MathRL Overview v1.0

2025 年 4 月 29 日

MathRL

Book Overview:

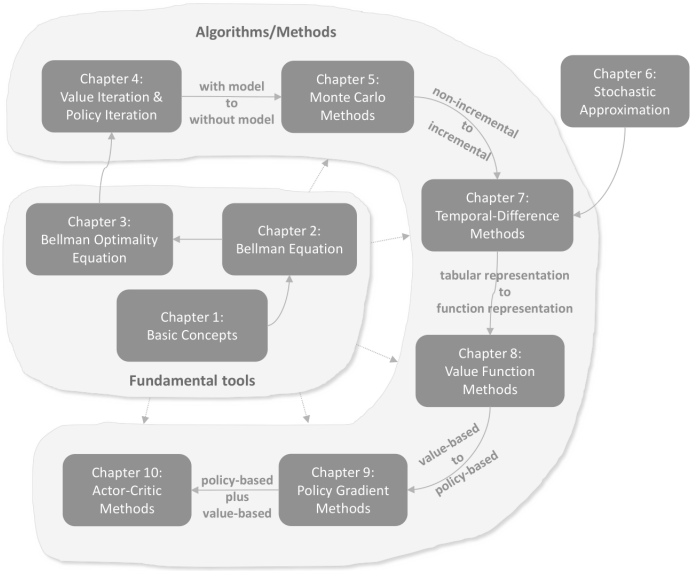
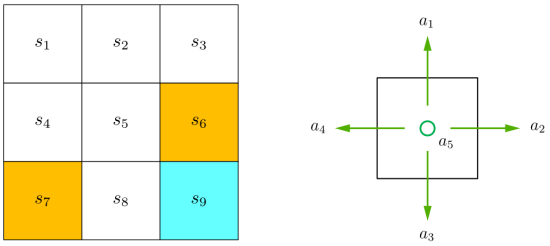


Fig. 1: book overview

Chapter 1: Basic Concepts

1.1 A Grid World Example



(a) States (b) Actions

Fig. 2: Illustrations of the state and action concepts

Agent and Environment: The interacting components in RL

1.2 State and Action

- **State (s):** Describes the agent's current situation or status within the environment. In the grid world, the state is the agent's location (e.g., $s_1, s_2, \ldots, s_9$)
- **State Space ($\mathcal{S}$):** The set of all possible states the agent can be in. $\mathcal{S} = \{s_1, \ldots, s_9\}$
- **Action (a):** An operation the agent can perform. In the grid world, actions are typically movements (up, down, left, right, stay) ($a_1, \ldots, a_5$)
- **Action Space ($\mathcal{A}$ or $\mathcal{A}(s)$):** The set of all actions available to the agent. This set can depend on the state s. $\mathcal{A}$ for all states

1.3 State Transition

- **Definition:** The process of moving from one state (s) to the next state (s') after the agent takes an action (a). Example: $s_1 \xrightarrow{a_2} s_2$
- **Representation:**
  - For deterministic transitions, a table can map each (s, a) pair to the next state s'

	a_1 (upward)	a_2 (rightward)	a_3 (downward)	a_4 (leftward)	a_5 (still)
s_1	s_1	s_2	s_4	s_1	s_1
s_2	s_2	s_3	s_5	s_1	s_2
s_3	s_3	s_3	s_6	s_2	s_3
s_4	s_1	s_5	s_7	s_4	s_4
s_5	s_2	s_6	s_8	s_4	s_5
s_6	s_3	s_6	s_9	s_5	s_6
s_7	s_4	s_8	s_7	s_7	s_7
s_8	s_5	s_9	s_8	s_7	s_8
s_9	s_6	s_9	s_9	s_8	s_9

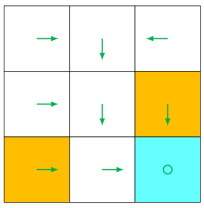
Fig. 3: A tabular representation of the state transition process

  - Described by the **state transition probability** $p(s'|s, a) = Pr(S_{t+1} = s'|S_t = s, A_t = a)$. For deterministic transitions, $p(s'|s, a)$ is 1 for the resulting state and 0 else

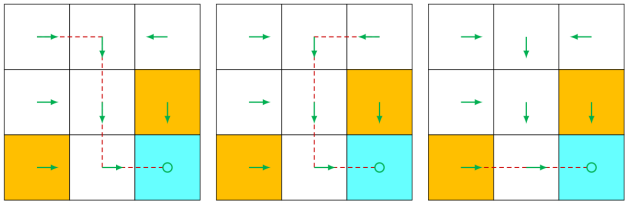
1.4 Policy (π)

- **Definition:** A rule or strategy the agent uses to choose actions in each state.
- **Types:**
  - **Deterministic Policy:** For each state s, the policy specifies a single action $a = \pi(s)$.
  - **Stochastic Policy:** For each state s, the policy specifies a probability distribution over actions, $\pi(a|s) = Pr(A_t = a|S_t = s)$. It must hold that $\sum_{a \in \mathcal{A}(s)} \pi(a|s) = 1$ for all s, normalization
- **Representation:**

– Visualized



(a) A deterministic policy



(b) Trajectories obtained from the policy

Fig. 4: A policy represented by arrows and some trajectories obtained by starting from different initial states

– Tabular representation: stores the probability $\pi(a|s)$ for every state-action pair. Function representation is mentioned for later chapters.

	a_1 (upward)	a_2 (rightward)	a_3 (downward)	a_4 (leftward)	a_5 (still)
s_1	0	0.5	0.5	0	0
s_2	0	0	1	0	0
s_3	0	0	0	1	0
s_4	0	1	0	0	0
s_5	0	0	1	0	0
s_6	0	0	1	0	0
s_7	0	1	0	0	0
s_8	0	1	0	0	0
s_9	0	0	0	0	1

Fig. 5: A tabular representation of a policy. Each entry indicates the probability of taking an action at a state

1.5 Reward (r or R_t)

- **Definition:** A scalar feedback signal received from the environment after taking action a in state s. It signals the immediate desirability of the transition. Denoted $r(s, a)$. Can be +, -, or 0
- **Purpose:** Guide the agent's learning
- **Example Rewards:** $r_{boundary} = -1$, $r_{forbidden} = -1$, $r_{target} = +1$, $r_{other} = 0$
- **Representation:**

	a_1 (upward)	a_2 (rightward)	a_3 (downward)	a_4 (leftward)	a_5 (still)
s_1	r_{boundary}	0	0	r_{boundary}	0
s_2	r_{boundary}	0	0	0	0
s_3	r_{boundary}	r_{boundary}	$r_{\text{forbidden}}$	0	0
s_4	0	0	$r_{\text{forbidden}}$	r_{boundary}	0
s_5	0	$r_{\text{forbidden}}$	0	0	0
s_6	0	r_{boundary}	r_{target}	0	$r_{\text{forbidden}}$
s_7	0	0	r_{boundary}	r_{boundary}	$r_{\text{forbidden}}$
s_8	0	r_{target}	r_{boundary}	$r_{\text{forbidden}}$	0
s_9	$r_{\text{forbidden}}$	r_{boundary}	r_{boundary}	0	r_{target}

Fig. 6: A tabular representation of the process of obtaining rewards. Here, the process is deterministic. Each cell indicates how much reward can be obtained after the agent takes an action at a given state.

- **Formal Representation:** Described by the **reward probability** $p(r|s,a) = Pr(R_{t+1} = r|S_t = s, A_t = a)$. The process can be stochastic. Often simplified to an expected reward $r(s,a) = E[R_{t+1}|S_t = s, A_t = a]$.
- The agent aims to maximize the *total* (cumulative) reward, not just the *immediate* reward.

1.6 Trajectories, Returns, and Episodes

- **Trajectory:** A state-action-reward chain, e.g., $S_t \xrightarrow{A_t, R_{t+1}} S_{t+1} \xrightarrow{A_{t+1}, R_{t+2}} \dots$.
- **Return (G_t):** Total accumulated reward starting from time step t. Crucial for evaluating policies.
  - Finite Return: sum $G_t = R_{t+1} + R_{t+2} + \cdots + R_T$.
  - Infinite return (Discounted Return): $G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$. $\gamma \in [0, 1)$ is the **discount rate**. Needed for infinite trajectories, balances immediate vs future rewards. Large γ pay more attention to future rewards and low γ pay attention to immediate rewards
- **Episode (Trial):** Trajectory from a start state to a **terminal state**. Usually finite.

1.7 Markov Decision Processes (MDPs)

- **Formal Framework:** Provides the standard mathematical description for RL problems.
- **Components:** An MDP is typically defined by a tuple $\langle \mathcal{S}, \mathcal{A}, p, \mathcal{R}, \gamma \rangle$, where:
  - $\mathcal{S}$: State space.
  - $\mathcal{A}$: Action space (possibly state-dependent $\mathcal{A}(s)$).
  - p: Transition dynamics, including state transition probability $p(s'|s,a)$ and reward probability $p(r|s,a)$. Often called the model
  - $\mathcal{R}$: Set of possible rewards.
  - γ: Discount factor.

- **Markov Property: The future (S_{t+1}, R_{t+1}) depends only on the present (S_t, A_t) and not on the past history. Mathematically (Eq 1.4):**
  - $p(s_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, \dots) = p(s_{t+1}|s_t, a_t)$
  - $p(r_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, \dots) = p(r_{t+1}|s_t, a_t)$

  This property is crucial for deriving Bellman equations.
- **MDP vs. Markov Process (MP)/Markov Chain:** An MDP becomes an MP (or Markov Chain for discrete time/states) once a fixed policy π is applied. Book focuses on finite MDPs.
- **Agent-Environment Interaction:** The standard loop where the agent observes state s_t, selects action a_t based on π, receives reward r_{t+1}, and transitions to state s_{t+1}
- **It can be simply argued that MDPs is Markov property**

Chapter 2: State Values and Bellman Equation

2.1 & 2.2 Motivating Examples

- **Core Idea:** Uses simple examples to demonstrate why returns are important for evaluating and comparing different policies. It introduces two perspectives for calculating returns/values:
  1. Direct summation of (discounted) rewards along a trajectory.
  2. Using the recursive relationship between values (**bootstrapping**), which naturally leads to the Bellman equation concept.

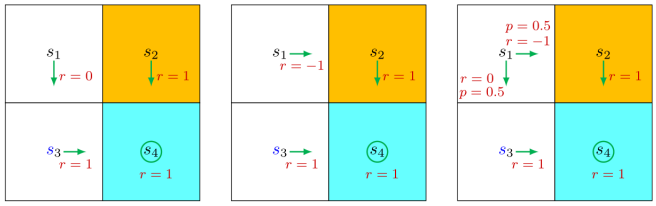


Fig. 7: Examples for demonstrating the importance of returns. The three examples have different policies for s_1

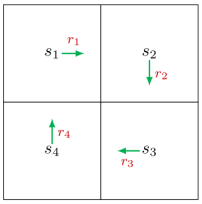


Fig. 8: An example for demonstrating how to calculate returns. There are no target or forbidden cells in this example.

- $v = r + \gamma P v$

2.3 State Values

- **Definition:** The **state-value function** $v_\pi(s)$ for a policy π is the expected discounted return starting from state s and subsequently following policy π.
- **Formula:** Consider a sequence of time steps $t = 0, 1, 2, \ldots$. At time t, the agent is in state S_t, and the action taken following a policy π is A_t. The next state is $S_t + 1$, and the immediate reward obtained is $R_t + 1$. This process can be expressed concisely as

  $$S_t \xrightarrow{A_t} S_{t+1}, R_{t+1}.$$

  Note that $S_t, S_{t+1}, A_t, R_{t+1}$ are all random variables. Moreover, $S_t, S_{t+1} \in \mathcal{S}, A_t \in \mathcal{A}(S_t)$, and $R_{t+1} \in \mathcal{R}(S_t, A_t)$.

  Starting from t, we can obtain a state-action-reward trajectory:

  $$S_t \xrightarrow{A_t} S_{t+1}, R_{t+1} \xrightarrow{A_{t+1}} S_{t+2}, R_{t+2} \xrightarrow{A_{t+2}} S_{t+3}, R_{t+3} \ldots.$$

  By definition, the discounted return along the trajectory is

  $$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots,$$

  where $\gamma \in (0, 1)$ is the discount rate. Note that G_t is a random variable since $R_t + 1, R_{t+2}, \ldots$ are all random variables.

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

  where G_t is the return from time t, $G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$. γ is the discount factor, and R_{t+k+1} is the reward received at time step $t + k + 1$.

- **Properties:**

- $v_\pi(s)$ depends on the state s. This is because its definition is a conditional expectation with the condition that the agent starts from $S_t = s$.
- $v_\pi(s)$ depends on π. This is because its definition is a conditional expectation with the condition that the agent starts from $S_t = s$.
- $v_\pi(s)$ is independent of the time step t. for stationary policies and MDPs. If the agent moves in the state space, t represents the current time step. The value of $v_\pi(s)$ is determined once the policy is given.
- Relationship to returns: In deterministic environments, $v_\pi(s)$ equals the return from s; in stochastic environments, $v_\pi(s)$ is the expected value over all possible returns starting from s.

- **Role:** A higher $v_\pi(s)$ indicates a better policy π when starting from state s. It's a core metric for policy evaluation.

2.4 Bellman Equation

- It can be simply understood as Dynamic Programming
- **Purpose:** Establishes a relationship between the value of a state s, $v_\pi(s)$, and the values of its potential successor states s', $v_\pi(s')$. It's the central tool for analyzing value functions.
- **Derivation Outline:** Starts from the recursive definition of return $G_t = R_{t+1} + \gamma G_{t+1}$. Taking the expectation conditioned on $S_t = s$: $v_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} | S_t = s] = \mathbb{E}_\pi[R_{t+1}|S_t = s] + \gamma \mathbb{E}_\pi[G_{t+1}|S_t = s]$. The terms are expanded using the law of total expectation and the Markov property:
  - Expected Immediate Reward: $\mathbb{E}_\pi[R_{t+1}|S_t = s] = \sum_{a\in\mathcal{A}} \pi(a|s) \sum_{r\in\mathcal{R}} p(r|s,a) r$.
  - Expected Future Return: $\mathbb{E}_\pi[G_{t+1}|S_t = s] = \sum_{a\in\mathcal{A}} \pi(a|s) \sum_{s'\in\mathcal{S}} p(s'|s,a) v_\pi(s')$.

$$
\begin{aligned}
v_\pi(s) &= \mathbb{E}[R_{t+1}|S_t = s] + \gamma \mathbb{E}[G_{t+1}|S_t = s], \\
&= \underbrace{\sum_{a\in\mathcal{A}} \pi(a|s) \sum_{r\in\mathcal{R}} p(r|s,a) r}_{\text{mean of immediate rewards}} + \underbrace{\gamma \sum_{a\in\mathcal{A}} \pi(a|s) \sum_{s'\in\mathcal{S}} p(s'|s,a) v_\pi(s')}_{\text{mean of future rewards}} \\
&= \sum_{a\in\mathcal{A}} \pi(a|s) \left[\sum_{r\in\mathcal{R}} p(r|s,a) r + \gamma \sum_{s'\in\mathcal{S}} p(s'|s,a) v_\pi(s') \right], \quad \text{for all } s \in \mathcal{S}.
\end{aligned}
$$

- **Bellman Equation (Elementwise Form for v_π):**

$$
v_\pi(s) = \sum_{a\in\mathcal{A}} \pi(a|s) \left[\sum_{r\in\mathcal{R}} p(r|s,a) r + \gamma \sum_{s'\in\mathcal{S}} p(s'|s,a) v_\pi(s') \right]
$$

  This equation holds for all states $s \in \mathcal{S}$, forming a system of linear equations for $v_\pi(s)$.

- **Remarks:**

- $v_\pi(s)$ and $v_\pi(s_0)$ are unknown state values to be calculated. It may be confusing to beginners how to calculate the unknown $v_\pi(s)$ given that it relies on another unknown $v_\pi(s_0)$. It must be noted that the Bellman equation refers to a set of linear equations for all states rather than a single equation. If we put these equations together, it becomes clear how to calculate all the state values.
- $\pi(a|s)$ is a given policy. Since state values can be used to evaluate a policy, solving the state values from the Bellman equation is a policy evaluation process, which is an important process in many reinforcement learning algorithms
- $p(r|s,a)$ and $p(s_0|s,a)$ represent the system model.

- **Simplify:** The value of the current state equals the expected immediate reward plus the expected discounted value of the next state, averaged over all possible actions taken according to the policy π.
- **Other Forms:** The book also mentions equivalent forms using joint probabilities $p(s', r|s, a)$ and the case where reward $r(s')$ depends only on the next state.

2.5 Examples for illustrating the Bellman equation

- Provides two specific 4-state grid world examples with deterministic and stochastic policies, respectively. It demonstrates how to write down the system of Bellman equations for each state and solve them simultaneously to find the state values $v_\pi(s)$. This helps build intuition about the equation's structure and application.

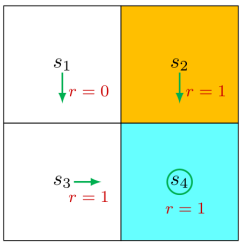


Fig. 9: An example for demonstrating the Bellman equation. The policy in this example is deterministic.

$$
\begin{aligned}
v_\pi(s_4) &= \frac{1}{1-\gamma}, \\
v_\pi(s_3) &= \frac{1}{1-\gamma}, \\
v_\pi(s_2) &= \frac{1}{1-\gamma}, \\
v_\pi(s_1) &= \frac{\gamma}{1-\gamma}.
\end{aligned}
$$

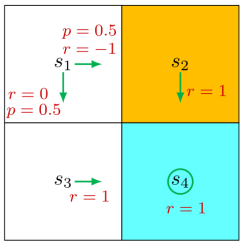


Fig. 10: An example for demonstrating the Bellman equation. The policy in this example is stochastic.

$$
\begin{aligned}
v_\pi(s_4) &= \frac{1}{1-\gamma}, \\
v_\pi(s_3) &= \frac{1}{1-\gamma}, \\
v_\pi(s_2) &= \frac{1}{1-\gamma}, \\
v_\pi(s_1) &= 0.5[0 + \gamma v_\pi(s_3)] + 0.5[-1 + \gamma v_\pi(s_2)], \\
&= -0.5 + \frac{\gamma}{1-\gamma}.
\end{aligned}
$$

2.6 Matrix-vector form of the Bellman equation

- **Purpose:** To express the system of Bellman equations concisely for easier theoretical analysis.
- **Definitions:**
  - $v_\pi \in \mathbb{R}^{|\mathcal{S}|}$: Column vector of state values, $[v_\pi]_s = v_\pi(s)$.
  - $r_\pi \in \mathbb{R}^{|\mathcal{S}|}$: Column vector of expected immediate rewards, $[r_\pi]_s = \sum_a \pi(a|s) \sum_r p(r|s,a) r$.
  - $P_\pi \in \mathbb{R}^{|\mathcal{S}| \times |\mathcal{S}|}$: State transition probability matrix under policy π, $[P_\pi]_{s,s'} = \sum_a \pi(a|s) p(s'|s,a)$.
- **Bellman Equation (Matrix Form):**

$$v_\pi = r_\pi + \gamma P_\pi v_\pi$$

- **Properties of P_π:** P_π is a **stochastic matrix** (non-negative entries, rows sum to 1, i.e., $P_\pi \mathbf{1} = \mathbf{1}$).

2.7 Solving state values from the Bellman equation

- **Policy Evaluation:** The process of calculating the state-value function v_π for a given policy π.
- **Solution Methods:**

1. **Closed-form Solution:** By matrix inversion:

$$v_\pi = (I - \gamma P_\pi)^{-1} r_\pi$$

The matrix $(I - \gamma P_\pi)$ is guaranteed to be invertible[cite: 647].

2. **Iterative Solution:** Using iterative updates:

$$v_{k+1} = r_\pi + \gamma P_\pi v_k$$

Starting from an arbitrary v_0, v_k converges to v_π as $k \to \infty$. This is often more practical than matrix inversion

2.8 From state value to action value

- **Action Value ($q_\pi(s, a)$):** Introduces the action-value function, representing the expected return after taking action a in state s and then following policy π.

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

- **Relationship to State Value:**

$$\underbrace{\mathbb{E}[G_t | S_t = s]}_{v_\pi(s)} = \sum_{a \in \mathcal{A}} \underbrace{\mathbb{E}[G_t | S_t = s, A_t = a]}_{q_\pi(s,a)} \pi(a|s).$$

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

$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left[\sum_{r \in \mathcal{R}} p(r|s, a) r + \gamma \sum_{s' \in \mathcal{S}} p(s'|s, a) v_\pi(s') \right],$$

$$q_\pi(s, a) = \sum_{r \in \mathcal{R}} p(r|s, a) r + \gamma \sum_{s' \in \mathcal{S}} p(s'|s, a) v_\pi(s').$$

1. $v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) q_\pi(s, a)$: State value is the expected action value under the policy.
2. $q_\pi(s, a) = \sum_r p(r|s, a) r + \gamma \sum_{s'} p(s'|s, a) v_\pi(s')$: Action value is the expected immediate reward plus the expected discounted value of the next state.

- **Importance:** These relationships allow deriving one value function from the other. The chapter emphasizes that even actions not selected by π have well-defined q_π values, crucial for policy improvement.

- **Bellman Equation for Action Values:** By substituting the expression for v_π into the equation for q_π, we get a recursive equation solely in terms of q_π:

$$q_\pi(s,a) = \sum_r p(r|s,a)r + \gamma \sum_{s'} p(s'|s,a) \sum_{a'} \pi(a'|s') q_\pi(s',a')$$

  Its matrix form is $q_\pi = \tilde{r} + \gamma P \Pi q_\pi$
  Where q_π is the action value vector indexed by the state-action pairs: its (s,a)th element is $[q_\pi]_{(s,a)} = q_\pi(s,a)$. $\tilde{r}$ is the immediate reward vector indexed by the state-action pairs: $[\tilde{r}]_{(s,a)} = \sum_{r\in\mathcal{R}} p(r|s,a)r$. The matrix P is the probability transition matrix, whose row is indexed by the state-action pairs and whose column is indexed by the states: $[P]_{(s,a),s'} = p(s'|s,a)$. Moreover, Π is a block diagonal matrix in which each block is a $1 \times |\mathcal{A}|$ vector: $\Pi_{s',(s',a')} = \pi(a'|s')$ and the other entries of Π are zero.

Chapter 3: Optimal State Values and Bellman Optimality Equation

3.1 Motivating example: How to improve policies?

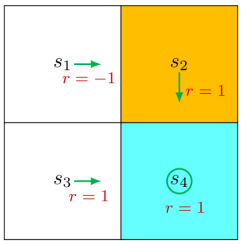


Fig. 11: An example for demonstrating policy improvement..

- **Core Idea:** Uses a simple grid world example to demonstrate that a non-optimal policy can be improved by comparing the **action values** ($q_\pi(s,a)$) for the current state s and choosing the action with the highest value. This motivates the need for a systematic way to find the best possible policy.

3.2 Optimal state values and optimal policies

- **Optimality Definition:**

- **Policy Comparison:** Policy π_1 is better than or equal to π_2 if $v_{\pi_1}(s) \geq v_{\pi_2}(s)$ for all states $s \in \mathcal{S}$.
- **Optimal Policy (π^*):** A policy π^* is optimal if it is better than or equal to all other policies π.
- **Optimal State Value ($v^*(s)$):** The maximum possible state value achievable by any policy: $v^*(s) \doteq \max_\pi v_\pi(s)$.
- **Optimal Action Value ($q^*(s,a)$):** The maximum possible action value achievable by any policy: $q^*(s,a) \doteq \max_\pi q_\pi(s,a)$.

- **Relationship:** The optimal state value is the maximum optimal action value for that state:

$$v^*(s) = \max_{a \in \mathcal{A}(s)} q^*(s,a)$$

- **Fundamental Questions:** Poses key questions about optimal policies: Do they exist? Are they unique? Are they stochastic or deterministic? How can we find them?

3.3 Bellman Optimality Equation (BOE)

- **Purpose:** The core tool for analyzing optimal values and policies. Its solution directly corresponds to the optimal value function
- **Bellman Optimality Equation (for v^*):**
  - **Elementwise Form:**

$$v^*(s) = \max_{a \in \mathcal{A}(s)} \left[\sum_{r \in \mathcal{R}} p(r|s,a) r + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) v^*(s') \right] = \max_{\pi(s) \in \Pi(s)} \sum_{a \in \mathcal{A}} \pi(a|s) q(s,a)$$

    This is equivalent to $v^*(s) = \max_{a \in \mathcal{A}(s)} q^*(s,a)$.

- **Bellman Optimality Equation (for q^*):**

$$q^*(s,a) = \sum_r p(r|s,a) r + \gamma \sum_{s'} p(s'|s,a) \max_{a' \in \mathcal{A}(s')} q^*(s',a')$$

- **Bellman Optimality Equation (Matrix-Vector Form for v^*):**

$$v^* = \max_{\pi \in \Pi} (r_\pi + \gamma P_\pi v^*)$$

$$[r_\pi]_s \doteq \sum_{a \in \mathcal{A}} \pi(a|s) \sum_{r \in \mathcal{R}} p(r|s,a) r, \quad [P_\pi]_{s,s'} = p(s'|s) \doteq \sum_{a \in \mathcal{A}} \pi(a|s) p(s'|s,a)$$

This is often viewed as a **fixed-point equation** $v = f(v)$, where $f(v) = \max_\pi(r_\pi + \gamma P_\pi v)$ is the Bellman optimality operator.

- **Contraction Mapping Theorem:**
  - Introduces the general theorem: If an operator f is a contraction mapping (i.e., there exists $\gamma \in [0, 1)$ such that for some norm $\|\cdot\|$, $\|f(x_1) - f(x_2)\| \leq \gamma\|x_1 - x_2\|$), then the equation $x = f(x)$ has a unique solution (fixed point) x^*, which can be found by iterating $x_{k+1} = f(x_k)$ from any x_0.
- **Contraction property of the right-hand side of the BOE (Theorem 3.2):** Proves that the Bellman optimality operator $f(v)$ is a contraction mapping under the max norm ($\|\cdot\|_\infty$) with the discount factor γ as the contraction factor. Specifically: $\|f(v_1) - f(v_2)\|_\infty \leq \gamma\|v_1 - v_2\|_\infty$.

3.4 Solving an optimal policy from the BOE

- **Existence and Uniqueness of v^* (Thm 3.3):** Applying the Contraction Mapping Theorem guarantees the existence and uniqueness of the solution v^* to the BOE
- **Algorithm (Value Iteration):** The unique v^* can be found by iterating the Bellman optimality operator (this forms the basis of the Value Iteration algorithm):

$$v_{k+1}(s) = \max_{a \in \mathcal{A}(s)} \left[\sum_r p(r|s, a)r + \gamma \sum_{s'} p(s'|s, a)v_k(s') \right]$$

- **Obtaining the Optimal Policy π^*:** Once v^* is found (or well-approximated):
  1. Compute the optimal action values: $q^*(s, a) = \sum_r p(r|s, a)r + \gamma \sum_{s'} p(s'|s, a)v^*(s')$.
  2. Choose the greedy action: $\pi^*(s) = \arg\max_{a \in \mathcal{A}(s)} q^*(s, a)$
- **Optimality of the Solution (Thm 3.4):** Proves that the unique solution v^* to the BOE is indeed the optimal state value function, i.e., $v^* \geq v_\pi$ for all policies π.
- **Properties of the Optimal Policy (Thm 3.5):**
  - **Determinism:** There always exists an optimal policy that is deterministic and greedy with respect to q^*.
  - **Uniqueness:** While v^* is unique, the optimal policy π^* that achieves it might **not be unique**

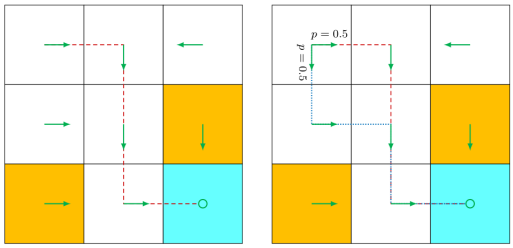


Fig. 12: Examples for demonstrating that optimal policies may not be unique. The two policies are different but are both optimal.

3.5 Factors that influence optimal policies

- **Influencing Factors:** The optimal policy and value depend on the immediate rewards r, the discount rate γ, and the system model (transition probabilities $p(s', r|s, a)$)
- **Impact of Discount Rate γ:**
  - Small γ (near 0): Agent focuses on **immediate rewards**, leading to **short-sighted** policies.
  - Large γ (near 1): Agent considers **future rewards** more heavily, leading to **far-sighted** policies, possibly taking short-term losses for long-term gains.

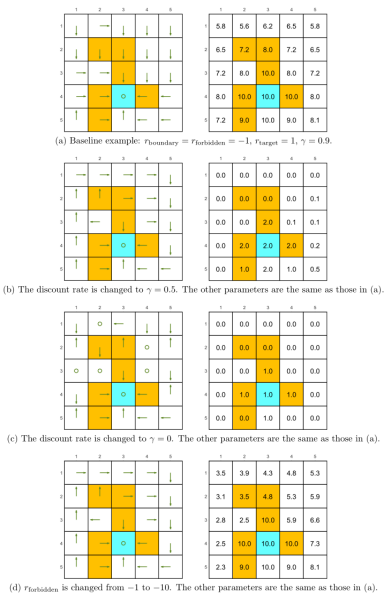


Fig. 13: The optimal policies and optimal state values given different parameter values

- **Impact of Reward r:**
  - Guides behavior (e.g., larger penalties for forbidden states)
  - **Invariance to Affine Transformation (Thm 3.6):** Optimal policy π^* remains unchanged if rewards r are transformed to $\alpha r + \beta$ (with $\alpha > 0$). Optimal state values transform to

$v' = \alpha v^* + \frac{\beta}{1-\gamma}\mathbf{1}$. This implies that the absolute scale and baseline of rewards do not affect optimal actions.

- **Avoiding Detours:** Even if intermediate steps have zero reward ($r_{other} = 0$), the discount factor $\gamma < 1$ inherently penalizes longer paths, encouraging optimal policies to reach the target quickly. Explicit negative rewards for steps are not required solely for this purpose.

Chapter 4: Value Iteration and Policy Iteration

4.1 Value Iteration (VI)

- **Core Idea:** Directly iterates the **Bellman Optimality Equation (BOE)** to compute the optimal state-value function v^*. It is essentially the algorithm suggested by Theorem 3.3 for solving the BOE
- **Iteration Formula:** Starting from an arbitrary initial value function v_0, VI repeatedly applies the Bellman optimality operator:
  - **Elementwise Form:**

$$v_{k+1}(s) = \max_{a \in \mathcal{A}(s)} \left[\sum_{r,s'} p(s', r|s, a)(r + \gamma v_k(s')) \right], \quad \forall s \in \mathcal{S}$$

    This means for each state s, compute the expected value for taking each action a (based on immediate reward and discounted value v_k of the next state), and take the maximum over all actions to get the new value $v_{k+1}(s)$.
  - **Vector Form (Operator notation):**

$$v_{k+1} = \max_{\pi}(r_\pi + \gamma P_\pi v_k)$$

- **Algorithm Steps:**
  1. Initialize value vector v_0 (e.g., all 0).
  2. **Loop** for $k = 0, 1, 2, \ldots$ until v_k converges (e.g., $\|v_{k+1} - v_k\|_\infty < \varepsilon$threshold):
     - For **all** states $s \in \mathcal{S}$:
       - For **all** actions $a \in \mathcal{A}(s)$, compute the action value using the previous iteration's state values: $q_k(s, a) = \sum_{r,s'} p(s', r|s, a)(r + \gamma v_k(s'))$
       - Update the state value: $v_{k+1}(s) = \max_a q_k(s, a)$
  3. **Output:** The converged value function $v^* \approx v_{k+1}$. The optimal policy can then be extracted greedily: $\pi^*(s) = \arg\max_a q^*(s, a)$.

Value iteration algorithm

Initialization: The probability models $p(r|s,a)$ and $p(s'|s,a)$ for all (s,a) are known. Initial guess v_0.

Goal: Search for the optimal state value and an optimal policy for solving the Bellman optimality equation.

While v_k has not converged in the sense that $\|v_k - v_{k-1}\|$ is greater than a predefined small threshold, for the kth iteration, do

For every state $s \in \mathcal{S}$, do

For every action $a \in \mathcal{A}(s)$, do

q-value: $q_k(s,a) = \sum_r p(r|s,a)r + \gamma \sum_{s'} p(s'|s,a)v_k(s')$

Maximum action value: $a_k^*(s) = \arg\max_a q_k(s,a)$

Policy update: $\pi_{k+1}(a|s) = 1$ if $a = a_k^*$, and $\pi_{k+1}(a|s) = 0$ otherwise

Value update: $v_{k+1}(s) = \max_a q_k(s,a)$

- **Convergence:** Guaranteed to converge to the unique v^* due to the Bellman optimality operator being a contraction mapping.
- **Example:** Shows the step-by-step computation in a 2x2 grid world.

q-table	a_1	a_2	a_3	a_4	a_5
s_1	$-1+\gamma v(s_1)$	$-1+\gamma v(s_2)$	$0+\gamma v(s_3)$	$-1+\gamma v(s_1)$	$0+\gamma v(s_1)$
s_2	$-1+\gamma v(s_2)$	$-1+\gamma v(s_2)$	$1+\gamma v(s_4)$	$0+\gamma v(s_1)$	$-1+\gamma v(s_2)$
s_3	$0+\gamma v(s_1)$	$1+\gamma v(s_4)$	$-1+\gamma v(s_3)$	$-1+\gamma v(s_3)$	$0+\gamma v(s_3)$
s_4	$-1+\gamma v(s_2)$	$-1+\gamma v(s_4)$	$-1+\gamma v(s_4)$	$0+\gamma v(s_3)$	$1+\gamma v(s_4)$

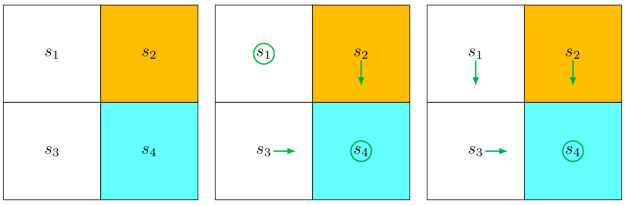


Fig. 14: The expression of q(s, a) for the example, An example for demonstrating the implementation of the value iteration algorithm

4.2 Policy Iteration (PI)

- **Core Idea:** Alternates between two steps: **Policy Evaluation** and **Policy Improvement**, until the policy stabilizes
- **Algorithm Steps:**

1. Initialize a (possibly arbitrary) policy π_0.
2. **Loop** for $k = 0, 1, 2, \ldots$:
   - **(PE) Policy Evaluation:** Evaluate the current policy π_k by computing its exact state-value function v_{π_k}. This involves solving the Bellman equation for π_k:

     $$v_{\pi_k} = r_{\pi_k} + \gamma P_{\pi_k} v_{\pi_k}$$

     This linear system is typically solved iteratively: start with $v^{(0)}$ and repeat $v^{(j+1)} = r_{\pi_k} + \gamma P_{\pi_k} v^{(j)}$ until $v^{(j)}$ converges to v_{π_k}
   - **(PI) Policy Improvement:** Improve the policy by acting greedily with respect to the computed value function v_{π_k}. For all $s \in \mathcal{S}$:
     (a) Compute action values: $q_{\pi_k}(s, a) = \sum_{r,s'} p(s', r|s, a)(r + \gamma v_{\pi_k}(s'))$.
     (b) Update the policy: $\pi_{k+1}(s) = \arg\max_{a \in \mathcal{A}(s)} q_{\pi_k}(s, a)$
3. **Stopping Condition:** If $\pi_{k+1} = \pi_k$, then the policy has stabilized and is optimal ($\pi^* = \pi_{k+1}$, $v^* = v_{\pi_k}$). Stop. Otherwise, continue to the next iteration.

Policy iteration algorithm

Initialization: The system model, $p(r|s, a)$ and $p(s'|s, a)$ for all (s, a), is known. Initial guess π_0.
Goal: Search for the optimal state value and an optimal policy.
While v_{π_k} has not converged, for the kth iteration, do
Policy evaluation:
Initialization: an arbitrary initial guess $v_{\pi_k}^{(0)}$
While $v_{\pi_k}^{(j)}$ has not converged, for the jth iteration, do
For every state $s \in \mathcal{S}$, do
$v_{\pi_k}^{(j+1)}(s) = \sum_a \pi_k(a|s) \left[\sum_r p(r|s, a)r + \gamma \sum_{s'} p(s'|s, a) v_{\pi_k}^{(j)}(s')\right]$
Policy improvement:
For every state $s \in \mathcal{S}$, do
For every action $a \in \mathcal{A}$, do
$q_{\pi_k}(s, a) = \sum_r p(r|s, a)r + \gamma \sum_{s'} p(s'|s, a) v_{\pi_k}(s')$
$a_k^*(s) = \arg\max_a q_{\pi_k}(s, a)$
$\pi_{k+1}(a|s) = 1$ if $a = a_k^*$, and $\pi_{k+1}(a|s) = 0$ otherwise

- **Policy Improvement Theorem (Lemma 4.1):** Guarantees that the improved policy π_{k+1} is always better than or equal to the previous policy π_k, i.e., $v_{\pi_{k+1}}(s) \geq v_{\pi_k}(s), \forall s \in \mathcal{S}$.
- **Convergence (Thm 4.1):** Policy Iteration is guaranteed to converge to an optimal policy π^* and the optimal value function v^* in a finite number of iterations (for finite MDPs).

- **Examples:** Illustrate the iterative evaluation and improvement steps in simple and more complex grid worlds.

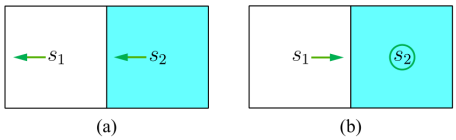


Fig. 15: An example for illustrating the implementation of the policy iteration algorithm

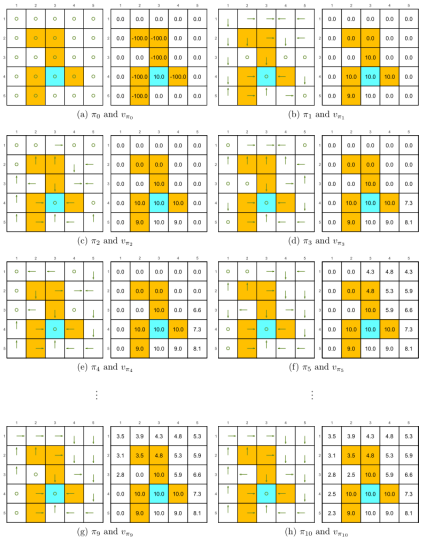


Fig. 16: The evolution processes of the policies generated by the policy iteration algorithm

4.3 Truncated Policy Iteration

- **Core Idea:** Compares VI and PI. Notes that the PE step in PI theoretically requires infinite iterations to solve for v_{π_k} exactly if using iterative methods. Truncated Policy Iteration performs only a **fixed, finite number** ($j_{truncate}$) of value update iterations within the Policy Evaluation step, rather than iterating until full convergence

- **Relationship**

  – VI can be seen as Truncated PI where only 1 iteration ($j_{truncate} = 1$) is performed in the PE step

  – PI can be seen as Truncated PI where infinite iterations ($j_{truncate} = \infty$) are performed in the PE step

	Policy iteration algorithm	Value iteration algorithm	Comments
1) Policy:	π_0	N/A	
2) Value:	$v_{\pi_0} = r_{\pi_0} + \gamma P_{\pi_0} v_{\pi_0}$	$v_0 \doteq v_{\pi_0}$	
3) Policy:	$\pi_1 = \arg\max_\pi (r_\pi + \gamma P_\pi v_{\pi_0})$	$\pi_1 = \arg\max_\pi (r_\pi + \gamma P_\pi v_0)$	The two policies are the same
4) Value:	$v_{\pi_1} = r_{\pi_1} + \gamma P_{\pi_1} v_{\pi_1}$	$v_1 = r_{\pi_1} + \gamma P_{\pi_1} v_0$	$v_{\pi_1} \geq v_1$ since $v_{\pi_1} \geq v_{\pi_0}$
5) Policy:	$\pi_2 = \arg\max_\pi (r_\pi + \gamma P_\pi v_{\pi_1})$	$\pi_2' = \arg\max_\pi (r_\pi + \gamma P_\pi v_1)$	
⋮	⋮	⋮	⋮

Fig. 17: A comparison between the implementation steps of policy iteration and value iteration

- **Algorithm (Alg 4.3):** Similar structure to PI, but the inner loop in PE runs for a fixed $j_{truncate}$ iterations

  Truncated policy iteration algorithm

  Initialization: The probability models $p(r|s,a)$ and $p(s'|s,a)$ for all (s,a) are known. Initial guess π_0.

  Goal: Search for the optimal state value and an optimal policy.

  While v_k has not converged, for the kth iteration, do

  Policy evaluation:

  Initialization: select the initial guess as $v_k^{(0)} = v_{k-1}$. The maximum number of iterations is set as j_{truncate}.

  While $j < j_{\text{truncate}}$, do

  For every state $s \in \mathcal{S}$, do

  $v_k^{(j+1)}(s) = \sum_a \pi_k(a|s) \left[\sum_r p(r|s,a) r + \gamma \sum_{s'} p(s'|s,a) v_k^{(j)}(s') \right]$

  Set $v_k = v_k^{(j_{\text{truncate}})}$

  Policy improvement:

  For every state $s \in \mathcal{S}$, do

  For every action $a \in \mathcal{A}(s)$, do

  $q_k(s,a) = \sum_r p(r|s,a) r + \gamma \sum_{s'} p(s'|s,a) v_k(s')$

  $a_k^*(s) = \arg\max_a q_k(s,a)$

  $\pi_{k+1}(a|s) = 1$ if $a = a_k^*$, and $\pi_{k+1}(a|s) = 0$ otherwise

- **Advantages:** Often converges faster overall than VI (more informed policy updates) and computationally cheaper per outer iteration than full PI (incomplete evaluation)

Chapter 5: Monte Carlo Methods

5.1 Motivating example: Mean estimation

- **Core Idea:** Introduces the fundamental principle behind MC methods using the problem of estimating the expected value (mean) of a random variable.
- **Method Comparison:**
  - **Model-based:** If the probability distribution $p(x)$ of a random variable X is known, the expectation is $E[X] = \sum_{x \in \mathcal{X}} p(x)x$.
  - **Model-free (MC Estimation):** If $p(x)$ is unknown, but we can draw independent and identically distributed (i.i.d.) samples $\{x_1, x_2, \ldots, x_n\}$ from X, we can estimate the expectation using the sample mean:

    $$E[X] \approx \bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i$$

    By the **Law of Large Numbers**, $\bar{x} \to E[X]$ as the number of samples $n \to \infty$.
- **RL Connection:** Since state values $v_\pi(s)$ and action values $q_\pi(s, a)$ are defined as expected returns $E[G_t]$, estimating values is fundamentally a mean estimation problem, solvable by averaging returns from multiple experienced episodes.

5.2 MC Basic: The simplest MC-based algorithm

- **Core Idea:** Adapts the Policy Iteration (PI) framework by replacing the model-based policy evaluation step with model-free MC estimation of action values. Estimating action values $q_\pi(s, a)$ is crucial because, without a model, $v_\pi(s)$ alone is insufficient for policy improvement.
- **MC Estimation of Action Values:**

$$q_{\pi_k}(s, a) = \sum_r p(r|s, a)r + \gamma \sum_{s'} p(s'|s, a)v_{\pi_k}(s').$$

$$\begin{aligned} q_{\pi_k}(s, a) &= \mathbb{E}[G_t|S_t = s, A_t = a] \\ &= \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots |S_t = s, A_t = a], \end{aligned}$$

$$q_{\pi_k}(s, a) = \mathbb{E}[G_t|S_t = s, A_t = a] \approx \frac{1}{n} \sum_{i=1}^{n} g_{\pi_k}^{(i)}(s, a).$$

$$q_{\pi_k}(s, a) = E_{\pi_k}[G_t|S_t = s, A_t = a] \approx q_k(s, a) = \frac{\sum_{i=1}^{N_{(s,a)}} g_{\pi_k}^{(i)}(s, a)}{N_{(s,a)}}$$

where $g_{\pi_k}^{(i)}(s, a)$ is the actual return observed following the i-th complete episode that started with state-action pair (s, a) under policy π_k, and $N_{(s,a)}$ is the number of such episodes.

- **MC Basic Algorithm:**

  1. Initialize policy π_0.
  2. **Loop** for $k = 0, 1, 2, \ldots$:
     - **Policy Evaluation:** For **all** state-action pairs (s, a):
       * Generate **many** complete episodes starting from (s, a) following the current policy π_k.
       * Calculate the average return $q_k(s, a)$ over these episodes as the estimate of $q_{\pi_k}(s, a)$.
     - **Policy Improvement:** Update the policy greedily based on the estimated action values: $\pi_{k+1}(s) = \arg\max_a q_k(s, a)$.
  3. Repeat until policy and value convergence.

     MC Basic (a model-free variant of policy iteration)

     Initialization: Initial guess π_0.
     Goal: Search for an optimal policy.
     For the kth iteration ($k = 0, 1, 2, \ldots$), do
     For every state $s \in \mathcal{S}$, do
     For every action $a \in \mathcal{A}(s)$, do
     Collect sufficiently many episodes starting from (s, a) by following π_k
     Policy evaluation:
     $q_{\pi_k}(s, a) \approx q_k(s, a) =$ the average return of all the episodes starting from (s, a)
     Policy improvement:
     $a_k^*(s) = \arg\max_a q_k(s, a)$
     $\pi_{k+1}(a|s) = 1$ if $a = a_k^*$, and $\pi_{k+1}(a|s) = 0$ otherwise

- **Characteristics:**
  - Conceptually simple, direct model-free adaptation of PI.
  - Inefficient: Requires numerous complete episodes for *each* (s, a) pair for evaluation.
  - Requires **Exploring Starts (ES):** Assumes that every (s, a) pair has a non-zero probability of being the starting pair of an episode to ensure all values can be estimated. This is often impractical.
- **Example:** Illustrates calculating the MC estimate for specific (s, a) pairs and performing one step of policy improvement.

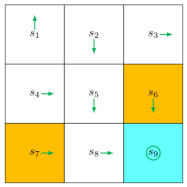


Fig. 18: An example for illustrating the MC Basic algorithm

5.3 MC Exploring Starts (MC ES)

- **Core Idea:** To improve the sample and update efficiency compared to MC Basic.
- **subepisode**
- **Sample Utilization Strategies:**
  - **First-Visit MC:** In a single episode, use only the return following the *first* time the pair (s, a) is visited to update the estimate for $q(s, a)$.
  - **Every-Visit MC:** In a single episode, use the return following *every* visit to the pair (s, a) to update the estimate for $q(s, a)$. Typically more sample efficient.
  - (MC Basic uses an "Initial-Visit" approach, only using the episode for the very first (s, a) pair).
- **Update Efficiency:** Instead of waiting to collect all episodes for a specific (s, a), updates can be performed **after each episode** for all (s, a) pairs visited within that episode (using either first-visit or every-visit returns to update the running average estimate of $q(s, a)$).
- **Algorithm (Alg 5.2):** Combines (typically) every-visit updates performed after each episode. Still requires the **Exploring Starts (ES)** assumption: every (s, a) pair has a non-zero probability of being selected as the starting pair.

  MC Exploring Starts (an efficient variant of MC Basic)

  Initialization: Initial policy $\pi_0(a|s)$ and initial value $q(s, a)$ for all (s, a). $\text{Returns}(s, a) = 0$ and $\text{Num}(s, a) = 0$ for all (s, a).

  Goal: Search for an optimal policy.

  For each episode, do

  Episode generation: Select a starting state-action pair (s_0, a_0) and ensure that all pairs can be possibly selected (this is the exploring-starts condition). Following the current policy, generate an episode of length T: $s_0, a_0, r_1, \ldots, s_{T-1}, a_{T-1}, r_T$.

  Initialization for each episode: $g \leftarrow 0$

  For each step of the episode, $t = T-1, T-2, \ldots, 0$, do

  $g \leftarrow \gamma g + r_{t+1}$

  $\text{Returns}(s_t, a_t) \leftarrow \text{Returns}(s_t, a_t) + g$

  $\text{Num}(s_t, a_t) \leftarrow \text{Num}(s_t, a_t) + 1$

Policy evaluation:
$q(s_t, a_t) \leftarrow \text{Returns}(s_t, a_t)/\text{Num}(s_t, a_t)$
Policy improvement:
$\pi(a|s_t) = 1$ if $a = \arg\max_a q(s_t, a)$ and $\pi(a|s_t) = 0$ otherwise

5.4 MC ϵ-Greedy: Learning without exploring starts

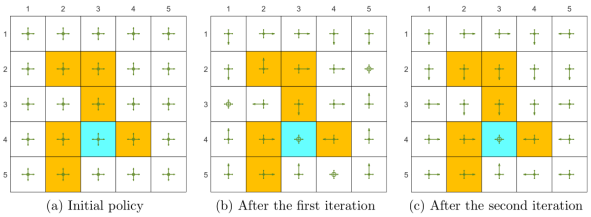
- **Core Idea:** Eliminate the need for the ES assumption by ensuring continued exploration through the use of **soft policies**. A soft policy has $\pi(a|s) > 0$ for all s, a.
- **ϵ-Greedy Policy:**
  - Definition: A common soft policy. With probability $1-\epsilon$, choose the action a^* that currently has the highest estimated Q-value ($\arg\max_{a'} q(s, a')$). With probability ϵ, choose an action uniformly at random from all available actions $|\mathcal{A}(s)|$.
  - Probability Formula:
$$\pi(a|s) = \begin{cases} 1 - \epsilon + \epsilon/|\mathcal{A}(s)| & \text{if } a = a^* \\ \epsilon/|\mathcal{A}(s)| & \text{if } a \neq a^* \end{cases}$$
    (Note: The probability of taking the greedy action is $1 - \epsilon$ plus its share $\epsilon/|\mathcal{A}(s)|$ from the random selection part).
- **MC ϵ-Greedy Algorithm (Alg 5.3):**
  - **Policy Evaluation:** Similar to MC ES, estimate $q(s, a)$ from episodes generated by the current ϵ-greedy policy π_k.
  - **Policy Improvement:** Generate a new ϵ-greedy policy π_{k+1} based on the updated $q(s, a)$ estimates.

Algorithm 5.3: MC *epsilon*-Greedy (a variant of MC Exploring Starts)
Initialization: Initial policy $pi_0(a|s)$ and initial value $q(s, a)$ for all (s, a). $\text{Returns}(s, a) = 0$ and $\text{Num}(s, a) = 0$ for all (s, a). $epsilonin(0, 1]$
Goal: Search for an optimal policy.
For each episode, do
Episode generation: Select a starting state-action pair (s_0, a_0) (the exploring starts condition is not required). Following the current policy, generate an episode of length T: $s_0, a_0, r_1, ldots, s_{T-1}, a_{T-1}, r_T$.
Initialization for each episode: $gleftarrow 0$
For each step of the episode, $t = T-1, T-2, ldots, 0$, do
$gleftarrow gammag + r_{t+1}$
$\text{Returns}(s_t, a_t) leftarrow \text{Returns}(s_t, a_t) + g$
$\text{Num}(s_t, a_t) leftarrow \text{Num}(s_t, a_t) + 1$
Policy evaluation:

$q(s_t, a_t) leftarrow$ Returns(s_t, a_t)/ Num(s_t, a_t)
Policy improvement:
Let $a^* = argmax_a q(s_t, a)$ and

$$\pi(a|s_t) = \begin{cases} 1 - \frac{|\mathcal{A}(s_t)|-1}{|\mathcal{A}(s_t)|}\epsilon, & a = a^* \\ \frac{1}{|\mathcal{A}(s_t)|}\epsilon, & a \neq a^* \end{cases}$$

- **Convergence:** This algorithm converges to the optimal ϵ-greedy policy. Converges to the optimal policy π^* if ϵ decays appropriately over time (GLIE condition - Greedy in the Limit with Infinite Exploration)
- **Example (Fig 5.5):** Shows MC ϵ-Greedy learning from a random initial policy using single episodes for updates.



(a) Initial policy (b) After the first iteration (c) After the second iteration

Fig. 19: The evolution process of the MC ϵ-Greedy algorithm based on single episodes

5.5 Exploration and exploitation of ϵ-greedy policies

- **Trade-off:** Discusses the fundamental balance managed by the ϵ parameter:
  - **Exploration:** Trying out different actions (potentially non-greedy ones) to gather information and possibly discover better policies ($\epsilon > 0$).
  - **Exploitation:** Choosing the action currently believed to be the best based on estimated Q-values to maximize expected return ($1 - \epsilon$ probability).
- **Impact of ϵ:**
  - **High ϵ (e.g., $\epsilon = 1$):** Maximizes exploration (policy is random). Ensures good coverage of state-action space but leads to suboptimal performance during learning and potentially suboptimal final ϵ-greedy policy.
  - **Low ϵ (e.g., $\epsilon = 0.1$):** Favors exploitation (policy is near-greedy). Leads to better performance according to current estimates but risks insufficient exploration, possibly getting stuck in suboptimal policies or having inaccurate value estimates for less explored pairs.
- **Examples:** Compares the resulting value functions, policy optimality, exploration trajectories, and state-action visit counts for different values of ϵ. Shows that larger ϵ degrades optimality but increases exploration coverage, while smaller ϵ improves near-term performance but reduces exploration.

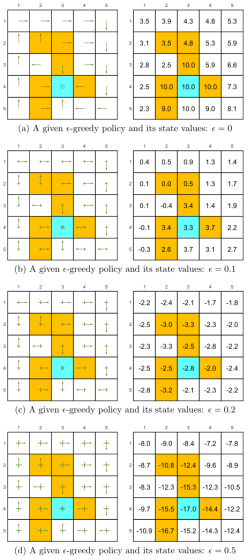


Fig. 20: The state values of some ϵ-greedy policies. These ϵ-greedy policies are consistent with each other in the sense that the actions with the greatest probabilities are the same. It can be seen that, when the value of ϵ increases, the state values of the ϵ-greedy policies decrease and hence their optimality becomes worse.

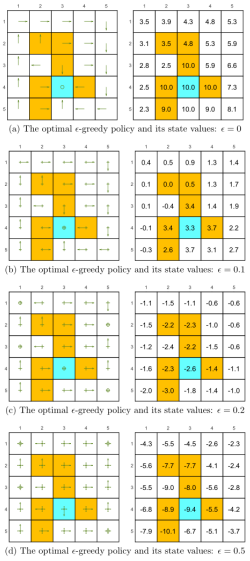


Fig. 21: The optimal ϵ-greedy policies and their corresponding state values under different values of ϵ.$Here, these\epsilon$-greedy policies are optimal among all ϵ-greedy ones (with the same value of ϵ). It can be seen that, when the value of ϵ increases, the optimal ϵ-greedy policies are no longer consistent with the optimal one as in (a).

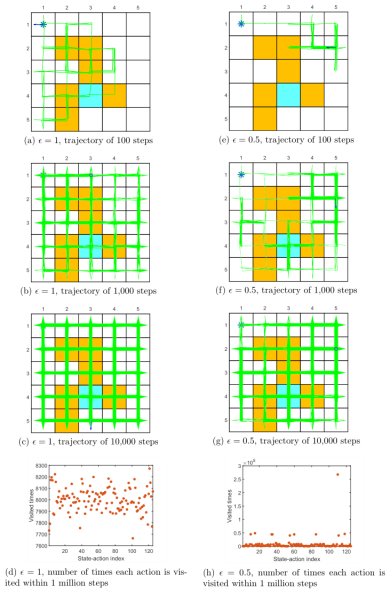


Fig. 22: Exploration abilities of ϵ-greedy policies with different values of ϵ.

- **Practical Approach:** Often, ϵ is annealed (gradually decreased) during learning, starting high for exploration and ending low for exploitation

Chapter 6: Stochastic Approximation

6.1 Motivating example: Mean estimation

- **Core Idea:** Introduces the concept of **incremental updates** using the problem of estimating the mean $E[X]$ of a random variable X.
- **Non-incremental vs. Incremental:** Compares two ways to compute the sample mean $\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$:
  - Non-incremental: Collect all n samples $\{x_1, \ldots, x_n\}$, then compute the average in one go. Requires storing all data and waiting until all data is available.
  - Incremental: Update the mean estimate w_k each time a new sample x_k arrives. The derived update rule is (equivalent to the common form):

$$w_{k+1} = w_k - \frac{1}{k}(w_k - x_k)$$

    where w_{k+1} is the estimate based on the first k samples. This avoids storing all data and allows for online updates.

- **General Form:** Introduces a more general incremental update form:

$$w_{k+1} = w_k - \alpha_k(w_k - x_k)$$

where $\alpha_k > 0$ is a step-size parameter (learning rate). Its convergence properties are analyzed later.

6.2 Robbins-Monro (RM) Algorithm

- **Problem Setting:** Addresses the **root-finding problem** $g(w) = 0$, where the exact function $g(w)$ is unknown, and only noisy observations $\tilde{g}(w, \eta) = g(w) + \eta$ are available. This represents a "black-box" scenario.
- **RM Algorithm:** The classic SA algorithm proposed by Robbins and Monro:

$$w_{k+1} = w_k - a_k\tilde{g}(w_k, \eta_k)$$

where w_k is the k-th estimate of the root w^*, and a_k is a positive step-size coefficient.

- **Example:** Demonstrates the RM algorithm converging to the root of $w^3 - 5 = 0$ even with noisy observations.

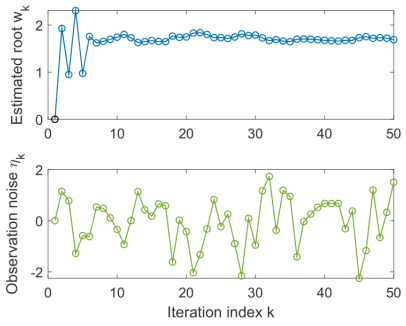


Fig. 23: An illustrative example of the RM algorithm

- **Convergence Intuition:** Uses the example $g(w) = \tanh(w - 1)$ to illustrate that the update step tends to move w_k closer to the true root w^*.

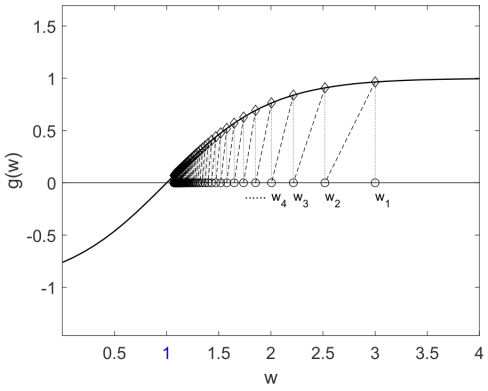


Fig. 24: An example for illustrating the convergence of the RM algorithm

- **Robbins-Monro Theorem (Thm 6.1):** Provides sufficient conditions for the algorithm to converge almost surely to the root w^*:
  (a) **Function Condition:** $g(w)$ is monotonically increasing with bounded slope: $0 < c_1 \leq \nabla_w g(w) \leq c_2$.
  (b) **Step-Size Conditions:** The sequence $\{a_k\}$ satisfies $\sum_{k=1}^{\infty} a_k = \infty$ and $\sum_{k=1}^{\infty} a_k^2 < \infty$.
  (c) **Noise Conditions:** The noise η_k has zero conditional mean $E[\eta_k|H_k] = 0$ and bounded conditional second moment $E[\eta_k^2|H_k] < \infty$ (H_k denotes history).
- **Condition Details:** Explains the significance of each condition. The step-size conditions (b) are crucial: $\sum a_k = \infty$ ensures the algorithm can overcome any initial error, while $\sum a_k^2 < \infty$ ensures the step sizes eventually become small enough to dampen the noise and allow convergence to a point. $a_k = 1/k$ is a typical sequence satisfying these conditions.
- **RM Application to Mean Estimation:** Shows that the incremental mean estimation algorithm $w_{k+1} = w_k - \alpha_k(w_k - x_k)$ is a special case of the RM algorithm applied to solve $g(w) = w - E[X] = 0$ (where the noise is $\eta_k = E[X] - x_k$). Its convergence under appropriate step sizes and i.i.d. samples is thus guaranteed by the RM theorem.

6.3 Dvoretzky's Convergence Theorem

- **Purpose:** Introduces a more general theorem for analyzing the convergence of stochastic approximation processes, applicable to RM and later TD algorithms.
- **Dvoretzky's Theorem (Thm 6.2):** Considers the stochastic process $\Delta_{k+1} = (1-\alpha_k)\Delta_k + \beta_k\eta_k$. It states that Δ_k converges almost surely to 0 if:
  (a) **Coefficient Conditions:** $\alpha_k, \beta_k \geq 0$ are stochastic sequences satisfying $\sum \alpha_k = \infty, \sum \alpha_k^2 < \infty, \sum \beta_k^2 < \infty$ (uniformly almost surely).
  (b) **Noise Conditions:** $E[\eta_k|H_k] = 0$ and $E[\eta_k^2|H_k] \leq C$ (almost surely).

- **Proof Idea (Box 6.1):** Based on (quasi)martingale convergence theory.
- **Applications:**
  - Used to re-prove the convergence of the incremental mean estimation algorithm.
  - Used to prove the Robbins-Monro Theorem (Thm 6.1).
- **Extension (Thm 6.3):** Mentions an extension of Dvoretzky's theorem for vector processes (multiple variables), which is useful for analyzing algorithms like Q-learning.
  Theorem 6.3. Consider a finite set $\mathcal{S}$ of real numbers. For the stochastic process

$$\Delta_{k+1}(s) = (1 - \alpha_k(s))\,\Delta_k(s) + \beta_k(s)\eta_k(s),$$

  it holds that $\Delta_k(s)$ converges to zero almost surely for every $s \in \mathcal{S}$ if the following conditions are satisfied for $s \in \mathcal{S}$:
  (a) $\sum_k \alpha_k(s) = \infty, \sum_k \alpha_k^2(s) < \infty, \sum_k \beta_k^2(s) < \infty$, and $\mathbb{E}\left[\beta_k(s)|\mathcal{H}_k\right] \leq \mathbb{E}\left[\alpha_k(s)|\mathcal{H}_k\right]$ uniformly almost surely;
  (b) $\left\|\mathbb{E}\left[\eta_k(s)|\mathcal{H}_k\right]\right\|_\infty \leq \gamma \left\|\Delta_k\right\|_\infty$, where $\gamma \in (0, 1)$;
  (c) $\operatorname{var}\left[\eta_k(s)|\mathcal{H}_k\right] \leq C\left(1 + \left\|\Delta_k(s)\right\|_\infty\right)^2$, where C is a constant.

  Here, $\mathcal{H}_k = \{\Delta_k, \Delta_{k-1}, \ldots, \eta_{k-1}, \ldots, \alpha_{k-1}, \ldots, \beta_{k-1}, \ldots\}$ represents the historical information. The term $\|\cdot\|_\infty$ refers to the maximum norm.

6.4 Stochastic Gradient Descent (SGD)

- **Problem Setting:** Optimizing an objective function expressed as an expectation: $J(w) = E[f(w, X)]$.
- **Gradient Descent (GD):** The standard update rule $w_{k+1} = w_k - \alpha_k \nabla_w J(w_k) = w_k - \alpha_k E[\nabla_w f(w_k, X)]$ requires computing the expected gradient, which is often infeasible.
- **SGD Algorithm:** Approximates the expected gradient using the gradient from a single random sample x_k:

$$w_{k+1} = w_k - \alpha_k \nabla_w f(w_k, x_k)$$

- **Relationship to RM:** SGD is a special case of the RM algorithm applied to find the root of $g(w) = E[\nabla_w f(w, X)] = 0$. The noisy observation is $\tilde{g}(w_k, \eta_k) = \nabla_w f(w_k, x_k)$, and the noise term $\eta_k = \nabla_w f(w_k, x_k) - E[\nabla_w f(w_k, X)]$ has zero (conditional) mean (see Box 6.1).
- **Application to Mean Estimation:** Shows the incremental mean estimation algorithm is a special case of SGD applied to minimize $J(w) = \frac{1}{2}E[(w - X)^2]$.
- **Convergence Pattern:** SGD typically exhibits fast initial convergence when far from the optimum w^*, similar to GD. As w_k approaches w^*, the noise in the gradient estimate becomes

relatively more significant, leading to slower convergence and potential oscillations around the minimum. The relative error $\delta_k \propto 1/|w_k - w^*|$.

- **Deterministic Formulation:** Discusses applying SGD to minimize average loss over a finite dataset $J(w) = \frac{1}{n}\sum_{i=1}^{n} f(w, x_i)$. This is shown to be equivalent to the stochastic formulation by defining a uniform distribution over the dataset, allowing the use of SGD (requires sampling x_k randomly from the dataset).
- **BGD, SGD, and MBGD:** Compares Batch Gradient Descent (BGD, uses all n samples), Stochastic Gradient Descent (SGD, uses 1 sample), and Mini-Batch Gradient Descent (MBGD, uses m samples where $1 < m < n$). MBGD is often a practical compromise, balancing computational efficiency and gradient variance.

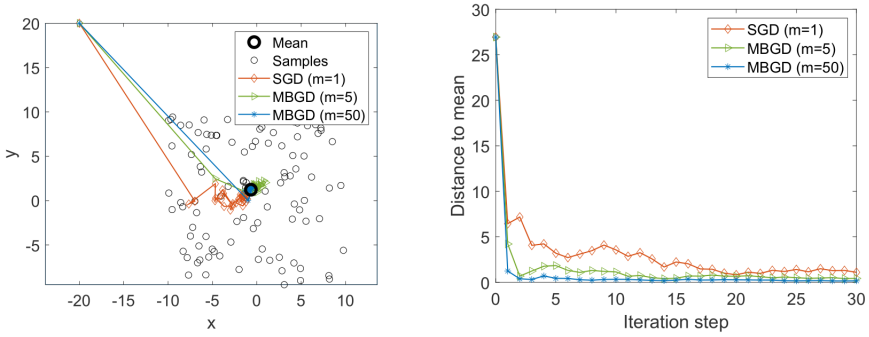


Fig. 25: An example for demonstrating stochastic and mini-batch gradient descent algorithms. The distribution of $X \in \mathbb{R}^2$ uniform in the square area centered at the origin with a side length as 20. The mean is $\mathbb{E}[X] = 0$. The mean estimation is based on 100 i.i.d. samples.

- **SGD Convergence (Thm 6.4):** Proves that SGD converges almost surely under conditions including strong convexity of the objective function, appropriate step sizes (satisfying RM conditions), and i.i.d. data samples.

Chapter 7: Temporal-Difference Methods

7.1 TD learning of state values

- **Purpose:** Introduces the most basic TD algorithm, often called **TD(0)**, used for **policy evaluation** (estimating the state-value function $v_\pi(s)$ for a given policy π).
- **TD(0) Algorithm:** Updates are based on observed transitions (s_t, r_{t+1}, s_{t+1}):

$$v_{t+1}(s_t) = v_t(s_t) - \alpha_t(s_t)\left[v_t(s_t) - (r_{t+1} + \gamma v_t(s_{t+1}))\right],$$

For all other states $s \neq s_t$, $v_{t+1}(s) = v_t(s)$.

- $\alpha_t(s_t)$ is the learning rate for state s_t at time t.
- $\bar{v}_t = r_{t+1} + \gamma v_t(s_{t+1})$ is called the **TD Target**.
- $\delta_t = v_t(s_t) - \bar{v}_t = v_t(s_t) - r_{t+1} - \gamma v_t(s_{t+1})$ is called the **TD Error**. The update can be written as $v_{t+1}(s_t) = v_t(s_t) - \alpha_t \delta_t$.

$$\underbrace{v_{t+1}(s_t)}_{\text{new estimate}} = \underbrace{v_t(s_t)}_{\text{current estimate}} - \alpha_t(s_t) \left[\overbrace{v_t(s_t) - \left(\underbrace{r_{t+1} + \gamma v_t(s_{t+1})}_{\text{TD target } \bar{v}_t} \right)}^{\text{TD error } \delta_t} \right],$$

- **Principle:** TD(0) can be viewed as a stochastic approximation algorithm solving the Bellman expectation equation $v_\pi(s) = E_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s]$.
- **Properties:** Incremental; uses bootstrapping (update depends on the estimate $v_t(s_{t+1})$); compared to MC, generally has lower variance but potentially introduces bias.

TD learning	**MC learning**
Incremental: TD learning is incremental. It can update the state/action values immediately after receiving an experience sample.	*Non-incremental:* MC learning is non-incremental. It must wait until an episode has been completely collected. That is because it must calculate the discounted return of the episode.
Continuing tasks: Since TD learning is incremental, it can handle both episodic and continuing tasks. Continuing tasks may not have terminal states.	*Episodic tasks:* Since MC learning is non-incremental, it can only handle episodic tasks where the episodes terminate after a finite number of steps.
Bootstrapping: TD learning bootstraps because the update of a state/action value relies on the previous estimate of this value. As a result, TD learning requires an initial guess of the values.	*Non-bootstrapping:* MC is not bootstrapping because it can directly estimate state/action values without initial guesses.
Low estimation variance: The estimation variance of TD is lower than that of MC because it involves fewer random variables. For instance, to estimate an action value $q_\pi(s_t, a_t)$, Sarsa merely requires the samples of three random variables: $R_{t+1}, S_{t+1}, A_{t+1}$.	*High estimation variance:* The estimation variance of MC is higher since many random variables are involved. For example, to estimate $q_\pi(s_t, a_t)$, we need samples of $R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots$. Suppose that the length of each episode is L. Assume that each state has the same number of actions as $\lvert\mathcal{A}\rvert$. Then, there are $\lvert\mathcal{A}\rvert^L$ possible episodes following a soft policy. If we merely use a few episodes to estimate, it is not surprising that the estimation variance is high.

Fig. 26: A comparison between TD learning and MC learning

- **Convergence (Thm 7.1):** Under standard stochastic approximation conditions for the step sizes ($\sum \alpha_t(s) = \infty, \sum \alpha_t^2(s) < \infty$) and assuming sufficient exploration of all states, $v_t(s)$ converges almost surely to the true $v_\pi(s)$.

7.2 Sarsa: TD learning of action values

- **Purpose:** Introduces the Sarsa algorithm for estimating the **action-value function** $q_\pi(s, a)$, which can be directly used for **control** (finding optimal policies) when combined with policy improvement.
- **Name Origin:** The update requires the tuple $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$, hence the name Sarsa.

- **Sarsa Algorithm :**

$$q_{t+1}(s_t, a_t) = q_t(s_t, a_t) - \alpha_t(s_t, a_t)\left[q_t(s_t, a_t) - (r_{t+1} + \gamma q_t(s_{t+1}, a_{t+1}))\right]$$

For all other state-action pairs $(s, a) \neq (s_t, a_t)$, $q_{t+1}(s, a) = q_t(s, a)$.

- **Principle:** Sarsa is a stochastic approximation method for solving the Bellman equation for action values: $q_\pi(s, a) = E_\pi[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1})|S_t = s, A_t = a]$.

- **Use for Control (Alg 7.1):** Combine Sarsa value estimation with ϵ-greedy policy improvement for an **on-policy** control algorithm. **Algorithm 7.1: Optimal policy learning by Sarsa**
Initialization: $\alpha_t(s, a) = \alpha > 0$ for all (s, a) and all t. $\epsilon \in (0, 1)$. Initial $q_0(s, a)$ for all (s, a). Initial ϵ-greedy policy π_0 derived from q_0.
Goal: Learn an optimal policy that can lead the agent to the target state from an initial state s_0.
For each episode, do
Generate a_0 at s_0 following $\pi_0(s_0)$
If s_t $(t = 0, 1, 2, \ldots)$ is not the target state, do
Collect an experience sample $(r_{t+1}, s_{t+1}, a_{t+1})$ given (s_t, a_t): generate r_{t+1}, s_{t+1} by interacting with the environment; generate a_{t+1} following $\pi_t(s_{t+1})$.
Update q-value for (s_t, a_t):

$$q_{t+1}(s_t, a_t) = q_t(s_t, a_t) - \alpha_t(s_t, a_t)\left[q_t(s_t, a_t) - (r_{t+1} + \gamma q_t(s_{t+1}, a_{t+1}))\right]$$

Update policy for s_t:

$$\pi_{t+1}(a|s_t) = 1 - \frac{\epsilon}{|\mathcal{A}(s_t)|}(|\mathcal{A}(s_t)| - 1) \text{ if } a = \arg\max_a q_{t+1}(s_t, a)$$

$$\pi_{t+1}(a|s_t) = \frac{\epsilon}{|\mathcal{A}(s_t)|} \text{ otherwise}$$

$s_t \leftarrow s_{t+1}$, $a_t \leftarrow a_{t+1}$

- **Convergence (Thm 7.2):** Under similar conditions to TD(0) (step sizes and sufficient exploration of all state-action pairs), $q_t(s, a)$ converges almost surely to the true action value $q_\pi(s, a)$ under the ϵ-greedy policy π.

- **Expected Sarsa (Box 7.4):** Variant using expected value for the next Q-value:

$$q_{t+1}(s_t, a_t) = q_t(s_t, a_t) - \alpha_t \left[q_t(s_t, a_t) - \left(r_{t+1} + \gamma \sum_{a'} \pi_t(a'|s_{t+1}) q_t(s_{t+1}, a') \right) \right]$$

  Lower variance, slightly more computation

7.3 n-step Sarsa

- **Core Idea:** Generalizes one-step Sarsa (n=1) and MC methods (n=∞). It looks ahead n steps using actual rewards and then bootstraps using the estimated Q-value at step n.

- **n-step Return:** The target for the update is based on the n-step return:

$$G_{t:t+n} \doteq R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n q_t(s_{t+n}, a_{t+n})$$

  (where q_t is the estimate available at the time of update, typically $t+n$).

- **n-step Sarsa Update:** The update for $q(s_t, a_t)$ is performed at time $t+n$:

$$q_{t+1}(s_t, a_t) = q_t(s_t, a_t) - \alpha_t(s_t, a_t) \left[q_t(s_t, a_t) - (r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^n q_t(s_{t+n}, a_{t+n})) \right].$$

- **Characteristics:** The parameter n allows a trade-off between bias (higher for small n) and variance (higher for large n).

7.4 Q-learning: TD learning of optimal action values

- **Purpose:** One of the most important and widely used RL algorithms. It directly learns the **optimal action-value function** $q^*(s, a)$, independent of the policy being followed to generate data.

- **Q-learning Algorithm:** Uses the transition $(s_t, a_t, r_{t+1}, s_{t+1})$ for the update:

$$q_{t+1}(s_t, a_t) = q_t(s_t, a_t) - \alpha_t(s_t, a_t) \left[q_t(s_t, a_t) - \left(r_{t+1} + \gamma \max_{a \in A(s_{t+1})} q_t(s_{t+1}, a) \right) \right]$$
$$q_{t+1}(s, a) = q_t(s, a), \quad \text{for all } (s, a) \neq (s_t, a_t),$$

- **Principle:** Q-learning is a stochastic approximation method for solving the **Bellman Optimality Equation** for action values:** $q^*(s, a) = E[R_{t+1} + \gamma \max_{a'} q^*(S_{t+1}, a') | S_t = s, A_t = a]$.

- **Off-policy vs On-policy:**

  - **Q-learning is Off-policy:** The update target uses $\max_{a'} q_t(s_{t+1}, a')$, which depends only on the next state s_{t+1} and the current value estimates, *not* on the action a_{t+1} that was actually taken next. This allows Q-learning to learn the optimal value function q^* using data

generated by any sufficiently exploratory **behavior policy** π_b, even if π_b is different from the optimal **target policy** π^*.

  - **Sarsa is On-policy:** Its update target depends on a_{t+1}, which is chosen according to the current policy π_t. It learns the value of the policy it follows.

- **Implementation (Alg 7.2 On-policy version, Alg 7.3 Off-policy version):** Shows how Q-learning can be implemented, either using an ϵ-greedy policy derived from its own Q-values for exploration (on-policy control style) or learning from data generated by an arbitrary behavior policy (true off-policy learning).

Algorithm 7.2: Optimal policy learning via Q-learning (on-policy version)

Initialization: $\alpha_t(s,a) = \alpha > 0$ for all (s,a) and all t. $\epsilon \in (0,1)$. Initial $q_0(s,a)$ for all (s,a). Initial ϵ-greedy policy π_0 derived from q_0.

Goal: Learn an optimal path that can lead the agent to the target state from an initial state s_0.

For each episode, do

If s_t $(t = 0, 1, 2, \ldots)$ is not the target state, do

Collect the experience sample (a_t, r_{t+1}, s_{t+1}) given s_t: generate a_t following $\pi_t(s_t)$; generate r_{t+1}, s_{t+1} by interacting with the environment.

Update q-value for (s_t, a_t):

$$q_{t+1}(s_t,a_t) = q_t(s_t,a_t) - \alpha_t(s_t,a_t)\left[q_t(s_t,a_t) - (r_{t+1} + \gamma \max_a q_t(s_{t+1},a))\right]$$

Update policy for s_t:

$$\pi_{t+1}(a|s_t) = 1 - \frac{\epsilon}{|\mathcal{A}(s_t)|}(|\mathcal{A}(s_t)| - 1) \text{ if } a = \arg\max_a q_{t+1}(s_t,a)$$

$$\pi_{t+1}(a|s_t) = \frac{\epsilon}{|\mathcal{A}(s_t)|} \text{ otherwise}$$

Algorithm 7.3: Optimal policy learning via Q-learning (off-policy version)

Initialization: Initial guess $q_0(s,a)$ for all (s,a). Behavior policy $\pi_b(a|s)$ for all (s,a). $\alpha_t(s,a) = \alpha > 0$ for all (s,a) and all t.

Goal: Learn an optimal target policy π_T for all states from the experience samples generated by π_b.

For each episode $\{s_0, a_0, r_1, s_1, a_1, r_2, \ldots\}$ generated by π_b, do

For each step $t = 0, 1, 2, \ldots$ of the episode, do

Update q-value for (s_t, a_t):

$$q_{t+1}(s_t, a_t) = q_t(s_t, a_t) - \alpha_t(s_t, a_t)\left[q_t(s_t, a_t) - (r_{t+1} + \gamma \max_a q_t(s_{t+1}, a))\right]$$

Update target policy for s_t:

$$\pi_{T,t+1}(a|s_t) = 1 \text{ if } a = \arg\max_a q_{t+1}(s_t, a)$$

$$\pi_{T,t+1}(a|s_t) = 0 \text{ otherwise}$$

- **Examples:** Demonstrates Q-learning's convergence, the utility of off-policy learning, and its dependence on the behavior policy providing sufficient exploration.

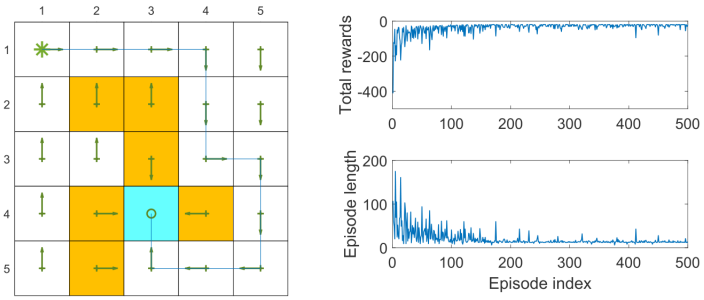


Fig. 27: An example for demonstrating Q-learning. All the episodes start from the top-left state and terminate after reaching the target state. The aim is to find an optimal path from the starting state to the target state. The reward settings are $r_{target} = 0, r_{forbidden} = r_{boundary} = -10, r_{other} = -1$. The learning rate is $\alpha = 0.1$ and the value of ϵ is 0.1. The left figure shows the final policy obtained by the algorithm. The right figure shows the total reward and length of every episode.

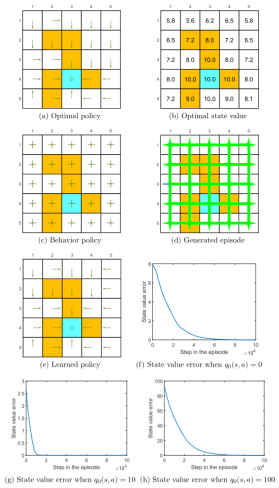


Fig. 28: Examples for demonstrating off-policy learning via Q-learning.The optimal policy and optimal state values are shown in (a) and (b), respectively.The behavior policy and the generated episode are shown in (c) and (d), respectively. The estimated policy and the estimation error evolution are shown in (e) and (f), respectively. The cases with different initial values are shown in (g) and (h).

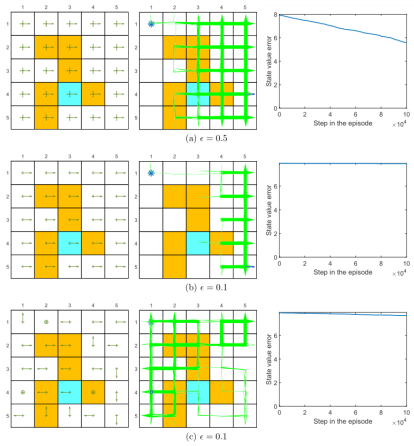


Fig. 29: The performance of Q-learning drops when the behavior policy is not exploratory. The figures in the left column show the behavior policies. The figures in the middle column show the generated episodes following the corresponding behavior policies. The episode in each example has 100,000 steps. The figures in the right column show the evolution of the root-mean-square error of the estimated state values.

7.5 A unified viewpoint

- **General Update Form:** Many TD (and MC) action-value algorithms fit:

$$q_{t+1}(s_t, a_t) = q_t(s_t, a_t) - \alpha_t(s_t, a_t)[q_t(s_t, a_t) - \bar{q}_t],$$

- **Different TD Targets $\bar{q}_t$:** Summarizes the specific target $\bar{q}_t$ used by Sarsa, n-step Sarsa, Q-learning, Expected Sarsa, and Monte Carlo.
- **Equations Solved:** Points out that Q-learning solves the Bellman Optimality Equation (BOE), whereas the other listed TD/MC methods solve the Bellman Equation (BE) for their respective (target or behavior) policies.

Algorithm	**Expression of the TD target $\bar{q}_t$ in (7.20)**
Sarsa	$\bar{q}_t = r_{t+1} + \gamma q_t(s_{t+1}, a_{t+1})$
n-step Sarsa	$\bar{q}_t = r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^n q_t(s_{t+n}, a_{t+n})$
Q-learning	$\bar{q}_t = r_{t+1} + \gamma \max_a q_t(s_{t+1}, a)$
Monte Carlo	$\bar{q}_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots$

Algorithm	**Equation to be solved**
Sarsa	BE: $q_\pi(s,a) = \mathbb{E}\left[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) \middle| S_t = s, A_t = a\right]$
n-step Sarsa	BE: $q_\pi(s,a) = \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^n q_\pi(S_{t+n}, A_{t+n}) \middle| S_t = s, A_t = a]$
Q-learning	BOE: $q(s,a) = \mathbb{E}\left[R_{t+1} + \gamma \max_a q(S_{t+1}, a) \middle| S_t = s, A_t = a\right]$
Monte Carlo	BE: $q_\pi(s,a) = \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots \middle| S_t = s, A_t = a]$

Fig. 30: A unified point of view of TD algorithms. Here, BE and BOE denote the Bellman equation and Bellman optimality equation, respectively.

Chapter 8: Value Function Methods

8.1 Value representation: From table to function

- **Core Idea:** Compares the traditional **tabular representation** of value functions with the **function approximation** approach.
- **Tabular Method:**
  - Stores an independent estimate $\hat{v}(s)$ (or $\hat{q}(s,a)$) for each state (or state-action pair).
  - Pros: Simple, theoretically well-understood.
  - Cons: Requires memory proportional to the state/action space size; lacks generalization (experience in one state doesn't inform others).
- **Function Approximation Method:**
  - Uses a parameterized function $\hat{v}(s,w) \approx v_\pi(s)$ or $\hat{q}(s,a,w) \approx q_\pi(s,a)$ to approximate the true value function, where w is a parameter vector.

- **Advantages:**
  - Memory Efficient: Only needs to store parameters w, whose dimension is typically much smaller than $|\mathcal{S}|$ or $|\mathcal{S}| \times |\mathcal{A}|$.
  - Generalization: Updating parameters w based on experience from one state can influence the value estimates for other (potentially unvisited but similar) states.
- **Disadvantages:**
  - Introduces approximation error: The function might not be able to represent the true value function exactly.
  - Interference: Updates for one state affect others.

- **Types of Functions:**
  - **Linear Function Approximation:** $\hat{v}(s, w) = \phi(s)^T w$ or $\hat{q}(s, a, w) = \phi(s, a)^T w$, where ϕ is a **feature vector**. Simple to analyze, but limited expressiveness; requires careful feature engineering.
  - **Non-linear Function Approximation:** E.g., **Artificial Neural Networks (ANNs)**. More powerful representation, can learn features automatically, forms the basis of deep RL.
- **Illustration:** Uses curve fitting examples (Fig 8.2, 8.4) to visually contrast the storage and update mechanisms of tabular vs. function approximation methods.

8.2 TD learning of state values based on function approximation

- **Goal:** Combine the TD(0) algorithm with function approximation $\hat{v}(s, w)$ for policy evaluation.
- **Objective Function $J(w)$:** Defines a metric to minimize the approximation error. A common choice is the **Mean Squared Value Error (MSVE)**:

$$J(w) = \mathbb{E}_{S \sim d_\pi}[(v_\pi(S) - \hat{v}(S, w))^2]$$

  where the expectation $\mathbb{E}_{S \sim d_\pi}[\cdot]$ is taken with respect to the **stationary distribution** d_π under policy π (detailed in Box 8.1). Other objectives mentioned include Bellman Error (BE) and Projected Bellman Error (PBE).

- **(Semi-)Gradient TD(0) Algorithm:** Adapts TD learning to update the parameters w using stochastic gradient descent ideas:

$$w_{t+1} = w_t + \alpha_t \underbrace{[r_{t+1} + \gamma \hat{v}(s_{t+1}, w_t) - \hat{v}(s_t, w_t)]}_{\text{TD Error } \delta_t} \nabla_w \hat{v}(s_t, w_t)$$

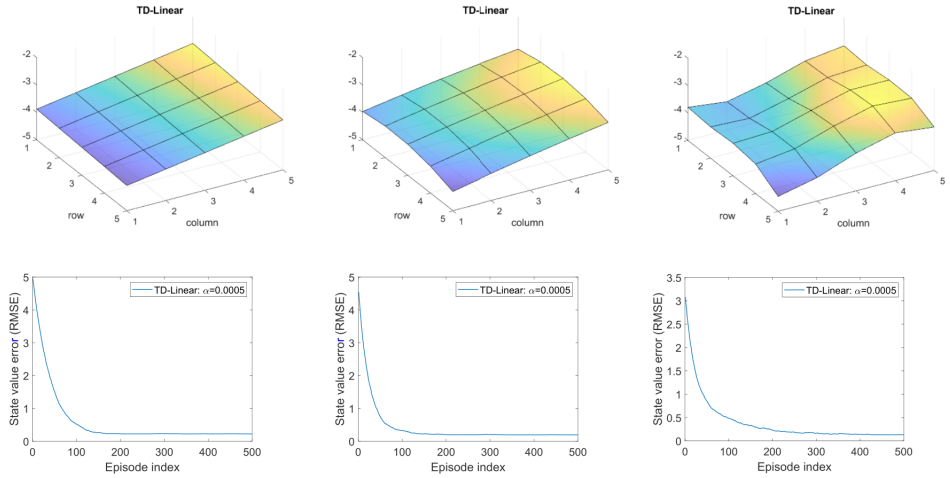
  This is often called "semi-gradient" because the gradient of the TD target term $r_{t+1} + \gamma \hat{v}(s_{t+1}, w_t)$ with respect to w_t is typically ignored in the update.

- **Linear Case:** When $\hat{v}(s, w) = \phi(s)^T w$, then $\nabla_w \hat{v}(s_t, w_t) = \phi(s_t)$, and the update becomes:

$$w_{t+1} = w_t + \alpha_t[r_{t+1} + \gamma\phi(s_{t+1})^T w_t - \phi(s_t)^T w_t]\phi(s_t)$$

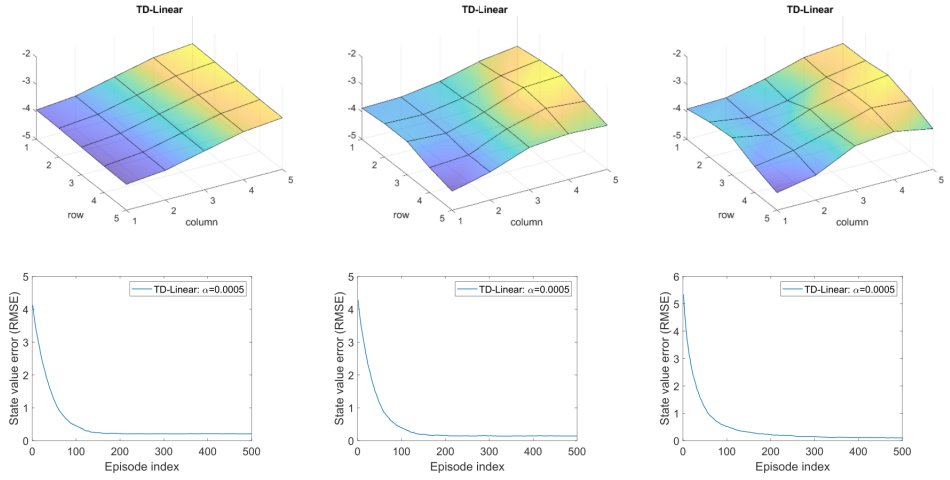
- **Feature Selection:** Discusses methods for choosing linear features, such as **Polynomial Features** and **Fourier Features**.

- **Examples (Fig 8.7, 8.8):** Show the convergence behavior and approximation error of TD-Linear using different feature vectors.



(a) $\phi(s) \in \mathbb{R}^3$ (b) $\phi(s) \in \mathbb{R}^6$ (c) $\phi(s) \in \mathbb{R}^{10}$

Fig. 31: TD-Linear estimation results obtained with the polynomial features



(a) $q = 1$ and $\phi(s) \in \mathbb{R}^4$ (b) $q = 2$ and $\phi(s) \in \mathbb{R}^9$ (c) $q = 3$ and $\phi(s) \in \mathbb{R}^{16}$

Fig. 32: TD-Linear estimation results obtained with the Fourier features

- **Theoretical Analysis (Linear Case):**
  - **Convergence Point:** The expected update of TD-Linear converges towards $w^* = A^{-1}b$, where $A = \mathbb{E}[\phi(S)(\phi(S) - \gamma\phi(S'))^T]$ and $b = \mathbb{E}[R_{t+1}\phi(S)]$ (expectations under stationary distribution). This w^* is the solution that minimizes the PBE.
  - **Error Bound:** The approximation error of the converged solution $\|\hat{v}(w^*) - v_\pi\|_D$ is bounded relative to the best possible approximation error within the function space.

$$\|\hat{v}(w^*) - v_\pi\|_D = \|\Phi w^* - v_\pi\|_D \leq \frac{1}{1-\gamma} \min_w \|\hat{v}(w) - v_\pi\|_D = \frac{1}{1-\gamma} \min_w \sqrt{J_E(w)}.$$

  - **LSTD (Least-Squares TD):** An alternative algorithm that directly estimates A and b from samples using least squares and computes $w^* = \hat{A}^{-1}\hat{b}$. It's typically more sample efficient but computationally heavier than TD-Linear and mainly suited for linear FA.

8.3 TD learning of action values based on function approximation

- **Core Idea:** Extends Sarsa and Q-learning algorithms to use action-value function approximation $\hat{q}(s, a, w)$.
- **Sarsa with Function Approximation:**

$$w_{t+1} = w_t + \alpha_t[r_{t+1} + \gamma\hat{q}(s_{t+1}, a_{t+1}, w_t) - \hat{q}(s_t, a_t, w_t)]\nabla_w\hat{q}(s_t, a_t, w_t)$$

- **Q-learning with Function Approximation:**

$$w_{t+1} = w_t + \alpha_t[r_{t+1} + \gamma \max_{a'} \hat{q}(s_{t+1}, a', w_t) - \hat{q}(s_t, a_t, w_t)]\nabla_w\hat{q}(s_t, a_t, w_t)$$

- **Note:** The policy $\pi(a|s)$ used for action selection is typically still derived from $\hat{q}$ (e.g., ϵ-greedy), even though the value itself is represented by a function.

8.4 Deep Q-learning (DQN)

- **Core Idea:** Combines Q-learning with **Deep Neural Networks (DNNs)** as powerful non-linear function approximators for $q(s, a, w)$. A seminal algorithm in deep RL
- **Objective Function (Approx. Squared Bellman Optimality Error):** Aims to minimize the expected squared difference between the current Q-value estimate and the TD target derived from the Bellman Optimality Equation:

$$J(w) = E\left[(y_T - \hat{q}(S, A, w))^2\right]$$

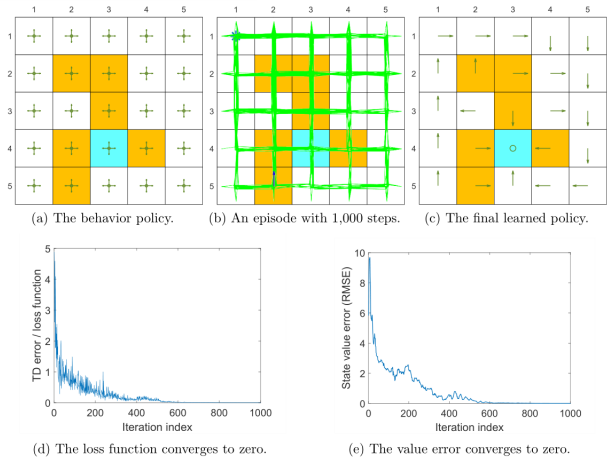
$$J = \mathbb{E}\left[\left(R + \gamma \max_{a \in \mathcal{A}(S')} \hat{q}(S', a, w) - \hat{q}(S, A, w)\right)^2\right]$$

where the TD target is $y_T = R + \gamma \max_{a'} \hat{q}(S', a', w^T)$.

- **Key Techniques:**
  1. **Experience Replay:** Stores transitions (s, a, r, s') in a large **replay buffer** B. During training, samples mini-batches randomly from B to update the network parameters w. This breaks temporal correlations in the data and improves data efficiency.
  2. **Target Network:** Uses a separate neural network (the target network) with parameters w^T that are updated less frequently (e.g., copied from the main network parameters w every C steps). The TD target y_T is calculated using this stable target network w^T. This helps stabilize the learning process by preventing the target values from changing too rapidly

- **Algorithm (based on DQN description):**
  1. Interact with environment using a behavior policy (e.g., ϵ-greedy based on current $\hat{q}(s, a, w)$) and store experiences in replay buffer B.
  2. Sample a random mini-batch of transitions from B.
  3. For each transition (s, a, r, s') in the batch, calculate the TD target $y_T = r + \gamma \max_{a'} \hat{q}(s', a', w^T)$ using the target network.
  4. Calculate the current Q-value estimate $\hat{q}(s, a, w)$ using the main network.
  5. Compute the loss (e.g., Mean Squared Error) between y_T and $\hat{q}(s, a, w)$ over the mini-batch.
  6. Perform a gradient descent step to update the main network parameters w.
  7. Periodically update the target network parameters: $w^T \leftarrow w$.

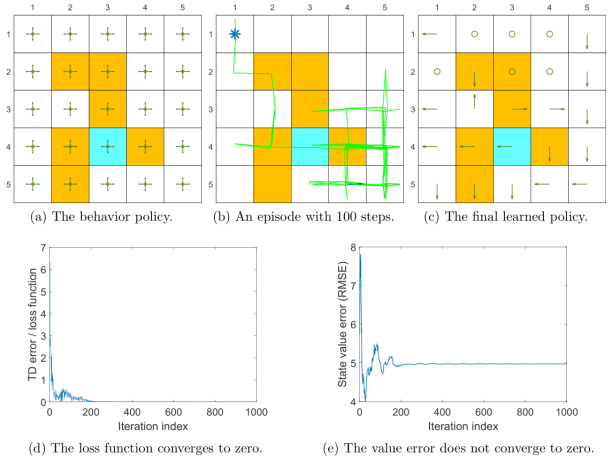
- **Examples (Fig 8.11, 8.12):** Demonstrate DQN's high sample efficiency compared to tabular methods, but also sensitivity to insufficient data.



(a) The behavior policy. (b) An episode with 1,000 steps. (c) The final learned policy.

(d) The loss function converges to zero. (e) The value error converges to zero.

Fig. 33: Optimal policy learning via deep Q-learning. Here, $\gamma = 0.9$, $r_{boundary} = r_{forbidden} = -10, r_{target} = 1$. The batch size is 100.



(a) The behavior policy. (b) An episode with 100 steps. (c) The final learned policy.

(d) The loss function converges to zero. (e) The value error does not converge to zero.

Fig. 34: Optimal policy learning via deep Q-learning. Here, $\gamma = 0.9$, $r_{boundary} = r_{forbidden} = -10, r_{target} = 1$. The batch size is 50

Chapter 9: Policy Gradient Methods

9.1 Policy representation: From table to function

- **Core Shift:** Moves away from representing policies using tables (storing $\pi(a|s)$ for all s, a) towards using parameterized functions $\pi(a|s, \theta)$, where θ is a vector of parameters (e.g., weights of a neural network).
- **Advantages:**
  - Can handle large or continuous state and action spaces.
  - Can learn stochastic policies naturally.
  - Often exhibit better generalization.
- **Differences from Tabular:** Policy updates involve modifying θ, not table entries; retrieving action probabilities requires function evaluation.

9.2 Metrics for defining optimal policies

- **Purpose:** To optimize a parameterized policy, a scalar performance metric $J(\theta)$ is needed to quantify the policy's quality.
- **Common Metrics:**
  1. **Average State Value:** $J(\theta) = \bar{v}_{\pi_\theta} = \sum_{s \in \mathcal{S}} d(s) v_{\pi_\theta}(s) = \mathbb{E}_{S \sim d}[v_{\pi_\theta}(S)]$. The state distribution $d(s)$ can be:
     - An independent starting state distribution $d_0(s)$ (metric denoted $\bar{v}^0_{\pi_\theta}$).

– The stationary distribution $d_{\pi_\theta}(s)$ under the policy π_θ (metric denoted $\bar{v}_{\pi_\theta}$).

Equivalent Form: $J(\theta) = \mathbb{E}[\sum_{t=0}^{\infty} \gamma^t R_{t+1}]$.

2. **Average Reward:** $J(\theta) = \bar{r}_{\pi_\theta} = \sum_{s \in \mathcal{S}} d_{\pi_\theta}(s) r_{\pi_\theta}(s) = \mathbb{E}_{S \sim d_{\pi_\theta}}[r_{\pi_\theta}(S)]$. Here $r_{\pi_\theta}(s) = \mathbb{E}[R_{t+1}|S_t = s]$ is the expected immediate reward from state s under policy π_θ. This metric is particularly relevant for the average-reward setting (or undiscounted case). Equivalent Form: $J(\theta) = \lim_{n \to \infty} \frac{1}{n} \mathbb{E}[\sum_{t=0}^{n-1} R_{t+1}]$.

- **Metric Relationship (Discounted Case, Lemma 9.1):** For $\gamma < 1$, the average reward and average value (under stationary distribution) are related: $\bar{r}_{\pi_\theta} = (1-\gamma)\bar{v}_{\pi_\theta}$.

9.3 Gradients of the metrics

- **Core Task:** To derive the gradient of the performance metric $J(\theta)$ with respect to the policy parameters θ, $\nabla_\theta J(\theta)$. This gradient indicates the direction in which to change θ to most rapidly increase performance.
- **Policy Gradient Theorem (Thm 9.1):** This is the central theoretical result of the chapter, providing a general expression for the policy gradient:

$$\nabla_\theta J(\theta) = \sum_{s \in \mathcal{S}} \eta(s) \sum_{a \in \mathcal{A}} \nabla_\theta \pi(a|s, \theta) q_{\pi_\theta}(s, a)$$

Or, more commonly written in expectation form:

$$\nabla_\theta J(\theta) = \mathbb{E}_{S \sim \eta, A \sim \pi(S,\theta)}[\nabla_\theta \ln \pi(A|S, \theta) q_{\pi_\theta}(S, A)]$$

  - $q_{\pi_\theta}(S, A)$ is the action-value function under policy π_θ.
  - $\eta(s)$ is a state distribution, the exact form of which depends on the definition of $J(\theta)$ and whether the setting is discounted or undiscounted (e.g., it could be d_0, d_{π_θ}, or ρ_{π_θ} related to discounted state visitation).
  - $\nabla_\theta \ln \pi(A|S, \theta)$ is known as the **score function**. Using the log-likelihood trick ($\nabla \ln \pi = \frac{\nabla \pi}{\pi}$) is key for deriving this expectation form and for practical algorithms.

- **Specific Derivations:** The chapter provides detailed derivations for:
  - Discounted case $\nabla_\theta \bar{v}^0_{\pi_\theta}$ (Thm 9.2), where $\eta = \rho_{\pi_\theta}$.
  - Discounted case $\nabla_\theta \bar{r}_{\pi_\theta}$ (Thm 9.3), where $\eta \approx d_{\pi_\theta}$ (approximation discussed).
  - Undiscounted case $\nabla_\theta \bar{r}_{\pi_\theta}$ (Thm 9.5), where $\eta = d_{\pi_\theta}$. The derivation involves the definition of action values in the undiscounted setting and the Poisson equation.

9.4 Monte Carlo policy gradient (REINFORCE)

- **Core Idea:** Applies the Policy Gradient Theorem to optimize the policy, using Monte Carlo methods to estimate the required q_{π_θ} term.

- **Gradient Ascent:** The optimization algorithm updates parameters using the gradient: $\theta_{t+1} = \theta_t + \alpha \nabla_\theta J(\theta_t)$.
- **Stochastic Gradient Approximation:** Replaces the true gradient $\mathbb{E}[\dots]$ with an estimate from samples.
- **REINFORCE Algorithm (Alg 9.1):**
  - Uses the full return of an episode starting from step t, $G_t = \sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1}$, as an **unbiased estimate** of $q_{\pi_\theta}(s_t, a_t)$.
  - **Update Rule (applied after an episode for each step t):**

$$\theta \leftarrow \theta + \alpha G_t \nabla_\theta \ln \pi(a_t|s_t, \theta)$$

Algorithm 9.1: Policy Gradient by Monte Carlo (REINFORCE)
Initialization: Initial parameter θ; $\gamma \in (0, 1)$; $\alpha > 0$.
Goal: Learn an optimal policy for maximizing $J(\theta)$.
For each episode, do
Generate an episode $\{s_0, a_0, r_1, \dots, s_{T-1}, a_{T-1}, r_T\}$ following $\pi(\theta)$.
For $t = 0, 1, \dots, T-1$:
Value update: $q_t(s_t, a_t) = \sum_{k=t+1}^{T} \gamma^{k-t-1} r_k$
Policy update: $\theta \leftarrow \theta + \alpha \nabla_\theta \ln \pi(a_t|s_t, \theta) q_t(s_t, a_t)$

- **Interpretation:** The update $\theta_{t+1} = \theta_t + \alpha \beta_t \nabla_\theta \pi(a_t|s_t, \theta_t)$ (where $\beta_t \approx G_t / \pi(a_t|s_t, \theta_t)$):
  - Increases the probability $\pi(a_t|s_t, \theta)$ for actions (a_t) that led to high returns G_t.
  - Decreases the probability for actions that led to low returns.
  - The division by $\pi(a_t|s_t, \theta_t)$ implicitly encourages exploration of less probable actions if they yield good returns.

Chapter 10: Actor-Critic Methods

10.1 The simplest actor-critic algorithm (QAC)

- **Core Idea:** The most basic AC structure. It directly uses an estimate of the action-value function $q_\pi(S, A)$ within the policy gradient formula $\nabla_\theta J(\theta) = \mathbb{E}[\nabla_\theta \ln \pi(A|S, \theta) q_\pi(S, A)]$. The q_π estimate is learned by the Critic using a TD method, often resembling Sarsa.
- **Actor (Policy Update):** Updates policy parameters θ using the Critic's Q-value estimate $q(s, a, w)$:

$$\theta_{t+1} = \theta_t + \alpha_\theta \nabla_\theta \ln \pi(a_t|s_t, \theta_t) q(s_t, a_t, w_t)$$

- **Critic (Value Update):** Updates the action-value function parameters w using a TD update (e.g., Sarsa-like):

$$w_{t+1} = w_t + \alpha_w[r_{t+1} + \gamma q(s_{t+1}, a_{t+1}, w_t) - q(s_t, a_t, w_t)]\nabla_w q(s_t, a_t, w_t)$$

- **Characteristics:** Simple structure illustrating the AC concept. Typically **on-policy** because the Critic update (if Sarsa-like) depends on the next action a_{t+1} taken by the current policy. Algorithm pseudocode is provided in Alg 10.1.

  Algorithm 10.1: The simplest actor-critic algorithm (QAC)

  Initialization: A policy function $\pi(a|s, \theta_0)$ where θ_0 is the initial parameter. A value function $q(s, a, w_0)$ where w_0 is the initial parameter. $\alpha_w, \alpha_\theta > 0$.

  Goal: Learn an optimal policy to maximize $J(\theta)$.

  At time step t in each episode, do

  Generate a_t following $\pi(a|s_t, \theta_t)$, observe r_{t+1}, s_{t+1}, and then generate a_{t+1} following $\pi(a|s_{t+1}, \theta_t)$.

  Actor (policy update):

$$\theta_{t+1} = \theta_t + \alpha_\theta \nabla_\theta \ln \pi(a_t|s_t, \theta_t) q(s_t, a_t, w_t)$$

  Critic (value update):

$$w_{t+1} = w_t + \alpha_w \left[r_{t+1} + \gamma q(s_{t+1}, a_{t+1}, w_t) - q(s_t, a_t, w_t)\right] \nabla_w q(s_t, a_t, w_t)$$

10.2 Advantage Actor-Critic (A2C)

- **Core Idea:** Aims to **reduce the variance** of the policy gradient estimate by introducing a **baseline**. It leverages the property that subtracting a state-dependent baseline $b(S)$ from q_π does not change the expected policy gradient.
- **Baseline Invariance:** Shows that $\mathbb{E}[\nabla_\theta \ln \pi(A|S, \theta)(q_\pi(S, A) - b(S))] = \mathbb{E}[\nabla_\theta \ln \pi(A|S, \theta) q_\pi(S, A)]$ because $\mathbb{E}[\nabla_\theta \ln \pi(A|S, \theta) b(S)] = 0$.
- **Advantage Function:** A common and effective baseline choice is the state-value function $b(S) = v_\pi(S)$. The term $q_\pi(S, A) - v_\pi(S)$ is called the **Advantage Function**, denoted $\delta_\pi(S, A)$. It represents how much better action A is compared to the average action under policy π in state S. The policy gradient becomes $\mathbb{E}[\nabla_\theta \ln \pi(A|S, \theta) \delta_\pi(S, A)]$.
- **Advantage Estimation:** In practice, the advantage function is often approximated by the **TD error** from a state-value Critic:

$$\delta_t = r_{t+1} + \gamma v(s_{t+1}, w_t) - v(s_t, w_t) \approx \delta_\pi(s_t, a_t)$$

This allows the Critic to only learn the state-value function $v(s, w)$.

- **A2C / TD Actor-Critic Algorithm (Alg 10.2):**
  - **Actor (Policy Update):** Uses the TD error δ_t as the advantage estimate:

$$\theta_{t+1} = \theta_t + \alpha_\theta \delta_t \nabla_\theta \ln \pi(a_t|s_t, \theta_t)$$

  - **Critic (Value Update):** Updates the state-value function parameters using the TD error:

$$w_{t+1} = w_t + \alpha_w \delta_t \nabla_w \hat{v}(s_t, w_t)$$

Algorithm 10.2: Advantage actor-critic (A2C) or TD actor-critic
Initialization: A policy function $\pi(a|s, \theta_0)$ where θ_0 is the initial parameter. A value function $v(s, w_0)$ where w_0 is the initial parameter. $\alpha_w, \alpha_\theta > 0$.
Goal: Learn an optimal policy to maximize $J(\theta)$.
At time step t in each episode, do
Generate a_t following $\pi(a|s_t, \theta_t)$ and then observe r_{t+1}, s_{t+1}.
Advantage (TD error):

$$\delta_t = r_{t+1} + \gamma v(s_{t+1}, w_t) - v(s_t, w_t)$$

Actor (policy update):

$$\theta_{t+1} = \theta_t + \alpha_\theta \delta_t \nabla_\theta \ln \pi(a_t|s_t, \theta_t)$$

Critic (value update):

$$w_{t+1} = w_t + \alpha_w \delta_t \nabla_w v(s_t, w_t)$$

- **Characteristics:** Typically results in lower variance updates compared to QAC or REINFORCE, leading to more stable learning. Usually implemented as **on-policy**. A3C is mentioned as an asynchronous variant.

10.3 Off-policy actor-critic

- **Core Idea:** Extends AC methods to the **off-policy** setting using the **Importance Sampling (IS)** technique. This allows learning a target policy $\pi(\theta)$ using data generated by a different behavior policy β.

- **Importance Sampling Review:** Recalls the principle: estimate $\mathbb{E}_{p_0}[X]$ using samples $\{x_i\}$ from p_1 via weighted average $\frac{1}{n}\sum_i \rho_i x_i$, where $\rho_i = \frac{p_0(x_i)}{p_1(x_i)}$ is the importance weight
- **Off-Policy Policy Gradient Theorem (Thm 10.1):** Provides the policy gradient for the target policy $\pi(\theta)$ using data from behavior policy β:

$$\nabla_\theta J(\theta) = \mathbb{E}_{S\sim\rho, A\sim\beta}\left[\frac{\pi(A|S,\theta)}{\beta(A|S)}\nabla_\theta \ln \pi(A|S,\theta) q_\pi(S,A)\right]$$

  Introduces the importance sampling ratio $\frac{\pi(A|S,\theta)}{\beta(A|S)}$.
- **Off-Policy AC Algorithm (Alg 10.3):** Applies the importance sampling ratio $\rho_t = \frac{\pi(a_t|s_t,\theta_t)}{\beta(a_t|s_t)}$ to the updates (shown for an A2C-like structure):
  - Actor: $\theta_{t+1} = \theta_t + \alpha_\theta \rho_t \delta_t \nabla_\theta \ln \pi(a_t|s_t,\theta_t)$
  - Critic: $w_{t+1} = w_t + \alpha_w \rho_t \delta_t \nabla_w \hat{v}(s_t, w_t)$ (Note: Critic update also needs IS correction).

  Algorithm 10.3: Off-policy actor-critic based on importance sampling
  Initialization: A given behavior policy $\beta(a|s)$. A target policy $\pi(a|s,\theta_0)$ where θ_0 is the initial parameter. A value function $v(s,w_0)$ where w_0 is the initial parameter. $\alpha_w, \alpha_\theta > 0$.
  Goal: Learn an optimal policy to maximize $J(\theta)$.
  At time step t in each episode, do
  Generate a_t following $\beta(s_t)$ and then observe r_{t+1}, s_{t+1}.
  Advantage (TD error):

$$\delta_t = r_{t+1} + \gamma v(s_{t+1}, w_t) - v(s_t, w_t)$$

  Actor (policy update):

$$\theta_{t+1} = \theta_t + \alpha_\theta \frac{\pi(a_t|s_t,\theta_t)}{\beta(a_t|s_t)}\delta_t \nabla_\theta \ln \pi(a_t|s_t,\theta_t)$$

  Critic (value update):

$$w_{t+1} = w_t + \alpha_w \frac{\pi(a_t|s_t,\theta_t)}{\beta(a_t|s_t)}\delta_t \nabla_w v(s_t, w_t)$$

- **Benefits:** Enables learning from offline data or using more exploratory behavior policies.
- **Challenges:** Importance sampling ratios can lead to high variance.

10.4 Deterministic actor-critic

- **Core Idea:** Adapts the AC framework for **Deterministic Policies**, denoted as $a = \mu(s, \theta)$. This is particularly useful for continuous action spaces.
- **Deterministic Policy Gradient Theorem (Thm 10.2):** The gradient for a deterministic policy differs from the stochastic case:

$$\nabla_\theta J(\theta) = \mathbb{E}_{S\sim\eta}\left[\nabla_\theta \mu(S,\theta)\nabla_a q_\mu(S,a)|_{a=\mu(S,\theta)}\right]$$

  Crucially, it depends on the gradient of the action-value function with respect to the *action*, $\nabla_a q_\mu$.

- **Algorithm (Deterministic Policy Gradient / DDPG foundation, Alg 10.4):**
  - **Actor (Policy Update):** Updates policy parameters θ along $\nabla_a q_\mu$:

$$\theta_{t+1} = \theta_t + \alpha_\theta \nabla_\theta \mu(s_t,\theta_t)\nabla_a \hat{q}(s_t,a,w_t)|_{a=\mu(s_t,\theta_t)}$$

  - **Critic (Value Update):** Learns the action-value function $\hat{q}(s,a,w)$, typically using a TD update similar to Q-learning (since DPG is naturally off-policy). The TD target often uses the Actor network's output for the next action $a' = \mu(s_{t+1},\theta_t)$:

$$\delta_t = r_{t+1} + \gamma \hat{q}(s_{t+1},\mu(s_{t+1},\theta_t),w_t) - \hat{q}(s_t,a_t,w_t)$$

$$w_{t+1} = w_t + \alpha_w \delta_t \nabla_w \hat{q}(s_t,a_t,w_t)$$

Algorithm 10.4: Deterministic policy gradient or deterministic actor-critic
Initialization: A given behavior policy $\beta(a|s)$. A deterministic target policy $\mu(s,\theta_0)$ where θ_0 is the initial parameter. A value function $q(s,a,w_0)$ where w_0 is the initial parameter. $\alpha_w, \alpha_\theta > 0$.
Goal: Learn an optimal policy to maximize $J(\theta)$.
At time step t in each episode, do

Generate a_t following β and then observe r_{t+1}, s_{t+1}.
TD error:

$$\delta_t = r_{t+1} + \gamma q(s_{t+1},\mu(s_{t+1},\theta_t),w_t) - q(s_t,a_t,w_t)$$

Actor (policy update):

$$\theta_{t+1} = \theta_t + \alpha_\theta \nabla_\theta \mu(s_t,\theta_t)\left(\nabla_a q(s_t,a,w_t)\right)\big|_{a=\mu(s_t)}$$

Critic (value update):

$$w_{t+1} = w_t + \alpha_w \delta_t \nabla_w q(s_t,a_t,w_t)$$

- **Characteristics:**
  - Naturally **off-policy** (gradient calculation doesn't directly involve behavior policy probabilities).
  - Well-suited for **continuous action spaces**.