Ludovic Bergeron
(盧維克)
ID :413853036

Temporal Difference Learning

1. Introduction

In a world where intelligent systems increasingly shape decision-making processes, reinforcement learning has become essential for solving complex problems involving sequential actions. Among its techniques, **Temporal Difference Learning (TD Learning)** stands out as a robust and adaptive approach. By blending the sampling nature of Monte Carlo methods with the iterative updates of dynamic programming, TD Learning enables efficient learning through incremental adjustments.

Unlike Monte Carlo methods, which update only after an episode ends, TD Learning refines state or action values step by step using differences between successive predictions, a process known as bootstrapping. This ability to learn progressively makes TD Learning particularly effective for dynamic environments with uncertain rewards and transitions or tasks of significant length.

Notable TD Learning algorithms, such as **SARSA** and **Q-Learning**, illustrate its versatility, powering applications ranging from strategic games like Go and Backgammon to robotics and logistics optimization. For instance, in strategic games, TD Learning allows agents to continuously improve strategies through iterative play, while in robotics, it enables rapid adaptation to new environments without prior knowledge.

This report explores the theoretical foundations of TD Learning, examines its algorithmic structure with a practical implementation example, and concludes with a discussion of its advantages, limitations, and future research directions.

2. Mathematics Involved

Key Concepts:

TD Learning relies on core mathematical ideas from dynamic programming and probability theory. The method is built on the concept of **value functions**, which estimate the long-term cumulative reward expected from a given state or state-action pair. TD Learning uses **bootstrapping** to update values incrementally, combining current estimates with next-state information without waiting for the episode's conclusion. Additionally, TD Learning incorporates the **Bellman equation**, a fundamental tool in reinforcement learning, to iteratively refine these estimates.

The algorithm also relies on **discounting future rewards** using a factor γ , which balances the importance of immediate versus long-term rewards. This ensures that the learning process remains computationally efficient while reflecting the diminishing impact of rewards that are farther in the future.

Core Equation:

The core of TD Learning is the **TD update rule**, which adjusts the value of a state based on the **TD error**, the difference between the current value estimate and the updated value, let's start with the classic TD Learning, or TD(0) :

$$V(s) \leftarrow V(s) + \alpha[r + \gamma V(s') - V(s)]$$

Where :

- $V(s)$: Current estimated value of state s.
- α : Learning rate, which determines how much the new information influences the existing estimate.
- r : Reward received after transitioning from state s to s'.
- γ : Discount factor, which balances the importance of immediate rewards versus future rewards (ranges between 0 and 1).
- $V(s')$: Estimated value of the next state s'.

Explication of the formula :

1. $r + \gamma V(s')$: **The Target Value**
   This term represents the **updated estimate** of the value of state s. It considers the immediate reward r and the discounted value of the next state $V(s')$, reflecting both short-term and long-term rewards.
2. $V(s)$: **Current Estimate**
   The current prediction of the value of state s. This is the baseline against which the target value is compared.
3. $r + \gamma V(s') - V(s)$: **The TD Error**
   This measures the difference between the current estimate ($V(s)$) and the target value ($r + \gamma V(s')$). It indicates how much the current estimate should be adjusted to better match the target.
4. α: **Step Size**
   The learning rate α controls how much of the TD error is used to update $V(s)$.

   A small α slows down learning, making updates more conservative.

   A large α accelerates learning but can lead to instability if it overshoots.

   How It Works:

   The TD update rule incrementally adjusts $V(s)$ by reducing the TD error, bringing the current estimate closer to the target value. This incremental approach allows the agent to refine its value estimates step by step as it interacts with the environment. Over time, as the agent observes more transitions, $V(s)$ converges to the true expected value of the state under the given policy.

VARIOUS EXTENSIONS OF TD LEARNING:

Transition to SARSA and Q-Learning (Contextualizing $Q(s, a)$):

SARSA and Q-Learning are extensions of **TD Learning**, they introduce the concept of **actions** into the learning process. Instead of evaluating only the states $V(s)$, they focus on evaluating **state-action pairs** $Q(s, a)$. This allows the algorithms to learn an **optimal policy** by choosing the actions that maximize future rewards.

Update Rules:

1. **SARSA (on-policy):**
   The update rule for SARSA is:

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)]$$

Here :

- The update depends on the **action actually chosen** (a') in the next state s'.
- SARSA learns values $Q(s, a)$ based on the policy currently being followed by the agent, making it **on-policy**.

2. **Q-Learning (off-policy):**
   The update rule for Q-Learning is:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma max(a')Q(s',a') - Q(s,a)]$$

Here :

- The update is based on the **theoretical optimal action** $(\max\,(a')Q(s',a'))$ in the next state s', regardless of the action actually taken.
- Q-Learning learns an **optimal policy** directly independently of the agent's making it **off-policy**.

3. Algorithm Mechanics

How It Works:

TD Learning works by updating the estimated value of states incrementally, based on the difference between the current estimate and a new target value. There are two main approaches to extending TD Learning for state-action pairs: **SARSA** and **Q-Learning**. Here is a step-by-step explanation of TD Learning, along with how these extensions work:

1. **Initialization**:
   - Start with an initial estimate of the value for all states ($V(s)$) for classic TD Learning) or all state-action pairs ($Q(s,a)$ for SARSA and Q-Learning).
   - Typically, these values are initialized to zero or random numbers.
2. **Interaction with the Environment**:
   - The agent begins in an initial state s.
   - It takes an action a (following a specific policy) and observes a reward r and the next state s'.
3. **Update Rule**:
   - For **TD(0)**, the agent updates the value of the current state s using the **TD update rule**: $V(s) \leftarrow V(s) + \alpha[r + \gamma V(s') - V(s)]$
   - For **SARSA**, the agent updates the value of the current state-action pair $Q(s,a)$: $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma Q(s',a') - Q(s,a)]$ Here, a' is the action actually taken in the next state s', based on the current policy.
   - For **Q-Learning**, the agent updates the value of the current state-action pair $Q(s,a)$: $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma max\ a'Q(s',a') - Q(s,a)]$ Here, $max\ a'Q(s',a')$ represents the value of the **best possible action** in the next state s', regardless of the action actually taken.
4. **Repeat**:
   - The agent moves to the next state s', repeats the process, and continues until the episode ends or the task is complete.

By repeating this process over multiple episodes, the agent learns more accurate estimates of $V(s)$ for classic TD Learning, or $Q(s,a)$ for SARSA and Q-Learning. This helps it make better decisions over time.

The learning process of the Q-Learning agent demonstrates an initial struggle to achieve high rewards due to exploration. Over time, the agent converges to an optimal policy that consistently yields positive rewards. This gradual improvement is typical of reinforcement learning algorithms, which refine performance through trial and error.

Strengths and Limitations:

Strengths:

1. **Efficiency**:
    - TD Learning, SARSA, and Q-Learning all update values incrementally, meaning they do not need to wait until the end of an episode (as in Monte Carlo methods). This makes them faster and suitable for environments with long or continuous episodes.
2. **Adaptability**:
    - **SARSA** learns directly from the agent's actual behavior (on-policy), which can be safer in uncertain environments.
    - **Q-Learning** learns the optimal policy independently of the agent's behavior (off-policy), which allows it to find the best actions faster.

Limitations:

1. **Dependence on Parameters**:
    - The performance of all these methods depends heavily on the choice of learning rate α and discount factor γ. Poor parameter settings can lead to slow convergence or instability and may not allow enough time to find the optimal solution.
2. **Trade-offs in Policy Learning**:
    - **SARSA** may converge more slowly to the optimal policy because it focuses on actions actually taken, which are influenced by exploration.
    - **Q-Learning** can be unstable in stochastic environments if exploration is not well-controlled, as it assumes the agent always takes the optimal action.

4. Example Implementation

Code Snippet

Here is the implementation of the **Q-Learning algorithm** in a **4x4 Grid World environment**. The goal is for the agent to learn an optimal policy to reach the terminal state efficiently while minimizing penalties.

Pseudocode for Q-Learning in a 4x4 Grid World

1. Initialize the environment

- Define a **4x4 grid** with 16 states.
- Define 4 possible **actions**:
    - **UP, DOWN, LEFT, RIGHT.**
- Set the **goal state** as the bottom-right corner (state 15).
- Create a **Q-table** of size (16, 4) initialized to 0.

2. Set the parameters

- **Learning rate** (α = 0.8 (controls how much new information overrides old).
- **Discount factor** (γ = 0.95 (importance of future rewards).
- **Exploration rate** (ϵ = 1.0 (starts with full exploration).

- **Decay rate** = 0.99 (reduces exploration over time).
- **Epochs** = 1000 (number of training episodes).

3. Q-Learning Algorithm

For each episode (epoch):

1. Start at the initial state (state 0).
2. Set **total_reward = 0**.
3. **Repeat until the goal state is reached**:
   a. **Choose an action** using the epsilon-greedy strategy:
      - With probability ϵ\epsilonϵ, choose a random action (**exploration**).
      - Otherwise, choose the action with the highest Q-value (**exploitation**).

b. **Move to the next state** based on the chosen action.
c. **Receive a reward**:
   - +1 if the goal state is reached (reward).
   - −0.1 for each additional step (penalty to encourage efficiency).

d. **Update the Q-value** for the current state-action pair:

$$Q(s,a) = Q(s,a) + \alpha[r + \gamma max\ Q(s',a') - Q(s,a)]$$

e. Update the current state $s \leftarrow s'$.
f. Add the reward to **total_reward**.

4. Reduce ϵ to decrease exploration (decay factor).
5. Save **total_reward** for this episode.

Visualizing Results

- Plot the **total rewards** over all episodes to see the learning performance.

Comments:

- The Q-table is a matrix where each row represents a state and each column represents an action.
- The algorithm learns by updating the Q-values based on **rewards** and **future Q-values**.
- The graph at the end shows how rewards improve over episodes as the agent learns the optimal policy.

  Visualization using matplotlib.pyplot :

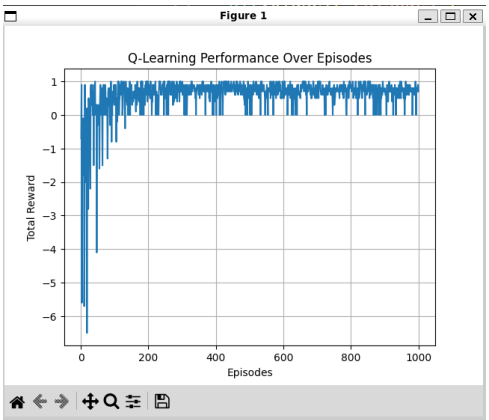
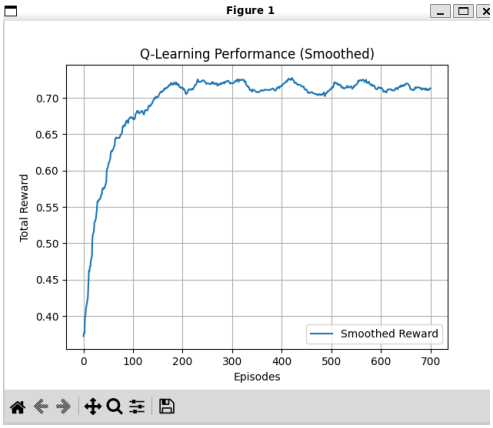


Figure 1 : **Q-Learning Performance Over Episodes**

This graph shows the evolution of total rewards as the agent learns. Initially, the rewards are negative as the agent explores the environment, but they improve over time as the agent learns the optimal policy.

Figure 2 : **Smoothed Q-Learning Performance**

This graph uses a moving average to smooth the total rewards over episodes. It provides a clearer view of the convergence of the learning process, with the rewards stabilizing near optimal values.



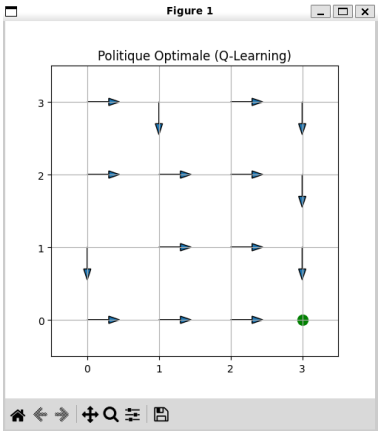


Figure 3 : **Optimal Policy Visualization**

This 4x4 grid illustrates the optimal policy learned by the Q-Learning algorithm. The arrows represent the best action to take in each state, and the terminal state (goal) is marked with a green dot.

Summary of Results

The Q-Learning algorithm successfully learned the optimal policy for navigating the 4x4 Grid World.

- The total rewards converged to stable values, as shown in the graphs.
- The optimal policy directs the agent efficiently toward the goal state while minimizing penalties.

5. Conclusion

Through this project, we explored Temporal Difference Learning with the example **Q-Learning algorithm**. TD learning is a fundamental method in reinforcement learning that enables an agent to learn an optimal policy by interacting with its environment. The algorithm relies on the **Bellman equation** to iteratively update Q-values, which represent the expected rewards for each state-action pair. By balancing **exploration** and **exploitation**, the agent gradually improves its decisions, converging towards an optimal policy. The **discount factor (γ)** and the **learning rate (α)**, plays a critical role in determining the speed and stability of convergence. Building on the insights gained from this exploration, the following section delves into key questions and potential avenues for enhancing the performance and applicability of Q-Learning in more complex and dynamic environments.

Are there ways to enhance or accelerate Q-Learning by integrating it with advanced techniques, and what role could methods like Deep Learning play in this process?