Revision: Monte Carlo and Temporal Difference

Minor in AI - IIT ROPAR

9th May, 2025

What is Monte Carlo in Reinforcement Learning?

In reinforcement learning, an agent interacts with an environment to maximize cumulative reward. In some cases, the environment's dynamics (i.e., state transition probabilities and reward functions) are fully known, allowing for planning via methods like Dynamic Programming. However, in many real-world scenarios, such models are unavailable. This setting calls for model-free reinforcement learning, where agents learn directly from experience.

Monte Carlo (MC) methods are a fundamental class of model-free approaches. The core idea is to learn from complete episodes. An episode is a sequence of states, actions, and rewards, starting from an initial state and ending in a terminal state. After each episode, the agent calculates the *return*—the total cumulative reward from the episode—and uses this to update its knowledge.

Rather than modeling the environment's dynamics step-by-step, Monte Carlo methods wait until the episode ends and then use the overall experience to make updates. Over many episodes, the agent estimates value functions by averaging observed returns, allowing it to learn which states (or state-action pairs) tend to yield higher cumulative rewards.

Why Monte Carlo Methods? Advantages

Monte Carlo methods are especially valuable in settings where the environment is unknown or hard to model. Their key advantages include:

- **Simplicity:** They do not require knowledge of transition probabilities or reward distributions. Learning is based purely on sampled experience.
- **Direct Sampling:** The agent learns directly from actual interactions with the environment, similar to how learning occurs in many natural systems.
- **Episodic Suitability:** MC methods are well-suited to environments where episodes have a clear beginning and end (e.g., games, tasks with natural termination points).

Learning to Predict: Monte Carlo for Value Estimation

A primary application of Monte Carlo methods is in prediction—estimating the value function under a given, fixed policy. This involves:

- Running multiple episodes while following the fixed policy.
- Recording the return (total cumulative reward) for each episode.
- Averaging the returns observed for each state (or state-action pair) over time.

The result is an estimate of the value function

$$V^{\pi}(s),$$

representing the expected return when starting in state s and following policy π. This provides insight into how effective the policy is.

Learning to Control: Monte Carlo for Policy Improvement

Beyond evaluating a policy, Monte Carlo methods can be used for control, i.e., improving the policy over time. The general procedure is as follows:

1. Evaluate the current policy using Monte Carlo estimation.
2. Improve the policy by acting greedily with respect to the estimated value function.
3. Repeat this process iteratively to converge toward an optimal policy.

A key aspect of this process is managing the exploration–exploitation trade-off. While the agent should favor actions that lead to higher returns (exploitation), it must also explore other actions to discover potentially better strategies. This is commonly managed using an ϵ-greedy policy, where the agent mostly chooses the best-known action but occasionally selects a random action.

Fundamental Concepts and Notation in Reinforcement Learning

In reinforcement learning, we model problems where an agent interacts with an environment over time to maximize some notion of cumulative reward. Below are the core concepts and mathematical notations used to define this process.

1. Return

The **return** G_t is the total accumulated reward from time step t onward, possibly discounted by a factor γ:

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

Where:

- R_{t+k+1} is the reward received $k+1$ steps after time t
- $\gamma \in [0, 1]$ is the *discount factor*, controlling the importance of future rewards
- T is the time step at which the episode ends (for episodic tasks)

Interpretation: The agent values immediate rewards more than future ones if $\gamma < 1$, which encourages short-term gains. If $\gamma = 1$, the agent values future rewards equally, useful in undiscounted tasks.

2. Recursive Form of Return

The return can be expressed recursively as:

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

Interpretation: The return at time t equals the immediate reward plus the discounted return from the next time step onward. This recursive formulation is fundamental to many RL algorithms, including Monte Carlo and Temporal Difference (TD) methods.

3. State-Value Function

The **state-value function** under a policy π is the expected return starting from state s and following policy π:

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

Where:

- $V^\pi(s)$ is the expected return when starting in state s and following policy π
- $\mathbb{E}_\pi[\cdot]$ denotes the expectation under policy π

Interpretation: It tells us how good it is to be in a particular state, assuming the agent follows policy π thereafter.

4. Action-Value Function

The **action-value function** under a policy π is the expected return starting from state s, taking action a, and then following policy π:

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

Where:

- $Q^{\pi}(s, a)$ represents the expected return after taking action a in state s and then behaving according to policy π

Interpretation: It evaluates the usefulness of an action in a given state under a specific policy.

5. Policy

A **policy** $\pi(a \mid s)$ is a mapping from states to probabilities of selecting each action:

$$\pi(a \mid s) = \Pr(A_t = a \mid S_t = s)$$

Where:

- π can be deterministic (one action per state) or stochastic (a probability distribution over actions)
- A_t is the action chosen at time t, and S_t is the current state

Interpretation: The policy is the agent's strategy—it defines how the agent chooses actions based on the current state.

6. Discount Factor

The **discount factor** γ determines the present value of future rewards:

$$0 \leq \gamma \leq 1$$

Where:

- $\gamma = 0$ makes the agent myopic (only cares about immediate rewards)
- $\gamma = 1$ considers future rewards as important as immediate ones (used in undiscounted tasks)

Interpretation: A smaller γ leads to short-term planning, while a higher γ encourages long-term strategies.

Monte Carlo Methods Overview

Monte Carlo (MC) methods are a class of model-free reinforcement learning techniques. They are used for learning value functions and making decisions based solely on experience, without requiring a model of the environment.

Main Categories

Monte Carlo algorithms can be broadly categorized into:

1. **Prediction (Policy Evaluation)**: Estimate the value function $V^\pi(s)$ for a fixed policy π.
   - **First-Visit MC Prediction**: Considers only the first occurrence of a state in each episode.
   - **Every-Visit MC Prediction**: Considers all occurrences of the state in each episode.
2. **Control (Policy Improvement)**: Improve the policy based on value estimates.
   - **MC with Exploring Starts**: Requires starting from random state-action pairs.
   - **MC Control with ϵ-Greedy**: Uses ϵ-greedy exploration for improved learning without the need for random starts.

First-Visit Monte Carlo Prediction

Goal

Estimate the state-value function $V^\pi(s)$ under a fixed policy π, by averaging the returns following the first visit to each state.

Key Idea

- Sample complete episodes using policy π.
- For each state s, consider only the **first occurrence** of s in the episode.
- Compute the return G_t from that point onward and update the estimate of $V(s)$.

Return

The return G_t from time step t onward is:

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

Update Rule

Let $G^{(i)}(s)$ be the return following the first visit of state s in the i-th episode. Then:

$$V(s) \leftarrow \frac{1}{N(s)} \sum_{i=1}^{N(s)} G^{(i)}(s)$$

Where:

- $N(s)$ is the number of episodes in which state s was visited **first**.

Algorithm (Pseudo-code Summary)

1. Initialize $Returns(s) \leftarrow 0$, $N(s) \leftarrow 0$ for all states s
2. For each episode:
   - Generate an episode $(s_0, a_0, r_0, \ldots, s_T)$
   - For each time step $t = T - 1$ down to 0:
     - If s_t is the first occurrence of state s in the episode:
       * Compute G_t
       * Update: $Returns(s_t) \leftarrow Returns(s_t) + G_t$
       * Increment: $N(s_t) \leftarrow N(s_t) + 1$
3. Compute $V(s) = \frac{Returns(s)}{N(s)}$

Every-Visit Monte Carlo Prediction

Goal

Estimate the state-value function $V^\pi(s)$ under a fixed policy π, by averaging the returns for **every occurrence** of a state in an episode.

Key Idea

- Sample full episodes using policy π
- For each occurrence of each state s, compute the return from that time step onward.
- Update $V(s)$ using all such returns.

Return

Same as First-Visit:

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

Update Rule

Let $G_j^{(i)}(s)$ be the return from the j-th occurrence of state s in episode i. Then:

$$V(s) \leftarrow \frac{1}{N(s)} \sum_{i,j} G_j^{(i)}(s)$$

Where:

- $N(s)$ is the total number of **times** state s is visited across all episodes.

Algorithm (Pseudo-code Summary)

1. Initialize $Returns(s) \leftarrow 0$, $N(s) \leftarrow 0$ for all states s
2. For each episode:
   - Generate an episode $(s_0, a_0, r_0, \ldots, s_T)$
   - For each time step $t = T - 1$ down to 0:
     - Compute G_t
     - Update: $Returns(s_t) \leftarrow Returns(s_t) + G_t$
     - Increment: $N(s_t) \leftarrow N(s_t) + 1$
3. Compute $V(s) = \frac{Returns(s)}{N(s)}$

Comparison: First-Visit vs Every-Visit

- **First-Visit MC:**
  - Updates based only on the first occurrence of each state per episode
  - Lower variance but slower updates
  - Good for theoretical convergence
- **Every-Visit MC:**
  - Updates based on all occurrences of a state in an episode
  - Faster learning but slightly higher variance

Monte Carlo Control with Exploring Starts

Goal

To find the optimal policy π^* by learning the optimal action-value function $Q^*(s, a)$, and improving the policy iteratively through experience.

Key Idea

- Generate episodes that start from randomly chosen **state-action pairs** — known as **Exploring Starts**.
- Estimate the action-value function $Q(s, a)$ using sampled returns.
- Improve the policy π by acting greedily with respect to $Q(s, a)$.

Return

The return G_t at time step t is:

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

Update Rule

Let $G^{(i)}(s, a)$ be the return following the first occurrence of pair (s, a) in episode i. Then:

$$Q(s, a) \leftarrow \frac{1}{N(s, a)} \sum_{i=1}^{N(s,a)} G^{(i)}(s, a)$$

$$\pi(s) \leftarrow \arg\max_a Q(s, a)$$

Requirements

- Every state-action pair must be visited infinitely often (Exploring Starts guarantees this).
- The policy must continually improve towards the optimal policy.

Pseudocode

Input: Set of states S, actions A, discount factor γ, number of episodes n_{ep}
Output: Optimal policy π^*, and corresponding action-value function $Q(s, a)$

1. Initialize:
   - $Q(s, a) \leftarrow 0$ for all $s \in S, a \in A$
   - $N(s, a) \leftarrow 0$ for all $s \in S, a \in A$
   - $\pi(s) \leftarrow \text{random}(A)$
2. For each episode $i = 1$ to n_{ep}:
   (a) Choose random state-action pair (s_0, a_0) as starting point (Exploring Start)
   (b) Generate an episode:
   $$(s_0, a_0, r_0), (s_1, a_1, r_1), \ldots, (s_{T-1}, a_{T-1}, r_{T-1})$$
   using policy π
   (c) For each time step $t = T - 1$ down to 0:
   - Compute return:
   $$G_t = \sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1}$$
   - Let (s_t, a_t) be the state-action pair at time t
   - Update:
   $$N(s_t, a_t) \leftarrow N(s_t, a_t) + 1$$
   $$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \frac{1}{N(s_t, a_t)} (G_t - Q(s_t, a_t))$$
   - Improve policy:
   $$\pi(s_t) \leftarrow \arg\max_a Q(s_t, a)$$

Remarks

- This method is guaranteed to converge to the optimal policy π^* and action-value function $Q^*(s, a)$, given infinite episodes and proper exploring starts.
- However, in practice, true exploring starts are often infeasible — which motivates alternatives like ϵ-greedy exploration in MC Control.

Monte Carlo Control with ε-Greedy Policy

Goal

To learn the optimal policy π^* using sampled episodes generated under an ε-greedy exploration strategy — without requiring exploring starts.

Key Idea

- Instead of starting episodes from arbitrary state-action pairs, encourage exploration using an ε-greedy policy.
- With probability ε, take a random action (exploration).
- With probability $1 - \varepsilon$, take the greedy action (exploitation).
- Estimate the action-value function $Q(s, a)$ from episodes.
- Improve the policy iteratively to become more greedy over time.

Exploration Policy Definition

Let $A(s)$ be the set of available actions in state s. The ε-greedy policy $\pi(a|s)$ is defined as:

$$\pi(a|s) = \begin{cases} 1 - \varepsilon + \dfrac{\varepsilon}{|A(s)|}, & \text{if } a = \arg\max_{a'} Q(s, a') \\ \dfrac{\varepsilon}{|A(s)|}, & \text{otherwise} \end{cases}$$

This ensures that:

- All actions are explored with non-zero probability.
- The action with the highest current value estimate is preferred.

Update Rule

For a given state-action pair (s_t, a_t) at time t, we compute the return G_t, and then use an incremental update for $Q(s_t, a_t)$:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[G_t - Q(s_t, a_t)\right]$$

where:

- G_t is the return following (s_t, a_t)
- $\alpha \in (0, 1]$ is the learning rate

Advantages

- Does not require exploring starts.
- Easily implemented in continuous, stochastic environments.
- Balances exploration and exploitation.

Pseudocode

Input: States S, Actions A, Discount factor γ, Exploration rate ε, Learning rate α, Number of episodes n_{ep}
Output: Approximated optimal action-value function $Q(s, a)$, and greedy policy π

1. Initialize:
   - $Q(s, a) \leftarrow 0$ for all $s \in S, a \in A$
2. For each episode $i = 1$ to n_{ep}:
   (a) Initialize s_0
   (b) Generate an episode $(s_0, a_0, r_1, s_1, a_1, r_2, \ldots, s_T)$ using ε-greedy policy π derived from Q
   (c) For each time step $t = T - 1$ down to 0:
   - Compute return:
   $$G_t = \sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1}$$
   - Update:
   $$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[G_t - Q(s_t, a_t)\right]$$

Remarks

- If $\varepsilon \to 0$ over time, the policy becomes greedy, and this method can converge to the optimal policy π^*.
- A decaying ε schedule is often used in practice to balance exploration and convergence.

Method	Task	Sampling Type	Output
First-Visit MC	Prediction	First occurrence of s	$V(s)$
Every-Visit MC	Prediction	Every occurrence of s	$V(s)$
MC with Exploring Starts	Control	Random (s, a) starts	$Q(s, a), \pi$
MC with ε-Greedy Policy	Control	ε-greedy policy over time	$Q(s, a), \pi$

Prediction methods estimate value functions for a fixed policy; control methods aim to improve the policy toward optimality.

What is Temporal-Difference (TD) Learning?

Temporal-Difference (TD) learning is a core method in reinforcement learning that enables agents to learn value functions directly from raw experience—without waiting for the end of an episode or requiring a model of the environment. TD methods combine the strengths of Monte Carlo methods and Dynamic Programming. They are:

- **Model-free:** TD does not need knowledge of transition dynamics.
- **Online and Incremental:** TD updates can be made after every step, making it suitable for continual learning.
- **Bootstrapped:** TD methods update estimates based on other learned estimates rather than waiting for complete returns.

Why Temporal-Difference (TD) Learning?

TD learning solves the core prediction problem in reinforcement learning without requiring a model or full episode completion. It blends:

- **Sampling (like Monte Carlo):** Learns from raw experience.
- **Bootstrapping (like DP):** Uses estimates of future values to update current values.

This combination allows for:

- Online, incremental updates
- Model-free learning
- Adaptability to both episodic and continuing tasks

The General TD Learning Rule

Core Idea:
Temporal-Difference (TD) methods are based on the principle of updating estimates using *other learned estimates*, rather than waiting for a final, complete return (as in Monte Carlo methods), or computing expectations with a known model (as in Dynamic Programming).

Generic TD Update Rule:

$$\text{New Estimate} \leftarrow \text{Old Estimate} + \alpha \cdot (\text{Target} - \text{Old Estimate})$$

This is the core update equation used across all TD methods. Each component plays an important role:

- $\alpha \in (0, 1]$ is the **learning rate**, which controls how quickly or cautiously the estimate is adjusted. A smaller α results in slower but more stable learning. A larger α leads to faster learning but can introduce instability.
- Old Estimate refers to the current estimated value of a quantity (e.g., value function $V(s)$ or action-value function $Q(s, a)$).

- Target is a **short-term approximation of the long-term return**. It typically includes the immediate reward received and a bootstrapped estimate of the next value.

Why This Rule Works:
This formula adjusts the old estimate in the direction of the new target. The amount of adjustment is proportional to the difference between the new target and the old value—this difference is called the **Temporal-Difference (TD) Error**:

$$\delta = \text{Target} - \text{Old Estimate}$$

Then the update rule becomes:

$$\text{New Estimate} = \text{Old Estimate} + \alpha \cdot \delta$$

This idea underlies both state-value learning and action-value learning in TD methods.

Applicability:

- This update rule is used for **value functions** such as $V(s)$, where s is a state.
- It is also used for **Q-functions** or action-value functions $Q(s, a)$, where the value is associated with taking action a in state s.

Understanding the TD Error

Temporal-Difference (TD) Error is a fundamental concept in TD learning. It measures how much the agent's prediction differs from what it just experienced—in other words, it quantifies the degree of **surprise**.

Definition:
The TD error at time step t, denoted δ_t, is defined as:

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

Let us break down each component:

- r_{t+1}: The immediate reward received after taking an action at time t.
- $\gamma \in [0, 1]$: The discount factor, which reduces the importance of future rewards. A higher γ places more emphasis on long-term outcomes.
- $V(s_{t+1})$: The current estimate of the value of the next state s_{t+1}.
- $V(s_t)$: The current estimate of the value of the present state s_t.

Together, the term $r_{t+1} + \gamma V(s_{t+1})$ is referred to as the **TD Target** or **Better Estimate**, since it represents an improved estimate of the return by combining the observed reward with the value of the next state.
The term $V(s_t)$ is the **Current Estimate**, the value we previously believed was accurate.

Interpretation:

- If $\delta_t = 0$, then the observed outcome perfectly matches our expectations. The estimate $V(s_t)$ is already accurate, and no update is needed.
- If $\delta_t \neq 0$, the TD error provides a signal indicating the direction and magnitude of the adjustment needed to improve the estimate.

Update Using TD Error:
We incorporate the TD error into the learning update:

$$V(s_t) \leftarrow V(s_t) + \alpha \cdot \delta_t$$

Substituting the expression for δ_t, this becomes:

$$V(s_t) \leftarrow V(s_t) + \alpha \cdot (r_{t+1} + \gamma V(s_{t+1}) - V(s_t))$$

This is equivalent to the generic TD learning rule:

$$\text{New Estimate} = \text{Old Estimate} + \alpha \cdot (\text{Target} - \text{Old Estimate})$$

Temporal-Difference Learning (TD(0)) — One-Step Learning

The goal of Temporal-Difference (TD) learning, specifically **TD(0)**, is to estimate the value function $V^\pi(s)$ for a given policy π, which defines the way the agent behaves in the environment. The value function $V^\pi(s)$ represents the expected return (cumulative discounted reward) from state s under policy π.

TD(0) is one of the simplest forms of TD prediction methods, where the updates to the value function are made after every individual time step, instead of waiting for an entire episode to conclude. TD(0) estimates the value of state s_t by incorporating the immediate reward r_{t+1} and the value of the next state s_{t+1}, making it an efficient method for online learning.

Detailed Explanation of the TD(0) Algorithm

Initialization

At the beginning, we initialize the value function $V(s)$ for all states s, except for terminal states (which might have predefined values like zero). These initial values are typically chosen arbitrarily, except for terminal states which can be set to zero or a known value. We also initialize the learning rate α, a small positive constant, and the discount factor γ, which is a value between 0 and 1.

$V(s)$ is initialized arbitrarily for all states s (except terminal states).

$\alpha \in (0, 1]$ is the learning rate.

$\gamma \in [0, 1)$ is the discount factor.

For Each Episode

We start the learning process by iterating through episodes. In each episode, the agent starts from a particular initial state s_0, and interacts with the environment by following the policy π. At each step, the agent will take an action, observe the reward and transition to a new state, then update the value estimate for the current state.

For each episode:

$s_0 \leftarrow$ Initial state

For Each Step

- The agent takes an action a_t according to the policy π. This is typically an action chosen based on the state s_t and the agent's policy, which could be deterministic or stochastic. - The agent then receives an immediate reward r_{t+1} and transitions to the next state s_{t+1}. - The agent updates the value function for the state s_t based on the new information it has acquired (the reward r_{t+1} and the value of the next state $V(s_{t+1})$) using the **TD(0) update rule**.

TD(0) Update Rule

The core of the TD(0) algorithm is the **update rule** that modifies the value of the current state s_t based on the new information. The update rule can be expressed as:

$$V(s_t) \leftarrow V(s_t) + \alpha \left(r_{t+1} + \gamma V(s_{t+1}) - V(s_t) \right)$$

- r_{t+1}: This is the immediate reward received by the agent after taking action a_t and transitioning to state s_{t+1}.
- γ: The **discount factor**, which determines how much importance the agent gives to future rewards. If γ is close to 1, the agent values future rewards almost as much as immediate rewards. If γ is close to 0, the agent only cares about immediate rewards.
- $V(s_t)$: The current estimate of the value of state s_t, before the update.
- $V(s_{t+1})$: The estimated value of the next state s_{t+1}.

The term $r_{t+1} + \gamma V(s_{t+1})$ is known as the **TD target**, which is the updated estimate of the value of state s_t. The difference between this TD target and the current value $V(s_t)$ is called the **TD error** δ_t, which reflects how much the current estimate deviates from the updated target.

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

The agent uses this error δ_t to adjust its estimate $V(s_t)$ in the direction of the new estimate.

End of Episode

Once the episode terminates (e.g., the agent reaches a terminal state or a predefined step limit), the value function for each state will have been updated based on the agent's experiences during the episode. The algorithm then proceeds to the next episode, where the value function is further refined as the agent continues to explore the environment.

Repeat

The process is repeated over multiple episodes, with the value estimates $V(s_t)$ gradually converging towards the true value function $V^\pi(s)$ for the given policy π. 0

TD Prediction Beyond TD(0) – Other Algorithms

While **TD(0)** is a simple and efficient method for estimating the value function $V^\pi(s)$ of a given policy π, it only looks at a one-step lookahead to update value estimates. This can be limiting in certain cases, as it doesn't take into account the wider context of the agent's future trajectory. To overcome this limitation, we introduce more general methods, such as **TD()** and **Expected Updates**, that strike a balance between TD and Monte Carlo methods.

These more general methods incorporate the benefits of multi-step returns, providing more flexibility in the way value functions are updated.

—

1. TD() – Eligibility Traces

TD() is a generalization of TD(0) that combines ideas from both Temporal-Difference methods and Monte Carlo methods. The key feature of TD() is the introduction of **eligibility traces**, which allow the algorithm to use information from multiple steps (instead of just one) when updating the value function.

Key Ideas:

- **Eligibility Traces:** TD() introduces eligibility traces, which are a way of maintaining a memory of states that have been visited. This allows the agent to update not only the value of the current state but also the values of previously visited states. The eligibility trace for each state is updated over time.
- **Multiple-Step Returns:** TD() updates the value function based on multiple-step returns, which include 1-step, 2-step, or even the full Monte Carlo return. The longer the horizon considered, the closer the algorithm approximates Monte Carlo.
- **Parameter λ:** The parameter λ controls how much influence past states have on the current update. It also controls the decay of eligibility traces.
  - $\lambda = 0$: Reduces to TD(0), where only the immediate next state is considered.
  - $\lambda = 1$: Approximates Monte Carlo methods, using the full return.
  - For $0 < \lambda < 1$: The updates are a mix between TD and Monte Carlo, where the influence of past states decreases as we look further into the future.

The Update Rule in TD():

In TD(), an eligibility trace is maintained for each state s visited during an episode. The eligibility trace $E(s)$ for a state s_t is updated as follows:

$$E(s_t) \leftarrow \gamma\lambda E(s_t) + 1_{\{s_t=s\}}$$

Where:

- $E(s_t)$: Eligibility trace for state s_t at time t,
- γ: Discount factor,
- λ: The decay parameter controlling the trace length,
- $1_{\{s_t=s\}}$ is an indicator function which is 1 if the state is s, and 0 otherwise.

The value function update in TD() is then:

$$V(s_t) \leftarrow V(s_t) + \alpha \left(r_{t+1} + \gamma V(s_{t+1}) - V(s_t)\right) E(s_t)$$

Where:

- r_{t+1}: The reward received after taking action a_t in state s_t,
- $V(s_t)$: The current estimate of the value of state s_t,
- $V(s_{t+1})$: The value estimate for the next state s_{t+1},
- $E(s_t)$: The eligibility trace for state s_t, determining how much influence past states should have on the update.

This update rule is a weighted combination of the immediate reward and the value of the next state, with the eligibility trace providing the weighting mechanism.

Effect of λ:

- For $\lambda = 0$, this update rule reduces to TD(0), where only the immediate reward and the next state's value contribute to the update. - For $\lambda = 1$, the update rule becomes Monte Carlo, where the update is based on the full return from the current state to the end of the episode.

—

2. Expected Updates (e.g., Expected SARSA)

Expected Updates methods, such as **Expected SARSA**, differ from TD methods like SARSA or Q-learning by using the expected value of the next state under a stochastic policy, rather than the actual next state value. This reduces the variance of updates and can improve the stability of learning.

Key Ideas:

- **Stochastic Policy:** In many environments, the agent may follow a stochastic policy, meaning that the action taken at a given state s_t is not deterministic but probabilistic. For instance, in an ϵ-greedy policy, the agent typically selects the best action with probability $1 - \epsilon$ and explores a random action with probability ϵ.
- **Expected Value:** In **Expected SARSA**, instead of updating based on the action actually taken at the next step, we average over all possible actions the agent might take at the next state. This reduces the variance of the update, which can make the learning process more stable.

The Update Rule in Expected SARSA:

The update rule for **Expected SARSA** is as follows:

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

Where:

- $Q(s_t, a_t)$: The action-value function for state s_t and action a_t,
- r_{t+1}: The reward obtained after taking action a_t in state s_t,
- γ: The discount factor,
- $\mathbb{E}_{a_{t+1}}[Q(s_{t+1}, a_{t+1})]$: The expected value of the next state's action-value function, averaged over all possible actions the agent could take at state s_{t+1} according to the policy π.

The key difference between Expected SARSA and standard SARSA is that Expected SARSA uses the expected value of $Q(s_{t+1}, a_{t+1})$, averaged over all possible actions a_{t+1}, rather than using the value corresponding to the actual action taken at s_{t+1}. This reduces variance and helps to stabilize the learning process.

SARSA – On-Policy Learning

SARSA stands for:

$$\text{State}(s_t) \rightarrow \text{Action}(a_t) \rightarrow \text{Reward}(r_{t+1}) \rightarrow \text{Next State}(s_{t+1}) \rightarrow \text{Next Action}(a_{t+1})$$

It is a **model-free**, **on-policy** Temporal-Difference (TD) control algorithm used in reinforcement learning. Let's break down its components and how it works in detail.

—

What Does SARSA Do?

SARSA is used to **learn the action-value function** $Q(s, a)$, which represents the expected return (or value) of performing a particular action a in a particular state s under the current policy π. The goal is to learn a policy that maximizes the cumulative expected reward by improving $Q(s, a)$ over time.

Key Characteristics of SARSA: - **On-policy:** SARSA is an on-policy algorithm because it learns about the action-value function based on the actions taken by the agent following the policy π that is being improved. Importantly, the same policy is used both for selecting actions and for updating the Q-values.

- **Exploration:** SARSA typically uses an **ϵ-greedy exploration strategy**. This means that most of the time the agent will pick the action that maximizes the expected reward according to the current estimate of $Q(s, a)$, but with a small probability ϵ, the agent will pick a random action to explore the environment.

—

How Does SARSA Update the Q-values?

SARSA's main contribution is in how it updates the **action-value function** $Q(s, a)$ during the learning process. The algorithm follows the **TD(0) approach**, which means it updates $Q(s, a)$ based on a **one-step lookahead**.

SARSA Update Rule:

At each time step t, when the agent is in state s_t and takes action a_t, it receives a reward r_{t+1} and ends up in state s_{t+1}. The agent then takes an action a_{t+1} in the next state s_{t+1} according to the same policy π. The Q-value update is:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \cdot [r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$

Where: - $Q(s_t, a_t)$ is the current action-value estimate for taking action a_t in state s_t, - α is the **learning rate**, controlling how much new information should be incorporated into the Q-value update, - r_{t+1} is the immediate reward received after taking action a_t in state s_t, - γ is the **discount factor**, which determines the importance of future rewards relative to immediate rewards, - $Q(s_{t+1}, a_{t+1})$ is the Q-value for the next state s_{t+1} and the next action a_{t+1} chosen according to the current policy.

The term:

$$r_{t+1} + \gamma Q(s_{t+1}, a_{t+1})$$

represents the **TD target**, or the updated estimate for the value of the current state-action pair, incorporating the reward received and the expected future value from state s_{t+1}.

The difference between this target and the current Q-value:

$$r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)$$

is the **TD error**. This error measures how much the current Q-value is off from the target value. The update rule moves $Q(s_t, a_t)$ towards the target by an amount proportional to the learning rate α.

—

Learning Process in SARSA

The learning process in SARSA involves iterating through multiple episodes where the agent interacts with the environment, takes actions, receives rewards, and updates the Q-values accordingly.

SARSA Learning Loop:

- **Choose an action** a_t from state s_t according to the **ϵ-greedy policy**. This means: - With probability $1 - \epsilon$, choose the action that maximizes $Q(s_t, a_t)$, - With probability ϵ, choose a random action for exploration.

- **Take the action** a_t and observe the reward r_{t+1} and the next state s_{t+1}.

- **Choose the next action** a_{t+1} from state s_{t+1} using the same policy π (i.e., use ϵ-greedy on $Q(s_{t+1}, a_{t+1})$).

- **Update** $Q(s_t, a_t)$ using the SARSA update rule:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \cdot [r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$

- **Repeat** this process for each time step until the episode ends.

- After each episode, the process starts again with the initial state.

—

Why is SARSA On-Policy?

The key feature that makes SARSA an **on-policy** method is that it updates its Q-values based on the **actions it actually takes** in the environment, following the **same policy** used to select those actions.

- The **"On-Policy"** nature means that the update depends on the **real actions taken by the agent**, including those that are part of the exploration process. Even if the action is random (due to exploration), the Q-value is updated based on that action.

This is in contrast to **Off-policy** methods like Q-learning, where the Q-values are updated based on the optimal actions, not necessarily the ones actually taken by the agent.

In SARSA, the update rule reflects the **real-world experience** of the agent, including its exploration of the environment, and doesn't assume that the agent will always take the optimal action.

Q-Learning – Off-Policy Learning

Q-Learning is a model-free, off-policy Temporal-Difference (TD) control algorithm used in reinforcement learning. The goal of Q-Learning is to learn the **optimal action-value function** $Q^*(s, a)$, which represents the maximum expected cumulative reward the agent can achieve by taking action a in state s and following the optimal policy thereafter.

—

What Does Q-Learning Do?

The fundamental goal of Q-Learning is to learn the optimal policy π^* that maximizes the expected reward over time. Unlike on-policy methods like SARSA, Q-Learning is **off-policy**, meaning that it learns the optimal policy even if the agent explores using a different policy.

Key Characteristics of Q-Learning: - **Off-policy:** Q-Learning is an off-policy method because it updates its Q-values assuming the agent is always taking the optimal action, regardless of the actions it actually takes during exploration. This contrasts with on-policy methods, which update the Q-values based on the actions the agent actually takes.

- **Exploration and Exploitation:** While Q-Learning assumes optimal behavior for Q-value updates, it still needs exploration to discover the environment's true dynamics. Typically, an ϵ-greedy strategy is used, where the agent chooses the best-known action most of the time but occasionally chooses a random action to explore the environment.

—

How Does Q-Learning Update the Q-values?

Q-Learning uses a **greedy** update rule, meaning it updates the Q-values as if the agent always selects the action that maximizes the expected future reward. This contrasts with SARSA, where the agent updates Q-values based on the actions it actually took.

Q-Learning Update Rule:

At each time step t, when the agent is in state s_t and takes action a_t, it receives a reward r_{t+1} and ends up in state s_{t+1}. The Q-value update is:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \cdot \left[r_{t+1} + \gamma \cdot \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right]$$

Where: - $Q(s_t, a_t)$ is the current action-value estimate for taking action a_t in state s_t, - α is the **learning rate**, controlling how much new information should be incorporated into the Q-value update, - r_{t+1} is the immediate reward received after taking action a_t in state s_t, - γ is the **discount factor**, which determines the importance of future rewards relative to immediate rewards, - $\max_a Q(s_{t+1}, a)$ represents the maximum Q-value of the next state s_{t+1} over all possible actions, i.e., the best possible future action.

The term:

$$r_{t+1} + \gamma \cdot \max_a Q(s_{t+1}, a)$$

is the **TD target**, or the updated estimate for the value of the current state-action pair, incorporating the reward received and the maximum expected future value from state s_{t+1}.

The difference between this target and the current Q-value:

$$r_{t+1} + \gamma \cdot \max_a Q(s_{t+1}, a) - Q(s_t, a_t)$$

is the **TD error**. This error measures how much the current Q-value is off from the target value. The update rule moves $Q(s_t, a_t)$ towards the target by an amount proportional to the learning rate α.

—

Learning Process in Q-Learning

Q-Learning learns the optimal action-value function by iterating through multiple episodes, where the agent interacts with the environment, takes actions, receives rewards, and updates the Q-values accordingly.

Q-Learning Learning Loop:

- **Choose an action** a_t from state s_t according to the **ϵ-greedy policy**. This means: - With probability $1 - \epsilon$, choose the action that maximizes $Q(s_t, a_t)$, - With probability ϵ, choose a random action for exploration.

- **Take the action** a_t and observe the reward r_{t+1} and the next state s_{t+1}.

- **Compute the best next action:** Calculate $\max_a Q(s_{t+1}, a)$, which is the maximum Q-value for the next state s_{t+1} over all possible actions.

- **Update** $Q(s_t, a_t)$ using the Q-Learning update rule:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \cdot \left[r_{t+1} + \gamma \cdot \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right]$$

- **Repeat** this process for each time step until the episode ends.

- After each episode, the process starts again with the initial state.

—

Why is Q-Learning Off-Policy?

Q-Learning is an **off-policy** method because it updates the Q-values assuming the agent always takes the optimal action, regardless of the action actually taken during exploration. Specifically: - The agent may explore the environment using an exploration policy (e.g., ϵ-greedy). - However, for each update, Q-Learning assumes that the optimal action was taken in the next state, i.e., it uses the greedy action according to the current Q-values.

This is in contrast to **on-policy** methods like SARSA, where the Q-values are updated using the action that was actually taken, regardless of whether it was optimal or exploratory.

By using the best possible next action in the update (i.e., $\max_a Q(s_{t+1}, a)$), Q-Learning ensures that it converges to the optimal policy, even if the agent does not always act optimally during learning.

Aspect	**SARSA (On-Policy)**	**Q-Learning (Off-Policy)**
Policy Type	Learns value of current behavior policy	Learns value of optimal greedy policy
Next Action	$a_{t+1} \sim \pi$ (actual action taken)	$\max_a Q(s_{t+1}, a)$ (greedy)
Update Rule	$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right]$	$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right]$
Exploration-aware	Yes	No
Risk Behavior	Safer, cautious learning	More aggressive, optimistic

Table 1: Comparison of SARSA (On-Policy) and Q-Learning (Off-Policy)

ϵ-Greedy Action Selection

The ϵ-greedy action selection strategy is commonly used in reinforcement learning algorithms, such as SARSA and Q-Learning, to balance exploration and exploitation. The key idea is that the agent will explore the environment by choosing random actions with a small probability ϵ, and exploit its knowledge of the environment (i.e., choose the action with the highest Q-value) with a probability of $1 - \epsilon$.

Explanation

- **Exploration:** With probability ϵ, the agent chooses a random action, even if it is not necessarily optimal. This is known as exploration, and it helps the agent gather information about the environment. - **Exploitation:** With probability $1 - \epsilon$, the agent selects the action that has the highest Q-value for the current state, which is the action that it believes will lead to the highest expected reward. This is known as exploitation, and it allows the agent to take advantage of the knowledge it has already learned.

The value of ϵ typically decreases over time, which means that the agent starts by exploring the environment more and gradually shifts towards exploitation as it learns more about the optimal policy. This is referred to as an ϵ-decay schedule, and it helps the agent strike a balance between learning and optimizing.

Pseudocode

The following pseudocode describes the ϵ-greedy action selection process:

```
def epsilon_greedy(Q, state, epsilon):
    if random() < epsilon:
        return random_action()  # Explore by choosing a random action
    else:
        return argmax_a Q[state][a]  # Exploit by choosing the action with the highest Q-value
```

Where: - Q is the action-value function, - *state* is the current state of the agent, - ϵ is the probability of choosing a random action (exploration), - $random()$ generates a random number between 0 and 1, - $random\_action()$ selects a random action from the set of possible actions, - $argmax_a Q[state][a]$ returns the action a that maximizes $Q(s, a)$, i.e., the action with the highest Q-value for the given state.

Usage

The ϵ-greedy action selection strategy is used in both:

- **SARSA:** An on-policy algorithm that updates the action-value function based on the actions the agent actually takes.
- **Q-Learning:** An off-policy algorithm that updates the action-value function as if the agent always takes the optimal action.

In both of these algorithms, ϵ-greedy helps balance the need for exploration (learning about the environment) and exploitation (using the best-known actions to maximize rewards).

Key Takeaways

- Monte Carlo methods learn from complete episodes, using the total reward from each episode to estimate the value of states or actions by summing rewards from a point to the end of the episode.
- MC is model-free, meaning it does not require knowledge of the environment's transition probabilities or reward function. It learns directly from the experience of the agent interacting with the environment.
- Value estimates are based on averaging actual returns observed during episodes. As more episodes are collected, the value estimates become more accurate, reflecting real rewards.
- In First-visit MC, only the first occurrence of a state in an episode is used to update its value, while Every-visit MC uses all occurrences of a state in an episode to calculate the average return.
- Exploring starts ensures that every state-action pair is visited by starting episodes from randomly selected states, guaranteeing that the agent explores diverse situations.
- On-policy MC control uses ϵ-soft policies, where the agent takes random actions with a small probability to ensure sufficient exploration of all actions while still improving the policy.
- MC methods are ideal for episodic tasks where episodes always terminate, as they rely on complete episodes to calculate returns and learn from them.
- TD learning updates value estimates after every step using both the immediate reward and a prediction of future rewards, enabling faster and more efficient learning.
- It learns directly from raw experience without requiring a model of the environment, making it suitable for model-free reinforcement learning.
- TD methods handle both episodic tasks (with clear endings) and continuing tasks (like ongoing control problems).
- TD error (δ) quantifies the difference between predicted and observed outcomes and drives learning—larger surprises trigger bigger updates.
- TD underpins powerful algorithms like Q-learning and SARSA, and scales to complex problems through function approximation (e.g., with neural networks).