Temporal-Difference Reinforcement Learning

Minor in AI - IIT ROPAR

22nd April, 2025

A Journey Into Temporal-Difference Learning

In a vast land where no maps had ever been drawn, a solitary traveler journeyed through the unknown. Hills rose like questions, rivers curved with mystery, and every path forked into uncertainty. No signs pointed the way, and no one could say what reward—or danger—waited beyond the next bend. The traveler had only one compass: experience.

At first, they walked long and winding paths, waiting until the end of each journey to pause and reflect. Only then did they count the treasures earned and the pitfalls endured. They traced their steps backward in memory, adjusting their judgment of each place based on the final tally. This was like learning from complete stories—Monte Carlo learning. It was honest, grounded in truth, but terribly slow. When paths were long or endless, the traveler often had to wait too long to learn anything useful.

One day, the traveler encountered a group of scholars who studied the land from towers of knowledge. They were the keepers of Dynamic Programming. They didn't need to walk the trails; they already knew the rules of the land—the likelihood of moving from one place to another, and the reward each action would bring. With this perfect model, they computed the value of every step through pure reason. It was powerful—almost magical. But the traveler could only watch in envy. The land held no such secrets for them. The world was a black box.

So, the traveler did something different.

Rather than waiting until the end of a journey, or hoping for a map they would never receive, they began to learn at every step. After each move, they took the reward at face value, then guessed what the future might hold, adjusting their beliefs immediately. They didn't need the full picture—just a glimpse ahead was enough. It wasn't perfect, but it was fast, flexible, and surprisingly wise over time.

This was Temporal-Difference learning: a balance between dreaming and remembering, between guessing and knowing. The traveler no longer feared the unknown. With every footfall, they learned—step by step, moment by moment—crafting a mental map not of the land itself, but of how to move wisely within it.

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.

$$V(s) \text{ is initialized arbitrarily for all states } s \text{ (except terminal states).}$$

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

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

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.

$$\text{For each episode:}$$

$$s_0 \leftarrow \text{Initial state}$$

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

- TD does learn from experience without needing a model of the environment – it figures things out by trial and error as it goes.
- TD updates guesses about future rewards after every step instead of waiting until the end of an episode, making learning faster.
- TD combines actual rewards received with its own predictions to improve its estimates over time.
- TD works for both tasks with clear endings (like games) and ongoing tasks that never stop (like robot control).
- TD can learn good strategies (policies) while still exploring random actions to discover better options.
- TD adjusts predictions based on how wrong they were – the bigger the surprise, the bigger the update.
- TD forms the foundation for popular methods like Q-learning and SARSA that power game AI and robotics.
- TD handles delayed rewards by gradually assigning credit to the right earlier actions.
- TD works with function approximation (like neural nets) to handle complex, real-world problems.
- TD balances between using what it knows works (exploitation) and trying new things (exploration).
- TD is more efficient than older methods that required complete trial-and-error runs before learning.
- TD naturally handles problems where outcomes are uncertain or partially random.