Reinforcement Learning

Lecture 6: Temporal-Difference Reinforcement Learning

Niranjan Deshpande

Minor in AI, IIT Ropar

April 22, 2025

Today's Agenda

1. Why TD Learning
2. Temporal Difference Learning
3. TD Algorithms

Why Temporal-Difference Learning?

- **Monte Carlo (MC):** Learns from complete episodes — must wait until the end.
- **Dynamic Programming (DP):** Requires a model of the environment — impractical for many real problems.
- **TD Learning:**
  - Learns directly from raw experience — like MC.
  - Does not need a model — like MC.
  - Updates after each step — like DP.

TD Learning = Sample-based like MC + Bootstrapped like DP

MC: Full Episode

Wait → Wait → Wait → Update

DP: Uses Model

Model → Evaluate → Update

TD: Online Update

Step → Observe → Update

MC vs DP vs TD: Key Comparison

Aspect	**MC**	**DP**	**TD**
Model required?	No	Yes	No
Learns from samples?	Yes	No (uses full model)	Yes
Bootstraps?	No	Yes	Yes
Updates after every step?	No (waits for episode end)	Yes	Yes
Converges with incomplete episodes?	No	Yes	Yes
Policy evaluation?	Yes	Yes	Yes
Suitable for online learning?	No	No	Yes

The General TD Learning Rule

Core idea: Temporal-Difference methods update estimates using other learned estimates.

Generic TD update rule:

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

- α: learning rate (how much we trust the new info).
- Target: a short-term approximation of long-term return.
- Works for both **value functions** and **Q-functions**.

"Update using what you just experienced and what you currently believe."

Understanding TD Error

TD Error measures how "surprised" the agent is by what it just observed:

$$\delta_t = \underbrace{r_{t+1} + \gamma V(s_{t+1})}_{\text{Better estimate (TD Target)}} - \underbrace{V(s_t)}_{\text{Current estimate}}$$

Why is this useful?

If TD Error $= 0$, then our current guess is already good.
If TD Error $\neq 0$, we use it to improve our estimate.

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

TD Algorithms Overview

Temporal-Difference methods can be divided into two broad categories:

- **TD Prediction:** Estimate the value function $V^{\pi}(s)$ for a given policy π.
  - TD(0)
  - TD(λ)
- **TD Control:** Learn the optimal policy π^{*} by improving the action-value function $Q(s, a)$.
  - SARSA (On-policy)
  - Q-Learning (Off-policy)
  - Expected SARSA
  - Double Q-Learning

All TD methods:

- Learn from experience
- Do not require a model of the environment
- Use bootstrapping — they update based on current estimates

TD Prediction
TD(0) – One-Step Learning

Goal: Estimate the value function $V^\pi(s)$ for a given policy π using experience.

TD(0) is the simplest TD prediction method:

- Updates value estimates after **every single step**.
- Uses the immediate reward and the value of the next state.
- Does not wait until the end of an episode.

Update Rule:

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

where:

- $r_{t+1} + \gamma V(s_{t+1})$: TD target (bootstrapped estimate)
- $V(s_t)$: current estimate
- δ_t: TD error = target - estimate

Tabular TD(0) for estimating v_π

Input: the policy π to be evaluated
Algorithm parameter: step size $\alpha \in (0, 1]$
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(\mathit{terminal}) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    $A \leftarrow$ action given by π for S
    Take action A, observe R, S'
    $V(S) \leftarrow V(S) + \alpha\big[R + \gamma V(S') - V(S)\big]$
    $S \leftarrow S'$
  until S is terminal

TD(0) uses a one-step lookahead to update value estimates. But there are more general methods that strike a balance between TD and Monte Carlo.

1. **TD(λ)** – Eligibility Traces

   - Combines ideas from TD(0) and Monte Carlo.
   - Uses multiple-step returns (1-step, 2-step, ..., full return).
   - Controlled by $\lambda \in [0, 1]$:
     - $\lambda = 0$: becomes TD(0)
     - $\lambda = 1$: approximates Monte Carlo

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

- Use expected value of the next state under a stochastic policy.
- Reduce variance by averaging over all actions.

These generalizations:

- Improve learning stability
- Create smoother trade-offs between bias and variance

SARSA stands for:

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

- It is an **on-policy** TD control method.
- Learns the action-value function $Q(s, a)$ while following the same policy used for acting.
- Usually uses an ϵ-greedy exploration strategy.

Key Idea: SARSA updates Q-values based on the action the agent actually took — including random actions due to exploration.

Goal: Learn $Q^{\pi}(s, a)$, the expected return of the current behavior policy.

SARSA – Update Rule and Learning Process

SARSA Update Rule:

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

Learning loop:

- Choose action a_t from s_t using ϵ-greedy.
- Take a_t, observe r_{t+1}, s_{t+1}.
- Choose a_{t+1} from s_{t+1} using same policy.
- Update $Q(s_t, a_t)$ using the rule above.
- Repeat.

Why On-Policy? The update uses the actual action a_{t+1} taken by the current policy — even if it was random.

SARSA - Algorithm

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(\mathit{terminal}, \cdot) = 0$

Loop for each episode:
  Initialize S
  Choose A from S using policy derived from Q (e.g., ε-greedy)
  Loop for each step of episode:
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ε-greedy)
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma Q(S', A') - Q(S, A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

Q-Learning is an **off-policy** TD control algorithm.

Goal: Learn the optimal action-value function $Q^*(s, a)$, regardless of how the agent behaves during learning.

Key Idea: Even if the agent explores (e.g., using ϵ-greedy), it always updates its Q-values as if it acted optimally.

- Ignores what action the agent actually took in the next state.
- Instead, it assumes the best possible action was taken.
- This makes it more optimistic — converges to the optimal policy.

Used widely in RL: Simple, powerful, and converges under mild conditions.

Q-Learning – Update Rule and Learning Process

Q-Learning Update Rule:

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

Learning loop:

- Choose action a_t from s_t using ϵ-greedy.
- Take a_t, observe r_{t+1}, s_{t+1}.
- Compute the best next action: $\max_a Q(s_{t+1}, a)$.
- Update $Q(s_t, a_t)$ using the rule above.
- Repeat.

Why Off-Policy? It updates using the *greedy* action, not the action actually taken during exploration.

Q-Learning - Algorithm

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

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
    $S \leftarrow S'$
  until S is terminal

SARSA vs Q-Learning
On-Policy vs Off-Policy

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

ϵ-Greedy Action Selection

- With probability ϵ: choose a random action (exploration)
- With probability $1 - \epsilon$: choose the action with the highest Q-value (exploitation)

Pseudocode:

```
def epsilon_greedy(Q, state, epsilon):
    if random() < epsilon:
        return random_action()
    else:
        return argmax_a Q[state][a]
```

Used in: Both SARSA and Q-Learning during action selection

Code Comparison: SARSA vs Q-Learning

SARSA: On-Policy

```
a_t = epsilon_greedy(Q, s_t)
s_t1, r = env.step(a_t)
a_t1 = epsilon_greedy(Q, s_t1)

Q[s_t][a_t] += alpha * (
    r + gamma * Q[s_t1][a_t1]
    - Q[s_t][a_t]
)
```

Q-Learning: Off-Policy

```
a_t = epsilon_greedy(Q, s_t)
s_t1, r = env.step(a_t)

Q[s_t][a_t] += alpha * (
    r + gamma * max(Q[s_t1])
    - Q[s_t][a_t]
)
```

Note: In SARSA, the next action is sampled and used in the update. In Q-learning, we assume the next action is always the best one.

Common ϵ-greedy Policy (used in both):

Summary: Temporal-Difference Learning

- **TD Learning** combines the strengths of:
  - **Monte Carlo:** Learns from raw experience (no model)
  - **Dynamic Programming:** Bootstraps estimates
- **TD(0)**: One-step prediction algorithm $V(s_t) \leftarrow V(s_t) + \alpha\left[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)\right]$
- **SARSA (On-Policy):** Learns from the action actually taken, even if it's random.
- **Q-Learning (Off-Policy):** Learns assuming the best possible action is always taken.
- ϵ-**greedy**: Balances exploration and exploitation in both methods.

TD = Learn while you go. Improve while you explore.