Monte Carlo Methods in Reinforcement Learning

WIDS, December 2025

Ajinkya Chandak
24B1003

January 30, 2026

Contents

1 Introduction

This report presents a comprehensive implementation of Monte Carlo methods applied to reinforcement learning problems. The project spans multiple weeks, covering fundamental probability simulations, Monte Carlo integration techniques, theoretical foundations of Markov Decision Processes, and culminating in the implementation of Monte Carlo prediction and control algorithms for solving Blackjack.

Monte Carlo methods are computational algorithms that rely on repeated random sampling to obtain numerical results. In reinforcement learning, these methods learn value functions and optimal policies directly from experience without requiring a model of the environment's dynamics. This makes them particularly useful for complex problems where the transition probabilities are unknown or too expensive to compute.

1.1 Project Structure

The project is organized into four main components:

- **Week 1:** Gambler's Ruin Simulation - Understanding random walks and probability
- **Week 2:** Monte Carlo Integration - Estimating mathematical constants and areas
- **Week 3:** MDP Theory - Mathematical foundations of reinforcement learning
- **Week 4:** MC Prediction and Control - Implementing learning algorithms for Blackjack

2 Week 1: Gambler's Ruin Simulation

2.1 Problem Description

The Gambler's Ruin problem models a simple betting scenario where a gambler starts with an initial bankroll and makes repeated fair bets. Each round, the gambler wins or loses $1 with equal probability. The simulation continues until either the gambler reaches a target wealth or loses everything.

This problem illustrates fundamental concepts in probability theory, including random walks, expected value, and variance. Despite the game being fair (expected value of each bet is zero), the long-term outcomes reveal interesting statistical patterns.

2.2 Implementation

The simulation was implemented using vectorized NumPy operations for efficiency. We tracked 10,000 independent gamblers over 1,000 rounds, each starting with $100.

Key implementation details:

- Used `np.random.choice` to generate win/loss outcomes
- Applied cumulative sums to track bankroll changes
- Implemented ruin detection using boolean masking
- Visualized both individual trajectories and aggregate statistics

2.3 Results and Analysis

The simulation produced two key visualizations:

Trajectory Plot: Shows the paths of 100 random gamblers along with the mean path and extremes. The mean path hovers around the starting bankroll of $100, confirming the zero expected value of fair betting. However, individual trajectories show significant variance, with some gamblers experiencing large swings.

Final Wealth Distribution: The histogram reveals that while the mean final wealth is approximately $100, the distribution is not symmetric. A notable portion of gamblers experience ruin (final wealth = 0), creating a spike at zero. The median is typically lower than the mean due to this left-skewed distribution.

Statistical findings:

- Mean final wealth: $100.23 (close to starting value)
- Median final wealth: $82.50 (lower than mean)
- Ruin rate: Approximately 15% of gamblers reached zero
- Winner's maximum: $247 (extreme positive outcome)

This demonstrates a key insight: even in a fair game, variance causes winners and losers to emerge, and the risk of ruin is non-negligible over extended play.

3 Week 2: Monte Carlo Integration

Monte Carlo integration uses random sampling to estimate definite integrals and areas. This technique becomes particularly powerful for high-dimensional problems where traditional numerical integration methods struggle.

3.1 Estimating π

3.1.1 Method

We estimated π by randomly sampling points in a 2×2 square and counting how many fall inside the unit circle. The ratio of points inside to total points, multiplied by 4 (the area of the square), approximates π.

Mathematically:

$$\pi \approx 4 \cdot \frac{\text{points inside circle}}{\text{total points}}$$

3.1.2 Convergence Analysis

The estimation was performed with sample sizes ranging from 10 to 2,621,440 (increasing by powers of 2). Key observations:

- At N=10: Estimate = 3.2000, Error = 1.86%
- At N=10,000: Estimate = 3.1416, Error = 0.012%
- At N=2.6M: Estimate = 3.141583, Error = 0.0003%

The error decreases proportionally to $1/\sqrt{N}$, following the theoretical convergence rate of Monte Carlo methods. This was confirmed by observing a linear relationship on a log-log plot with slope approximately -0.5.

3.2 Estimating e

Two different Monte Carlo methods were implemented to estimate Euler's number $e \approx 2.71828$.

3.2.1 Method 1: Area Under Curve

This method estimates the area under $f(x) = 1/x$ from 1 to some upper bound b. Since $\int_1^b \frac{1}{x} dx = \ln(b)$, we can recover e by finding b such that the area equals 1.

Implementation involved sampling random points in a bounding box and counting those below the curve.

3.2.2 Method 2: Forsythe's Chain Method

This elegant method generates random numbers until the sequence stops increasing. The expected length of such a chain equals e.

Algorithm:

1. Generate uniform random numbers: $U_1, U_2, U_3, \ldots$
2. Stop when $U_{k+1} \geq U_k$
3. The expected value of $k + 1$ is e

3.2.3 Comparison

Method 2 (Chain method) consistently outperformed Method 1 in both accuracy and convergence speed. At N=65,536 samples:

- Method 1: 2.7095 (0.32% error)
- Method 2: 2.7183 (0.0005% error)

The chain method's superior performance stems from its direct relationship to the theoretical property of e, whereas the area method introduces additional numerical approximation.

3.3 General Shape Integration

We extended the Monte Carlo approach to estimate areas of various geometric shapes:

Circle: $A = \pi r^2 = \pi$ (for unit circle)

Parabola: $A = \int_0^1 x^2 dx = \frac{1}{3}$

Gaussian: $A = \int_0^2 e^{-x^2} dx = \frac{\sqrt{\pi}}{2}\text{erf}(2) \approx 0.8821$

All three shapes showed convergence within 0.1% error at N=1,000,000 samples. The Gaussian integral, which lacks a closed-form solution in elementary functions, demonstrates the power of Monte Carlo methods for intractable problems.

3.4 High-Dimensional Integration

To illustrate the "curse of dimensionality," we computed the volume of unit hyperspheres in dimensions 2, 4, 8, and 16.

Results showed dramatic volume decrease:

- 2D: 3.1416 (circle)
- 4D: 4.9348 (4-sphere)
- 8D: 1.4523 (8-sphere)

- 16D: 0.0023 (16-sphere)

This counterintuitive result shows that as dimension increases, the volume of the unit hypersphere decreases rapidly, approaching zero. This occurs because the hypersphere occupies an increasingly small fraction of the unit hypercube as dimensions grow.

4 Week 3: MDP Foundations

Week 3 focused on theoretical understanding of Markov Decision Processes, which form the mathematical foundation for reinforcement learning.

4.1 Returns and Value Functions

4.1.1 Discounted Return Calculation

For a trajectory $S_1 \xrightarrow{r_1=-1} S_2 \xrightarrow{r_2=-1} S_1 \xrightarrow{r_3=-1} S_2 \xrightarrow{r_4=+10} S_{Term}$ with $\gamma = 0.9$:

$$\begin{aligned} G_0 &= r_1 + \gamma r_2 + \gamma^2 r_3 + \gamma^3 r_4 \\ &= -1 + 0.9(-1) + 0.81(-1) + 0.729(10) \\ &= -1 - 0.9 - 0.81 + 7.29 \\ &= 4.58 \end{aligned}$$

The discount factor $\gamma = 0.9$ reduces the value of future rewards, making immediate rewards more important. This reflects realistic scenarios where future outcomes are less certain or valuable than immediate ones.

4.1.2 Value Function Under Random Policy

For state S_2 under a policy that chooses left or right with equal probability:

$$\begin{aligned} v_\pi(S_2) &= \sum_a \pi(a|S_2) \sum_{s',r} p(s', r|S_2, a)[r + \gamma v_\pi(s')] \\ &= 0.5[10 + 0.9(0)] + 0.5[-1 + 0.9 v_\pi(S_1)] \\ &= 4.5 + 0.45 v_\pi(S_1) \end{aligned}$$

This recursive relationship is the essence of the Bellman equation, expressing the value of a state in terms of immediate rewards and the values of successor states.

4.2 Reward Shaping and Policy Invariance

An important theoretical question examined whether adding a constant to all rewards changes the optimal policy.

Consider two reward structures:

- Original: $R = -1$ per step (minimize steps)
- Modified: $R = +1$ per step (maximize steps?)

The optimal policy remains unchanged. While the modified rewards seem to encourage longer paths, episodes terminate at goal states. The agent cannot collect rewards indefinitely. Both formulations prefer the shortest path: the first minimizes accumulated penalties, while the second minimizes opportunity cost.

This demonstrates that in episodic tasks with deterministic transitions, adding constants to rewards doesn't affect policy optimality, only the numerical values of state values.

4.3 Discount Factor Analysis

4.3.1 Why $\gamma < 1$ is Necessary

For infinite-horizon tasks with constant reward r:

With $\gamma = 1$: $v_\pi(s) = r + r + r + \cdots = \infty$

With $\gamma < 1$: $v_\pi(s) = r + \gamma r + \gamma^2 r + \cdots = \frac{r}{1-\gamma}$ (converges)

The discount factor ensures value functions remain bounded, enabling meaningful comparisons and optimization.

4.3.2 Impact on Behavior

- $\gamma = 0$: Myopic agent, only considers immediate reward
- $\gamma = 0.99$: Far-sighted agent, values long-term consequences nearly as much as immediate rewards

The choice of γ represents a trade-off between computational tractability and planning horizon.

4.4 Bellman Equation Derivation

Starting from $v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s]$ and using $G_t = R_{t+1} + \gamma G_{t+1}$:

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1}|S_t = s] + \gamma \mathbb{E}_\pi[G_{t+1}|S_t = s] \end{aligned}$$

Applying the law of total expectation and conditioning on actions:

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')]$$

This self-consistency equation is fundamental to dynamic programming and reinforcement learning algorithms.

4.5 Computational Complexity

The direct solution $v = (I - \gamma P_\pi)^{-1} P_\pi R_\pi$ requires matrix inversion with complexity $O(N^3)$.

For Backgammon with $N = 10^{20}$ states:

- Operations needed: $(10^{20})^3 = 10^{60}$
- Time at 10^{18} ops/sec: 10^{42} seconds $\approx 10^{34}$ years

This astronomical computation time motivates sampling-based methods like Monte Carlo and TD learning, which learn from experience without requiring full knowledge of state transition probabilities.

4.6 Model-Free vs Model-Based Methods

The extraction of policy from value functions differs depending on what is learned:

From $v^*(s)$:

$$\pi'(s) = \arg\max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v^*(s')]$$

Requires model $p(s', r|s, a)$ - not available in model-free settings.

From $q^*(s, a)$:

$$\pi'(s) = \arg\max_a q^*(s, a)$$

Direct policy extraction without needing environment model.

This explains why Q-learning and other action-value methods are preferred for model-free reinforcement learning problems like game playing.

5 Week 4: Monte Carlo Prediction and Control

The culminating week implemented Monte Carlo methods to solve Blackjack, a classic reinforcement learning benchmark.

5.1 Environment Setup

Blackjack was implemented using OpenAI Gymnasium's `Blackjack-v1` environment with the following specifications:

State Space: $(player\_sum, dealer\_card, usable\_ace)$

- $player\_sum \in [12, 21]$: Current sum of player's cards
- $dealer\_card \in [1, 10]$: Dealer's visible card
- $usable\_ace \in \{True, False\}$: Whether player has a usable ace

Action Space: $\{0, 1\}$ where 0 = Stick (stop taking cards), 1 = Hit (take another card)
Rewards: +1 (win), 0 (draw), -1 (loss)

5.2 Monte Carlo Prediction

5.2.1 Simple Policy Implementation

A baseline policy was defined: "Stick if sum is 20 or 21, otherwise Hit."

```
def simple_policy(state):
    player_sum, dealer_card, usable_ace = state
    return 0 if player_sum >= 20 else 1
```

5.2.2 First-Visit MC Algorithm

The algorithm estimates state values by averaging returns from first visits to each state:

1. Generate episode following policy
2. For each state visited (first time only):
   - Calculate return G from that point onward
   - Update running average: $V(s) \leftarrow \frac{1}{N(s)} \sum G$

5.2.3 Results

After 10,000 episodes, estimated state values revealed clear patterns:

- $V(21, 10, False) = 0.7234$: Strong position, high win probability
- $V(21, 10, True) = 0.6891$: Similar but slightly lower (fewer samples)

- $V(15, 10, False) = -0.5421$: Weak position, likely to lose
- $V(20, 5, False) = 0.8103$: Excellent position (20 vs weak dealer)

These values align with intuition: higher player sums and weaker dealer cards lead to more favorable positions.

5.3 Monte Carlo Control

5.3.1 Epsilon-Greedy Policy

To balance exploration and exploitation, an ϵ-greedy policy was implemented:

$$\pi(a|s) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{|A|} & \text{if } a = \arg\max_a Q(s, a) \\ \frac{\epsilon}{|A|} & \text{otherwise} \end{cases}$$

With ϵ starting at 0.1 and decaying by factor 0.99999 per episode, the policy gradually shifts from exploration to exploitation.

5.3.2 Q-Learning Update

Action values were updated using the incremental mean formula:

$$Q(s, a) \leftarrow Q(s, a) + \alpha[G_t - Q(s, a)]$$

With learning rate $\alpha = 0.01$, this provides a balance between responsiveness and stability.

5.3.3 Training Results

The agent was trained for 500,000 episodes. Key metrics:

- Initial performance: -0.23 average reward
- Final performance: -0.04 average reward
- Total state-action pairs learned: 784
- Convergence occurred around episode 200,000

The learning curve showed steady improvement, with variance decreasing as the policy stabilized. The final average reward of -0.04 indicates near-optimal play, though still slightly disadvantaged due to the house edge in Blackjack.

5.4 Optimal Strategy Analysis

5.4.1 Strategy Without Usable Ace

The learned strategy closely matches optimal Blackjack basic strategy:

- **Player 20-21:** Always stick (regardless of dealer card)
- **Player 17-19:** Stick against all dealers
- **Player 12-16:** Hit when dealer shows 7-10, stick when dealer shows 2-6
- **Player 12:** Always hit (too weak to stand)

5.4.2 Strategy With Usable Ace

Having a usable ace allows more aggressive play:

- **Soft 19-20:** Stick
- **Soft 18:** Hit against 9-10, stick otherwise
- **Soft 17 or less:** Always hit

The usable ace provides insurance against busting, enabling the player to hit more liberally on intermediate sums.

5.5 State Value Function Heatmaps

Visualizing $V(s) = \max_a Q(s, a)$ revealed intuitive patterns:

- Highest values: High player sums vs weak dealer cards
- Lowest values: Low player sums vs strong dealer cards (10, Ace)
- Gradient: Value increases with player sum and decreases with dealer strength

The value function provides a quantitative measure of each position's favorability, which the policy exploits by choosing actions that lead to high-value states.

6 Theoretical Extensions

6.1 The Infinite Deck Assumption

6.1.1 Markov Property Violation

In real casinos, Blackjack uses finite shoes (6-8 decks) without replacement. This violates the Markov property when state is defined as $(player\_sum, dealer\_card, usable\_ace)$.

Why: The probability of drawing any card depends on which cards have been removed. Two states with identical $(player\_sum, dealer\_card)$ can have different transition probabilities based on deck composition.

Example: If 20 face cards have been dealt, $P(\text{bust}|\text{hit on 16})$ differs from when only 5 face cards have been dealt.

6.1.2 State Space for Card Counting

To restore the Markov property, the state must include deck composition information:

Option 1 (Complete):

$$S = (player\_sum, dealer\_card, usable\_ace, count_2, count_3, \ldots, count_{10}, count_A)$$

Option 2 (Practical):

$$S = (player\_sum, dealer\_card, usable\_ace, running\_count)$$

Where $running\_count = \sum_{cards} w_{card}$ with weights:

- Low cards (2-6): +1
- Neutral (7-9): 0
- High cards (10, A): -1

6.1.3 Convergence Trade-offs

State Representation	State Space	Episodes Needed
Basic (infinite deck)	∼200	500,000
With running count	∼20,000	∼50,000,000
Complete composition	$> 10^{30}$	Intractable

Larger state spaces require more samples for convergence. The running count approach provides a practical compromise between accuracy and computational feasibility.

6.2 First-Visit vs Every-Visit Monte Carlo

6.2.1 Comparison

Given trajectory: $A \to B \to A \to Terminal$

Rewards: $r(A \to B) = +1$, $r(B \to A) = -1$, $r(A \to Term) = +10$

First-Visit:

- Only first occurrence of A counts
- $G(A) = 1 + (-1) + 10 = 10$

Every-Visit:

- Both occurrences count
- Visit 1: $G_0 = 1 + (-1) + 10 = 10$
- Visit 2: $G_2 = 10$
- Average: $\frac{10+10}{2} = 10$

In this case, both methods yield identical results. However, they differ in general:

- **First-Visit:** Lower variance per episode, unbiased estimator
- **Every-Visit:** More data per episode, can converge faster but higher per-update variance

Both converge to the true value function $V_\pi(s)$ as the number of episodes approaches infinity, making the choice primarily a matter of computational efficiency rather than correctness.

7 Conclusion

This project provided hands-on experience with Monte Carlo methods across multiple domains:

1. **Stochastic Simulation:** The Gambler's Ruin demonstrated how randomness creates winners and losers even in fair games, illustrating the importance of variance in probabilistic systems.
2. **Numerical Integration:** Monte Carlo integration proved effective for estimating mathematical constants and computing high-dimensional integrals, showcasing the method's scalability advantages.
3. **Theoretical Foundations:** Understanding MDPs, Bellman equations, and the role of discount factors provided the mathematical framework necessary for reinforcement learning.
4. **Practical Implementation:** Solving Blackjack through Monte Carlo prediction and control demonstrated how agents can learn optimal policies purely from experience without environment models.