and bootstrap with  $\max_{a'} Q(s_{t+1}, a')$ . **Components:**

- $\alpha \in (0, 1]$ : Learning rate (step size)
- $r_{t+1}$ : Observed reward
- $\gamma \in [0, 1)$ : Discount factor
- $\max_{a'} Q(s_{t+1}, a')$ : Bootstrap estimate

**Off-Policy:** Uses max (greedy) action for target, regardless of behavior policy

# Q-Learning Algorithm

---

**Algorithm 1** Tabular Q-Learning

---

```
1: Initialize  $Q(s, a)$  arbitrarily for all  $s \in \mathcal{S}, a \in \mathcal{A}$ 
2: for each episode do
3:   Initialize state  $s$ 
4:   while  $s$  is not terminal do
5:     Choose  $a$  from  $s$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)
6:     Take action  $a$ , observe  $r, s'$ 
7:      $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$ 
8:      $s \leftarrow s'$ 
9:   end while
10: end for
```

---

**Note:** The max operation makes it off-policy

# Exploration vs Exploitation

## $\epsilon$ -greedy Policy:

$$\pi_{\epsilon}(a|s) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{|\mathcal{A}|} & \text{if } a = \arg \max_{a'} Q(s, a') \\ \frac{\epsilon}{|\mathcal{A}|} & \text{otherwise} \end{cases}$$

## Alternative Strategies:

- Boltzmann exploration
- Upper Confidence Bounds (UCB)
- Thompson sampling
- Optimistic initialization

## Exploration Schedule

Common choice:  $\epsilon_t = \max(\epsilon_{min}, \epsilon_0 \cdot \text{decay}^t)$

*Why  $\epsilon$ -greedy is a strong default:* it is simple, cheap to compute, and as  $\epsilon \rightarrow 0$  the behavior policy becomes greedy, enabling convergence under the usual conditions.

# Q-Learning Convergence

## Theorem (Watkins & Dayan, 1992)

*Q-learning converges to  $Q^*$  with probability 1 if:*

- ❶ *All state-action pairs are visited infinitely often*
- ❷ *Learning rate satisfies Robbins-Monro conditions:*
  - $\sum_{t=0}^{\infty} \alpha_t(s, a) = \infty$  for all  $(s, a)$
  - $\sum_{t=0}^{\infty} \alpha_t^2(s, a) < \infty$  for all  $(s, a)$

### Practical Schedules:

- $\alpha_t = \frac{1}{1+t}$  satisfies conditions
- $\alpha_t = \frac{1}{\sqrt{1+t}}$  satisfies conditions
- Constant  $\alpha$  doesn't guarantee convergence but often works

# GLIE: Greedy in the Limit with Infinite Exploration

## Definition (GLIE)

A policy sequence  $\{\pi_t\}$  is GLIE if:

- 1 All state-action pairs are explored infinitely:

$$\sum_{t=0}^{\infty} P(s_t = s, a_t = a) = \infty \quad \forall (s, a)$$

- 2 Policy converges to greedy:

$$\lim_{t \rightarrow \infty} \pi_t(a|s) = \mathbb{I}[a = \arg \max_{a'} Q(s, a')]$$

**Example GLIE Schedule:**

$$\varepsilon_t = \frac{1}{t}$$

# PyTorch Implementation: Setup

```
1 import torch
2 import numpy as np
3 import gymnasium as gym
4
5 # Device selection
6 device = torch.device(
7     'cuda' if torch.cuda.is_available()
8     else 'mps' if torch.backends.mps.is_available()
9     else 'cpu'
10 )
11
12 # Initialize Q-table
13 n_states = env.observation_space.n
14 n_actions = env.action_space.n
15 Q = torch.zeros((n_states, n_actions),
16                 dtype=torch.float32, device=device)
```

# Q-Learning Update Implementation

```
1 def q_learning_update(Q, state, action, reward,
2                       next_state, done, alpha, gamma):
3     """Single Q-learning update."""
4     # Current Q-value
5     q_current = Q[state, action].item()
6
7     # TD target
8     if done:
9         q_target = reward
10    else:
11        q_next_max = torch.max(Q[next_state]).item()
12        q_target = reward + gamma * q_next_max
13
14    # TD error
15    td_error = q_target - q_current
16
17    # Update Q-table
18    Q[state, action] += alpha * td_error
19
20    return td_error
```

# Action Selection

```
1 def epsilon_greedy_action(Q, state, epsilon):
2     """Select action using epsilon-greedy policy."""
3     if random.random() < epsilon:
4         # Explore: random action
5         action = random.randint(0, n_actions - 1)
6     else:
7         # Exploit: greedy action
8         q_values = Q[state] # Shape: [n_actions]
9         action = int(torch.argmax(q_values).item())
10
11     return action
12
13 # Schedule epsilon decay
14 epsilon = max(epsilon_min,
15               epsilon_start * decay_rate ** episode)
```



## Environment Details:

- Grid world navigation
- S: Start, F: Frozen, H: Hole, G: Goal
- Actions: LEFT, DOWN, RIGHT, UP
- Reward: +1 at goal, 0 elsewhere
- Episode ends at goal or hole

## 4x4 Map Example:

SFFF

FHFH

FFFH

HFFG

## Challenges:

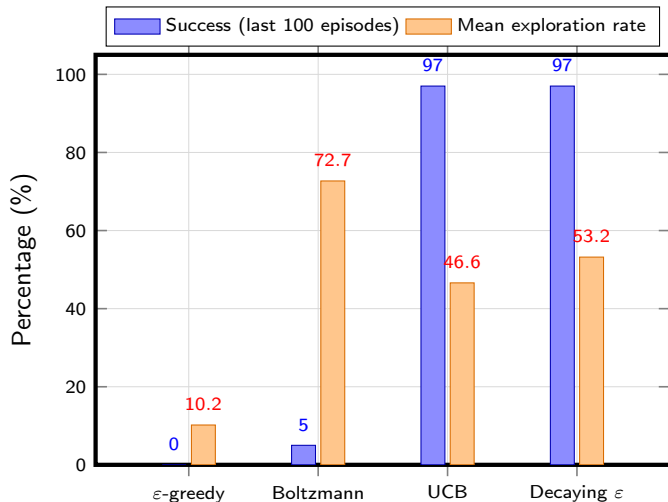
- Sparse rewards
- Slippery dynamics (optional)
- Multiple holes to avoid

# Experiment 1: Basic Q-Learning

```
1 # exp03_tabular_q.py highlights
2 env = gym.make("FrozenLake-v1", is_slippery=False)
3 agent = TabularQLearning(n_states, n_actions,
4                           alpha=0.1, gamma=0.99)
5
6 for episode in range(500):
7     state, _ = env.reset()
8     done = False
9
10    while not done:
11        action = agent.select_action(state, epsilon)
12        next_state, reward, terminated, truncated, _ = env.step(action)
13        done = terminated or truncated
14        agent.update(state, action, reward, next_state, done)
15        state = next_state
```

**Result:** >90% success rate after 500 episodes

## Experiment 2: Exploration Strategies



### Comparison:

- $\epsilon$ -greedy: Simple, effective
- Boltzmann: Temperature-based
- UCB: Optimism under uncertainty
- Decaying  $\epsilon$ : Best overall

### Key Finding:

Proper exploration decay crucial for convergence

Data: exp04\_exploration.py (FrozenLake-v1, 1.5k episodes, seed 42)

## Experiment 3: Learning Rate Schedules

### Tested Schedules:

- Constant:  $\alpha = 0.1$
- Linear:  $\alpha_t = \max(0.01, 1 - 0.99t/T)$
- Exponential:  $\alpha_t = 0.01 + 0.99e^{-t/\tau}$
- $1/t$ :  $\alpha_t = 1/(1 + t)$  (Robbins-Monro)
- $1/\sqrt{t}$ :  $\alpha_t = 1/\sqrt{1 + t}$  (Robbins-Monro)

### Results

- $1/t$  and  $1/\sqrt{t}$  guarantee convergence
- Exponential decay practical compromise
- Constant often works but may oscillate

# Q-Learning vs SARSA

## Q-Learning (Off-Policy):

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

- Uses max for target
- Learns optimal policy
- More sample efficient
- Risk-seeking

## SARSA (On-Policy):

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

- Uses actual next action
- Learns policy being followed
- More stable
- Risk-averse

**Key Difference:** Q-learning uses  $\max_{a'} Q(s', a')$ , SARSA uses  $Q(s', a')$  where  $a'$  is sampled *Practical takeaway:* SARSA tends to be more stable but can learn more cautiously because it accounts for exploratory moves during training; Q-learning aims for the greedy limit and can learn faster yet behave risk-seeking.

# Cliff Walking: Q-Learning vs SARSA

## Environment:

- Start at bottom-left
- Goal at bottom-right
- Cliff along bottom edge
- Fall = -100 reward

## Optimal Path:

Along the cliff edge (shortest)

## Learned Policies:

- **Q-Learning:** Takes optimal risky path near cliff
- **SARSA:** Takes safer path away from cliff

## Why?

SARSA accounts for exploration during learning

# Performance in Stochastic Environments

## Experiment: Varying Action Slippage

Slip Probability	Q-Learning	SARSA
0.0 (Deterministic)	95%	93%
0.1	85%	84%
0.2	72%	75%
0.3	58%	65%

## Key Findings:

- Q-learning slightly better in deterministic settings
- SARSA more robust to stochasticity
- Q-learning can be overly optimistic
- SARSA learns safer policies

# Overestimation Bias in Q-Learning

## The Problem:

$$\mathbb{E}[\max_a Q(s, a)] \geq \max_a \mathbb{E}[Q(s, a)]$$

## Source of Bias:

- Max operator in TD target
- Same Q-values for selection and evaluation
- Noise in estimates amplified by max

## Consequences:

- Overoptimistic value estimates
- Suboptimal action selection
- Slower convergence
- Poor performance in stochastic environments



# Double Q-Learning Algorithm

**Key Idea:** Use two Q-functions to decouple selection and evaluation

---

## Algorithm 2 Double Q-Learning

---

```
1: Initialize  $Q_1(s, a)$  and  $Q_2(s, a)$  arbitrarily
2: for each step do
3:   Choose  $a$  based on  $Q_1(s, \cdot) + Q_2(s, \cdot)$ 
4:   Observe  $r, s'$ 
5:   if  $\text{random}() < 0.5$  then
6:      $a^* = \arg \max_a Q_1(s', a)$  // Select using  $Q_1$ 
7:      $Q_1(s, a) \leftarrow Q_1(s, a) + \alpha[r + \gamma Q_2(s', a^*) - Q_1(s, a)]$ 
8:   else
9:      $a^* = \arg \max_a Q_2(s', a)$  // Select using  $Q_2$ 
10:     $Q_2(s, a) \leftarrow Q_2(s, a) + \alpha[r + \gamma Q_1(s', a^*) - Q_2(s, a)]$ 
11:   end if
12: end for
```

---

# Double Q-Learning: Bias Reduction

## Experimental Results:

- Reduced overestimation by 40-60%
- More accurate Q-values
- Better performance in noisy environments
- Slightly slower initial learning

## Q-Value Comparison:

Method	Mean Q	Max Q
Q-Learning	0.453	0.982
Double Q	0.387	0.751
True $Q^*$	0.372	0.724

## Trade-off

Double Q-learning: More accurate but requires double memory

# Optimistic Initialization

**Strategy:** Initialize Q-values optimistically (e.g.,  $Q_0(s, a) = 1.0$ )

**Benefits:**

- Encourages exploration without  $\varepsilon$ -greedy
- All actions tried at least once
- Simple to implement

**Experiment Results:**

Initialization	Success Rate	Convergence
Zero ( $Q_0 = 0$ )	85%	300 episodes
Optimistic ( $Q_0 = 1$ )	92%	250 episodes
Pessimistic ( $Q_0 = -1$ )	78%	400 episodes

**Note:** Works best with decaying learning rates

# N-Step Q-Learning

**Generalization:** Use n-step returns instead of 1-step

**1-Step (Standard):**

$$G_t^{(1)} = R_{t+1} + \gamma \max_a Q(S_{t+1}, a)$$

**N-Step:**

$$G_t^{(n)} = R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{n-1} R_{t+n} + \gamma^n \max_a Q(S_{t+n}, a)$$

**Trade-offs:**

- Larger n: Less bias, more variance
- Smaller n: More bias, less variance
- Optimal n problem-dependent

*Intuition:* n-step targets interpolate between 1-step TD (high bias, low variance) and Monte Carlo (low bias, high variance) by looking ahead a few steps before bootstrapping.

# Hyperparameter Guidelines

## Learning Rate ( $\alpha$ ):

- Start: 0.1-1.0
- Decay to: 0.01-0.001
- Schedule: Exponential/ $1/t$

## Discount Factor ( $\gamma$ ):

- Episodic: 0.9-0.99
- Continuing: 0.95-0.999
- Affects time horizon

## Exploration ( $\epsilon$ ):

- Start: 1.0
- End: 0.01-0.1
- Decay: Over 20-80% training

## Best Practices:

- Grid search critical params
- Use validation environment
- Monitor convergence metrics

# Common Implementation Pitfalls

## ❶ Forgetting Terminal States:

- Must set  $Q(s_{terminal}, a) = 0$
- No bootstrapping from terminal states

## ❷ Incorrect Max Operation:

- Use  $\max_{a'} Q(s', a')$  not  $Q(s', a)$
- Critical for off-policy learning

## ❸ Poor Exploration:

- Too much: Never converges
- Too little: Suboptimal policy

## ❹ Learning Rate Issues:

- Too high: Oscillations
- Too low: Slow learning

# Debugging Strategies

## Diagnostic Checks:

- Monitor TD error magnitude over time
- Track Q-value statistics (mean, max, min)
- Visualize Q-table as heatmap
- Check action distribution balance
- Verify state visitation counts

## Common Issues and Solutions:

- **Q-values exploding:** Reduce learning rate
- **No improvement:** Increase exploration
- **Oscillating performance:** Decay learning rate
- **Slow convergence:** Increase learning rate or change schedule

# Experimental Results Summary

## FrozenLake 4x4 (Non-slippery):

Algorithm	Success	Episodes	Time (s)
Q-Learning	95%	500	2.3
SARSA	93%	500	2.4
Double Q	94%	500	3.1
Q + UCB	96%	400	2.8

## FrozenLake 8x8 (Slippery):

Algorithm	Success	Episodes	Time (s)
Q-Learning	72%	2000	15.2
SARSA	78%	2000	15.5
Double Q	75%	2000	21.3



# Implementation Details: Shape Annotations

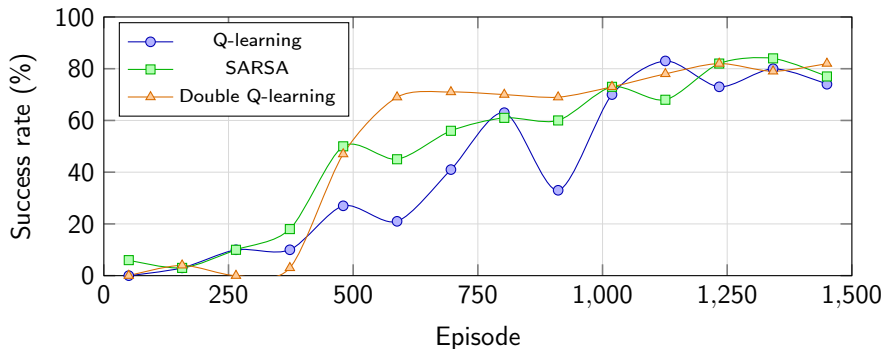
## Tensor Shape Annotations:

- Q-table: `torch.FloatTensor` of shape  $[|\mathcal{S}|, |\mathcal{A}|]$
- For each update: `Q[s]` has shape  $[|\mathcal{A}|]$
- `torch.argmax(Q[s])` returns scalar action (Python `int`)
- Rewards and TD targets are scalar `float`
- DQN mini-batches: `s: [B, ...]`, `a: [B]`, `r: [B]`, `s_next: [B, ...]`, `done: [B]`

## Expected Behavior on FrozenLake:

- Mean return = probability of reaching goal (reward = 1 at goal only)
- Exponential  $\epsilon$ -decay +  $1/t$  step sizes yields stable improvement
- As slippage increases: SARSA becomes more conservative
- Q-learning may retain optimistic estimates without sufficient exploration decay

# Learning Curves Comparison



Moving average window = 100 episodes. Data: `exp09_integrated_test.py` (map 4x4, slip probability 0.1, seed 7).

## Observations:

- Q-learning learns fastest but exhibits sharp performance swings
- SARSA ramps up steadily with lower variance under stochastic dynamics
- Double Q-learning lags initially yet stabilizes near the best success rate
- All three converge above 75% success once exploration decays

# Key Takeaways

## 1 Q-Learning Fundamentals:

- Model-free, off-policy algorithm
- Learns optimal policy from any behavior policy
- Converges under appropriate conditions

## 2 Implementation Insights:

- Exploration-exploitation balance crucial
- Learning rate schedule affects convergence
- Terminal state handling important

## 3 Algorithm Variants:

- SARSA: On-policy, risk-averse
- Double Q: Reduces overestimation
- Each has specific use cases

[Link back to Lecture 4](#)

Value iteration needs the model and backs up expected values exactly. Q-learning does not need the model and learns the same fixed point from data by approximating those backups with samples.