A Comprehensive Analysis of Core Reinforcement Learning Algorithms: From Dynamic Programming to Model-Free Learning

Taha Majlesi

Part I
Model-Free Learning from Experience

The dynamic programming methods discussed in the previous section are powerful but suffer from a major limitation: they require a perfect and complete model of the environment's dynamics ($P(s'|s,a)$). In most real-world problems, such a model is not available. This necessitates a shift from **planning** to **learning**, where the agent learns optimal behavior directly from its interactions with the environment.

1 The Planning vs. Learning Distinction

The fundamental difference between planning and learning in reinforcement learning hinges on the availability of the environment's model.

1.1 Planning (Model-Based)

In this paradigm, the agent has access to the model. [11, 19] Algorithms like Value Iteration and Policy Iteration leverage this model to compute value functions and policies. They can perform "thought experiments" or simulations, considering the outcomes of actions without ever needing to take them in the real world. For example, a chess program with a perfect model (the rules of chess) can plan by exploring future move sequences in its internal simulation. [11, 22]

1.2 Learning (Model-Free)

In this paradigm, the model is unknown. [11, 19] The agent cannot predict the next state or reward. To learn, it must interact with the environment to collect experience—sequences of states, actions, and rewards. [1] This experience serves as the training data from which the agent estimates value functions and improves its policy. An agent learning to play a video game for the first time without instructions is a classic example of model-free learning.

The Q-value update equation on page 26 of the presentation encapsulates this distinction perfectly:

$$Q^*(s,a) = \sum_{s'} P(s'|s,a) \Big(R(s,a,s') + \gamma \max_{a'} Q^*(s',a') \Big) \tag{1}$$

To execute this update, a planning algorithm needs $P(s'|s,a)$ and $R(s,a,s')$. A learning algorithm, lacking this knowledge, must find alternative ways to estimate the expected value on the right-hand side, typically by sampling from experience.

It is important to note that this distinction is not a strict dichotomy but rather a spectrum. An agent can use model-free methods to learn a model of the environment from experience, and then use that learned model for planning. This hybrid approach, known as **model-based reinforcement learning**, aims to combine the data efficiency of planning with the flexibility of learning in unknown environments.

2 Monte Carlo Prediction

Monte Carlo (MC) methods represent a straightforward approach to model-free learning. The core idea is to learn value functions by averaging the returns from sample episodes. [3, 4]

2.1 Core Principle and Key Characteristics

MC methods estimate the value of a state, $V^\pi(s)$, by running many complete episodes under policy π, recording the return G_t that follows each visit to state s, and then averaging these returns. By the law of large numbers, as the number of observed returns approaches infinity, this average will converge to the true expected value, $V^\pi(s)$. [1]

MC methods have several defining characteristics:

- **Model-Free:** They operate directly on sample episodes of states, actions, and rewards, requiring no knowledge of the environment's transition dynamics $P(s'|s, a)$. [3, 23]
- **Episodic Tasks:** Standard MC methods are defined only for episodic tasks, i.e., tasks that have a clear start and end. [4, 23] This is because the value update can only be performed after an episode is complete and the final return G_t is known.
- **No Bootstrapping:** MC updates for a state s are based on the entire sequence of rewards received until the end of the episode. The estimate for $V(s)$ is independent of the current value estimates of any other state. This contrasts sharply with DP methods, which bootstrap: they update the value of a state based on the estimated values of its successor states. [1, 4]

2.2 First-Visit vs. Every-Visit MC

The presentation introduces two variants of MC prediction, which differ in how they handle multiple visits to the same state within a single episode.

2.2.1 First-Visit MC

As detailed in the algorithm on page 29, when calculating the value for a state s, one only considers the return following the **first time** s was visited in each episode. [1, 30] If s is visited again later in the same episode, that visit is ignored for the purpose of averaging. The algorithm loops through each step of a generated episode backwards from $t = T - 1$ down to 0. For each step t, it calculates the return $G \leftarrow \gamma G + R_{t+1}$. It then checks if the state S_t has already appeared in the episode history up to that point $(S_0, \ldots, S_{t-1})$. If not, it appends the calculated return G to the list of returns for S_t and updates the value estimate $V(S_t)$ by averaging this list.

2.2.2 Every-Visit MC

As described on page 31, this method is simpler. It averages the returns for **every visit** to a state s in every episode. [1, 30] If a state is visited three times in one episode, three different return values (calculated from each of those time steps to the end of the episode) will be added to the list for that state.

Theoretically, first-visit MC is often preferred because the returns for each episode are independent and identically distributed (i.i.d.) samples of $V^\pi(s)$, making the estimates unbiased. [33] Every-visit MC estimates are slightly biased, but both methods are guaranteed to converge to the true value function as the number of visits approaches infinity. [1, 33]

3 Monte Carlo Control and Incremental Updates

The goal of RL is not just to evaluate a policy (prediction) but to find an optimal policy (control).

3.1 Monte Carlo Control

To achieve control without a model, an agent must estimate action-values, $Q(s, a)$, rather than state-values, $V(s)$. This is because improving a policy from V-values requires the model to perform the one-step lookahead ($\arg\max_a \sum P(\ldots)$), whereas improving from Q-values is model-free: one simply takes the action with the highest Q-value ($\arg\max_a Q(s, a)$). [3]

MC control follows the general pattern of GPI: it alternates between policy evaluation and policy improvement.

1. **Policy Evaluation:** Estimate $Q^\pi(s, a)$ by running many episodes under policy π and averaging the returns for every state-action pair.
2. **Policy Improvement:** Update the policy to be greedy with respect to the current Q-function: $\pi(s) \leftarrow \arg\max_a Q(s, a)$.

3.2 The Incremental Mean Update

A naive implementation of MC would require storing a list of all returns for each state and recalculating the average after every episode. This is computationally and memory-intensive. Pages 31 and 32 of the presentation introduce a more efficient **incremental update** mechanism. [32, 43]

3.2.1 Batch Update (from p. 31)

The standard average is calculated as:

$$V^\pi(s) = \frac{G(s)}{N(s)} \tag{2}$$

where $G(s)$ is the sum of all returns observed for state s and $N(s)$ is the number of visits.

3.2.2 Incremental Update (from p. 32)

This update rule allows the mean to be updated with each new return $G_{i,t}$ without storing past returns: [45, 46]

$$V^\pi(s) \leftarrow V^\pi(s) + \frac{1}{N(s)}(G_{i,t} - V^\pi(s)) \tag{3}$$

Here, $N(s)$ is the count of visits after the current one. This formula can be derived from the definition of the mean. Let V_N be the mean of N returns and G_N be the N^{th} return.

$$\begin{aligned} V_N &= \frac{1}{N}\sum_{k=1}^{N} G_k \\ &= \frac{1}{N}\left(G_N + \sum_{k=1}^{N-1} G_k\right) \\ &= \frac{1}{N}(G_N + (N-1)V_{N-1}) \\ &= \frac{1}{N}G_N + \frac{N-1}{N}V_{N-1} \\ &= V_{N-1} + \frac{1}{N}(G_N - V_{N-1}) \end{aligned}$$

This matches the form on page 32.

This incremental update is a specific instance of a more general form presented on pages 36-37:

$$V^\pi(s) \leftarrow V^\pi(s) + \alpha(G_{i,t} - V^\pi(s)) \tag{4}$$

Here, α is a step-size parameter. When $\alpha = 1/N(s)$, it computes a true sample average. However, using a small, constant α is often preferred, especially in non-stationary environments where the reward dynamics might change over time. A constant α gives more weight to recent experiences, allowing the agent to adapt. This general update rule, *NewEstimate* $\leftarrow$ *OldEstimate* + *StepSize* $\times$ (*Target* - *OldEstimate*), is one of the most fundamental ideas in reinforcement learning and forms the basis for more advanced algorithms like Temporal-Difference (TD) learning. [46]

3.3 The Exploration-Exploitation Dilemma and On-Policy vs. Off-Policy Learning

A significant challenge in MC control arises from the policy improvement step. If the policy is deterministic and greedy, it will always choose the action with the highest estimated Q-value. This means if an action initially appears suboptimal, it may never be chosen again. The agent might get stuck in a locally optimal policy, never exploring other actions that could lead to a better overall return. This is the classic **exploration-exploitation dilemma**: the agent must balance exploiting its current knowledge to maximize rewards with exploring new actions to discover potentially better strategies. [5, 10, 13]

To ensure that all state-action pairs are continually visited and evaluated, MC control methods typically use a "soft" policy, such as an **ϵ-greedy policy**. This policy chooses the greedy action most of the time (with probability $1 - \epsilon$) but explores a random action with a small probability ϵ. [5]

This leads to another important distinction in RL algorithms: on-policy versus off-policy learning. [9, 12]

3.3.1 On-Policy Methods

These methods attempt to evaluate and improve the **same policy** that is used to make decisions and generate experience. [9, 14] For example, an on-policy MC control algorithm would use an ϵ-greedy policy to explore the environment and then use the collected returns to improve that same ϵ-greedy policy. **SARSA** is a well-known on-policy algorithm. [7, 8]

3.3.2 Off-Policy Methods

These methods evaluate and improve a policy that is **different** from the one used to generate the data. [9, 14] An off-policy method uses two distinct policies: a **target policy** (the policy it is trying to learn, which is typically deterministic and greedy) and a **behavior policy** (the policy used to explore the environment, which is typically soft, like ϵ-greedy). [8] This allows the agent to learn about the optimal way to behave while still ensuring sufficient exploration. **Q-learning** is a classic off-policy algorithm. [7, 9]

Part II
Synthesis and Comparative Analysis

The algorithms presented—Value Iteration, Policy Iteration, and Monte Carlo methods—represent two distinct philosophical approaches to solving MDPs: model-based planning and model-free learning. Understanding their relative strengths, weaknesses, and underlying mechanics is crucial for selecting the appropriate tool for a given problem.

4 A Comparative Framework for RL Algorithms

The three algorithmic families can be systematically compared across several key dimensions, revealing fundamental trade-offs in reinforcement learning.

4.1 The Bias-Variance Tradeoff

The comparison table illuminates a fundamental tradeoff in reinforcement learning that distinguishes DP from MC methods. This tradeoff is between **bias** and **variance**. [34, 41]

Dynamic Programming (VI and PI) methods are bootstrapping methods. The update for a state's value is based on the estimated values of its successor states. For example, in the VI update $V_{k+1}(s) \leftarrow \dots \gamma V_k(s') \dots$, the term $V_k(s')$ is not the true optimal value but an estimate from the previous iteration. Using an estimate to update another estimate introduces a systematic error, or **bias**. [37] This bias is reduced with each iteration, but it is present throughout the learning process. Because DP methods use a perfect model to compute an expectation over all possible outcomes, there is no **variance** introduced from random sampling.

Monte Carlo methods, in contrast, are non-bootstrapping. The update for $V^\pi(s)$ is based on the full, actual return G_t observed in a sample episode. This return is a noisy but **unbiased** sample of

Feature	Value Iteration (VI)	Policy Iteration (PI)	Monte Carlo (MC) Methods
Paradigm	Model-Based (Planning)	Model-Based (Planning)	Model-Free (Learning)
Model Requirement	Requires full knowledge of $P(s'\|s,a)$ and $R(s,a,s')$	Requires full knowledge of $P(s'\|s,a)$ and $R(s,a,s')$	None (learns from experience)
Core Operation	Iteratively solve the Bellman Optimality Equation for V^*	Alternate between solving the Bellman Expectation Equation for V^π (evaluation) and greedy policy updates (improvement)	Average the returns (G_t) from many complete sample episodes to estimate V^π or Q^π
Bootstrapping	Yes (updates $V_k(s)$ based on the previous estimate $V_{k-1}(s')$)	Yes (updates $V^\pi(s)$ based on estimates of $V^\pi(s')$)	No (updates are based on the actual, full return G_t, not on other value estimates)
Source of Updates	Full expectation over all possible next states s', computed from the model	Full expectation over next states s' for a fixed policy, computed from the model	Sampled trajectories generated by interacting with the environment
Sample Efficiency	N/A (does not use samples)	N/A (does not use samples)	Low (can require a very large number of episodes to converge due to high variance)
Bias/Variance	No variance from sampling; potential bias from bootstrapping on imperfect estimates	No variance from sampling; potential bias from bootstrapping	High variance from sampling random episodes; low bias as it uses true returns
Applicability	Finite, known MDPs where the state space is manageable	Finite, known MDPs; often faster than VI if policy converges quickly	Episodic tasks where a model is unavailable or too complex to build

Table 1: Comparison of RL Algorithm Families

the true value $V^\pi(s)$. [34, 37] The **high variance** arises because the return of any single episode can deviate significantly from the true expected value due to the stochasticity in the policy and environment. However, by averaging an infinite number of these unbiased samples, the estimate is guaranteed to converge to the true value.

In essence, DP methods have low variance but are subject to bias from their own estimates, while MC methods have low bias but suffer from high variance due to their reliance on random sampling. [34]

5 Strategic Algorithm Selection

The choice between DP and MC methods depends entirely on the characteristics of the problem at hand.

5.1 When to use Dynamic Programming (Value or Policy Iteration)

DP methods are the algorithms of choice when a perfect model of the environment is available and the state and action spaces are small enough to be computationally feasible. This is common in problems like board games (e.g., chess, Go, where the rules are the model) or in inventory management and other operations research problems where the system dynamics can be precisely defined.

Between the two, Policy Iteration is often preferred when the number of policies is much smaller than the number of states, as it can converge in very few iterations, even if each iteration is computationally heavy.

Value Iteration is simpler to implement and may be preferred when the state space is small, or as a conceptual building block for more complex algorithms.

5.2 When to use Monte Carlo Methods

MC methods are essential when the environment's model is unknown or intractable. [4] They are the foundational approach for learning directly from raw experience. The primary requirements are that the task must be episodic (so that returns can be calculated) and that it must be possible to generate a large number of sample episodes through interaction. While MC methods can be slow to converge due to high variance, their ability to learn without a model makes them far more broadly applicable than DP. [4] They serve as the conceptual starting point for more advanced model-free algorithms.

5.3 The Path Forward: Bridging the Gap with Temporal-Difference Learning

The limitations of both DP and MC methods naturally motivate a third class of algorithms not covered in the presentation: **Temporal-Difference (TD) Learning**. TD learning combines the key strengths of both approaches: [2, 6, 41]

- Like MC, TD learning is model-free and learns directly from experience. [20, 41]
- Like DP, TD learning bootstraps, updating value estimates based on other learned estimates. [2, 41]

An exemplary TD update rule (for an algorithm called TD(0)) would be: [6, 21, 44]

$$V(S_t) \leftarrow V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t)) \tag{5}$$

This update is performed at each time step, rather than waiting for the end of an episode like in MC methods. [6, 26] The term $R_{t+1} + \gamma V(S_{t+1})$ is the **TD target**, and the difference between the TD target and the current value $V(S_t)$ is the **TD error**. [21] TD learning forms the basis for highly influential algorithms like Q-Learning and SARSA. [6, 8]