Temporal-Difference Learning: A Bridge Between Monte Carlo and Dynamic Programming

Niranjan Deshpande

April 22, 2025

Abstract

Temporal-Difference (TD) learning stands as a central idea in reinforcement learning, combining the strengths of Monte Carlo (MC) methods and Dynamic Programming (DP). This document provides a comprehensive explanation of TD learning, drawing parallels and contrasts with MC and DP, and explains foundational concepts such as sampling, bootstrapping, TD prediction, and TD control. The aim is to provide an intuitive and mathematical understanding of how TD learning operates and why it plays a pivotal role in learning optimal policies.

Contents

1 Introduction

Temporal-Difference (TD) learning is a fundamental concept in reinforcement learning (RL), known for its hybrid nature that draws inspiration from both Monte Carlo (MC) methods and Dynamic Programming (DP). While MC learns entirely from experience and DP relies entirely on a model of the environment, TD combines the best of both worlds: learning from sampled experiences *without a model* (like MC) and *bootstrapping* from existing estimates (like DP).

To truly appreciate the significance of TD learning, it is essential to first recall the defining characteristics of Monte Carlo and Dynamic Programming approaches to the prediction problem in RL.

1.1 Dynamic Programming (DP)

Dynamic Programming requires a complete model of the environment, including the transition probabilities $P(s'|s,a)$ and reward function $R(s,a,s')$. DP algorithms use these to compute expected values over all possible next states and actions.

Key properties of DP:

- Requires full knowledge of the environment.
- Performs *full backups* using expectations over all next states.
- Updates are performed *after every step*, but using the **model**.

The Bellman Expectation Equation used in DP for policy evaluation is:

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s'} P(s'|s,a) \left[R(s,a,s') + \gamma v_\pi(s')\right]$$

1.2 Monte Carlo (MC) Methods

Monte Carlo methods, in contrast, learn value functions based purely on experience. They sample complete episodes from the environment and use the **actual return** from each episode to update value estimates.

Key properties of MC:

- Does not require a model of the environment.
- Uses *sample returns* to update values.
- Must wait until the **end of the episode** to perform updates.

Monte Carlo prediction updates value estimates using the return G_t from a state:

$$V(s_t) \leftarrow V(s_t) + \alpha \left[G_t - V(s_t)\right]$$

1.3 Why Temporal-Difference Learning?

Temporal-Difference (TD) learning was introduced to bridge the gap between the two classic approaches to reinforcement learning: Monte Carlo (MC) and Dynamic Programming (DP). TD methods inherit strengths from both paradigms:

- **Sampling like Monte Carlo:** TD methods learn directly from actual interaction with the environment. They do not require knowledge of the transition dynamics $P(s'|s, a)$. Instead, they rely on **sampled episodes**, just like Monte Carlo methods. This allows learning to happen in unknown or complex environments through raw experience.
- **Bootstrapping like Dynamic Programming:** Unlike Monte Carlo methods, TD methods do not need to wait for the end of an episode. They update estimates using a **one-step lookahead** — incorporating the immediate reward and the estimated value of the next state. This approach, known as **bootstrapping**, enables fast and incremental updates.

Sampling: This refers to the process of learning from observed state transitions of the form:

$$(s_t, a_t, r_{t+1}, s_{t+1})$$

Instead of using expected values over all possible future outcomes (as in DP), TD methods use these actual samples to drive learning.

Bootstrapping: The term *bootstrapping* means to lift or improve oneself using one's own resources. In TD learning, it refers to improving an estimate based on other current estimates, rather than waiting for the final true outcome. Specifically, TD learning updates the value of the current state using the estimated value of the next state.

This concept will be explored in depth in the next section, where we introduce the idea of the **TD error** — a central signal that quantifies the difference between the current estimate and the bootstrapped target.

2 The General Learning Rule in TD Methods

2.1 A Fundamental Principle of Learning

At the heart of many learning algorithms — including Temporal-Difference (TD) methods — lies a simple but powerful idea:

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

This is a general **incremental learning rule**, where:

- $\alpha \in (0, 1]$ is the **learning rate**,
- The **Target** is a better estimate of what the value should be,
- The difference Target − Old Estimate is the **error signal**.

Key Intuition: You already have a belief about something (e.g., how good a state is), and you just received new information that helps refine it. You don't completely replace the old value — instead, you nudge it a little bit toward the new insight.

This idea is used in many areas of machine learning and control theory.

2.2 Application to TD Learning

In Temporal-Difference learning, the quantity being estimated is the value of a state (or a state-action pair).

The general TD learning rule becomes:

$$V(s_t) \leftarrow V(s_t) + \alpha \left(\text{Target} - V(s_t)\right)$$

Now, the central question becomes: **What is the target?**

This leads us to the concept of the **TD target**, and the difference between the target and current value is what we define as the **TD error**.

We explore this in detail in the next section.

3 Understanding the Temporal-Difference (TD) Error

3.1 What Are We Trying to Estimate?

In reinforcement learning, the core goal of prediction is to estimate the expected return from a state under a given policy π. This expected return is formally defined as the value function:

$$v_\pi(s_t) = \mathbb{E}_\pi \left[G_t \mid s_t \right] = \mathbb{E}_\pi \left[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots \mid s_t \right]$$

By the recursive structure of the Bellman expectation equation, this is also equivalent to:

$$v_\pi(s_t) = \mathbb{E}_\pi \left[r_{t+1} + \gamma v_\pi(s_{t+1}) \mid s_t \right]$$

However, in practice, we do not know v_π and must instead build an estimate, denoted $V(s)$, based on experience.

3.2 TD Target: Bootstrapped Estimate of Return

Instead of waiting for the full return G_t at the end of an episode, TD methods construct a one-step estimate of that return:

$$\text{TD Target} = r_{t+1} + \gamma V(s_{t+1})$$

This target uses:

- r_{t+1}: the immediate reward observed after taking action a_t,
- $V(s_{t+1})$: the current estimate of the value of the next state.

This is a **bootstrapped estimate** because it uses the current estimate $V(s_{t+1})$ rather than the actual return from that state.

3.3 What is the TD Error?

The **Temporal-Difference (TD) error** is the difference between the TD target (our new, better guess) and our current estimate $V(s_t)$. It is defined as:

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

This measures how surprised we are by the new experience — if the TD error is large, it means the previous estimate $V(s_t)$ was significantly off and needs to be adjusted.

3.4 Analogy: Restaurant Rating with Learning Rule

Imagine you are trying to predict how much your friend will like a new restaurant.

Initially, you guess they'll rate it a 6/10. After trying the starter, your friend texts you: "So far it's great — if things stay like this, I'd say 8/10!"

Rather than replacing your original guess (6) with this new estimate (8), you apply a slight adjustment using the learning rule:

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

In this case:

- Old Estimate = 6,
- Target = 8 (friend's partial feedback),
- Learning rate $\alpha = 0.5$ (say).

Then:

$$\text{New Estimate} = 6 + 0.5 \cdot (8 - 6) = 6 + 1 = 7$$

So you revise your prediction to 7/10 — a balance between your prior belief and the new evidence.

This is exactly what Temporal-Difference learning does:

- It uses recent feedback (reward),
- And the current estimate of what's next (value of the next state),
- To incrementally improve the value of earlier states — without waiting for the final outcome.

Analogy Concept	TD Concept
Initial guess of the restaurant	$V(s_t)$: Estimate of how good the current state is before receiving new feedback
Friend tastes starter	Agent takes an action and observes an immediate reward
Friend says "so far, 8/10"	TD target: $r_{t+1} + \gamma V(s_{t+1})$, an improved estimate based on reward + next state
You revise your original guess	Value update: $V(s_t) \leftarrow V(s_t) + \alpha \cdot \delta_t$
You didn't wait for final score	TD updates occur before the full episode ends — using bootstrapped estimates

Table 1: Mapping the restaurant rating analogy to TD learning concepts

4 TD Prediction: The TD(0) Algorithm

4.1 Problem Setup

In the prediction problem, the goal is to estimate the value function $v_\pi(s)$ — the expected return when starting from state s and following a given policy π.

Formally:

$$v_\pi(s) = \mathbb{E}_\pi \left[G_t \mid s_t = s \right]$$

Unlike Monte Carlo methods, which wait for an entire episode to end to calculate G_t, TD(0) uses only the first reward and the estimated value of the next state to incrementally update the value of s_t.

4.2 TD(0) Update Rule

The TD(0) method performs the following update after each step:

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

Where:

- $V(s_t)$ is the current estimate of the value of state s_t,
- r_{t+1} is the reward received after taking action a_t,
- $V(s_{t+1})$ is the estimated value of the next state,
- α is the learning rate,
- γ is the discount factor.

This update is applied after every transition, making the algorithm online and incremental.

4.3 Pseudocode: Tabular TD(0)

Algorithm: TD(0) for Policy Evaluation

Input: Policy π, discount factor $\gamma \in [0, 1]$, step size $\alpha \in (0, 1]$
Initialize $V(s)$ arbitrarily for all $s \in \mathcal{S}$
Repeat (for each episode):
  Initialize s
  Repeat (for each step of episode):
    Take action $a \sim \pi(\cdot|s)$; observe reward r, next state s'
    Update:
      $V(s) \leftarrow V(s) + \alpha[r + \gamma V(s') - V(s)]$
    $s \leftarrow s'$
  until s is terminal

4.4 Remarks

- TD(0) uses only the most recent experience and the value estimate of the next state to update the current state.
- This enables **faster learning** compared to MC in many environments, especially continuing tasks.
- It does not require a model of the environment.
- TD(0) can be used in both episodic and continuing environments.

4.5 Extensions of TD(0): Multi-Step TD and TD(λ)

Motivation: While TD(0) updates value estimates based only on a single step into the future, it is often beneficial to use more future rewards when forming the target. This leads to multi-step TD methods, including TD(n) and TD(λ), which generalize the update mechanism of TD(0).

TD(n): Multi-Step Temporal Difference Learning

TD(n) methods look ahead n steps before bootstrapping:

$$G_t^{(n)} = r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^{n-1} r_{t+n} + \gamma^n V(s_{t+n})$$

The value update is then:

$$V(s_t) \leftarrow V(s_t) + \alpha \left[G_t^{(n)} - V(s_t) \right]$$

Trade-Off:

- Small $n \rightarrow$ More bootstrapping, faster updates, but less future reward info.
- Large $n \rightarrow$ Closer to Monte Carlo, more accurate long-term return, but slower convergence and higher variance.

TD(λ): Eligibility Traces and Mixed Returns

TD(λ) combines all n-step returns using an exponentially decaying weighting factor $\lambda \in [0, 1]$:

$$G_t^{(\lambda)} = (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_t^{(n)}$$

Rather than storing full episodes to compute these returns, TD(λ) is implemented efficiently using **eligibility traces**, which track how recently and frequently each state has been visited.

Update Rule with Eligibility Trace:

$$V(s) \leftarrow V(s) + \alpha \cdot \delta_t \cdot e_t(s)$$

Where:

- $\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$ is the TD error,
- $e_t(s)$ is the eligibility trace for state s,
- $e_t(s)$ decays over time and increases when state s is visited.

Special Cases:

- TD(0) is a special case of TD(λ) with $\lambda = 0$.
- Monte Carlo prediction corresponds to TD(λ) with $\lambda = 1$.

Why TD(λ) Matters:

- TD(λ) allows a smooth trade-off between bias and variance.
- It's widely used in both tabular and function approximation settings.
- It forms the foundation of more advanced algorithms (e.g., TD(λ) with function approximation, actor-critic methods).

5 TD Control Methods

5.1 SARSA: On-Policy TD Control

SARSA (State–Action–Reward–State–Action) is an **on-policy Temporal-Difference (TD) control algorithm**. It estimates the action-value function $Q(s, a)$ under the same policy that the agent uses to act — usually an ϵ-greedy policy that occasionally explores. Unlike off-policy methods, SARSA learns from the actual actions taken, including those resulting from exploration.

Objective

SARSA aims to learn the expected return starting from state s, taking action a, and thereafter following the current policy π. That is:

$$Q(s, a) \approx \mathbb{E}[G_t \mid s_t = s, a_t = a]$$

Update Rule

SARSA updates its Q-values after observing the following transition:

$$(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$$

The update rule is:

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

where:

- α is the learning rate,
- γ is the discount factor,
- a_{t+1} is chosen using the same behavior policy (e.g., ϵ-greedy).

Pseudocode (Tabular SARSA)

SARSA Algorithm
Initialize $Q(s, a)$ arbitrarily for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
For each episode:
Initialize state s
Choose action $a \sim \epsilon$-greedy$(Q(s, \cdot))$
Repeat (for each step of episode):
Take action a, observe reward r and next state s'
Choose next action $a' \sim \epsilon$-greedy$(Q(s', \cdot))$
Update: $Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma Q(s', a') - Q(s, a)\right]$
Set $s \leftarrow s'$, $a \leftarrow a'$
Until s is terminal

What Does "On-Policy" Mean in SARSA?

The term **on-policy** means that the agent uses the same policy:

- To select actions while interacting with the environment,
- And to update the value function based on those actions.

In SARSA, the agent follows a behavior policy such as ϵ-greedy and updates its Q-values using the actual next action it took. This includes both greedy and exploratory actions.

How Does ϵ-greedy Work? At any state s, the agent:

- Picks a random action with probability ϵ (exploration),
- Picks the action with the highest Q-value with probability $1 - \epsilon$ (exploitation).

Concrete Example: Suppose the agent is in state s_{t+1}, and its Q-values are:

$$Q(s_{t+1}, a_1) = 5$$

$$Q(s_{t+1}, a_2) = 2$$

$$Q(s_{t+1}, a_3) = 4$$

$$Q(s_{t+1}, a_4) = 3$$

If $\epsilon = 0.1$:

- 90% chance: pick a_1 (greedy)
- 10% chance: pick randomly from all 4 actions.

Now suppose the agent randomly chooses a_3. SARSA will update using:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma Q(s_{t+1}, a_3) - Q(s_t, a_t) \right]$$

Even though a_1 was the best action, SARSA reflects what actually happened. This makes SARSA more realistic and safer in environments where risky exploration must be acknowledged.

Key Characteristics

- **On-policy:** Learns the value of the policy actually followed (including exploratory moves).
- **Risk-aware:** If the policy includes randomness, SARSA incorporates that into learning.
- **Slower convergence** in some cases than Q-learning, but often leads to safer behavior.

5.2 Q-Learning: Off-Policy TD Control

Q-learning is an **off-policy Temporal-Difference (TD) control algorithm**. It aims to learn the optimal action-value function $Q^*(s, a)$, regardless of the policy actually used to explore the environment.

Unlike SARSA, Q-learning updates the Q-value of the current state-action pair by assuming that the agent will act optimally from the next state onward — even if it doesn't. This makes Q-learning an **off-policy method**, because the update assumes a greedy policy while the agent might be following an exploratory one.

Objective

Q-learning seeks to estimate the optimal value function:

$$Q^*(s, a) = \mathbb{E}\left[r_{t+1} + \gamma \max_{a'} Q^*(s_{t+1}, a') \mid s_t = s, a_t = a\right]$$

This is done by interacting with the environment and updating the Q-values using the maximum value of the next state, regardless of the next action actually taken.

Update Rule

After observing a transition $(s_t, a_t, r_{t+1}, s_{t+1})$, the Q-learning update rule is:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\left[r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t)\right]$$

Where:

- α is the learning rate,
- γ is the discount factor,
- $\max_{a'} Q(s_{t+1}, a')$ is the greedy estimate of the next state's value.

Pseudocode (Tabular Q-Learning)

Q-Learning Algorithm
Initialize $Q(s, a)$ arbitrarily for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
For each episode:
Initialize state s
Repeat (for each step of episode):
Choose action $a \sim \epsilon$-greedy$(Q(s, \cdot))$
Take action a, observe reward r, next state s'
Update: $Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma \max_{a'} Q(s', a') - Q(s, a)\right]$
Set $s \leftarrow s'$
Until s is terminal

Why is Q-Learning Off-Policy?

Although the agent often behaves using an ϵ-greedy policy, the update is performed assuming it will act greedily from the next state onward.

This means:

- The agent may explore in s_{t+1}, but the update still assumes it chooses the **best** action.
- The learning is disconnected from the actual trajectory taken — hence, off-policy.

Example:

Suppose the agent reaches state s_{t+1}, with Q-values:

$$Q(s_{t+1}, a_1) = 5$$

$$Q(s_{t+1}, a_2) = 2$$

$$Q(s_{t+1}, a_3) = 4$$

$$Q(s_{t+1}, a_4) = 3$$

Even if the agent (exploring with $\epsilon = 0.1$) takes action a_3, the update uses the **maximum Q-value** among all actions, i.e., $Q(s_{t+1}, a_1) = 5$.

So the update will be:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \cdot 5 - Q(s_t, a_t)\right]$$

Q-learning is thus more **optimistic** — it always assumes the best possible future behavior.

Key Characteristics

- **Off-policy:** Updates are based on the greedy policy, not the behavior policy.
- **Optimistic:** Learns the value of the best possible actions — even if not followed.
- **Faster convergence** to the optimal policy in many cases — but can be riskier due to ignoring exploration behavior in learning.

5.3 Comparison: SARSA vs. Q-Learning

Both SARSA and Q-Learning are Temporal-Difference (TD) control algorithms that use experience to update action-value functions. They often share the same behavior policy (e.g., ϵ-greedy), but differ in how they use the outcomes of that policy to perform updates.

Behavior Policy: ϵ-greedy (Same in Both)

In both algorithms, the agent usually selects actions using an ϵ-greedy policy:

- With probability ϵ, a random action is chosen (exploration).
- With probability $1 - \epsilon$, the action with the highest Q-value is chosen (exploitation).

So yes — both behave greedily most of the time. But the **update step** is where the key difference lies.

SARSA (On-policy)

- Uses the actual action taken in the next state a_{t+1}, even if it's exploratory.
- Update Rule:

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

- a_{t+1} is sampled using the same current policy (e.g., ϵ-greedy).
- This includes randomness in learning and reflects what the current policy actually does.

Q-Learning (Off-policy)

- Assumes the agent acts optimally in the next state.
- Update Rule:

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

- Always uses the greedy action for updating, regardless of the actual action taken.
- Learns the value of the optimal policy, not necessarily the one being followed.

Comparison Table

Aspect	**SARSA (On-Policy)**	**Q-Learning (Off-Policy)**
Policy Type	On-policy (updates using actual action taken)	Off-policy (updates assuming optimal action)
Update Target	$r + \gamma Q(s_{t+1}, a_{t+1})$	$r + \gamma \max_a Q(s_{t+1}, a)$
Behavior Policy	ϵ-greedy	ϵ-greedy
Update Based On	Action taken (including exploration)	Best possible action (greedy)
Learns Value of	Behavior policy (possibly suboptimal)	Optimal policy
Learning Style	Safe, cautious	Aggressive, optimistic
Convergence To	Behavior policy's performance	Optimal policy performance

Table 2: Comparison of SARSA and Q-Learning

Pseudocode Snapshot

SARSA:

```
# Step 1: choose action a_t using epsilon-greedy
# Step 2: take a_t, observe r, s_{t+1}
# Step 3: choose next action a_{t+1} using epsilon-greedy
# Step 4: Q-update:
Q[s_t][a_t] += alpha * (r + gamma * Q[s_{t+1}][a_{t+1}] - Q[s_t][a_t])
```

Q-Learning:

```
# Step 1: choose action a_t using epsilon-greedy
# Step 2: take a_t, observe r, s_{t+1}
# Step 3: Q-update:
Q[s_t][a_t] += alpha * (r + gamma * max(Q[s_{t+1}]) - Q[s_t][a_t])
```

Analogy: Learning to Drive

- **SARSA:** Like learning to drive by evaluating what actually happened on the road — including wrong turns and missed exits.
- **Q-Learning:** Like learning to drive by imagining you always took the best path, even when you didn't.

When to Use Which?

- Use **SARSA** if safe learning is critical, especially in environments where exploration can be risky.
- Use **Q-Learning** when long-term performance is the goal and exploration risks are acceptable.