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

An Explanatory Document by Taha Majlesi

July 12, 2025

Contents

Part I

Model-Free Learning from Experience

The previous part of this report detailed Dynamic Programming (DP) methods, a class of algorithms capable of finding optimal policies in a known environment. Their power, however, is predicated on a critical and often unavailable assumption: that a perfect and complete model of the environment's dynamics, specifically the state transition probabilities $P(s'|s,a)$ and the reward function $R(s,a,s')$, is provided to the agent. In the vast majority of real-world scenarios, from a robot learning to navigate a new building to an algorithm trading on the stock market, such a model is a luxury we do not possess.

This fundamental limitation necessitates a paradigm shift from **planning** within a known model to **learning** from direct interaction. This part of the report delves into the world of model-free reinforcement learning, where an agent must learn to behave optimally by trial and error, using only the raw experience it gathers from its environment.

1 The Planning vs. Learning Distinction

The most crucial dividing line in reinforcement learning algorithms is whether they require a model of the environment. This distinction fundamentally changes the nature of the problem and the methods used to solve it.

1.1 Planning with a Model (Model-Based RL)

In the model-based paradigm, the agent is gifted with the "rules of the game." [11, 19] It has full access to the MDP's components, $P(s'|s,a)$ and $R(s,a,s')$. Algorithms like Value Iteration and Policy Iteration are quintessential planning algorithms. They leverage this model to perform "thought experiments" or internal simulations.

- **How it Works:** A planning algorithm can compute the expected outcome of any action from any state without actually executing it in the real world. For example, a chess program, knowing the rules of chess (the model), can explore deep into the game tree to evaluate move sequences. [11, 22] The Bellman equations can be solved directly because the expectation over next states, $\sum_{s'} P(s'|s,a)[\dots]$, is fully computable.

- **Primary Advantage (Data Efficiency):** Planning is extremely data-efficient. The agent can "learn" about the entire environment and find the optimal policy without taking a single, potentially costly, real-world step.

- **Primary Disadvantage (The Curse of the Model):** Its greatest strength is its greatest weakness. The approach is entirely dependent on the availability and accuracy of the model. If the model is even slightly inaccurate, the resulting "optimal" policy may be severely suboptimal or even catastrophic in the real environment.

1.2 Learning from Experience (Model-Free RL)

In the model-free paradigm, the agent is thrown into the environment with no prior knowledge of its dynamics. [11, 19] It does not know what state it will end up in or what reward it will receive after taking an action. To learn, it must actively interact with the environment to collect experience, which takes the form of trajectories: $(s_0, a_0, r_1, s_1, a_1, r_2, s_2, \dots)$. [1]

- **How it Works:** The agent uses the collected experience as its sole source of information. Instead of computing expectations with P and R, it *estimates* these expectations by averaging the outcomes it has observed. An agent learning to play a new video game without instructions is a perfect example; it learns which actions are good by trying them and observing the resulting score.
- **Primary Advantage (Generality):** Model-free methods are far more broadly applicable than planning methods. They can be applied to any problem where interaction is possible, even if the underlying rules are immensely complex, stochastic, or completely unknown.
- **Primary Disadvantage (Sample Inefficiency):** The agent must gather a vast amount of experience (samples) to form reliable estimates of values. This can be a very slow process, requiring millions of interactions, which may be expensive or time-consuming in the real world.

The Bellman optimality equation for Q-values highlights this difference:

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

A planning algorithm computes the summation on the right-hand side directly. A learning algorithm, lacking P and R, must find a way to approximate this value by sampling experiences from the environment.

1.3 The Middle Ground: Learning a Model

It is worth noting that this distinction is not a strict dichotomy. A sophisticated agent can operate on a spectrum between these two extremes. In an approach known as **model-**

based reinforcement learning (or indirect RL), the agent first learns a model from experience and then uses that learned model for planning.

1. **Learn the Model:** The agent interacts with the environment to collect data. It uses this data to learn an approximate model, $\hat{P}(s'|s, a)$ and $\hat{R}(s, a, s')$.

2. **Plan with the Learned Model:** The agent then feeds this approximate model into a planning algorithm like Value Iteration to compute an optimal policy.

This hybrid approach attempts to gain the data efficiency of planning while retaining the flexibility of learning. However, it introduces a new challenge: any inaccuracies in the learned model will lead to biases in the planned policy.

2 Monte Carlo Prediction

Monte Carlo (MC) methods are the most straightforward and intuitive family of model-free learning algorithms. Their core idea is simple and powerful: to learn the value of a state or action, simply experience it many times and average the results. [3, 4]

2.1 Core Principle and Key Characteristics

MC methods learn value functions by averaging the empirical returns observed in sample episodes. To estimate $V^\pi(s)$, the agent, following policy π, runs many complete episodes. Every time state s is visited, the agent records the total discounted reward (the return G_t) that followed that visit. By the law of large numbers, as the number of observed returns for state s approaches infinity, their average will converge to the true expected value, $V^\pi(s)$. [1]

MC methods are defined by a few key characteristics:

- **Model-Free:** This is their defining advantage. They learn directly from episodes of (s, a, r) tuples and require zero knowledge of the environment's transition or reward dynamics. [3, 23]

- **Episodic Tasks:** Standard MC methods are defined only for tasks that are episodic—that is, tasks that are guaranteed to terminate. [4, 23] This is a critical limitation, as the central update mechanism relies on calculating the complete return $G_t = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{T-t-1} R_T$, which is only possible once the terminal time step T is reached.

- **No Bootstrapping:** MC updates are deeply honest. The value estimate for a state $V(s)$ is updated based on the actual, full return G_t that was received. This update is completely independent of the current value estimates of any other states. This stands in stark contrast to DP methods, which *bootstrap*—they update the value of a state based on the estimated (and possibly incorrect) values of its successor states. [1, 4]

2.2 First-Visit vs. Every-Visit MC

A single episode may involve visiting the same state multiple times. This leads to two variants of MC prediction that differ in how they handle these multiple visits.

2.2.1 First-Visit MC

As detailed on page 29 of the presentation, this method adheres to a strict rule: for each episode, it only considers the return that follows the **very first time** a state s is visited. [1, 30] Any subsequent visits to s within that same episode are ignored for the purpose of updating $V(s)$.

- **Algorithm:** 1. Run a full episode under policy π. 2. For each state s that appeared in the episode: 3. Find the time step t of its *first* occurrence. 4. Calculate the return from that point: $G_t = \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$. 5. Add this return G_t to a list of returns for state s. 6. Update $V(s)$ to be the average of this list.

The samples (returns) collected for each state are independent and identically distributed (i.i.d.), which makes this method theoretically clean and unbiased. [33]

2.2.2 Every-Visit MC

As described on page 31, this method is simpler and more liberal with data. [1, 30] It averages the returns for **every single visit** to a state s in every episode.

- **Algorithm:** 1. Run a full episode under policy π. 2. For each time step t in the episode: 3. Let the state be S_t. 4. Calculate the return from that point: $G_t = \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$. 5. Add this return G_t to a list of returns for state S_t. 6. Update $V(S_t)$ to be the average of this list.

If a state is visited three times in an episode, it will contribute three different return values to its average. While slightly biased, 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 ultimate goal of reinforcement learning is not just to predict the value of a policy, but to find the optimal policy—a problem of **control**.

3.1 From Prediction to Control with Q-Functions

To achieve control in a model-free setting, an agent must estimate action-values, $Q(s, a)$, rather than state-values, $V(s)$. The reason is fundamental:

- If an agent only knows $V(s)$, it cannot decide which action is best without a model. To improve its policy, it would need to compute $\arg\max_a \sum_{s'} P(s'|s, a)[\dots]$, which is impossible without P.
- If an agent learns $Q(s, a)$, policy improvement becomes trivial and model-free. It can simply act greedily by choosing the action with the highest Q-value in any given state: $\pi(s) = \arg\max_a Q(s, a)$. [3]

MC control, therefore, applies the same logic of averaging sample returns, but to state-action pairs instead of just states. It follows the general pattern of Generalized Policy Iteration (GPI):

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

This loop of evaluation and improvement is repeated until the policy converges.

3.2 The Efficiency of Incremental Updates

A naive implementation of MC would require storing a growing list of all returns for each state-action pair and recalculating the average after every episode. This is computationally and memory-intensive. [32, 43] Pages 31 and 32 of the presentation introduce a far more elegant and efficient **incremental update** mechanism. [45, 46]

Instead of recomputing the mean from scratch, we can update it with each new return, G, using a simple rule. The update for the N-th return is:

$$V_N \leftarrow V_{N-1} + \frac{1}{N}(G_N - V_{N-1}) \tag{2}$$

This can be generalized to a more flexible form using a step-size parameter, α:

$$\text{NewEstimate} \leftarrow \text{OldEstimate} + \alpha(\text{Target} - \text{OldEstimate}) \tag{3}$$

For MC, the update for $Q(s, a)$ becomes:

$$Q(s, a) \leftarrow Q(s, a) + \alpha(G_t - Q(s, a)) \tag{4}$$

Here, G_t is the "target" we are trying to move our estimate towards.

- **Sample Average with $\alpha = 1/N(s, a)$:** This computes the true average and is guaranteed to converge in a stationary environment. However, the learning rate diminishes over time, making it slow to adapt if the environment changes.

- **Constant α:** Using a small, constant α is common for non-stationary problems. It allows the agent to "forget" old experiences and continuously adapt by giving more weight to recent returns. This is a form of an exponentially weighted moving average. [46]

3.3 The Exploration-Exploitation Dilemma

A major challenge in MC control arises from the "greedy" policy improvement step. If the agent always exploits its current knowledge by choosing the action with the highest estimated Q-value, it may never explore other actions that could lead to a better long-term reward. This is the classic **exploration-exploitation dilemma**. [5, 10, 13]

Imagine an agent finds a strategy that yields a decent reward. A purely greedy policy will get stuck executing this strategy forever, never discovering a different, truly optimal strategy. To find the optimal policy, the agent *must* explore.

The most common solution is to adopt a "soft" policy, such as an **ϵ-greedy policy**. This policy behaves greedily most of the time but, with a small probability ϵ, chooses an action at random. [5]

- With probability $1 - \epsilon$: choose $\arg\max_a Q(s, a)$ (exploit).

- With probability ϵ: choose a random action from $A(s)$ (explore).

This ensures that, over time, all state-action pairs will be visited, allowing their Q-values to be learned accurately.

3.4 On-Policy vs. Off-Policy Learning

The need for exploration gives rise to another crucial distinction in RL algorithms. [9, 12]

3.4.1 On-Policy Methods

These methods "learn on the job." They attempt to evaluate and improve the **exact same policy** that is being used to make decisions and generate experience. [9, 14] For example, an on-policy MC control algorithm uses an ϵ-greedy policy to explore the environment, and the data it gathers is used to improve that very same ϵ-greedy policy. The goal is to find the best possible exploratory policy. **SARSA** is a famous on-policy algorithm. [7, 8]

3.4.2 Off-Policy Methods

These methods are more flexible; they 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** (π): The policy the agent is trying to learn. This is typically the deterministic, greedy policy.
- A **behavior policy** (b): The policy the agent actually uses to explore the environment. This is typically a soft policy like ϵ-greedy.

This separation allows the agent to learn about the optimal (greedy) way to behave while still behaving in an exploratory way to gather information. [8] It can even learn from data generated by other agents or from past, suboptimal policies. **Q-learning** is the 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 fundamentally different philosophies for solving MDPs. The first two are model-based planning algorithms, while the third is a model-free learning algorithm. Understanding their relative strengths, weaknesses, and underlying mechanics is crucial for selecting the appropriate tool for a given reinforcement learning problem.

4 A Comparative Framework for RL Algorithms

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

4.1 The Bias-Variance Tradeoff Explained

The comparison table illuminates the most fundamental theoretical tradeoff in reinforcement learning, which distinguishes DP from MC methods: the tradeoff between **bias** and **variance**. [34, 41]

- **Dynamic Programming's Bias:** DP 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 $V_{k+1}(s) \leftarrow \cdots + \gamma V_k(s')$, the term $V_k(s')$ is not the

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 model** of $P(s'\|s,a)$ and $R(s,a,s')$. Its biggest limitation.	**Requires full model** of $P(s'\|s,a)$ and $R(s,a,s')$. Its biggest limitation.	**None**. Learns directly from sampled experience. Its biggest advantage.
Core Operation	Iteratively applies the Bellman Optimality Equation to find V^*.	Alternates between full policy evaluation (using Bellman Expectation Eq.) and greedy policy improvement.	Averages 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, which is independent of other value estimates.
Bias/Variance	**Biased** due to bootstrapping from its own estimates. No sampling variance.	**Biased** due to bootstrapping. No sampling variance.	**Unbiased** as it uses true returns. **High variance** due to reliance on random sample episodes.
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: A Detailed Comparison of RL Algorithm Families.

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 a full expectation over all possible outcomes, there is no randomness or **variance** introduced from sampling.

- **Monte Carlo's Variance:** MC 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 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. One lucky episode can result in a very high return, while an unlucky one can result in a very low return. Only by averaging a vast number of these high-variance samples can the estimate reliably converge.

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 is not a matter of one being universally superior; it 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 **if and only if** 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 domains where the rules are perfectly defined:

- Board games like chess or Go (the rules are the model).
- Inventory management and logistics problems.
- Certain resource allocation or operations research problems.

Between the two, **Policy Iteration** is often preferred when the number of distinct policies is much smaller than the number of states, as it can converge in very few, powerful iterations. **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**. They are the foundational approach for learning directly from raw experience. The primary requirements are:

- The task must be **episodic**, so that finite returns can be calculated.
- It must be possible to generate a large number of sample episodes through interaction, either in simulation or the real world.

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. They serve as the conceptual starting point for more advanced and practical model-free algorithms.

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

The limitations of both DP and MC methods naturally motivate a third, revolutionary class of algorithms: **Temporal-Difference (TD) Learning**. TD learning brilliantly combines the key strengths of both approaches, creating a method that is both model-free and more efficient than Monte Carlo. [2, 6, 41]

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

The canonical TD update rule (for an algorithm called TD(0)) for learning $V(s)$ is:

$$V(S_t) \leftarrow V(S_t) + \alpha \underbrace{\left(\overbrace{R_{t+1} + \gamma V(S_{t+1})}^{\text{TD Target}} - V(S_t) \right)}_{\text{TD Error}} \tag{5}$$

Let's break this down:

- **Online Learning:** This update is performed at *every time step* $(t+1)$, not at the end of an episode. The agent learns immediately from each transition. [6, 26]
- **The TD Target:** The term $R_{t+1} + \gamma V(S_{t+1})$ is the **TD Target**. Instead of waiting for the full return G_t (like MC), TD forms a target by taking the actual immediate reward R_{t+1} and adding the *discounted current estimate* of the next state's value, $V(S_{t+1})$.
- **The TD Error:** The term in the parentheses is the **TD Error**. [21] It measures the difference between the agent's current estimate, $V(S_t)$, and the better, one-step-lookahead estimate provided by the TD target. This error signal drives the learning.

TD learning hits the sweet spot between DP and MC. It has lower variance than MC because it doesn't depend on an entire random trajectory, only on the next step. It can be used in non-episodic tasks and learns much faster. While it introduces the bias of bootstrapping (like DP), this trade-off is often highly favorable. This powerful combination of model-free learning and bootstrapping makes TD methods, and their control variants like **Q-Learning** and **SARSA**, the foundation of modern reinforcement learning. [6, 8]