1. Reinforcement Learning: Sequential Decision Making

Notes based on Sutton & Barto

March 23, 2025

Contents

1 Index of Topics

2 Introduction to Sequential Decision Making

Reinforcement learning is fundamentally about sequential decision making - the study of how agents learn to make a sequence of decisions that maximize cumulative reward over time. Unlike supervised learning where correct input-output pairs are presented, in reinforcement learning the agent must learn by interacting with its environment and discovering which actions yield the most reward.

The key challenge in sequential decision making is that decisions not only affect immediate rewards but may also influence future situations and, consequently, future rewards. This creates a temporal credit assignment problem: how do we attribute credit or blame to decisions that have delayed consequences?

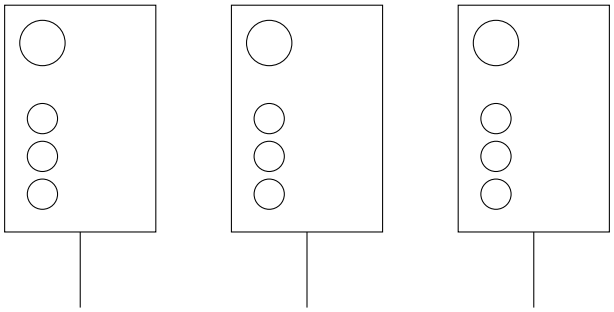
2.1 Multi-armed Bandit Problem

The multi-armed bandit problem represents the simplest form of sequential decision making. The name comes from imagining a gambler at a row of slot machines (known as "one-armed bandits"), who must decide which machines to play, how many times to play each machine, and in which order to play them, to maximize reward.

This problem formulation is fundamental to reinforcement learning for several reasons:

- It isolates the exploration-exploitation dilemma that is central to all reinforcement learning problems
- It provides a simplified setting with no state transitions, allowing us to focus solely on action selection
- It has important real-world applications such as clinical trials, website optimization, and adaptive routing

While simplified, the multi-armed bandit captures the essential challenge of learning from interaction: we must make decisions based on limited information, and our choices affect what information we receive in the future. This creates a feedback loop between learning and decision making.



Each machine has an unknown reward distribution

Figure 1: Illustration of a 3-armed bandit problem

2.1.1 Problem Formulation

In the multi-armed bandit problem:

- You face k different options or "arms"
- Each time step t, you select one arm $A_t \in \{1, 2, ..., k\}$
- After choosing arm a, you receive a reward R_t drawn from a probability distribution associated with that arm
- The goal is to maximize the total reward $\sum_{t=1}^{T} R_t$ over some time horizon T

In mathematical terms, choosing an arm a at time t is formalized as an action from a set $\mathcal{A}$. When we choose action A_t, we receive a reward R_t drawn from an unknown probability distribution $p(r|a)$.

$$\text{Action selection:} \quad A_t \in \mathcal{A} \tag{1}$$

Notation Overview

- A_t: The action chosen at time t
- $\mathcal{A}$: The set of all possible actions
- $p(r|a)$: The probability distribution of reward r given action a
- R_t: The reward received at time t after taking action A_{t-1} (following Sutton & Barto's indexing)

2.2 Actions, Rewards, and Policies

In the bandit problem, the agent's objective is to maximize the expected reward over time. The three fundamental components are:

2.2.1 Actions

Actions represent the choices available to the agent. In the multi-armed bandit setting, an action corresponds to selecting one of the k arms. We denote the action taken at time t as $A_t \in \mathcal{A}$, where $\mathcal{A}$ is the set of all possible actions.

2.2.2 Rewards

After taking action A_t at time t, the agent receives a numerical reward R_{t+1}. This reward is sampled from a probability distribution associated with the selected action: $R_{t+1} \sim p(r|A_t)$. The distribution $p(r|a)$ is initially unknown to the agent and must be learned through experience.

2.2.3 Policies

A policy defines the agent's strategy for selecting actions. In the context of bandits, a policy π is a mapping from actions to selection probabilities:

$$\pi(a) = P(A_t = a) \tag{2}$$

Notation Overview

- $\pi(a)$: The probability of selecting action a under policy π
- $P(A_t = a)$: The probability that the action at time t equals a

We can define more sophisticated policies that depend on past actions and rewards, but in the most basic form, a policy simply assigns probabilities to each action.

2.3 Evaluative Feedback

In reinforcement learning, the agent receives evaluative feedback rather than instructive feedback. The distinction is crucial:

- **Instructive feedback** (as in supervised learning) tells the agent what the correct action would have been
- **Evaluative feedback** only indicates how good the selected action was, but not whether it was the best possible action or what the best action would have been

To understand the difference more clearly:

- In supervised learning for image classification, if the model predicts "cat" when the true label is "dog", the model receives the correct label "dog" as feedback.
- In reinforcement learning, if an agent takes action A_t and receives reward R_{t+1}, it only knows the value of that specific actionnot what reward it would have received had it taken a different action or which action would have been optimal.

This approach mirrors many real-world learning scenarios. When a child touches a hot stove, they learn that this action led to pain, but they don't automatically know what alternative action would have been better. They must explore different approaches to build knowledge through experience.

Evaluative feedback forces the agent to actively explore to gather information about the environment, rather than passively receiving correct answers. This exploration process is what makes reinforcement learning fundamentally different from other machine learning paradigms.

The key challenge is to find the action a that produces the maximum expected reward:

$$\max_a \mathbb{E}[p(r|a)] \tag{3}$$

Notation Overview

- $\mathbb{E}[p(r|a)]$: The expected value of the reward distribution for action a
- $\max_a$: The operation of finding the action a that maximizes the subsequent expression

The true expected reward of each action is unknown, so the agent must estimate it from experience. This leads to the concept of action value.

2.3.1 Action Value

The value of an action a, denoted $q(a)$, is the expected reward when that action is selected:

$$q(a) = \mathbb{E}[R_t|A_{t-1} = a] \tag{4}$$

Notation Overview

- $q(a)$: The true value of action a (expected reward)
- $\mathbb{E}[R_t|A_{t-1} = a]$: The expected reward at time t given that action a was chosen at time $t-1$

In practice, we estimate $q(a)$ based on observed rewards. Let $Q_t(a)$ denote our estimate of $q(a)$ at time t. As we gather more experience, we refine this estimate to approach the true value.

$$Q_t(a) = \frac{1}{N_t(a)} \sum_{i=1}^{t} R_i \cdot \mathbf{1}_{A_{i-1}=a} \tag{5}$$

Notation Overview

- $Q_t(a)$: The estimated value of action a at time t
- $N_t(a)$: The number of times action a has been selected up to time t
- R_i: The reward received at time i
- $\mathbf{1}_{A_{i-1}=a}$: An indicator function that equals 1 if $A_{i-1} = a$ and 0 otherwise

2.4 Incremental Update of Action-Values

Calculating $Q_t(a)$ using the formula above would require storing all past rewards, which becomes inefficient as t grows. Instead, we can update our estimate incrementally after each new observation.

Incremental updating is a fundamental concept in reinforcement learning for several important reasons:

- **Memory efficiency**: We don't need to store the entire history of rewards for each action
- **Computational efficiency**: We can update estimates with constant time complexity, regardless of how many steps we've taken so far
- **Online learning**: We can immediately incorporate new information as it becomes available

- **Non-stationarity**: With appropriate step sizes, we can track changing reward distributions over time

The incremental update approach is also aligned with how humans and animals appear to learn from experience. We don't reprocess our entire history of experiences when we learn something new; instead, we adjust our existing understanding based on new observations.

Let's derive the incremental update rule:

$$Q_{t+1}(a) = \frac{1}{N_{t+1}(a)} \sum_{i=1}^{t+1} R_i \cdot \mathbf{1}_{A_{i-1}=a} \tag{6}$$

$$= \frac{1}{N_{t+1}(a)} \left(R_{t+1} \cdot \mathbf{1}_{A_t=a} + \sum_{i=1}^{t} R_i \cdot \mathbf{1}_{A_{i-1}=a} \right) \tag{7}$$

$$= \frac{1}{N_{t+1}(a)} \left(R_{t+1} \cdot \mathbf{1}_{A_t=a} + N_t(a) \cdot Q_t(a) \right) \tag{8}$$

If $A_t = a$, then $N_{t+1}(a) = N_t(a) + 1$ and the update becomes:

$$Q_{t+1}(a) = Q_t(a) + \frac{1}{N_{t+1}(a)} [R_{t+1} - Q_t(a)] \tag{9}$$

Notation Overview

- $Q_{t+1}(a)$: The updated estimate of the value of action a after observing the $(t+1)$-th reward
- $Q_t(a)$: The previous estimate of the value of action a
- R_{t+1}: The reward received at time $t+1$
- $\frac{1}{N_{t+1}(a)}$: The learning rate, which decreases as we observe more samples of action a

This is known as the incremental update rule. It adjusts the current estimate $Q_t(a)$ by a fraction of the prediction error $[R_{t+1} - Q_t(a)]$. The prediction error represents the difference between the observed reward and the expected reward.

A more general form replaces $\frac{1}{N_{t+1}(a)}$ with a step-size parameter $\alpha \in (0, 1]$:

$$Q_{t+1}(a) = Q_t(a) + \alpha [R_{t+1} - Q_t(a)] \tag{10}$$

Notation Overview

- α: The step-size parameter or learning rate, controlling how quickly the estimates are updated
- $[R_{t+1} - Q_t(a)]$: The prediction error or TD error

Using a constant α can be advantageous in non-stationary problems where the true action values change over time. The constant step size gives more weight to recent rewards, allowing the estimates to track changing values.

2.5 Exploitation vs. Exploration

The multi-armed bandit problem presents a fundamental dilemma: exploitation versus exploration.

- **Exploitation** means selecting the action that currently appears best based on accumulated knowledge
- **Exploration** means trying different actions to gather more information about their values

If we always exploit (by selecting the action with the highest estimated value), we risk missing better actions whose value we have underestimated due to limited samples. If we spend too much time exploring, we may waste opportunities to gain higher rewards from the best actions.

The exploration-exploitation dilemma extends far beyond reinforcement learning:

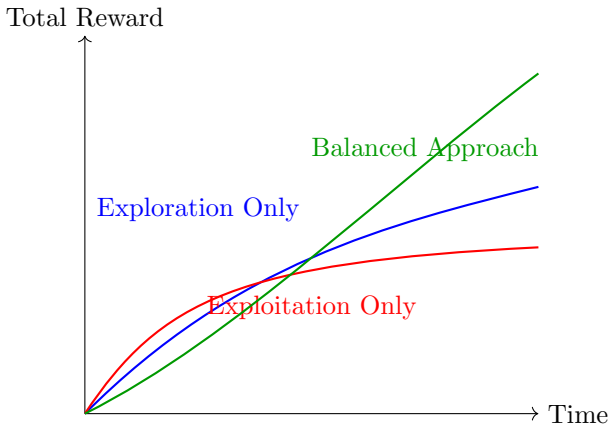


Figure 2: Conceptual illustration of different exploration-exploitation strategies and their long-term reward outcomes

- A restaurant-goer must decide whether to visit a new restaurant (exploration) or return to a favorite one (exploitation)
- A business must choose between investing in new products (exploration) or optimizing existing ones (exploitation)
- A researcher must decide whether to pursue novel research directions (exploration) or build on established findings (exploitation)

Finding the right balance is criticaltoo much exploitation leads to suboptimal solutions, while too much exploration wastes resources on unpromising options. Different contexts require different balances, and the optimal balance often changes over time as knowledge accumulates.

2.5.1 Greedy Action Selection

The simplest policy is to always select the action with the highest estimated value:

$$A_t = \arg\max_a Q_t(a) \tag{11}$$

Notation Overview

- $\arg\max_a Q_t(a)$: The action a that maximizes the estimated value $Q_t(a)$

This policy is known as the greedy policy. It always exploits current knowledge but never explores, which can lead to suboptimal performance if the initial estimates are inaccurate.

2.6 Action Sampling Methods

Several methods have been developed to balance exploration and exploitation. Each has unique characteristics that make it suitable for different scenarios:

2.6.1 ϵ-Greedy Selection

The ϵ-greedy approach introduces random exploration by selecting a random action with probability ϵ and the greedy action with probability $1 - \epsilon$:

$$\pi(a) = \begin{cases} 1 - \epsilon & \text{if } a = \arg\max_a Q_t(a) \\ \frac{\epsilon}{|\mathcal{A}|-1} & \text{otherwise} \end{cases} \tag{12}$$

Notation Overview

- $\pi(a)$: Probability of selecting action a
- ϵ: The exploration rate ($0 \leq \epsilon \leq 1$)
- $|\mathcal{A}|$: The number of actions (cardinality of the action set)
- $\arg\max_a Q_t(a)$: The action with the highest estimated value

The ϵ-greedy approach ensures that every action has some probability of being selected, which allows the agent to continue learning about all actions.

Advantages of ϵ-greedy:

- Simple to implement and understand
- Computationally efficient
- Guaranteed to eventually visit all actions
- Can be effective in practice despite its simplicity

Implementation considerations:

- A common practice is to start with a high value of ϵ (e.g., 0.1) and gradually decrease it over time
- This approach, called ϵ-*decay*, allows more exploration early in learning when uncertainty is high
- As more information is gathered, ϵ decreases to favor exploitation

Algorithm 1 ϵ-Greedy Action Selection

```
procedure EpsilonGreedy(Q_t, ε)
    p ← Random(0, 1)
    if p < ε then
        return Random action from A
    else
        return arg max_a Q_t(a)
    end if
end procedure
```

2.6.2 Softmax Selection

While ϵ-greedy treats all non-greedy actions equally, the softmax approach (also known as Boltzmann exploration) selects actions with probabilities relative to their estimated values:

$$\pi(a) = \frac{e^{Q_t(a)/\tau}}{\sum_{a' \in \mathcal{A}} e^{Q_t(a')/\tau}} \tag{13}$$

Notation Overview

- $\pi(a)$: Probability of selecting action a
- $e^{Q_t(a)/\tau}$: The exponential of the estimated value divided by temperature
- τ: The temperature parameter controlling the randomness of the selection
- $\sum_{a' \in \mathcal{A}} e^{Q_t(a')/\tau}$: Normalization term to ensure probabilities sum to 1

The temperature parameter τ controls the degree of exploration:

- As $\tau \to 0$, softmax approaches greedy selection
- As $\tau \to \infty$, softmax approaches uniform random selection

2.6.3 Upper Confidence Bound (UCB) Selection

The UCB algorithm addresses the exploration-exploitation dilemma by selecting actions based not only on their estimated values but also on the uncertainty in those estimates:

$$A_t = \arg\max_a \left[Q_t(a) + c\sqrt{\frac{\ln t}{N_t(a)}} \right] \tag{14}$$

Notation Overview

- $Q_t(a)$: The estimated value of action a at time t
- c: A parameter controlling the degree of exploration
- $\ln t$: Natural logarithm of the current time step
- $N_t(a)$: The number of times action a has been selected up to time t
- $\sqrt{\frac{\ln t}{N_t(a)}}$: The uncertainty term, which increases with time and decreases with the number of selections

The UCB approach favors actions that either have high estimated values or have been selected infrequently. As an action is selected more often, its uncertainty term decreases, reducing the incentive for further exploration unless its estimated value is truly high.

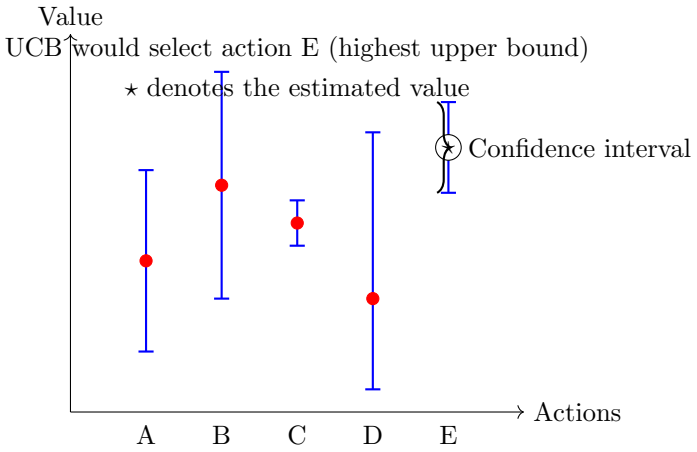


Figure 3: Illustration of Upper Confidence Bound selection

The UCB algorithm has several important theoretical and practical advantages:

- It is **principled**: The exploration term is derived from statistical confidence bounds
- It is **adaptive**: Unlike ϵ-greedy, which explores randomly, UCB directs exploration toward actions with high uncertainty
- It has **regret guarantees**: Under certain conditions, UCB algorithms have provably optimal regret bounds
- It is **parameter-sensitive**: The constant c controls the balance between exploration and exploitation

UCB exemplifies the principle of *optimism in the face of uncertainty*: by optimistically selecting actions based on the upper bound of their confidence intervals, the algorithm efficiently explores the action space and converges to the optimal policy.

3 Summary

Sequential decision making forms the foundation of reinforcement learning. The multi-armed bandit problem provides a simplified setting where:

- An agent repeatedly chooses among k actions
- Each action yields a reward from an unknown distribution
- The agent must balance exploration (trying different actions to learn their values) and exploitation (selecting the best-known action to maximize reward)

Key concepts include:

- Action values and their estimation
- Incremental updating of estimates
- Exploration strategies like ϵ-greedy, softmax, and UCB

These concepts extend to more complex reinforcement learning problems where actions also affect the environment state, leading to Markov Decision Processes and the full reinforcement learning framework.

2. Reinforcement Learning: From Bandits to Full RL

March 23, 2025

Contents

1 Introduction

This document covers the transition from the multi-armed bandit problem to full reinforcement learning, exploring how the introduction of states transforms the problem and solution approaches. We follow the notation and framework established in Sutton and Barto's Reinforcement Learning book.

2 Associativity: Reward Dependency on States and Actions

2.1 From Simple Bandits to State-Dependent Rewards

In the multi-armed bandit problem, rewards depend only on the selected action:

$$p(r_t|a_t) \tag{1}$$

Notation Overview

- $p(r_t|a_t)$ - The probability distribution of reward r_t given action a_t
- r_t - The reward received at time step t
- a_t - The action taken at time step t

Associativity introduces the concept of state, where rewards now depend on both the current state and the action taken:

$$p(r_t|s_t, a_t) \tag{2}$$

Notation Overview

- $p(r_t|s_t, a_t)$ - The probability distribution of reward r_t given state s_t and action a_t
- r_t - The reward received at time step t
- s_t - The state at time step t
- a_t - The action taken at time step t

In reinforcement learning with associativity, the agent recognizes that the environment is not stationary but exists in different states. The reward

received for a particular action depends not only on which action was chosen but also on the environmental state in which it was chosen.

2.2 The Agent-Environment Interface

The standard agent-environment interface in reinforcement learning extends the bandit formulation to include states:

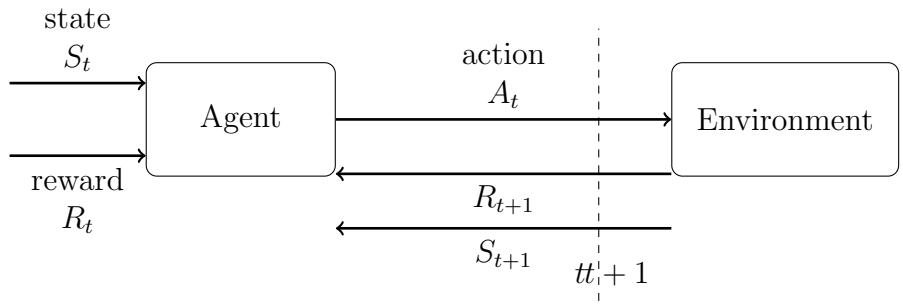


Figure 1: The agent-environment interaction in reinforcement learning (Sutton & Barto)

At each time step t:

- The agent observes the current state $S_t \in \mathcal{S}$
- Based on this state, the agent selects an action $A_t \in \mathcal{A}(S_t)$
- The environment responds with a reward $R_{t+1} \in \mathcal{R}$ and a new state S_{t+1}
- The agent then selects a new action based on the new state, and the process continues

This creates a sequence of states, actions, and rewards:

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, \ldots \tag{3}$$

Notation Overview

- S_t - The state at time step t
- A_t - The action taken at time step t
- R_{t+1} - The reward received after taking action A_t in state S_t
- $\mathcal{S}$ - The set of all possible states
- $\mathcal{A}(S_t)$ - The set of actions available in state S_t
- $\mathcal{R}$ - The set of possible rewards

2.3 The Markov Property

A key concept in reinforcement learning is the Markov property, which states that the future is independent of the past given the present. Formally, an environment has the Markov property if:

$$P(S_{t+1} = s', R_{t+1} = r | S_t, A_t, R_t, S_{t-1}, A_{t-1}, \ldots, R_1, S_0, A_0) = P(S_{t+1} = s', R_{t+1} = r | S_t, A_t) \tag{4}$$

Notation Overview

- $P(S_{t+1} = s', R_{t+1} = r | S_t, A_t, \ldots)$ - The probability of transitioning to state s' and receiving reward r given the entire history
- $P(S_{t+1} = s', R_{t+1} = r | S_t, A_t)$ - The probability of transitioning to state s' and receiving reward r given only the current state and action
- S_t - The state at time step t
- A_t - The action taken at time step t
- R_t - The reward received at time step t

The Markov property is crucial because it allows us to make decisions based solely on the current state without needing to remember the entire history of states and actions. This simplifies the problem significantly while still capturing the essential dynamics of many real-world situations.

3 Extension to Contextual Bandits

3.1 Definition and Properties

Contextual bandits represent an intermediate step between multi-armed bandits and full reinforcement learning. In this setting:

- The environment presents a state (or context) S_t to the agent
- The agent chooses an action A_t based on the state
- The environment provides a reward R_{t+1} that depends on both S_t and A_t
- The next state S_{t+1} is **not influenced** by the previous action A_t

The key distinguishing feature of contextual bandits is that while rewards depend on states, actions do not influence future states. The state transitions are independent of the agent's actions:

$$p(s_{t+1}|s_t, a_t) = p(s_{t+1}|s_t) \tag{5}$$

Notation Overview

- $p(s_{t+1}|s_t, a_t)$ - The probability of transitioning to state s_{t+1} given the current state s_t and action a_t
- $p(s_{t+1}|s_t)$ - The probability of transitioning to state s_{t+1} given only the current state s_t, independent of the action taken
- s_t - The state at time step t
- a_t - The action taken at time step t
- s_{t+1} - The next state at time step $t + 1$

3.2 Applications of Contextual Bandits

Contextual bandits are particularly useful in scenarios where:

- The state/context changes independently of the agent's actions
- Decisions only affect immediate rewards but not future states

- The agent needs to adapt to changing contexts

Some practical applications include:

- News article recommendation: The user's interests (state) change over time independently of which articles are recommended, but the reward (click-through rate) depends on both the user's interests and the selected article.
- Dynamic pricing: Market conditions (state) change due to external factors, while the price decision (action) affects immediate revenue but not future market conditions.
- Clinical trials with rotating patients: Each patient represents a new state, and the treatment decision affects the outcome for that patient but not the characteristics of future patients.

3.3 Action-Value Function for Contextual Bandits

In contextual bandits, the action-value function is defined as the expected immediate reward when taking action a in state s:

$$Q(s, a) = \mathbb{E}[R_{t+1}|S_t = s, A_t = a] \tag{6}$$

Notation Overview

- $Q(s, a)$ - The action-value function, representing the expected reward for taking action a in state s
- $\mathbb{E}[\cdot]$ - The expected value
- R_{t+1} - The reward received at time step $t + 1$
- S_t - The state at time step t
- A_t - The action taken at time step t

This differs from the action-value function in full reinforcement learning, which includes future rewards.

3.4 Estimating Action Values in Contextual Bandits

The action-value function in contextual bandits can be estimated from experience:

$$Q_t(s,a) = \frac{1}{n_t(s,a)} \sum_{i=1}^{t} r_i \cdot \mathbb{I}(s_i = s, a_i = a) \tag{7}$$

Notation Overview

- $Q_t(s,a)$ - The estimated action-value after t steps
- $n_t(s,a)$ - The number of times action a has been selected in state s up to time t
- r_i - The reward received at step i
- $\mathbb{I}(s_i = s, a_i = a)$ - An indicator function that equals 1 if $s_i = s$ and $a_i = a$, and 0 otherwise
- s_i - The state at time step i
- a_i - The action taken at time step i

An incremental update rule for the action values can also be derived:

$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha[r_{t+1} - Q_t(s_t, a_t)] \tag{8}$$

Notation Overview

- $Q_{t+1}(s_t, a_t)$ - The updated estimate of the action-value
- $Q_t(s_t, a_t)$ - The current estimate of the action-value
- α - The step-size parameter (learning rate), $0 < \alpha \leq 1$
- r_{t+1} - The observed reward
- s_t - The state at time step t
- a_t - The action taken at time step t

3.5 Policy in Contextual Bandits

A policy π in contextual bandits maps states to probabilities of selecting each possible action:

$$\pi(a|s) = P(A_t = a|S_t = s) \tag{9}$$

Notation Overview

- $\pi(a|s)$ - The probability of selecting action a in state s under policy π
- $P(A_t = a|S_t = s)$ - The probability that the agent selects action $A_t = a$ when in state $S_t = s$
- A_t - The action taken at time step t
- S_t - The state at time step t

Various policy types can be used in contextual bandits:

- **Greedy Policy**: Always selects the action with the highest estimated value:

$$\pi(a|s) = \begin{cases} 1 & \text{if } a = \arg\max_{a'} Q(s, a') \\ 0 & \text{otherwise} \end{cases} \tag{10}$$

- **ϵ-Greedy Policy**: Selects the best action most of the time, but occasionally explores:

$$\pi(a|s) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{|\mathcal{A}(s)|} & \text{if } a = \arg\max_{a'} Q(s, a') \\ \frac{\epsilon}{|\mathcal{A}(s)|} & \text{otherwise} \end{cases} \tag{11}$$

- **Softmax Policy**: Selects actions with probabilities proportional to their estimated values:

$$\pi(a|s) = \frac{e^{Q(s,a)/\tau}}{\sum_{a'} e^{Q(s,a')/\tau}} \tag{12}$$

  where τ is a temperature parameter controlling exploration.

3.6 Example: Contextual Multi-Armed Bandit

Consider a modified version of the multi-armed bandit problem where there are visible weather conditions (sunny, rainy, cloudy) that affect the rewards for each arm (e.g., different types of investments). The rewards for each arm depend on the weather, but today's action doesn't affect tomorrow's weather.

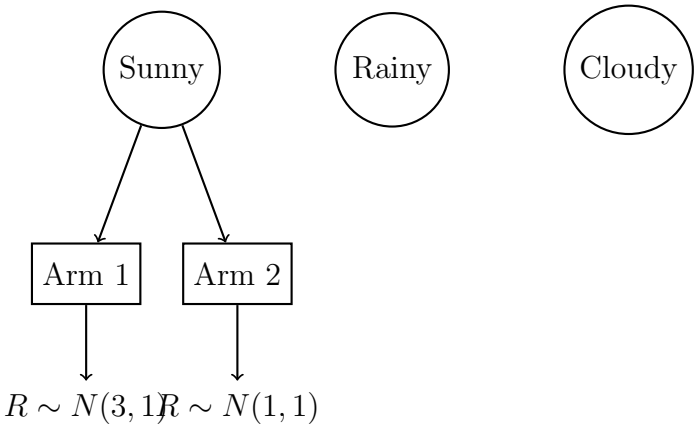


Figure 2: A simplified contextual bandit example with weather states

The key insight is that while the agent's action choice depends on the observed state (weather), and the rewards depend on both state and action, the agent's actions do not influence which state occurs next.

4 Full RL with State Transitions

4.1 The Critical Addition: Actions Affect Future States

The defining characteristic of full reinforcement learning is that actions not only influence immediate rewards but also affect future states, and through them, future rewards:

$$p(s_{t+1}|s_t, a_t) \tag{13}$$

Notation Overview

- $p(s_{t+1}|s_t, a_t)$ - The probability of transitioning to state s_{t+1} given the current state s_t and action a_t
- s_t - The state at time step t
- a_t - The action taken at time step t
- s_{t+1} - The next state at time step $t + 1$

This introduces a critical complexity: actions have long-term consequences through their influence on future states. The agent must consider not just immediate rewards but also how actions shape future opportunities.

4.2 Markov Decision Processes (MDPs)

Full reinforcement learning problems are typically formalized as Markov Decision Processes (MDPs), which are defined by:

- A set of states $\mathcal{S}$
- A set of actions $\mathcal{A}$
- A reward function $r(s, a) = \mathbb{E}[R_{t+1}|S_t = s, A_t = a]$
- A state transition probability function $p(s'|s, a) = P(S_{t+1} = s'|S_t = s, A_t = a)$
- A discount factor $\gamma \in [0, 1]$

The dynamics of an MDP are formally defined by:

$$p(s', r|s, a) = P(S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a) \tag{14}$$

Notation Overview

- $p(s', r|s, a)$ - The probability of transitioning to state s' and receiving reward r, given that the agent was in state s and took action a
- $P(S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a)$ - The probability that the state at time t is s' and the reward is r, given that the state at time $t - 1$ was s and the action taken was a
- S_t - The state at time step t
- R_t - The reward received at time step t
- A_t - The action taken at time step t

From this joint probability function, we can derive other important quantities:

$$p(s'|s, a) = \sum_{r \in \mathcal{R}} p(s', r|s, a) \tag{15}$$

$$r(s, a) = \mathbb{E}[R_t|S_{t-1} = s, A_{t-1} = a] = \sum_{r \in \mathcal{R}} r \sum_{s' \in \mathcal{S}} p(s', r|scla, a) \tag{16}$$

$$r(s, a, s') = \mathbb{E}[R_t | S_{t-1} = s, A_{t-1} = a, S_t = s'] = \sum_{r \in \mathcal{R}} r \frac{p(s', r|s, a)}{p(s'|s, a)} \tag{17}$$

Notation Overview

- $p(s'|s, a)$ - The probability of transitioning to state s' given state s and action a
- $r(s, a)$ - The expected reward when taking action a in state s
- $r(s, a, s')$ - The expected reward when transitioning from state s to state s' via action a
- $\mathcal{R}$ - The set of possible rewards
- $\mathcal{S}$ - The set of possible states

4.3 Episodic vs. Continuing Tasks

Reinforcement learning tasks can be categorized as either episodic or continuing:

- **Episodic Tasks**: These have a clear endpoint or terminal state. For example, a game of chess ends with either a win, loss, or draw. The sequence of states, actions, and rewards from the start to the terminal state is called an episode.
- **Continuing Tasks**: These go on indefinitely without a natural endpoint. For example, an ongoing process control system or a stock trading agent that operates continuously.

4.4 The Return and Value Functions in Full RL

In full RL, the objective is to maximize the expected return, which is the cumulative discounted reward:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{18}$$

Notation Overview

- G_t - The return at time step t
- R_{t+k} - The reward received k steps after time t
- γ - The discount factor, $0 \leq \gamma \leq 1$

For episodic tasks with a clear terminal state, we can alternatively define the return as:

$$G_t = \sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1} \tag{19}$$

where T is the final time step of the episode.

To unify the notation for both episodic and continuing tasks, we can use:

$$G_t = \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k \tag{20}$$

with the understanding that $T = \infty$ for continuing tasks, and $\gamma < 1$ for continuing tasks to ensure the sum is finite.

4.5 State-Value and Action-Value Functions

The state-value function represents the expected return when starting in state s and following policy π:

$$v_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s \right] \tag{21}$$

Notation Overview

- $v_\pi(s)$ - The state-value function for policy π
- $\mathbb{E}_\pi[\cdot]$ - The expected value when following policy π
- G_t - The return at time step t
- S_t - The state at time step t
- R_{t+k+1} - The reward received $k+1$ steps after time t
- γ - The discount factor, $0 \leq \gamma \leq 1$

Similarly, the action-value function represents the expected return when starting in state s, taking action a, and thereafter following policy π:

$$q_\pi(s,a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s, A_t = a\right] \quad (22)$$

Notation Overview

- $q_\pi(s,a)$ - The action-value function for policy π
- $\mathbb{E}_\pi[\cdot]$ - The expected value when following policy π
- G_t - The return at time step t
- S_t - The state at time step t
- A_t - The action taken at time step t
- R_{t+k+1} - The reward received $k+1$ steps after time t
- γ - The discount factor, $0 \leq \gamma \leq 1$

4.6 The Relationship Between State-Value and Action-Value Functions

The state-value and action-value functions are related by:

$$v_\pi(s) = \sum_a \pi(a|s) q_\pi(s,a) \quad (23)$$

This shows that the value of a state is the expected value of the actions that might be taken in that state, weighted by their probability under policy π.

Conversely, we can express the action-value in terms of the state-value:

$$q_\pi(s,a) = \sum_{s',r} p(s',r|s,a)[r + \gamma v_\pi(s')] \quad (24)$$

This indicates that the value of taking action a in state s is the expected immediate reward plus the discounted value of the next state.

4.7 The Bellman Equations

The recursive relationship between value functions is captured by the Bellman equations. For the state-value function:

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')] \tag{25}$$

Notation Overview

- $v_\pi(s)$ - The state-value function for policy π
- $\pi(a|s)$ - The probability of selecting action a in state s under policy π
- $p(s', r|s, a)$ - The probability of transitioning to state s' and receiving reward r, given state s and action a
- r - The reward
- γ - The discount factor, $0 \leq \gamma \leq 1$

And for the action-value function:

$$q_\pi(s, a) = \sum_{s',r} p(s', r|s, a) \left[r + \gamma \sum_{a'} \pi(a'|s') q_\pi(s', a') \right] \tag{26}$$

Notation Overview

- $q_\pi(s, a)$ - The action-value function for policy π
- $p(s', r|s, a)$ - The probability of transitioning to state s' and receiving reward r, given state s and action a
- $\pi(a'|s')$ - The probability of selecting action a' in state s' under policy π
- r - The reward
- γ - The discount factor, $0 \leq \gamma \leq 1$

The Bellman equations express a fundamental property: the value of a state (or state-action pair) equals the expected immediate reward plus the discounted value of the next state (or states).

4.8 Optimal Value Functions and Policies

A policy π is defined as better than or equal to a policy π' if its expected return is greater than or equal to that of π' for all states:

$$\pi \geq \pi' \iff v_\pi(s) \geq v_{\pi'}(s) \text{ for all } s \in \mathcal{S} \tag{27}$$

There is always at least one policy that is better than or equal to all other policies, called an optimal policy, denoted π_*. All optimal policies share the same optimal state-value function, v_*:

$$v_*(s) = \max_\pi v_\pi(s) \text{ for all } s \in \mathcal{S} \tag{28}$$

They also share the same optimal action-value function, q_*:

$$q_*(s,a) = \max_\pi q_\pi(s,a) \text{ for all } s \in \mathcal{S}, a \in \mathcal{A}(s) \tag{29}$$

Notation Overview

- $\pi \geq \pi'$ - Policy π is better than or equal to policy π'
- $v_\pi(s)$ - The state-value function for policy π
- $v_*(s)$ - The optimal state-value function
- $q_*(s,a)$ - The optimal action-value function
- $\max_\pi$ - The maximum over all possible policies
- $\mathcal{S}$ - The set of all possible states
- $\mathcal{A}(s)$ - The set of actions available in state s

4.9 The Bellman Optimality Equations

The Bellman optimality equations characterize the optimal value functions:

$$v_*(s) = \max_a \sum_{s',r} p(s',r|s,a)[r + \gamma v_*(s')] \tag{30}$$

$$q_*(s,a) = \sum_{s',r} p(s',r|s,a)[r + \gamma \max_{a'} q_*(s',a')] \tag{31}$$

Notation Overview

- $v_*(s)$ - The optimal state-value function
- $q_*(s, a)$ - The optimal action-value function
- $\max_a$ - The maximum over all actions $a \in \mathcal{A}(s)$
- $\max_{a'}$ - The maximum over all actions $a' \in \mathcal{A}(s')$
- $p(s', r|s, a)$ - The probability of transitioning to state s' and receiving reward r, given state s and action a
- r - The reward
- γ - The discount factor, $0 \leq \gamma \leq 1$

Given the optimal value functions, it is straightforward to determine an optimal policy:

$$\pi_*(s) = \arg\max_a q_*(s, a) \tag{32}$$

4.10 The Credit Assignment Problem

One of the fundamental challenges introduced by state transitions in full RL is the credit assignment problem: determining which actions in a sequence were responsible for a later reward.

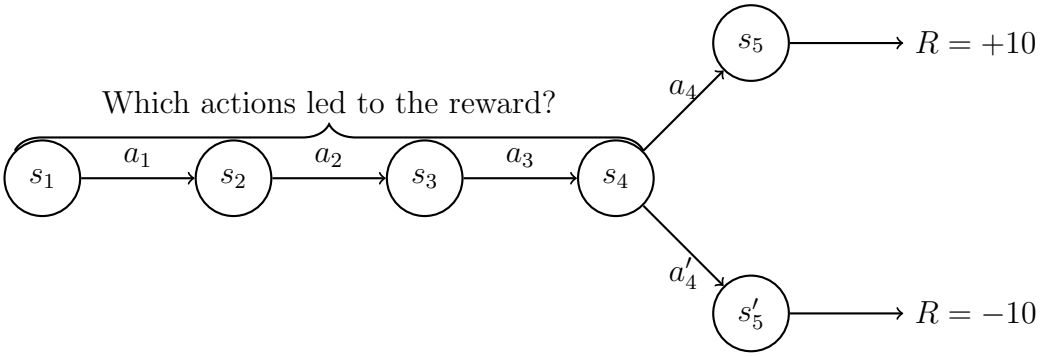


Figure 3: The credit assignment problem: determining which actions in a sequence contributed to the final reward

In the diagram, a positive reward is received after reaching state s_5, and a negative reward after reaching state s'_5. The challenge is to determine which

of the earlier actions in the sequence (a, a, a, a/a') were most responsible for the final outcome.

This problem becomes particularly difficult when:

- Rewards are delayed (occur many steps after the critical actions)
- Actions have long-term consequences through complex state transitions
- The environment is noisy or stochastic

Reinforcement learning algorithms address this problem through various methods:

- **Temporal Difference Learning**: Updates value estimates based on the difference between successive predictions
- **Eligibility Traces**: Maintain a record of which states and actions have been visited recently and are thus "eligible" for updates
- **Monte Carlo Methods**: Use complete episode returns to update action values

4.11 Comparison: Bandits vs. Contextual Bandits vs. Full RL

Feature	**Bandits**	**Contextual Bandits**	**Full RL**
States	No	Yes	Yes
Reward depends on state	No	Yes	Yes
Actions affect future states	No	No	Yes
Time horizon	Immediate	Immediate	Multiple steps
Value function	$Q(a)$	$Q(s, a)$	$q_\pi(s, a), v_\pi(s)$
Considers future rewards	No	No	Yes
Credit assignment	Simple	Simple	Complex
Optimal policy computation	Easy	Moderate	Hard
Exploration-exploitation	Simple	Moderate	Complex

Table 1: Comparison of key features across bandit problems, contextual bandits, and full RL

5 Conclusion

The transition from bandits to full reinforcement learning represents a significant increase in complexity and expressive power:

- **Bandits** involve only actions and immediate rewards with no concept of state.
- **Contextual Bandits** introduce states that influence rewards, but actions don't affect future states.
- **Full RL** incorporates state transitions influenced by actions, creating sequential decision problems where actions have long-term consequences.

This progression captures the core challenge of reinforcement learning: making decisions that optimize long-term cumulative reward when actions influence not just immediate rewards but also future states and opportunities.

The key concepts introduced in full reinforcement learning include:

- The Markov property, which allows decisions to be based solely on the current state
- Value functions, which estimate the expected future reward
- The Bellman equations, which express the recursive relationship between values of different states
- Optimal policies, which maximize the expected return
- The credit assignment problem, which involves determining which actions led to observed rewards

Understanding these fundamental concepts provides the foundation for more advanced reinforcement learning methods, including temporal difference learning, Monte Carlo methods, and function approximation techniques.

6 Summary of Key Equations

Here is a summary of all the key equations covered in this document:

6.1 Reward Dependencies

$$\begin{aligned} &\text{Bandits:} && p(r_t|a_t) && (33)\\ &\text{Contextual Bandits:} && p(r_t|s_t, a_t) && (34)\\ &\text{Full RL:} && p(r_t|s_t, a_t) \text{ and } p(s_{t+1}|s_t, a_t) && (35) \end{aligned}$$

6.2 MDP Dynamics

Joint Probability:
$$p(s', r|s, a) = P(S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a) \tag{36}$$

State Transition:
$$p(s'|s, a) = \sum_{r \in \mathcal{R}} p(s', r|s, a) \tag{37}$$

Expected Reward:
$$r(s, a) = \sum_{r \in \mathcal{R}} r \sum_{s' \in \mathcal{S}} p(s', r|s, a) \tag{38}$$

Expected Reward with Next State:
$$r(s, a, s') = \sum_{r \in \mathcal{R}} r \frac{p(s', r|s, a)}{p(s'|s, a)} \tag{39}$$

6.3 Return

Continuing Tasks:
$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{40}$$

Episodic Tasks:
$$G_t = \sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1} \tag{41}$$

Unified Notation:
$$G_t = \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k \tag{42}$$

6.4 Value Functions

State-Value Function:
$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] \tag{43}$$

Action-Value Function:
$$q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \tag{44}$$

Relationship:
$$v_\pi(s) = \sum_a \pi(a|s) q_\pi(s, a) \tag{45}$$

Inverse Relationship:
$$q_\pi(s, a) = \sum_{s', r} p(s', r|s, a)[r + \gamma v_\pi(s')] \tag{46}$$

6.5 Bellman Equations

For State-Value:
$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')] \tag{47}$$

For Action-Value:
$$q_\pi(s, a) = \sum_{s',r} p(s', r|s, a) \left[r + \gamma \sum_{a'} \pi(a'|s') q_\pi(s', a') \right] \tag{48}$$

6.6 Optimal Value Functions

Optimal State-Value:
$$v_*(s) = \max_\pi v_\pi(s) \tag{49}$$

Optimal Action-Value:
$$q_*(s, a) = \max_\pi q_\pi(s, a) \tag{50}$$

6.7 Bellman Optimality Equations

For State-Value:
$$v_*(s) = \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_*(s')] \tag{51}$$

For Action-Value:
$$q_*(s, a) = \sum_{s',r} p(s', r|s, a)[r + \gamma \max_{a'} q_*(s', a')] \tag{52}$$

6.8 Policies

Policy Definition:
$$\pi(a|s) = P(A_t = a|S_t = s) \tag{53}$$

Greedy Policy:
$$\pi(a|s) = \begin{cases} 1 & \text{if } a = \arg\max_{a'} Q(s, a') \\ 0 & \text{otherwise} \end{cases} \tag{54}$$

ϵ-Greedy Policy:
$$\pi(a|s) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{|\mathcal{A}(s)|} & \text{if } a = \arg\max_{a'} Q(s, a') \\ \frac{\epsilon}{|\mathcal{A}(s)|} & \text{otherwise} \end{cases} \tag{55}$$

Softmax Policy:
$$\pi(a|s) = \frac{e^{Q(s,a)/\tau}}{\sum_{a'} e^{Q(s,a')/\tau}} \tag{56}$$

Optimal Policy:
$$\pi_*(s) = \arg\max_a q_*(s, a) \tag{57}$$

6.9 Estimating Action Values

$$\text{Averaging Approach:} \qquad Q_t(s,a) = \frac{1}{n_t(s,a)} \sum_{i=1}^{t} r_i \cdot \mathbb{I}(s_i = s, a_i = a) \quad (58)$$

$$\text{Incremental Update:} \qquad Q_{t+1}(s_t,a_t) = Q_t(s_t,a_t) + \alpha[r_{t+1} - Q_t(s_t,a_t)] \quad (59)$$

3. Reinforcement Learning: Markov Decision Processes

Reinforcement Learning Notes

March 23, 2025

Contents

1 Introduction to Markov Decision Processes

1.1 Motivation and Context

Markov Decision Processes (MDPs) provide the formal mathematical framework for modeling reinforcement learning problems where an agent interacts with an environment through sequential decision-making. MDPs extend upon simpler paradigms like multi-armed bandits and contextual bandits by introducing:

- A fully specified notion of state
- State transitions influenced by agent actions
- Long-term consequences of decisions

In a multi-armed bandit problem, only the immediate reward is considered, with no concept of state. In contextual bandits, while rewards depend on states, the actions do not influence future states. MDPs provide the next evolutionary step, where an agent's actions both affect immediate rewards and influence the probability distribution of future states.

1.2 The Agent-Environment Interface

The standard agent-environment interface in reinforcement learning is formalized through MDPs:

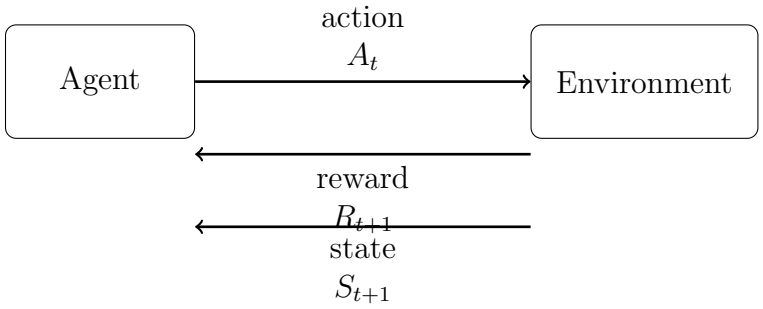


Figure 1: The agent-environment interaction in reinforcement learning (Sutton & Barto)

The interaction proceeds as follows:

- At each time step t, the agent observes the current state $S_t \in \mathcal{S}$

- Based on this state, the agent selects an action $A_t \in \mathcal{A}(S_t)$
- The environment responds with a reward $R_{t+1} \in \mathcal{R}$ and transitions to a new state S_{t+1}
- This creates a sequence of states, actions, and rewards: $S_0, A_0, R_1, S_1, A_1, R_2, S_2, \ldots$

The goal of the agent is to select actions that maximize the expected cumulative reward over time.

2 Components of MDPs

2.1 States, Actions, Rewards, and Transitions

A Markov Decision Process is defined by four essential components:

1. **States** ($\mathcal{S}$): The set of all possible states of the environment. A state $s \in \mathcal{S}$ provides all relevant information needed for decision-making. The state at time t is denoted S_t.
2. **Actions** ($\mathcal{A}$): The set of all possible actions the agent can take. For each state s, the agent can choose from a set of available actions $\mathcal{A}(s)$. The action at time t is denoted A_t.
3. **Transitions**: The state transition probability function $p(s'|s, a)$ defines the dynamics of the environment. It specifies the probability of transitioning to state s' given that the agent was in state s and took action a.
4. **Rewards** ($\mathcal{R}$): The reward function $r(s, a)$ defines the immediate reward received after taking action a in state s. The reward at time $t+1$ (after taking action A_t in state S_t) is denoted R_{t+1}.

2.2 The Markov Property

The defining characteristic of MDPs is the Markov property, which states that the future is independent of the past given the present. Formally, a state S_t is Markov if and only if:

$$P(S_{t+1} = s', R_{t+1} = r|S_t, A_t, R_t, S_{t-1}, A_{t-1}, \ldots, R_1, S_0, A_0) = P(S_{t+1} = s', R_{t+1} = r|S_t, A_t) \tag{1}$$

Notation Overview

- $P(S_{t+1} = s', R_{t+1} = r|S_t, A_t, \ldots)$ - The probability of transitioning to state s' and receiving reward r given the entire history
- $P(S_{t+1} = s', R_{t+1} = r|S_t, A_t)$ - The probability of transitioning to state s' and receiving reward r given only the current state and action
- S_t - The state at time step t
- A_t - The action taken at time step t
- R_t - The reward received at time step t

The Markov property implies that the current state captures all relevant information from the history of the process. This is a crucial assumption that simplifies the decision-making problem while still capturing the essential dynamics of many real-world sequential decision problems.

3 Formal MDP Definition

3.1 Mathematical Notation

A finite Markov Decision Process is formally defined as a 5-tuple $(\mathcal{S}, \mathcal{A}, p, r, \gamma)$ where:

- $\mathcal{S}$ is a finite set of states
- $\mathcal{A}$ is a finite set of actions
- $p : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \times \mathcal{R} \to [0, 1]$ is the state transition probability function
- $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the reward function
- $\gamma \in [0, 1]$ is the discount factor

The dynamics of the environment are fully specified by the joint probability distribution:

$$p(s', r|s, a) = P(S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a) \tag{2}$$

Notation Overview

- $p(s', r|s, a)$ - The probability of transitioning to state s' and receiving reward r, given that the agent was in state s and took action a
- $P(S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a)$ - The probability that the state at time t is s' and the reward is r, given that the state at time $t-1$ was s and the action taken was a
- S_t - The state at time step t
- R_t - The reward received at time step t
- A_t - The action taken at time step t

From this joint probability function, we can derive other important quantities:

3.2 State Transition Probabilities

The state transition probabilities are derived from the joint probability by summing over all possible rewards:

$$p(s'|s, a) = \sum_{r \in \mathcal{R}} p(s', r|s, a) \tag{3}$$

Notation Overview

- $p(s'|s, a)$ - The probability of transitioning to state s' given state s and action a
- $p(s', r|s, a)$ - The joint probability of transitioning to state s' and receiving reward r, given state s and action a
- $\mathcal{R}$ - The set of all possible rewards

3.3 Expected Reward Functions

There are several ways to define the expected reward, each useful in different contexts:

$$r(s, a) = \mathbb{E}[R_t|S_{t-1} = s, A_{t-1} = a] = \sum_{r \in \mathcal{R}} r \sum_{s' \in \mathcal{S}} p(s', r|s, a) \tag{4}$$

Notation Overview

- $r(s, a)$ - The expected reward when taking action a in state s
- $\mathbb{E}[\cdot]$ - The expected value operator
- R_t - The reward at time t
- S_{t-1} - The state at time $t - 1$
- A_{t-1} - The action taken at time $t - 1$
- $p(s', r|s, a)$ - The joint probability of transitioning to state s' and receiving reward r, given state s and action a
- $\mathcal{S}$ - The set of all possible states
- $\mathcal{R}$ - The set of all possible rewards

3.4 Example of Expected Reward Calculation

Let's illustrate the calculation of the expected reward with a simple example:
Consider a tiny MDP with:

- 2 states: s_1 and s_2
- 1 action: a
- 2 possible rewards: 0 and 1

The transition probabilities are defined as:

- $p(s_1, 0|s_1, a) = 0.3$ (30% chance of staying in s_1 with reward 0)
- $p(s_2, 0|s_1, a) = 0.2$ (20% chance of moving to s_2 with reward 0)
- $p(s_1, 1|s_1, a) = 0.1$ (10% chance of staying in s_1 with reward 1)
- $p(s_2, 1|s_1, a) = 0.4$ (40% chance of moving to s_2 with reward 1)

To calculate the expected reward $r(s_1, a)$, we apply the formula:

$$\begin{aligned}
r(s_1, a) &= \sum_{r \in \mathcal{R}} r \sum_{s' \in \mathcal{S}} p(s', r | s_1, a) \\
&= 0 \cdot \sum_{s' \in \mathcal{S}} p(s', 0 | s_1, a) + 1 \cdot \sum_{s' \in \mathcal{S}} p(s', 1 | s_1, a) \\
&= 0 \cdot (p(s_1, 0 | s_1, a) + p(s_2, 0 | s_1, a)) + 1 \cdot (p(s_1, 1 | s_1, a) + p(s_2, 1 | s_1, a)) \\
&= 0 \cdot (0.3 + 0.2) + 1 \cdot (0.1 + 0.4) \\
&= 0 \cdot 0.5 + 1 \cdot 0.5 \\
&= 0.5
\end{aligned}$$

This example shows how the inner sum (over s') collects probabilities for the same reward across different next states, and the outer sum (over r) adds up the weighted rewards.

We can also define the expected reward for state-action-next-state triples:

$$r(s, a, s') = \mathbb{E}[R_t | S_{t-1} = s, A_{t-1} = a, S_t = s'] = \sum_{r \in \mathcal{R}} r \frac{p(s', r | s, a)}{p(s' | s, a)} \tag{5}$$

Notation Overview

- $r(s, a, s')$ - The expected reward when transitioning from state s to state s' via action a
- $\mathbb{E}[\cdot]$ - The expected value operator
- R_t - The reward at time t
- S_{t-1} - The state at time $t - 1$
- A_{t-1} - The action taken at time $t - 1$
- S_t - The state at time t
- $p(s', r | s, a)$ - The joint probability of transitioning to state s' and receiving reward r, given state s and action a
- $p(s' | s, a)$ - The probability of transitioning to state s' given state s and action a
- $\mathcal{R}$ - The set of all possible rewards

3.5 MDP Constraints

For a valid MDP, the following constraints must be satisfied:

1. $\sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r|s, a) = 1$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$
2. $p(s', r|s, a) \geq 0$ for all $s, s' \in \mathcal{S}$, $r \in \mathcal{R}$, and $a \in \mathcal{A}(s)$

These constraints ensure that $p(s', r|s, a)$ is a valid probability distribution.

4 Policy Definition

4.1 Formalization of Policies

A policy is a mapping from states to probabilities of selecting each possible action. Formally, a policy π is defined as:

$$\pi(a|s) = P(A_t = a|S_t = s) \tag{6}$$

Notation Overview

- $\pi(a|s)$ - The probability of selecting action a in state s under policy π
- $P(A_t = a|S_t = s)$ - The probability that the agent selects action $A_t = a$ when in state $S_t = s$
- A_t - The action taken at time step t
- S_t - The state at time step t

4.2 Types of Policies

Policies can be categorized based on their stochasticity:

1. **Deterministic Policies**: For each state, exactly one action is selected with probability 1, and all other actions have probability 0. A deterministic policy can be written as $\pi(s) = a$, indicating that action a is always selected in state s.
2. **Stochastic Policies**: Actions are selected according to a probability distribution. Stochastic policies are useful for exploration and for solving problems with partial observability or non-deterministic dynamics.

4.3 Stationary vs. Non-Stationary Policies

Another categorization of policies is based on their time-dependence:

1. **Stationary Policies**: The policy does not change over time. The mapping from states to action probabilities remains constant regardless of the time step.

2. **Non-Stationary Policies**: The policy changes over time. The mapping from states to action probabilities depends on the time step.

In most reinforcement learning contexts, we focus on finding optimal stationary policies, as they are simpler and often sufficient for solving MDPs.

4.4 Examples of Policies

Several common policy types used in reinforcement learning include:

- **Greedy Policy**: Always selects the action with the highest estimated value:
$$\pi(a|s) = \begin{cases} 1 & \text{if } a = \arg\max_{a'} Q(s, a') \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

- **ϵ-Greedy Policy**: Selects the best action most of the time, but occasionally explores:
$$\pi(a|s) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{|\mathcal{A}(s)|} & \text{if } a = \arg\max_{a'} Q(s, a') \\ \frac{\epsilon}{|\mathcal{A}(s)|} & \text{otherwise} \end{cases} \tag{8}$$

- **Softmax Policy**: Selects actions with probabilities proportional to their estimated values:
$$\pi(a|s) = \frac{e^{Q(s,a)/\tau}}{\sum_{a'} e^{Q(s,a')/\tau}} \tag{9}$$

  where τ is a temperature parameter controlling exploration.

Notation Overview

- $\pi(a|s)$ - The probability of selecting action a in state s under policy π
- $Q(s, a)$ - The estimated value of taking action a in state s
- $\arg\max_{a'}$ - The action that maximizes the given function
- ϵ - A small probability for random exploration
- $|\mathcal{A}(s)|$ - The number of possible actions in state s
- τ - Temperature parameter in softmax, controlling exploration (higher values increase randomness)

5 Episodic vs. Continuing Tasks

5.1 Episodic Tasks

Episodic tasks are those with a clear endpoint or terminal state:

- Each episode starts from an initial state and ends when a terminal state is reached
- Examples include board games (chess, Go), where games have a definite beginning and end
- The interaction breaks naturally into subsequences called "episodes"
- Each episode ends in a special terminal state, often with different rewards for different outcomes

In episodic tasks, we denote the set of all non-terminal states as $\mathcal{S}$ and the set of all states, including the terminal state, as $\mathcal{S}^+$.

5.2 Continuing Tasks

Continuing tasks are those that go on indefinitely without a natural endpoint:

- The agent-environment interaction does not break into episodes and continues without limit

- Examples include ongoing process control, perpetual robot operation, or stock portfolio management
- There is no terminal state, so the agent must learn to perform well over an indefinite horizon

5.3 Unified Notation

To unify the notation for both episodic and continuing tasks, we can use a single formulation by considering episodic tasks as continuing tasks with a special absorbing terminal state that transitions only to itself with a reward of zero.

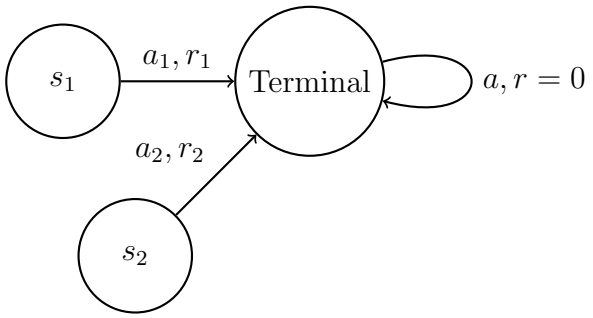


Figure 2: Representation of a terminal state in an episodic task

This unification allows us to treat both episodic and continuing tasks within the same mathematical framework.

6 Discounted Future Return

6.1 Definition of Return

In reinforcement learning, the agent aims to maximize not just immediate rewards but the cumulative reward over time. The return is the function of reward sequence that the agent seeks to maximize.

For a continuing task, the simplest formulation would be the sum of rewards:

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \dots \tag{10}$$

Notation Overview

- G_t - The return at time step t
- R_{t+k} - The reward received k steps after time t

However, this sum could be infinite for continuing tasks, making it unsuitable as an optimization criterion.

6.2 Discounting

To address this issue, we introduce discounting, which reduces the importance of future rewards:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{11}$$

Notation Overview

- G_t - The discounted return at time step t
- R_{t+k+1} - The reward received $k+1$ steps after time t
- γ - The discount factor, $0 \leq \gamma \leq 1$
- $\sum_{k=0}^{\infty}$ - The sum over all future time steps

The discount factor γ determines how much the agent values future rewards:

- γ close to 0: Myopic evaluation, immediate rewards are strongly preferred
- γ close to 1: Far-sighted evaluation, future rewards are valued almost as much as immediate ones

6.3 Reasons for Discounting

There are several reasons to use discounting:

1. **Mathematical Convenience**: Ensures the sum is finite for continuing tasks, provided $\gamma < 1$
2. **Uncertainty about the Future**: Future rewards are more uncertain, which may justify valuing them less

3. **Alignment with Human Preferences**: People often prefer immediate rewards to delayed ones

4. **Avoidance of Infinite Returns**: Without discounting, optimal policies might not exist for continuing tasks

6.4 Episodic Returns

For episodic tasks with a terminal time step T, the return can be defined as:

$$G_t = \sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1} \tag{12}$$

Notation Overview

- G_t - The return at time step t
- R_{t+k+1} - The reward received $k+1$ steps after time t
- γ - The discount factor, $0 \leq \gamma \leq 1$
- T - The final time step of the episode
- $T-t-1$ - The number of remaining steps in the episode after time t

6.5 Unified Return Formulation

To unify notation for both episodic and continuing tasks, we can use:

$$G_t = \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k \tag{13}$$

Notation Overview

- G_t - The return at time step t
- R_k - The reward received at time step k
- γ - The discount factor, $0 \leq \gamma \leq 1$
- T - The final time step (could be ∞ for continuing tasks)
- $k - t - 1$ - The number of time steps between receiving reward R_k and time t

This formulation handles both episodic tasks (where T is finite) and continuing tasks (where $T = \infty$). Note that for continuing tasks, $\gamma < 1$ is required to ensure the sum is finite.

6.6 Recursive Relationship for Return

The return has a useful recursive relationship:

$$G_t = R_{t+1} + \gamma G_{t+1} \tag{14}$$

Notation Overview

- G_t - The return at time step t
- R_{t+1} - The immediate reward received after time t
- γ - The discount factor, $0 \leq \gamma \leq 1$
- G_{t+1} - The return at time step $t + 1$

This recursive relationship is fundamental to many reinforcement learning methods, particularly those based on bootstrapping.

7 Value Functions

7.1 State-Value Function

The state-value function $v_\pi(s)$ represents how good it is for an agent to be in a particular state s when following policy π. Formally, it is defined as the expected return starting from state s and following policy π thereafter:

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] \tag{15}$$

Notation Overview

- $v_\pi(s)$ - The state-value function for policy π
- $\mathbb{E}_\pi[\cdot]$ - The expected value when following policy π
- G_t - The return at time step t
- S_t - The state at time step t

We can expand this definition using the recursive relationship for the return:

$$v_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|S_t = s] \tag{16}$$

Notation Overview

- $v_\pi(s)$ - The state-value function for policy π
- $\mathbb{E}_\pi[\cdot]$ - The expected value when following policy π
- R_{t+1} - The immediate reward
- γ - The discount factor, $0 \leq \gamma \leq 1$
- G_{t+1} - The return at time step $t + 1$
- S_t - The state at time step t

7.2 Action-Value Function

The action-value function $q_\pi(s, a)$ represents how good it is to take a specific action a in state s when following policy π thereafter. Formally:

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \tag{17}$$

Notation Overview

- $q_\pi(s, a)$ - The action-value function for policy π
- $\mathbb{E}_\pi[\cdot]$ - The expected value when following policy π
- G_t - The return at time step t
- S_t - The state at time step t
- A_t - The action taken at time step t

7.3 Relationship Between Value Functions

The state-value and action-value functions are related by:

$$v_\pi(s) = \sum_a \pi(a|s) q_\pi(s, a) \tag{18}$$

Notation Overview

- $v_\pi(s)$ - The state-value function for policy π
- $\pi(a|s)$ - The probability of taking action a in state s under policy π
- $q_\pi(s, a)$ - The action-value function for policy π
- $\sum_a$ - The sum over all possible actions in state s

This equation shows that the value of a state equals the expected value of the actions that might be taken in that state, weighted by their probability under policy π.

Conversely, the action-value function can be expressed in terms of the state-value function:

$$q_\pi(s, a) = \sum_{s', r} p(s', r|s, a)[r + \gamma v_\pi(s')] \tag{19}$$

Notation Overview

- $q_\pi(s, a)$ - The action-value function for policy π
- $p(s', r|s, a)$ - The probability of transitioning to state s' and receiving reward r, given state s and action a
- r - The immediate reward
- γ - The discount factor, $0 \leq \gamma \leq 1$
- $v_\pi(s')$ - The state-value function for policy π at the next state s'
- $\sum_{s',r}$ - The sum over all possible next states and rewards

This equation shows that the value of taking action a in state s equals the expected immediate reward plus the discounted value of the next state.

7.4 The Bellman Equation for State-Value Function

The Bellman equation expresses the recursive relationship in value functions. For the state-value function:

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')] \tag{20}$$

Notation Overview

- $v_\pi(s)$ - The state-value function for policy π
- $\pi(a|s)$ - The probability of taking action a in state s under policy π
- $p(s', r|s, a)$ - The probability of transitioning to state s' and receiving reward r, given state s and action a
- r - The immediate reward
- γ - The discount factor, $0 \leq \gamma \leq 1$
- $v_\pi(s')$ - The state-value function for policy π at the next state s'
- $\sum_a$ - The sum over all possible actions in state s
- $\sum_{s',r}$ - The sum over all possible next states and rewards

The Bellman equation states that the value of a state equals the expected return for taking an action according to policy π, which consists of the immediate reward plus the discounted value of the next state.

7.5 The Bellman Equation for Action-Value Function

Similarly, the Bellman equation for the action-value function is:

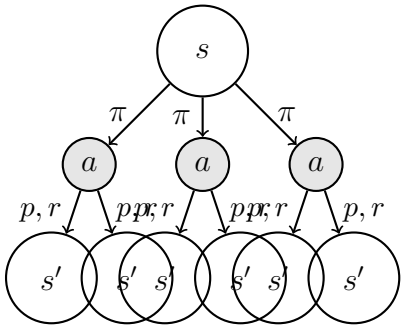
$$q_\pi(s,a) = \sum_{s',r} p(s',r|s,a) \left[r + \gamma \sum_{a'} \pi(a'|s') q_\pi(s',a') \right] \tag{21}$$

Notation Overview

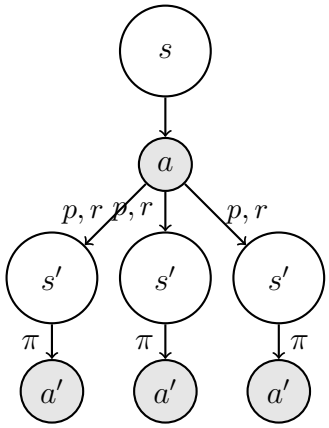
- $q_\pi(s,a)$ - The action-value function for policy π
- $p(s',r|s,a)$ - The probability of transitioning to state s' and receiving reward r, given state s and action a
- r - The immediate reward
- γ - The discount factor, $0 \leq \gamma \leq 1$
- $\pi(a'|s')$ - The probability of taking action a' in the next state s' under policy π
- $q_\pi(s',a')$ - The action-value function for policy π at the next state-action pair (s',a')
- $\sum_{s',r}$ - The sum over all possible next states and rewards
- $\sum_{a'}$ - The sum over all possible actions in the next state s'

7.6 Backup Diagrams for Bellman Equations

Backup diagrams provide a graphical representation of the Bellman equations, illustrating how values are propagated from future states back to the current state.



(a) State-value backup



(b) Action-value backup

Figure 3: Backup diagrams for (a) the state-value function v_π and (b) the action-value function q_π

In these diagrams:

- Solid circles represent states
- Open circles represent actions
- The root node at the top represents the state or state-action pair being evaluated
- The branches show possible trajectories
- Nodes are traversed from top to bottom when following trajectories
- Values are backed up from bottom to top when computing the value function

8 Optimal Policies and Value Functions

8.1 Definition of Optimal Policies

A policy π is defined as better than or equal to a policy π' if its expected return is greater than or equal to that of π' for all states:

$$\pi \geq \pi' \iff v_\pi(s) \geq v_{\pi'}(s) \text{ for all } s \in \mathcal{S} \tag{22}$$

Notation Overview

- $\pi \geq \pi'$ - Policy π is better than or equal to policy π'
- $v_\pi(s)$ - The state-value function for policy π
- $v_{\pi'}(s)$ - The state-value function for policy π'
- $\mathcal{S}$ - The set of all possible states

There is always at least one policy that is better than or equal to all other policies, called an optimal policy, denoted π_*. All optimal policies share the same optimal state-value function, v_*:

$$v_*(s) = \max_\pi v_\pi(s) \text{ for all } s \in \mathcal{S} \tag{23}$$

Notation Overview

- $v_*(s)$ - The optimal state-value function
- $\max_\pi$ - The maximum over all possible policies
- $v_\pi(s)$ - The state-value function for policy π
- $\mathcal{S}$ - The set of all possible states

They also share the same optimal action-value function, q_*:

$$q_*(s, a) = \max_\pi q_\pi(s, a) \text{ for all } s \in \mathcal{S}, a \in \mathcal{A}(s) \tag{24}$$

Notation Overview

- $q_*(s, a)$ - The optimal action-value function
- $\max_\pi$ - The maximum over all possible policies
- $q_\pi(s, a)$ - The action-value function for policy π
- $\mathcal{S}$ - The set of all possible states
- $\mathcal{A}(s)$ - The set of actions available in state s

8.2 The Bellman Optimality Equations

The Bellman optimality equations characterize the optimal value functions:

$$v_*(s) = \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_*(s')] \tag{25}$$

Notation Overview

- $v_*(s)$ - The optimal state-value function
- $\max_a$ - The maximum over all actions $a \in \mathcal{A}(s)$
- $p(s', r|s, a)$ - The probability of transitioning to state s' and receiving reward r, given state s and action a
- r - The immediate reward
- γ - The discount factor, $0 \leq \gamma \leq 1$
- $v_*(s')$ - The optimal state-value function at the next state s'
- $\sum_{s',r}$ - The sum over all possible next states and rewards

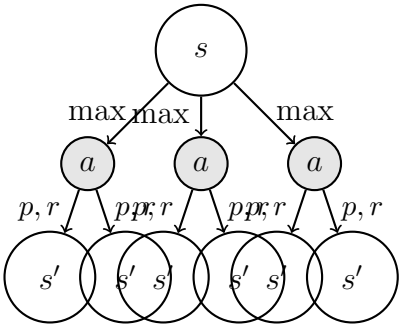
And for the action-value function:

$$q_*(s, a) = \sum_{s',r} p(s', r|s, a)[r + \gamma \max_{a'} q_*(s', a')] \tag{26}$$

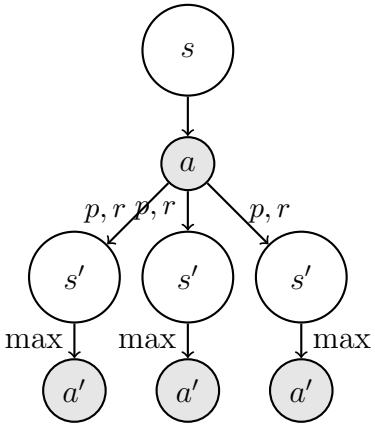
Notation Overview

- $q_*(s, a)$ - The optimal action-value function
- $p(s', r|s, a)$ - The probability of transitioning to state s' and receiving reward r, given state s and action a
- r - The immediate reward
- γ - The discount factor, $0 \leq \gamma \leq 1$
- $\max_{a'}$ - The maximum over all actions $a' \in \mathcal{A}(s')$
- $q_*(s', a')$ - The optimal action-value function at the next state-action pair (s', a')
- $\sum_{s',r}$ - The sum over all possible next states and rewards

8.3 Backup Diagrams for Bellman Optimality Equations



(a) Optimal state-value backup



(b) Optimal action-value backup

Figure 4: Backup diagrams for (a) the optimal state-value function v_* and (b) the optimal action-value function q_*

The key difference between these backup diagrams and those for v_π and q_π is the replacement of the policy distribution π with the max operator, indicating that the agent selects the action that maximizes value.

8.4 Deriving Optimal Policies

Given the optimal value functions, it is straightforward to determine an optimal policy:

$$\pi_*(s) = \arg\max_a q_*(s, a) \tag{27}$$

Notation Overview

- $\pi_*(s)$ - The optimal policy
- $\arg\max_a$ - The action that maximizes the given function
- $q_*(s, a)$ - The optimal action-value function

Alternatively, using the state-value function:

$$\pi_*(s) = \arg\max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_*(s')] \tag{28}$$

Notation Overview

- $\pi_*(s)$ - The optimal policy
- $\arg\max_a$ - The action that maximizes the given function
- $p(s', r|s, a)$ - The probability of transitioning to state s' and receiving reward r, given state s and action a
- r - The immediate reward
- γ - The discount factor, $0 \leq \gamma \leq 1$
- $v_*(s')$ - The optimal state-value function at the next state s'
- $\sum_{s',r}$ - The sum over all possible next states and rewards

Any policy that is greedy with respect to the optimal value function(s) is an optimal policy. There may be multiple optimal policies that share the same optimal value function.

9 Solving MDPs

9.1 Methods for Finding Optimal Policies

Several methods can be used to solve MDPs:

1. **Dynamic Programming (DP)**: When the MDP's dynamics (transition probabilities and reward function) are known, DP methods like policy iteration and value iteration can be used to compute optimal policies.
2. **Monte Carlo (MC) Methods**: When the dynamics are unknown, MC methods use sample episodes to estimate values and improve the policy.
3. **Temporal Difference (TD) Learning**: Combines ideas from DP and MC, updating value estimates based on other learned estimates without waiting for the end of an episode.

4. **Model-Free Methods**: Methods like Q-learning and SARSA that learn directly from interaction with the environment without explicitly modeling its dynamics.

5. **Model-Based Methods**: Methods that learn a model of the environment's dynamics and then use that model for planning.

9.2 The Generalized Policy Iteration Framework

Many reinforcement learning methods follow a generalized policy iteration (GPI) framework, which alternates between policy evaluation and policy improvement:

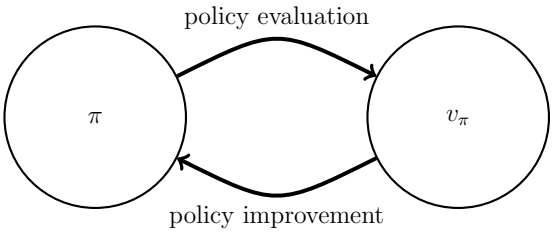


Figure 5: The generalized policy iteration framework

- **Policy Evaluation**: Compute the value function v_π for the current policy π
- **Policy Improvement**: Improve the policy by making it greedy with respect to the current value function
- This process continues until convergence to the optimal policy and value function

Different methods vary in how they implement these two steps and how they interleave them.

10 Example Applications of MDPs

10.1 Grid World Example

A classic example of an MDP is the grid world problem:

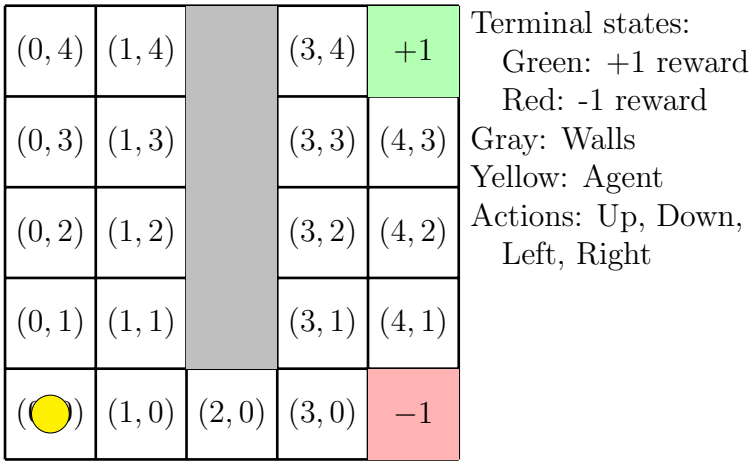


Figure 6: Example grid world MDP

In this environment:

- States are the grid positions (x, y)
- Actions are movements in four directions: up, down, left, right
- Transitions are deterministic (moving in a direction always results in the corresponding adjacent cell, unless blocked by a wall or grid boundary)
- Rewards are -1 for each step, except when reaching terminal states
- Terminal states have rewards of +1 (green) and -1 (red)
- The goal is to find the policy that maximizes the expected return (i.e., reach the green state with the minimum number of steps)

10.2 Robot Navigation

MDPs can model robot navigation problems:

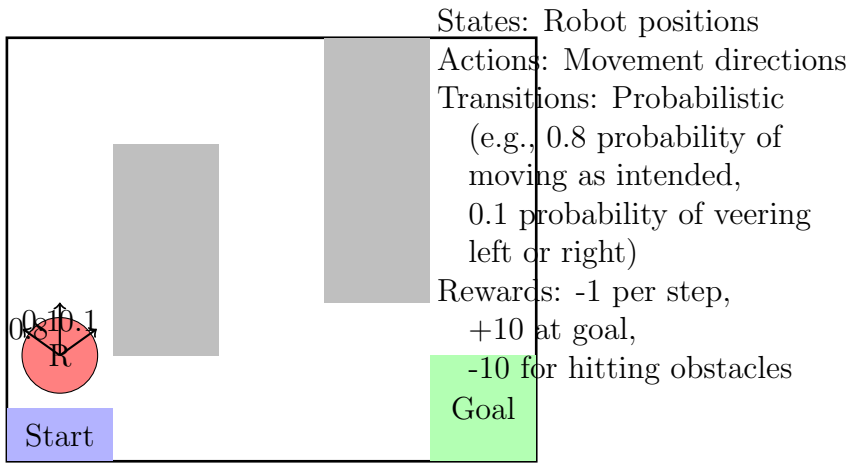


Figure 7: Robot navigation as an MDP

10.3 Resource Management

MDPs can also model resource allocation problems:

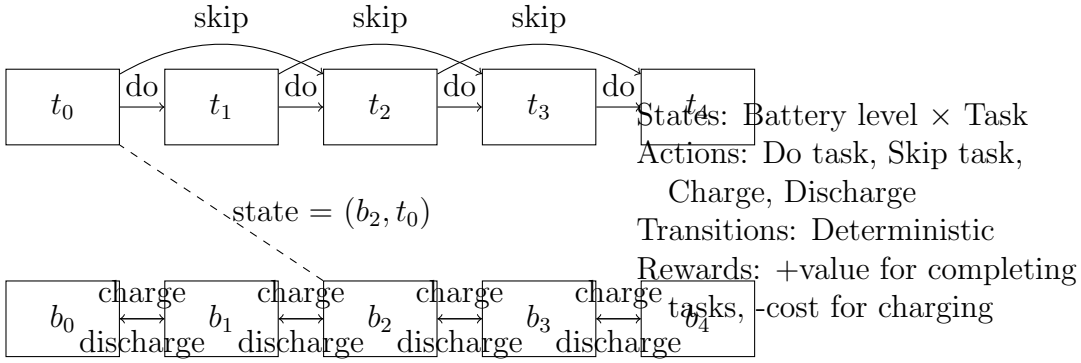


Figure 8: Resource management as an MDP

11 Conclusion

Markov Decision Processes provide the formal mathematical framework for modeling sequential decision-making problems. They extend simpler frameworks like multi-armed bandits by introducing state transitions that are influenced by the agent's actions, allowing for the modeling of long-term consequences.

Key concepts covered in this chapter include:

- The four essential components of an MDP: states, actions, transitions, and rewards
- The formal MDP definition using mathematical notation and constraints
- Policies as mappings from states to action probabilities
- Episodic versus continuing tasks and the unification of their notation
- The discounting of future rewards and its mathematical and practical justifications
- Value functions and their recursive relationships expressed in the Bellman equations
- Optimal policies and value functions characterized by the Bellman optimality equations
- Various methods for solving MDPs to find optimal policies

Understanding MDPs is crucial for reinforcement learning, as they provide the theoretical foundation upon which algorithms are built. The concepts of value functions, policies, and the Bellman equations are fundamental to virtually all reinforcement learning methods, from classic techniques like dynamic programming to modern approaches based on deep neural networks.

4. Reinforcement Learning Notes: Policy Iteration

Rahul Sawhney

March 23, 2025

Contents

1 Introduction to Policy Iteration

Policy iteration is a method for finding an optimal policy in a Markov Decision Process (MDP). It works by alternating between two main steps:

1. **Policy Evaluation:** Computing the state-value function v_π for a given policy π.
2. **Policy Improvement:** Improving the policy by making it greedy with respect to the current value function.

This process continues until the policy converges to an optimal policy, denoted as π^*. Policy iteration is at the core of many reinforcement learning methods as it provides a fundamental approach to solving MDPs when the model (transition probabilities and rewards) is known.

2 Value Functions

2.1 State-Value Function

The state-value function $v_\pi(s)$ represents how good it is for an agent to be in a particular state s when following a policy π. Formally, it is defined as the expected return (expected cumulative discounted reward) when starting in state s and following policy π thereafter:

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t | S_t = s] \tag{1}$$

Notation Overview

- $v_\pi(s)$: State-value function for state s under policy π
- $\mathbb{E}_\pi$: Expected value when following policy π
- G_t: Return (cumulative discounted reward) starting from time step t
- S_t: State at time step t
- $\doteq$: Defined as (definitional equality)

Utilizing the recursive property of the return, we can also express this as:

$$v_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} | S_t = s] \tag{2}$$

Notation Overview

- $v_\pi(s)$: State-value function for state s under policy π
- $\mathbb{E}_\pi$: Expected value when following policy π
- R_{t+1}: Reward received after taking action in state S_t
- γ: Discount factor, $0 \leq \gamma \leq 1$, determines the present value of future rewards
- G_{t+1}: Return starting from time step $t+1$
- S_t: State at time step t

2.2 Action-Value Function

The action-value function $q_\pi(s, a)$ represents how good it is for an agent to take a specific action a in state s and then follow policy π thereafter. Formally, it is defined as:

$$q_\pi(s, a) \doteq \mathbb{E}_\pi[G_t | S_t = s, A_t = a] \tag{3}$$

Notation Overview

- $q_\pi(s, a)$: Action-value function for taking action a in state s under policy π
- $\mathbb{E}_\pi$: Expected value when following policy π
- G_t: Return (cumulative discounted reward) starting from time step t
- S_t: State at time step t
- A_t: Action taken at time step t

2.3 Bellman Equations for Value Functions

The Bellman equations provide a recursive definition of value functions, establishing a fundamental relationship between the value of a state and the values of its successor states.

For the state-value function, the Bellman equation is:

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')] \tag{4}$$

Notation Overview

- $v_\pi(s)$: State-value function for state s under policy π
- $\pi(a|s)$: Probability of taking action a in state s under policy π
- $p(s', r|s, a)$: Probability of transitioning to state s' and receiving reward r when taking action a in state s
- γ: Discount factor
- s': Next state
- r: Reward

For the action-value function, the Bellman equation is:

$$q_\pi(s, a) = \sum_{s', r} p(s', r|s, a)[r + \gamma \sum_{a'} \pi(a'|s') q_\pi(s', a')] \tag{5}$$

Notation Overview

- $q_\pi(s, a)$: Action-value function for taking action a in state s under policy π
- $p(s', r|s, a)$: Probability of transitioning to state s' and receiving reward r when taking action a in state s
- $\pi(a'|s')$: Probability of taking action a' in state s' under policy π
- γ: Discount factor
- s': Next state
- r: Reward
- a': Next action

2.4 Relationship Between Value Functions

The state-value function and action-value function are closely related. The relationship can be expressed as:

$$v_\pi(s) = \sum_a \pi(a|s) q_\pi(s, a) \tag{6}$$

Notation Overview

- $v_\pi(s)$: State-value function for state s under policy π
- $q_\pi(s, a)$: Action-value function for taking action a in state s under policy π
- $\pi(a|s)$: Probability of taking action a in state s under policy π

And conversely:

$$q_\pi(s, a) = \sum_{s', r} p(s', r|s, a)[r + \gamma v_\pi(s')] \tag{7}$$

Notation Overview

- $q_\pi(s, a)$: Action-value function for taking action a in state s under policy π
- $p(s', r|s, a)$: Probability of transitioning to state s' and receiving reward r when taking action a in state s
- γ: Discount factor
- $v_\pi(s')$: State-value function for state s' under policy π
- s': Next state
- r: Reward

3 Optimal Value Functions

3.1 Definitions and Properties

An optimal policy is a policy that achieves the maximum possible expected return from all states. The optimal state-value function, denoted as $v_*(s)$, gives the maximum value achievable for each state:

$$v_*(s) \doteq \max_\pi v_\pi(s) \tag{8}$$

Notation Overview

- $v_*(s)$: Optimal state-value function for state s
- $\max_\pi$: Maximum over all possible policies π
- $v_\pi(s)$: State-value function for state s under policy π

Similarly, the optimal action-value function, denoted as $q_*(s, a)$, gives the maximum value achievable for each state-action pair:

$$q_*(s, a) \doteq \max_\pi q_\pi(s, a) \tag{9}$$

Notation Overview

- $q_*(s, a)$: Optimal action-value function for taking action a in state s
- $\max_\pi$: Maximum over all possible policies π
- $q_\pi(s, a)$: Action-value function for taking action a in state s under policy π

3.2 Bellman Optimality Equations

The Bellman optimality equation for the state-value function is:

$$v_*(s) = \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_*(s')] \tag{10}$$

Notation Overview

- $v_*(s)$: Optimal state-value function for state s
- $\max_a$: Maximum over all possible actions a
- $p(s', r|s, a)$: Probability of transitioning to state s' and receiving reward r when taking action a in state s
- γ: Discount factor
- s': Next state
- r: Reward

The Bellman optimality equation for the action-value function is:

$$q_*(s, a) = \sum_{s',r} p(s', r|s, a)[r + \gamma \max_{a'} q_*(s', a')] \tag{11}$$

Notation Overview

- $q_*(s, a)$: Optimal action-value function for taking action a in state s
- $p(s', r|s, a)$: Probability of transitioning to state s' and receiving reward r when taking action a in state s
- $\max_{a'}$: Maximum over all possible next actions a'
- γ: Discount factor
- s': Next state
- r: Reward
- a': Next action

3.3 Relationship Between Optimal Value Functions

The relationship between the optimal state-value function and the optimal action-value function can be expressed as:

$$v_*(s) = \max_a q_*(s, a) \tag{12}$$

Notation Overview

- $v_*(s)$: Optimal state-value function for state s
- $q_*(s, a)$: Optimal action-value function for taking action a in state s
- $\max_a$: Maximum over all possible actions a

This shows that the optimal value of a state is the maximum of the optimal action-values for that state, which corresponds to taking the best possible action in that state.

4 Policy Evaluation

Policy evaluation is the process of computing the state-value function v_π for a given policy π. This is a crucial step in policy iteration.

4.1 Iterative Policy Evaluation

The Bellman equation for the state-value function provides a system of linear equations with one equation for each state. For a finite MDP, this system can be

solved directly. However, iterative methods are often more practical, especially for large state spaces.

The iterative policy evaluation algorithm updates the value function as follows:

$$v_{k+1}(s) = \sum_{a} \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_k(s')] \tag{13}$$

Notation Overview

- $v_{k+1}(s)$: Estimated state-value of state s after $k+1$ iterations
- $v_k(s)$: Estimated state-value of state s after k iterations
- $\pi(a|s)$: Probability of taking action a in state s under policy π
- $p(s', r|s, a)$: Probability of transitioning to state s' and receiving reward r when taking action a in state s
- γ: Discount factor
- s': Next state
- r: Reward

This is a form of dynamic programming update, which converges to the true value function v_π as the number of iterations approaches infinity.

4.2 Iterative Policy Evaluation Algorithm

Algorithm 1 Iterative Policy Evaluation (for estimating $V \approx v_\pi$)

Input: A policy π to be evaluated
Parameter: A small threshold $\theta > 0$ determining the accuracy of estimation
Initialize $V(s) = 0$, for all $s \in \mathcal{S}^+$ (except for terminal states where $V(terminal) = 0$)
repeat
    $\Delta \leftarrow 0$
    for each $s \in \mathcal{S}$ **do**
        $v \leftarrow V(s)$
        $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$
        $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
    end for
until $\Delta < \theta$
return $V \approx v_\pi$

Notation Overview

- $V(s)$: Estimated state-value function
- $\pi(a|s)$: Probability of taking action a in state s under policy π
- $p(s', r|s, a)$: Probability of transitioning to state s' and receiving reward r when taking action a in state s
- γ: Discount factor
- $\mathcal{S}$: Set of all non-terminal states
- $\mathcal{S}^+$: Set of all states, including terminal states
- Δ: Maximum change in value function
- θ: Threshold for convergence

5 Policy Improvement

Policy improvement is the process of making a policy better by making it greedy with respect to the current value function.

5.1 Policy Improvement Theorem

The policy improvement theorem states that if we have two policies π and π' such that, for all states $s \in \mathcal{S}$:

$$q_\pi(s, \pi'(s)) \geq v_\pi(s) \tag{14}$$

Notation Overview

- $q_\pi(s, \pi'(s))$: Action-value of taking action prescribed by policy π' in state s, then following policy π
- $v_\pi(s)$: State-value of state s under policy π
- $\pi'(s)$: Action prescribed by policy π' in state s
- $\mathcal{S}$: Set of all non-terminal states

Then policy π' is at least as good as π, meaning that $v_{\pi'}(s) \geq v_\pi(s)$ for all $s \in \mathcal{S}$. Furthermore, if there exists at least one state where $q_\pi(s, \pi'(s)) > v_\pi(s)$, then π' is strictly better than π in at least one state.

The proof of this theorem is based on the recursive nature of value functions. If we start from any state s where $q_\pi(s, \pi'(s)) > v_\pi(s)$, we can show that following π' will lead to higher expected returns than following π.

5.2 Greedy Policy Improvement

A natural way to improve a policy is to make it greedy with respect to the current value function. For any deterministic policy π, the improved policy π' is:

$$\pi'(s) = \arg\max_a q_\pi(s,a) = \arg\max_a \sum_{s',r} p(s',r|s,a)[r + \gamma v_\pi(s')] \tag{15}$$

Notation Overview

- $\pi'(s)$: Action prescribed by the improved policy in state s
- $\arg\max_a$: The action a that maximizes the following expression
- $q_\pi(s,a)$: Action-value of taking action a in state s, then following policy π
- $p(s',r|s,a)$: Probability of transitioning to state s' and receiving reward r when taking action a in state s
- γ: Discount factor
- $v_\pi(s')$: State-value of state s' under policy π

The policy improvement theorem guarantees that this new policy π' will be strictly better than π unless π is already optimal.

6 Policy Iteration Algorithm

Policy iteration combines policy evaluation and policy improvement in an iterative process, which converges to an optimal policy.

Algorithm 2 Policy Iteration (for estimating $\pi \approx \pi_*$)

Initialize: $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$
repeat
    // Policy Evaluation
    repeat
        $\Delta \leftarrow 0$
        for each $s \in \mathcal{S}$ **do**
            $v \leftarrow V(s)$
            $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$
            $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
        end for
    until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)
    // Policy Improvement
    policy-stable $\leftarrow$ true
    for each $s \in \mathcal{S}$ **do**
        old-action $\leftarrow \pi(s)$
        $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$
        if old-action $\neq \pi(s)$ **then**
            policy-stable $\leftarrow$ false
        end if
    end for
until policy-stable
return $V \approx v_*$ and $\pi \approx \pi_*$

Notation Overview

- $V(s)$: Estimated state-value function
- $\pi(s)$: Current policy (deterministic)
- $\mathcal{S}$: Set of all non-terminal states
- $\mathcal{A}(s)$: Set of actions available in state s
- $p(s', r|s, a)$: Probability of transitioning to state s' and receiving reward r when taking action a in state s
- γ: Discount factor
- θ: Threshold for convergence in policy evaluation
- Δ: Maximum change in value function
- policy-stable: Boolean indicating whether the policy has changed
- old-action: Action prescribed by the old policy
- $\arg\max_a$: The action a that maximizes the following expression

6.1 Convergence of Policy Iteration

Policy iteration is guaranteed to converge to the optimal policy π_* and the optimal value function v_* in a finite number of iterations for any finite MDP. This is because:

1. The set of policies for a finite MDP is finite.
2. Each policy improvement step produces a strictly better policy unless the current policy is already optimal.
3. Value functions are bounded for finite MDPs with a discount factor $\gamma < 1$.

Therefore, policy iteration must converge to an optimal policy in a finite number of iterations.

7 Value Iteration

Value iteration is a variant of policy iteration that combines the policy evaluation and policy improvement steps into a single update. Instead of waiting for policy evaluation to converge completely before policy improvement, value iteration performs a single sweep of policy evaluation followed immediately by policy improvement.

The update rule for value iteration is:

$$v_{k+1}(s) = \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_k(s')] \tag{16}$$

Notation Overview

- $v_{k+1}(s)$: Estimated state-value of state s after $k+1$ iterations
- $v_k(s)$: Estimated state-value of state s after k iterations
- $\max_a$: Maximum over all possible actions a
- $p(s', r|s, a)$: Probability of transitioning to state s' and receiving reward r when taking action a in state s
- γ: Discount factor
- s': Next state
- r: Reward

Algorithm 3 Value Iteration (for estimating $\pi \approx \pi_*$)

Initialize: $V(s) = 0$, for all $s \in \mathcal{S}^+$
repeat
    $\Delta \leftarrow 0$
    for each $s \in \mathcal{S}$ **do**
        $v \leftarrow V(s)$
        $V(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$
        $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
    end for
until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)
// Output a deterministic policy
for each $s \in \mathcal{S}$ **do**
    $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$
end for
return $\pi \approx \pi_*$

Notation Overview

- $V(s)$: Estimated state-value function
- $\mathcal{S}$: Set of all non-terminal states
- $\mathcal{S}^+$: Set of all states, including terminal states
- $\max_a$: Maximum over all possible actions a
- $\arg\max_a$: The action a that maximizes the following expression
- $p(s', r|s, a)$: Probability of transitioning to state s' and receiving reward r when taking action a in state s
- γ: Discount factor
- θ: Threshold for convergence
- Δ: Maximum change in value function
- $\pi(s)$: Deterministic optimal policy

8 Example: Grid World

Let's consider a simple grid world example to illustrate policy iteration. The grid world is a 4x4 grid with states labeled from (0,0) to (3,3). The agent can move in four directions: up, down, left, and right. If the agent attempts to move off the grid, it remains in the same state. The agent receives a reward of -1 for each move, except when it reaches the goal state (3,3), where it receives a reward of +10 and the episode terminates.

			Goal
(0,3)	(1,3)	(2,3)	(3,3)
(0,2)	(1,2)	(2,2)	(3,2)
(0,1)	(1,1)	(2,1)	(3,1)
(0,0)	(1,0)	(2,0)	(3,0)

Figure 1: 4x4 Grid World

8.1 Initial Policy and Value Function

We start with an arbitrary policy, such as moving right in all states, and initialize the value function to zero for all states (except terminal states).

8.2 Policy Evaluation

Using the policy evaluation algorithm, we iteratively update the value function for the current policy. For example, for state (0,0), following the "move right" policy:

$$\begin{aligned} v_\pi(0,0) &= \sum_a \pi(a|(0,0)) \sum_{s',r} p(s',r|(0,0),a)[r + \gamma v_\pi(s')] \\ &= 1 \cdot \sum_{s',r} p(s',r|(0,0),\text{right})[r + \gamma v_\pi(s')] \\ &= 1 \cdot [(-1) + 0.9 \cdot v_\pi(1,0)] \end{aligned}$$

Assuming $\gamma = 0.9$, and continuing this process for all states, we eventually converge to the value function for the initial policy.

8.3 Policy Improvement

Once we have the value function for the current policy, we improve the policy by making it greedy with respect to the value function:

$$\pi'(s) = \arg\max_a \sum_{s',r} p(s',r|s,a)[r + \gamma v_\pi(s')]$$

For example, for state (0,0), we would consider all four actions (up, down, left, right) and choose the one that gives the highest expected return. After improving the policy for all states, we check if the policy has changed. If it has, we go back to the policy evaluation step with the new policy.

8.4 Convergence

We repeat the policy evaluation and policy improvement steps until the policy no longer changes. At this point, we have found an optimal policy for the grid world, which tells the agent the best action to take in each state to maximize the expected return.

9 Generalized Policy Iteration

Generalized Policy Iteration (GPI) is a term that refers to the general idea of letting policy evaluation and policy improvement processes interact, regardless

of how they are implemented. Most reinforcement learning methods can be viewed as approximations to GPI.

In GPI, the value function is repeatedly pushed toward the value function for the current policy (policy evaluation), while the policy is repeatedly improved with respect to the current value function (policy improvement). These two processes work together, both pushing toward the optimal value function and optimal policy.

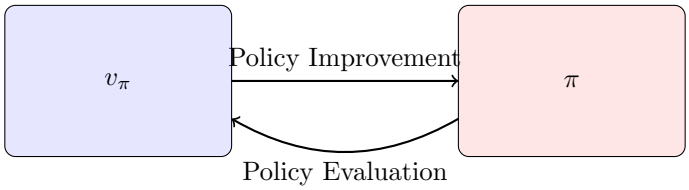


Figure 2: Generalized Policy Iteration

10 Efficiency of Policy Iteration vs Value Iteration

Both policy iteration and value iteration are guaranteed to converge to an optimal policy, but they have different computational characteristics:

- **Policy Iteration**: Requires fewer iterations but each iteration is more computationally expensive due to the complete policy evaluation step.
- **Value Iteration**: Requires more iterations but each iteration is less computationally expensive due to the single backup operation.

In practice, a modified version of policy iteration is often used, where the policy evaluation step is truncated after a fixed number of iterations or when a certain convergence criterion is met. This approach, known as truncated policy iteration, combines the advantages of both policy iteration and value iteration.

11 Theoretical Guarantees

Policy iteration is guaranteed to converge to an optimal policy and value function in a finite number of iterations for any finite MDP. This is because:

1. The policy improvement step always yields a strictly better policy unless the current policy is already optimal.
2. There are only a finite number of deterministic policies for a finite MDP.

Similarly, value iteration is also guaranteed to converge to the optimal value function for any finite MDP, as long as the discount factor $\gamma < 1$ (for continuing tasks) or all episodes terminate (for episodic tasks).

12 Conclusion

Policy iteration is a fundamental algorithm in reinforcement learning for finding optimal policies in MDPs. It consists of two main steps: policy evaluation, which computes the value function for a given policy, and policy improvement, which makes the policy greedy with respect to the computed value function. The algorithm alternates between these two steps until convergence.

Value iteration is a variant of policy iteration that combines the policy evaluation and policy improvement steps into a single update, often resulting in faster convergence for large state spaces.

Both policy iteration and value iteration are instances of the more general concept of Generalized Policy Iteration (GPI), which forms the foundation of many reinforcement learning algorithms.

When the MDP model (transition probabilities and rewards) is known, policy iteration and value iteration provide powerful tools for solving reinforcement learning problems. However, in many practical scenarios, the model is not known, leading to the development of model-free reinforcement learning methods that estimate value functions directly from experience.

5. Reinforcement Learning: Other Solution Methods

Based on Sutton and Barto's Reinforcement Learning Book

Contents

1 Introduction to Other Solution Methods

In previous sections, we explored policy iteration and value iteration methods for solving reinforcement learning problems, which require a complete model of the environment. However, in many practical scenarios, we either don't have a complete model or it's computationally expensive to use one. In this section, we explore alternative approaches to solving reinforcement learning problems that either relax the requirement for a complete model or approach the problem from different angles.

These alternative methods include:

- Understanding the limitations of on-policy methods
- Monte Carlo techniques for estimating value functions
- Temporal Difference Learning, which combines ideas from Monte Carlo methods and dynamic programming
- Q-Learning as an off-policy temporal difference method
- Policy Gradient methods that directly optimize policy parameters

These alternative approaches are crucial for extending reinforcement learning to a wide variety of practical applications where complete models are unavailable or impractical.

2 Limitations of On-Policy Methods

On-policy methods evaluate and improve the same policy that is used to make decisions. While conceptually straightforward, this approach has several limitations:

2.1 Definition and Characteristics

On-policy methods learn the value functions and improve the policy based on actions taken according to the current policy. The key characteristic is that the policy being evaluated and the policy generating behavior are the same.

2.2 Limitations

2.2.1 Exploration-Exploitation Tradeoff

One fundamental limitation is the exploration-exploitation tradeoff. To find the optimal policy, the agent needs to explore various states and actions. However, to maximize rewards, it should exploit its current knowledge by choosing the actions it believes are best. On-policy methods must balance these competing objectives using the same policy.

For example, in ϵ-greedy policies, the agent follows the greedy policy with probability $1 - \epsilon$ and takes a random action with probability ϵ. While this introduces exploration, it also means the policy being learned is not truly optimal but is constrained by the exploration requirements.

2.2.2 Sample Inefficiency

On-policy methods can be sample inefficient because they can only learn from data collected under the current policy. When the policy changes, previously collected experience becomes less relevant or even irrelevant for evaluating the new policy. This limitation is particularly significant in environments where data collection is expensive or time-consuming.

2.2.3 Necessity for Incremental Policy Improvement

On-policy methods typically require incremental policy improvements to ensure stability. Large policy changes can lead to drastic changes in the data distribution, making it difficult to accurately estimate value functions.

2.3 On-Policy vs. Off-Policy Learning

To address these limitations, we can use off-policy methods, which allow the agent to learn from data that was collected using a different policy (behavior policy) than the one being evaluated and improved (target policy). This separation offers more flexibility but introduces its own challenges.

3 Monte Carlo Techniques

Monte Carlo (MC) methods learn directly from complete episodes of experience without requiring a model of the environment's dynamics. They are based on averaging complete returns from episodes of experience.

3.1 Properties of Monte Carlo Methods

- Only applicable to episodic tasks (tasks with a definite ending point)
- Learn from complete episodes: no bootstrapping
- Can be off-policy, learning about a target policy while following a behavior policy
- Do not require knowledge about environment dynamics
- Simple conceptually and can handle environments with unknown dynamics

3.2 Monte Carlo Prediction (Evaluation)

Monte Carlo prediction estimates the state-value function v_π for a given policy π by averaging returns observed after visits to a state.

Algorithm 1 First-visit Monte Carlo Prediction

Input: policy π to be evaluated
Initialize $V(s)$ for all $s \in \mathcal{S}$
Initialize $Returns(s)$ as an empty list for all $s \in \mathcal{S}$
for each episode **do**
    Generate an episode following π: $S_0, A_0, R_1, S_1, A_1, \ldots, S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    for $t = T-1, T-2, \ldots, 0$ **do**
        $G \leftarrow \gamma G + R_{t+1}$
        if S_t appears for the first time in the episode **then**
            Append G to $Returns(S_t)$
            $V(S_t) \leftarrow average(Returns(S_t))$
        end if
    end for
end for

Notation Overview

- π - policy to be evaluated
- $V(s)$ - estimated value function for state s
- $Returns(s)$ - list of returns observed after visiting state s
- S_t - state at time t
- A_t - action at time t
- R_t - reward at time t
- G - return (cumulative discounted reward)
- γ - discount factor
- T - final time step of the episode

The first-visit MC method estimates $v_\pi(s)$ as the average of returns following the first visits to state s, while the every-visit MC method averages returns following all visits to s.

3.3 Monte Carlo Control

Monte Carlo control methods find the optimal policy by alternating between policy evaluation and policy improvement, similar to policy iteration but using Monte Carlo methods for the evaluation step.

Algorithm 2 Monte Carlo Control with Exploring Starts

Initialize $Q(s, a)$ arbitrarily for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
Initialize $\pi(s)$ arbitrarily for all $s \in \mathcal{S}$
Initialize $Returns(s, a)$ as empty lists for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
for each episode **do**
    Choose $S_0 \in \mathcal{S}$ and $A_0 \in \mathcal{A}(S_0)$ such that all pairs have probability > 0
    Generate an episode from S_0, A_0 following π: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    for $t = T-1, T-2, \ldots, 0$ **do**
        $G \leftarrow \gamma G + R_{t+1}$
        if (S_t, A_t) pair appears for the first time in the episode **then**
            Append G to $Returns(S_t, A_t)$
            $Q(S_t, A_t) \leftarrow average(Returns(S_t, A_t))$
            $\pi(S_t) \leftarrow \arg\max_a Q(S_t, a)$
        end if
    end for
end for

Notation Overview

- $Q(s, a)$ - action-value function for state s and action a
- $\pi(s)$ - policy that maps states to actions
- $Returns(s, a)$ - list of returns observed after taking action a in state s
- S_t - state at time t
- A_t - action at time t
- R_t - reward at time t
- G - return (cumulative discounted reward)
- γ - discount factor
- T - final time step of the episode
- $\mathcal{S}$ - set of all states
- $\mathcal{A}(s)$ - set of all actions available in state s

3.4 Off-Policy Monte Carlo Methods

Off-policy Monte Carlo methods allow learning about one policy (the target policy) while following another policy (the behavior policy). This approach addresses the exploration-exploitation tradeoff by using a separate exploratory policy for data collection.

Algorithm 3 Off-policy Monte Carlo Prediction

Input: target policy π to be evaluated
Initialize $V(s)$ for all $s \in \mathcal{S}$
Initialize $C(s)$ to 0 for all $s \in \mathcal{S}$
for each episode **do**
    Generate an episode using behavior policy b: $S_0, A_0, R_1, S_1, A_1, \ldots, S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    $W \leftarrow 1$
    for $t = T-1, T-2, \ldots, 0$ **do**
        $G \leftarrow \gamma G + R_{t+1}$
        $C(S_t) \leftarrow C(S_t) + W$
        $V(S_t) \leftarrow V(S_t) + \frac{W}{C(S_t)}[G - V(S_t)]$
        $W \leftarrow W \frac{\pi(A_t|S_t)}{b(A_t|S_t)}$
        if $W = 0$ **then**
            break
        end if
    end for
end for

Notation Overview

- π - target policy to be evaluated
- b - behavior policy used to generate episodes
- $V(s)$ - estimated value function for state s
- $C(s)$ - cumulative weight for state s
- W - importance sampling weight
- S_t - state at time t
- A_t - action at time t
- R_t - reward at time t
- G - return (cumulative discounted reward)
- γ - discount factor
- T - final time step of the episode
- $\pi(A_t|S_t)$ - probability of taking action A_t in state S_t under policy π
- $b(A_t|S_t)$ - probability of taking action A_t in state S_t under policy b

The term $\frac{\pi(A_t|S_t)}{b(A_t|S_t)}$ is the importance sampling ratio, which corrects for the difference in action selection probabilities between the target and behavior policies.

4 Temporal Difference Learning

Temporal Difference (TD) learning combines ideas from Monte Carlo methods and dynamic programming. Like Monte Carlo methods, TD methods can learn directly from experience without a model of the environment. Like DP, TD methods update estimates based in part on other learned estimates, without waiting for a final outcome (they bootstrap).

4.1 TD Prediction

The simplest TD method, known as TD(0), updates the value estimate for a state based on the observed reward and the estimated value of the next state:

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

Notation Overview

- $V(S_t)$ - estimated value of state S_t
- R_{t+1} - reward received after taking action in state S_t
- S_{t+1} - next state after taking action in state S_t
- α - step-size parameter (learning rate)
- γ - discount factor
- $R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$ - TD error, often denoted as δ_t

The term $R_{t+1} + \gamma V(S_{t+1})$ is called the TD target, and $R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$ is called the TD error.

Algorithm 4 Tabular TD(0) for estimating v_π

Input: policy π to be evaluated
Parameters: step size $\alpha \in (0, 1]$
Initialize $V(s)$ arbitrarily for all $s \in \mathcal{S}$
for each episode **do**
    Initialize S
    for each step of episode **do**
        Take action A according to $\pi(\Delta|S)$
        Observe reward R and next state S'
        $V(S) \leftarrow V(S) + \alpha[R + \gamma V(S') - V(S)]$
        $S \leftarrow S'$
    end for
end for

4.2 Advantages of TD Learning

TD learning has several advantages over both Monte Carlo methods and dynamic programming:

- TD methods don't require a model of the environment (unlike DP)
- TD methods can learn online, after each step, without waiting for the end of an episode (unlike MC)
- TD methods can learn from incomplete episodes, making them applicable to continuing (non-terminating) tasks
- TD methods are generally more sample efficient than Monte Carlo methods

4.3 SARSA: On-Policy TD Control

SARSA (State-Action-Reward-State-Action) is an on-policy TD control method that learns action-value functions. The update rule for SARSA is:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)] \tag{2}$$

Notation Overview

- $Q(S_t, A_t)$ - estimated value of taking action A_t in state S_t
- R_{t+1} - reward received after taking action A_t in state S_t
- S_{t+1} - next state after taking action A_t in state S_t
- A_{t+1} - next action taken in state S_{t+1}
- α - step-size parameter (learning rate)
- γ - discount factor

Algorithm 5 SARSA (on-policy TD control) for estimating $Q \approx q_*$

Parameters: step size $\alpha \in (0, 1]$, small $\epsilon > 0$
Initialize $Q(s, a)$ arbitrarily for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$
for each episode **do**
    Initialize S
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    for each step of episode **do**
        Take action A, observe R, S'
        Choose A' from S' using policy derived from Q (e.g., ϵ-greedy)
        $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma Q(S', A') - Q(S, A)]$
        $S \leftarrow S'$
        $A \leftarrow A'$
    end for
end for

5 Q-Learning

Q-Learning is an off-policy TD control method. It directly approximates the optimal action-value function, q_*, independent of the policy being followed.

5.1 Q-Learning Algorithm

The update rule for Q-Learning is:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \tag{3}$$

Notation Overview

- $Q(S_t, A_t)$ - estimated value of taking action A_t in state S_t
- R_{t+1} - reward received after taking action A_t in state S_t
- S_{t+1} - next state after taking action A_t in state S_t
- $\max_a Q(S_{t+1}, a)$ - maximum estimated value obtainable from state S_{t+1}
- α - step-size parameter (learning rate)
- γ - discount factor

Algorithm 6 Q-Learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Parameters: step size $\alpha \in (0, 1]$, small $\epsilon > 0$
Initialize $Q(s, a)$ arbitrarily for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$
for each episode **do**
    Initialize S
    for each step of episode **do**
        Choose A from S using policy derived from Q (e.g., ϵ-greedy)
        Take action A, observe R, S'
        $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$
        $S \leftarrow S'$
    end for
end for

5.2 Differences Between SARSA and Q-Learning

The key difference between SARSA and Q-Learning lies in their update targets:

- SARSA uses the Q-value of the next state-action pair actually taken: $Q(S_{t+1}, A_{t+1})$, making it on-policy.
- Q-Learning uses the maximum Q-value for the next state: $\max_a Q(S_{t+1}, a)$, regardless of what action is actually taken next, making it off-policy.

This means Q-Learning can learn the optimal policy even while following an exploratory policy, while SARSA learns the optimal policy subject to the constraint of the exploration strategy being used.

5.3 Expected SARSA

Expected SARSA is a variant of SARSA that uses the expected value of the next state-action pair instead of the sample:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \sum_a \pi(a|S_{t+1})Q(S_{t+1}, a) - Q(S_t, A_t)] \tag{4}$$

Notation Overview

- $Q(S_t, A_t)$ - estimated value of taking action A_t in state S_t
- R_{t+1} - reward received after taking action A_t in state S_t
- S_{t+1} - next state after taking action A_t in state S_t
- $\pi(a|S_{t+1})$ - probability of taking action a in state S_{t+1} under policy π
- $\sum_a \pi(a|S_{t+1})Q(S_{t+1}, a)$ - expected value of the next state-action pair
- α - step-size parameter (learning rate)
- γ - discount factor

Expected SARSA can be more computationally expensive than SARSA but can reduce variance in the updates and lead to better performance in some environments.

6 Policy Gradient Methods

Policy gradient methods take a different approach by directly parameterizing the policy and updating the parameters to maximize the expected return.

6.1 Policy Parameterization

The policy is represented as a parameterized function:

$$\pi(a|s,\theta) = P(A_t = a|S_t = s, \theta_t = \theta) \tag{5}$$

where θ is a vector of policy parameters that we adjust to optimize performance.

Notation Overview

- $\pi(a|s,\theta)$ - probability of taking action a in state s under the policy parameterized by θ
- θ - vector of policy parameters
- A_t - action at time t
- S_t - state at time t
- θ_t - policy parameters at time t

6.2 Performance Measure

The performance measure to be maximized is the expected return:

$$J(\theta) = \mathbb{E}_{\pi_\theta}[G_0] \tag{6}$$

where G_0 is the return starting from the initial state.

Notation Overview

- $J(\theta)$ - expected return under the policy parameterized by θ
- $\mathbb{E}_{\pi_\theta}$ - expectation over trajectories generated by following policy π_θ
- G_0 - return (cumulative discounted reward) starting from the initial state
- π_θ - policy parameterized by θ

6.3 Policy Gradient Theorem

The policy gradient theorem provides a way to compute the gradient of $J(\theta)$ without requiring differentiation of the state distribution:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi(A_t|S_t,\theta) \cdot G_t] \tag{7}$$

Notation Overview

- $\nabla_\theta J(\theta)$ - gradient of the expected return with respect to policy parameters
- $\mathbb{E}_{\pi_\theta}$ - expectation over trajectories generated by following policy π_θ
- $\nabla_\theta \log \pi(A_t|S_t, \theta)$ - gradient of the log probability of taking action A_t in state S_t
- G_t - return (cumulative discounted reward) from time step t
- π_θ - policy parameterized by θ

6.4 REINFORCE Algorithm

REINFORCE is a Monte Carlo policy gradient method that implements the policy gradient theorem:

Algorithm 7 REINFORCE (Monte Carlo Policy Gradient)

Parameters: step size $\alpha > 0$
Initialize policy parameters θ arbitrarily
for each episode **do**
    Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$ following $\pi(\Delta|\Delta, \theta)$
    for $t = 0, 1, \ldots, T-1$ **do**
        $G \leftarrow$ return from step t
        $\theta \leftarrow \theta + \alpha \gamma^t G \nabla_\theta \log \pi(A_t|S_t, \theta)$
    end for
end for

Notation Overview

- θ - policy parameters
- α - step-size parameter (learning rate)
- γ - discount factor
- G - return (cumulative discounted reward)
- $\nabla_\theta \log \pi(A_t|S_t, \theta)$ - gradient of the log probability of taking action A_t in state S_t
- S_t - state at time t
- A_t - action at time t
- R_t - reward at time t
- T - final time step of the episode

6.5 Advantages of Policy Gradient Methods

Policy gradient methods have several advantages:

- They can learn stochastic policies, which is important in partially observable environments
- They naturally handle continuous action spaces

- They can incorporate prior knowledge by initializing the policy to specific behaviors
- Convergence properties are often better than value-based methods
- Function approximation is more stable with policy gradients than with value-based methods

7 Comparison of Methods

Method	**Model Required**	**On/Off-Policy**	**Bootstrapping**	**Updates**
Dynamic Programming	Yes	On-policy	Yes	Sweep
Monte Carlo	No	Both	No	Episode
TD Learning	No	Both	Yes	Step
Q-Learning	No	Off-policy	Yes	Step
SARSA	No	On-policy	Yes	Step
Policy Gradient	No	Typically on-policy	Varies	Episode/Step

Table 1: Comparison of Reinforcement Learning Methods

8 Conclusion

We have explored various alternative solution methods for reinforcement learning problems beyond the classical dynamic programming approaches. Each method has its strengths and weaknesses, making them suitable for different types of problems and environments.

Monte Carlo methods are conceptually simple and learn directly from complete episodes, but they can only be applied to episodic tasks and may require more samples. Temporal Difference methods combine ideas from Monte Carlo and dynamic programming, allowing learning from incomplete sequences and bootstrapping from existing estimates. Q-Learning provides an off-policy approach that can learn the optimal policy while following an exploratory policy. Finally, policy gradient methods offer a direct approach to optimizing policies, which is particularly useful for continuous action spaces and stochastic policies.

The choice of method depends on the specific problem characteristics, such as whether the environment is episodic or continuing, whether a model is available, and whether the state and action spaces are discrete or continuous.

6. Deep Reinforcement Learning

Reinforcement Learning Notes

Contents

1 Deep Reinforcement Learning

1.1 Introduction

Reinforcement learning has shown significant success in various domains; however, traditional RL methods face challenges when dealing with high-dimensional state spaces. Deep Reinforcement Learning (DRL) addresses this limitation by combining deep neural networks with reinforcement learning techniques, allowing agents to learn directly from raw sensory inputs.

The key advantage of deep reinforcement learning is its ability to automatically extract relevant features from raw input data, eliminating the need for manual feature engineering. This makes DRL particularly useful for tasks where the state space is large and complex, such as computer vision, natural language processing, and game playing.

1.2 Deep Q Learning

1.2.1 Neural Networks for Atari Games

Deep Q-Learning was first successfully applied to Atari games by Mnih et al. (2013/2015) at DeepMind. The key innovation was using convolutional neural networks to process raw pixel inputs from Atari games and learn control policies directly from this high-dimensional sensory input.

Input Processing The input to the neural network consists of the current and three previous game frames, stacked together to provide temporal information. This allows the network to infer motion and dynamics from the static frames.

Network Architecture The Deep Q-Network (DQN) uses:

- Convolutional layers to process the visual input
- Fully-connected layers for decision making
- Output layer with one node for each possible action, representing the estimated Q-value

Mathematical Formulation The neural network approximates the action-value function $Q(s, a)$ using a parameterized function $Q(s, a; \theta)$, where θ represents the network weights.

$$Q(s, a; \theta) \approx Q^*(s, a) \tag{1}$$

Notation Overview

- $Q(s, a; \theta)$: Approximated action-value function with parameters θ
- $Q^*(s, a)$: Optimal action-value function
- s: State (preprocessed game frames)
- a: Action (game controls)
- θ: Neural network parameters

1.2.2 Target Networks

A significant challenge in using neural networks for Q-learning is stability. The Q-learning update rule relies on the current estimate of the action-value function, which can lead to oscillations or divergence when function approximation is used.

The Problem The standard Q-learning update rule is:

$$\theta_{t+1} = \theta_t + \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a; \theta_t) - Q(s_t, a_t; \theta_t) \right] \nabla_{\theta_t} Q(s_t, a_t; \theta_t) \tag{2}$$

Notation Overview

- θ_t: Network parameters at time t
- α: Learning rate
- r_{t+1}: Reward received after taking action a_t in state s_t
- γ: Discount factor
- $\max_a Q(s_{t+1}, a; \theta_t)$: Maximum Q-value for the next state
- $Q(s_t, a_t; \theta_t)$: Current Q-value estimate
- $\nabla_{\theta_t} Q(s_t, a_t; \theta_t)$: Gradient of the Q-value with respect to the parameters

The problem is that the target $r_{t+1} + \gamma \max_a Q(s_{t+1}, a; \theta_t)$ depends on the same parameters θ_t that are being updated, creating a moving target.

The Solution: Target Network To stabilize learning, DQN uses a separate target network with parameters θ^- that are periodically updated to match the online network parameters θ:

$$\theta_{t+1} = \theta_t + \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a; \theta_t^-) - Q(s_t, a_t; \theta_t) \right] \nabla_{\theta_t} Q(s_t, a_t; \theta_t) \tag{3}$$

Notation Overview

- θ_t^-: Target network parameters at time t
- $Q(s_{t+1}, a; \theta_t^-)$: Q-value estimated by the target network

The target network parameters θ^- are updated every C steps by copying the online network parameters θ, effectively fixing the target for C updates:

$$\theta_t^- = \theta_{t-C} \tag{4}$$

This approach reduces the correlation between the target and the current estimate, leading to more stable learning.

1.2.3 Experience Replay

Another challenge in deep reinforcement learning is the correlation between consecutive samples, which can lead to inefficient learning and instability.

The Problem In standard online reinforcement learning, experiences $(s_t, a_t, r_{t+1}, s_{t+1})$ are used immediately and then discarded. This leads to:

- Strong correlations between consecutive samples
- Inefficient use of data (experiences are used only once)
- Forgetting of rare but potentially important experiences

The Solution: Experience Replay Experience replay addresses these issues by storing experiences in a replay memory and sampling from this memory for training:

Algorithm 1 Deep Q-Learning with Experience Replay

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
for episode = 1, M **do**
    Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
    for $t = 1, T$ **do**
        With probability ϵ select a random action a_t
        otherwise select $a_t = \arg\max_a Q(\phi(s_t), a; \theta)$
        Execute action a_t in emulator and observe reward r_t and image x_{t+1}
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
        Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
        Perform gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to θ
        Every C steps reset $\hat{Q} = Q$ (i.e., set $\theta^- = \theta$)
    end for
end for

Notation Overview

- D: Replay memory with capacity N
- ϕ: Preprocessing function (e.g., frame stacking, normalization)
- ϵ: Exploration rate for the ϵ-greedy policy
- y_j: Target value for the Q-function
- $\hat{Q}$: Target network for computing target values

Benefits of Experience Replay

- Breaks correlations between consecutive samples by random sampling
- More efficient use of data by reusing experiences
- Smooths the training distribution over many past behaviors
- Increases stability by reducing variance in updates

1.3 AlphaGo

AlphaGo, developed by Silver et al. at DeepMind, was the first computer program to defeat a world champion in the game of Go. It combines deep neural networks with Monte Carlo tree search (MCTS) to achieve superhuman performance.

1.3.1 Challenges in Go

Go presents several unique challenges that make it significantly harder for computers than games like chess:

High Branching Factor The game of Go has approximately 250 legal moves per position, compared to about 35 for chess. This creates a much larger search space.

Game Length Go games typically involve around 150 moves, further expanding the total number of possible game states.

Evaluation Difficulty Unlike chess, evaluating Go positions is extremely challenging. As Müller (2002) noted, "No simple yet reasonable evaluation function will ever be found for Go."

Mathematical Complexity The game tree complexity of Go is estimated to be around 10^{170}, making exhaustive search completely infeasible.

1.3.2 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a key component of AlphaGo, used to explore the game tree efficiently.

Basic MCTS Algorithm The MCTS algorithm consists of four main steps:

1. **Selection:** Starting from the root node, traverse the tree using a tree policy until reaching a leaf node.
2. **Expansion:** Add one or more child nodes to the current leaf.
3. **Simulation:** From the new node, simulate a complete game using a rollout policy.
4. **Backpropagation:** Update the value estimates of all traversed nodes based on the simulation outcome.

UCT Selection Formula The Upper Confidence Bound applied to Trees (UCT) formula guides the selection phase:

$$a_t = \arg\max_a \left(Q(s_t, a) + c \cdot \frac{\sqrt{\ln N(s_t)}}{N(s_t, a)} \right) \tag{5}$$

Notation Overview

- a_t: Action selected at time t
- $Q(s_t, a)$: Estimated value of taking action a in state s_t
- c: Exploration constant
- $N(s_t)$: Number of times state s_t has been visited
- $N(s_t, a)$: Number of times action a has been taken in state s_t

1.3.3 Deep Neural Networks for Go

AlphaGo uses multiple deep neural networks to guide the MCTS process and evaluate positions.

Neural Network Types AlphaGo uses three different networks:

- **Policy Network:** Suggests promising moves for tree expansion
- **Value Network:** Evaluates board positions without simulation
- **Rollout Policy Network:** Guides fast rollout simulations

Policy Network The policy network $p_\sigma(a|s)$ is trained to predict expert human moves:

$$p_\sigma(a|s) \approx p(a|s) \tag{6}$$

Where $p(a|s)$ is the probability of an expert human player choosing action a in state s.

Notation Overview

- $p_\sigma(a|s)$: Policy network with parameters σ
- $p(a|s)$: Expert human policy
- s: Board position
- a: Move

The policy network is trained in two phases:

1. **Supervised Learning:** Training to predict expert moves from a database of human games
2. **Reinforcement Learning:** Further training by self-play and policy gradient methods

Value Network The value network $v_\theta(s)$ approximates the expected outcome from position s:

$$v_\theta(s) \approx v^\pi(s) = \mathbb{E}[z_t|s_t = s, a_{t...T} \sim \pi] \tag{7}$$

Notation Overview

- $v_\theta(s)$: Value network with parameters θ
- $v^\pi(s)$: True value function under policy π
- z_t: Final game outcome from time t (win=+1, loss=-1)
- π: Policy used for self-play games

The value network is trained on positions from self-play games using the reinforcement-learned policy network, with the game outcomes as targets.

Rollout Policy Network The rollout policy is a simpler, faster network used for MCTS simulations:

$$p_\psi(a|s) \approx p(a|s) \tag{8}$$

Notation Overview

- $p_\psi(a|s)$: Rollout policy network with parameters ψ
- ψ: Parameters of a linear network (much simpler than σ)

The rollout policy is designed for speed (about 1000 times faster than the policy network) at the cost of some accuracy.

1.3.4 AlphaGo's MCTS Algorithm

AlphaGo combines MCTS with the neural networks to select moves:

Algorithm 2 AlphaGo's MCTS Algorithm

function SEARCHTREE(position s, simulations n)
  for $i = 1$ to n **do**
    $s' \leftarrow s$ ▷ Copy the root position
    $path \leftarrow \emptyset$ ▷ Empty array of (state, action) pairs
    while s' has been visited before and is not terminal **do**
      $a \leftarrow \arg\max_a \left(Q(s', a) + u(s', a)\right)$ ▷ Select with exploration bonus
      Append (s', a) to $path$
      $s' \leftarrow$ next state after action a
    end while
    if s' is terminal **then**
      $v \leftarrow$ outcome of s' for the player to move in the root position
    else if s' has not been visited before **then**
      $P(s', \cdot) \leftarrow p_\sigma(\cdot|s')$ ▷ Use policy network for probabilities
      $v \leftarrow v_\theta(s')$ ▷ Use value network for evaluation
    else
      $a \sim p_\psi(\cdot|s')$ ▷ Sample from rollout policy
      Simulate a rollout from s' until terminal state s_T
      $v \leftarrow$ outcome of s_T for the player to move in the root position
    end if
    for each (s, a) in $path$ **do**
      $N(s, a) \leftarrow N(s, a) + 1$ ▷ Increment visit count
      $Q(s, a) \leftarrow Q(s, a) + \frac{v - Q(s,a)}{N(s,a)}$ ▷ Update value estimate
    end for
  end for
  return $\arg\max_a N(s, a)$ ▷ Return most visited action
end function

Notation Overview

- $u(s, a)$: Exploration bonus based on $P(s, a)$ and visit counts
- $P(s, a)$: Prior probability of selecting action a in state s from policy network
- $N(s, a)$: Visit count for state-action pair
- $Q(s, a)$: Action value estimate from search

1.4 AlphaGo Zero

AlphaGo Zero represents a significant advancement over the original AlphaGo by learning entirely from self-play, without using any human gameplay data.

1.4.1 Learning Without Human Knowledge

AlphaGo Zero starts from random play and learns entirely through self-play reinforcement learning. This approach offers several advantages:

- Avoids human biases and limitations
- Discovers novel strategies beyond human knowledge
- Simplifies the training pipeline
- Demonstrates the possibility of tabula rasa learning in complex domains

1.4.2 Network Architecture

AlphaGo Zero uses a single neural network instead of the separate policy and value networks in the original AlphaGo:

$$(p(a|s), v(s)) = f_\theta(s) \tag{9}$$

Notation Overview

- f_θ: Combined policy-value network with parameters θ
- $p(a|s)$: Policy output (probability distribution over moves)
- $v(s)$: Value output (predicted game outcome)
- s: Board position

This single network outputs both a move probability distribution and a position evaluation, with shared parameters between the policy and value components.

1.4.3 Self-Play Reinforcement Learning

The training process for AlphaGo Zero consists of three main components that operate in a continuous cycle:

Neural Network Training The network parameters θ are trained to minimize the loss:

$$l = (z - v)^2 - \pi^T \log p + c\|\theta\|^2 \tag{10}$$

Notation Overview

- z: Game outcome (win=+1, loss=-1)
- v: Predicted value by the network
- π: Improved policy from MCTS
- p: Policy output from the network
- c: Regularization parameter
- $\|\theta\|^2$: L2 regularization term

Self-Play with MCTS Games are generated by self-play, with each move selected by MCTS guided by the current neural network:

$$\pi(a|s) = \frac{N(s,a)^{1/\tau}}{\sum_b N(s,b)^{1/\tau}} \tag{11}$$

Notation Overview

- $\pi(a|s)$: Search policy (MCTS output)
- $N(s,a)$: Visit count for action a in state s
- τ: Temperature parameter controlling exploration

Data Generation Each state s_t encountered during self-play is stored along with:

- The MCTS policy π_t
- The game outcome z_t

These data points (s_t, π_t, z_t) are used to train the neural network, completing the reinforcement learning loop.

1.4.4 AlphaGo Zero Algorithm

Algorithm 3 AlphaGo Zero Training Algorithm

Initialize neural network parameters θ randomly
Initialize replay buffer D with capacity N
for iteration $= 1, 2, ...$ **do**
  for episode $= 1, 2, ..., E$ **do**
    $s_1 \leftarrow$ initial board position
    for $t = 1$ until terminal state **do**
      $\pi_t \leftarrow \text{MCTS}(s_t, \theta)$ ▷ Run MCTS guided by neural network
      Sample action $a_t \sim \pi_t$ ▷ Select move based on MCTS policy
      $s_{t+1} \leftarrow$ next state after a_t
    end for
    $z \leftarrow$ final outcome of the game
    for each step t in the episode **do**
      Store (s_t, π_t, z_t) in D where $z_t = \pm z$ depending on player
    end for
  end for
  Sample mini-batch of (s, π, z) from D
  Update θ by gradient descent on loss $(z - v_\theta(s))^2 - \pi^T \log p_\theta(s) + c\|\theta\|^2$
end for

1.4.5 Comparison with Original AlphaGo

Feature	AlphaGo	AlphaGo Zero
Training data	Human expert games + self-play	Only self-play
Neural networks	Separate policy, value, and rollout networks	Single policy-value network
Input features	Handcrafted features + raw board position	Only raw board position
MCTS rollouts	Uses fast rollout policy	No rollouts, relies on value network
Training process	Multiple stages (SL then RL)	End-to-end reinforcement learning
Performance	Defeated world champion	Stronger than all previous versions

2 Summary

Deep Reinforcement Learning represents a powerful combination of deep learning and reinforcement learning techniques. The key innovations discussed in this chapter include:

- **Deep Q-Learning:** Combines Q-learning with deep neural networks, stabilized by target networks and experience replay to learn directly from high-dimensional sensory inputs.
- **AlphaGo:** A breakthrough system that combines deep neural networks with Monte Carlo Tree Search to master the game of Go, using separate policy and value networks trained on expert human games and self-play.
- **AlphaGo Zero:** A more advanced version that learns tabula rasa through pure self-play reinforcement learning, using a unified policy-value network and achieving superhuman performance without human knowledge.

These methods demonstrate the power of deep neural networks as function approximators in reinforcement learning, enabling solutions to previously intractable problems with high-dimensional state spaces. The success of these approaches has led to further advancements in robotics, game playing, and other domains requiring sequential decision making from complex sensory inputs.

3 Further Reading

- Mnih, V., Kavukcuoglu, K., Silver, D., et al. (2015). Human-level control through deep reinforcement learning. Nature, 518(7540), 529-533.
- Silver, D., Huang, A., Maddison, C. J., et al. (2016). Mastering the game of Go with deep neural networks and tree search. Nature, 529(7587), 484-489.
- Silver, D., Schrittwieser, J., Simonyan, K., et al. (2017). Mastering the game of Go without human knowledge. Nature, 550(7676), 354-359.
- Silver, D., Hubert, T., Schrittwieser, J., et al. (2017). Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. arXiv preprint arXiv:1712.01815.